English

FUJITSU Software BS2000

AID V3.4B

Debugging of C/C++ Programs

User Guide

Edition June 2018

Comments… Suggestions… Corrections…

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

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

Documentation creation according to DIN EN ISO 9001:2015

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

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

Copyright and Trademarks

Contents

1 Preface

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

With AID, the user can debug both on the symbolic level of the relevant programming language as well as on machine code level. Symbolic debugging of a C/C++ program allows you to use the names defined in the source code to address statements, functions and data items and to use the source reference generated by the compiler to address statements which have no name.

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.

With AID, you can debug pure BS2000 or POSIX programs or mixed mode programs. Pure POSIX programs run entirely in the POSIX shell. BS2000 programs which use the POSIX interfaces are known as mixed mode programs.

In addition, the options for accessing the data and statements of a C++ program as described in this manual require the C/C++ compiler as of V3.0.

Please refer to the appendix for an overview of the main differences when debugging programs compiled with the C/C++ compiler as of V3.0 and older objects.

AID provides you with source-based debugging of programs compiled with the new C/C++ compiler, in a graphical interface on your PC as standard. Graphical debugging is a much more convenient way of debugging as the program section currently being executed is always displayed on your screen and you can input AID commands with a simple mouse-click.

1.1 Objectives and target groups of the AID documentation

AID is targeted to all software developers working in BS2000 with the programming languages COBOL, FORTRAN, C, C++, PL/I or ASSEMBH or those who wish to debug or correct programs on machine code level. This manual is intended for those involved in debugging C and C++ 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 Ready Reference, giving the syntax of all the commands and the operands with brief explanatory notes. It also includes 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 topics 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 repertoire of AID commands is presented. The other chapters describe prerequisites for debugging; command input; the subcommand, complex memory reference and medium-and-quantity operands; AID literals and keywords. The manual also includes the BS2000 commands not permitted in command sequences.

AID User Guides

The User Guides deal with the commands in alphabetical order, and they describe all simple memory references. Apart from the present manual,
AID - Debugging of C and C++ Programs,

the available User Guides are:
AID - Debugging of COBOL Programs [3]
AID - Debugging of FORTRAN Programs [4]
AID - Debugging under POSIX [5]
AID - Debugging of ASSEMBH Programs [6]

In these 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 functionality for machine code debugging is described in
AID - Debugging on Machine Code Level [2]

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

Readme file

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

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

Information under BS2000

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

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

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

Additional product information

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

1.3 Changes since the last edition of this manual

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

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

1.4 Notational conventions

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

bold Text to be highlighted is printed in ***bold***. In addition, bold print is also used in the syntax notations to differentiate special characters and lowercase letters which must be entered as shown as opposed to metasyntax elements and operand names. Typical examples include the square brackets `[]`, which enclose the subscript of an array in C/C++, and the `sizeof()` operator, which must always be entered in lowercase.

This symbol identifies points to be specially noted, e.g. cases where different addresses are calculated in AID and C++ even though the syntax is identical (e.g. because AID and C/C++ differ in their treatment of individual address operands, etc).

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.

```
⎧ alternative ⎫
⎨     ...     ⎬
⎩ alternative ⎭
```

```
{ alternative | ... | alternative }
```
: Alternatives; one of these alternatives must be selected. 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 or stands for a *prequalification* (see also the %QUALIFY statement) or is the operator for a byte offset or is 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 debugging

For symbolic debugging, AID requires a "List for Symbolic Debugging" (LSD) which contains the symbolic names defined in a program. This LSD information is generated by the compiler and can be taken over at the time of linkage and also be loaded. This chapter briefly describes the control statements required for generating the LSD with the C/C++ compiler at both BS2000 and POSIX levels. In addition, the following sections also list the operands you have to specify during compiling, linking and loading to create and run a program under POSIX. General information on LSD records, linking, loading, and starting can be found in the chapter on “Prerequisites for debugging with AID” in the AID Core Manual.

In addition, AID offers an option that allows the LSD information to be dynamically loaded if the program was initially loaded without the LSD. The LSD must have been stored with the program concerned in one PLAM library for this. It can have been directly stored there by the compiler during compilation or, if the program was compiled under POSIX, you can copy it with the LSD records from the POSIX file system into a PLAM library. This chapter also contains a brief description of the POSIX `bs2cp` command which you need to transfer the program from POSIX into BS2000.
The final section of this chapter contains a summary of the commands you should always use to start a debugging session.

3.1 Compiling in BS2000

You control the generation of LSD information with the following option of the C/C++ compiler V3.0:

```
//MODIFY-TEST-PROPERTIES TEST-SUPPORT = {*UNCHANGED|*YES|*NO}
```

*UNCHANGED
: The last value defined with a MODIFY-TEST-PROPERTIES statement is taken over. *NO applies if no value was defined in the current compilation run

*YES
: The compiler will generate LSD information.

*NO
: With the presetting NO, the compiler will not generate LSD information. Call backtracing (%SDUMP %NEST) is possible even without this LSD information.

LSD generation is possible for non-optimized programs only. If optimization is turned on anyway (cf. the MODIFY-OPTIMIZATION-PROPERTIES statement), the compiler sets the optimization level to *LOW, and issues a corresponding message.
LSD generation for C++ programs also has an impact on the way functions are generated. Inline functions are generated as outline functions. The option INLINING=*YES, if specified, is reset by the compiler to INLINING=*NO.

Furthermore, note that if you do not plan to dynamically load the LSD information with %SYMLIB when required, you will also have to ensure that the LSD information is included in the compiler statement that controls the linking of the module:

```
//MODIFY-BIND-PROPERTIES ...,TEST-SUPPORT = {*UNCHANGED|*YES|*NO}
```

*UNCHANGED
: The last value defined with a MODIFY-TEST-PROPERTIES statement is taken over. *NO applies if no MODIFY-TEST-PROPERTIES statement was specified in the current compilation run.

*YES
: The LSD information is linked into the module.

*NO
: With the default *NO setting, the LSD information is not linked in.

A further option of the MODIFY-BIND-PROPERTIES statement that affects debugging with AID is STDLIB. This is assigned the value *DYNAMIC by default, which means that the C runtime system is loaded dynamically. In the case of some program errors, e.g. when some portions of the code are overwritten by library functions, it may not be possible for AID to display the entire call hierarchy, i.e., the last function before the error occurred may be missing. If this occurs, you could help yourself by specifying STDLIB=*STATIC at linkage and thus ensure that the runtime system is statically linked to the program (see also the "C/C++ Compiler“ User Guide).

You must specify the following two options if the program uses POSIX interfaces:

– The _OSD_POSIX define must be set before the first #include statement in the program. The simplest way to do this is to specify the following compile option:
```
//MODIFY-SOURCE-PROPERTIES DEFINE = _OSD_POSIX
```

– In order to find the standard include headers during compilation, you must specify the SYSLIB.POSIX-HEADER library, which contains the standard include elements for POSIX functions, in addition to the CRTE library SYSLNK.CRTE.
This can be done with the following option:
```
//MODIFY-INCLUDE-LIBRARIES STD-INCLUDE-LIBRARY=
  (*STD-LIBRARY,$.SYSLIB.POSIX-HEADER)
```

The following option must be set if the program is to read in the parameters for the main function, as is usual with UNIX:

```
//MODIFY-RUNTIME-PROPERTIES PARAMETER-PROMPTING = *YES
```

This causes the program to be halted immediately after starting and you are then prompted to input the parameters for the main function or redirections for stdin/stdout or stderr. Specifying this operand is meaningless if the program is started in the POSIX shell as parameters and redirections are input directly in the command line as in UNIX.

A complete description of the operands which control compilation can be found in the C/ C++ User Guide [9].

3.2 Linking, loading and starting in BS2000

To be able to debug symbolically, you also have to ensure that the LSD information is included during linking, loading and starting.

Compiled programs can be linked, loaded and started by using standard SDF commands which are valid for all languages. These commands are described in the chapter on “Prerequisites for debugging with AID” in the AID Core Manual. The same chapter also describes which parameter is needed to pass the LSD information generated by the compiler to the link editor (BINDER) or the dynamic binder loader DBL. It is also possible to dynamically load LSD information from a PLAM library using the %SYMLIB command (see the section “Loading the LSD dynamically” on page 18).

If you want to use the C runtime system POSIX functions, you must specify the `SYSLNK.CRTE.POSIX` link switch library when linking. The module in this library must be linked in before modules from other CRTE libraries. With dynamic linking using the DBL, you therefore have to assign the `SYSLNK.CRTE.POSIX` library a lower link name BLSLIB*nn* than any subsequent, further CRTE libraries.

Example

```
ADD-FILE-LINK FILE-NAME=$.SYSLNK.CRTE.POSIX,LINK-NAME=BLSLIB00
ADD-FILE-LINK FILE-NAME=$.SYSLNK.CRTE.PARTIAL-BIND,LINK-NAME=BLSLIB01
LOAD-PROGRAM ...
```

If you link statically using BINDER and link in the `SYSLNK.CRTE.POSIX` library with an `INCLUDE-MODULES` statement, this ensures that the module from the link switch library is linked in before the runtime system modules:

```
INCLUDE-MODULES *LIB(LIB = $.SYSLNK.CRTE.POSIX, ELEM = *ALL)
```

More information on the common runtime environment CRTE can be found in the manual “CRTE - Common RunTime Environment” [12].

3.3 Compiling and linking under POSIX

The following POSIX commands are available to you in the POSIX shell for compiling and linking C or C++ programs:

cc, c89 Calls the compiler as C compiler

CC Calls the compiler as C++ compiler

The C/C++ compiler generates LSD information if you specify the `-g` option. Note that this option also suppresses the inlining of functions in the C/C++ source program and the standard optimizations (`-O`).

If you do not specify `-g`, you cannot debug the program symbolically. However, you can debug the program at machine code level.

The `cc`, `c89` and `CC` commands are described in detail in the manual “POSIX Commands of the C/C++ Compiler” [9].

3.4 Loading and starting under POSIX

You use the POSIX `debug` command to load the program with the LSD. This command is described in detail in the chapter “Special notes on debugging under POSIX” on page 295. After loading, AID outputs message AID0348, which contains the process number (pid) of the created process. You are then presented with the debug mode prompt and can input AID commands. You can start the program with %RESUME.

If you load and start the program directly in the POSIX shell, i.e. without using the `debug` command, the program is unloaded if an error termination occurs. In contrast to the BS2000 level, you then have no possibility of examining the error environment and error cause immediately if you want to try to eliminate the error and continue program execution.

3.5 Loading the LSD dynamically

Programs used in production are generally loaded without the LSD. It is also meaningful to load very large programs, in which only separate modules are to be debugged symbolically, without the LSD. In such cases, AID can still access the relevant LSD at a later stage, provided the module was stored together with the LSD in a PLAM library. This is done by specifying the PLAM library containing the program with the LSD information in the %SYMLIB command (see page 278). If you subsequently access a symbolic memory reference with an AID command, AID opens the PLAM library and searches for the required information in it. You can also use this procedure if the program is running in the POSIX shell. Since %SYMLIB does not support accessing POSIX files, the program must be stored with the LSD in a PLAM library in BS2000 in this case as well. If the program was compiled in the POSIX shell, you will need to copy the created object into BS2000 with the POSIX command `bs2cp` and store it there as a type L element in a PLAM library.

It is fundamentally not possible to dynamically load the LSD for programs invoked via an `exec()` call from another program. In this case, you always have to use the procedure described above if you wish to debug symbolically.

3.6 bs2cp

`bs2cp` copies files from the POSIX file system into BS2000 and vice versa. BS2000 files may be DVS files or BS2000 PLAM library elements. You will find a detailed description of `bs2cp` in the manual “POSIX Commands” [11].

3.7 Commands on starting a debugging session

When debugging C/C++ programs, it is advisable to enter the following command at the start of each debugging session:

```
%AID C=YES
```

This enables the handling of `char` arrays as C strings, and thus allows you to work with C strings in AID just as you would in C/C++. At the same time, the setting `C=YES` also enables `LOW=ALL` and `SYMCHARS=NOSTD`.

– `LOW=ALL` means that AID distinguishes between uppercase and lowercase in names from the source program and does not convert lowercase names of translation units and other BLS names into uppercase. All other specifications in BS2000 and AID commands are converted as usual to uppercase. You can thus continue to enter command and operand names and all other inputs in lowercase. Note, however, that if you are not debugging under POSIX, it is better to set `%AID LOW=ON`, since you would then not have to worry about entering the names of translation units in uppercase.

– `SYMCHARS=NOSTD` sets the hyphen to be always interpreted as the minus sign. Since C and C++ do not allow the use of hyphens in names, all hyphens in inputs can only represent minus signs.

Note that the entry `%AID C=NO` does not affect the settings of `LOW` and `SYMCHARS`, i.e. the values `LOW=ALL` and `SYMCHARS=NOSTD`, which are set implicitly by `C=YES`, are not reset by `C=NO`.
The current settings of global parameters can be displayed during a debugging session with %SHOW %AID (see the description of the command %SHOW on page 272).

Immediately after loading, the program counter (PC) is in the superblock. As a result, you can only reference global data and data declared as static. AID requires the appropriate qualification for access. There is no call hierarchy until the program counter is on the first instruction in your program, and only then can AID address local data and execute the %SDUMP command. You get to this point using:

```
%insert main;%r
```

or

```
%trace 1 in s=srcname
```

srcname is the name of the translation unit which contains the `main` function.

In the case of C programs and C++ programs without virtual functions or constructors, both options have the same significance, i.e. the program is halted before the first executable statement of the `main` function.
However, C++ programs usually begin with a compiler-generated function in which, among other things, constructors are interpreted and tables are constructed for virtual functions. Following the %TRACE command, the program is halted at the start of this function. The

name of this function (_ _STI_ _) is generated by the compiler and is also output in the stop message of the %TRACE. To ensure that even a C++ program will always halt immediately before the first executable statement of the main function after loading, you can use the command:

```
%trace 1 in main
```

The following option
OVERFLOW-CONTROL=*USER-ACKNOWLEDGE
must be set with the
/MODIFY-TERMINAL-OPTIONS
command to enable interruption of an extensive AID output with the K2 key.

4 Addressing in C and C++ programs

This chapter describes only those memory references which are used for the symbolic debugging of both C and C++ programs. In chapter 5 you will find descriptions of additional address operands specific to C++ programs. A general description of addressing methods can be found in the chapter on “Addressing in AID” in the AID Core Manual.
All data names and statement names from the program which are listed in the LSD records as well as the source references generated by the compiler can be used as symbolic memory references. In some cases, preceding qualifications may be required as described below .
In all operands in which *compl-memref* is possible, you can choose as you like between the memory references described in this manual and those for debugging at machine code level [2].

4.1 Qualifications

You use qualifications in the following cases:

– If you wish to access a memory object that is not in the current AID work area.

– If the interrupt point is not in the scope of the addressed memory object.

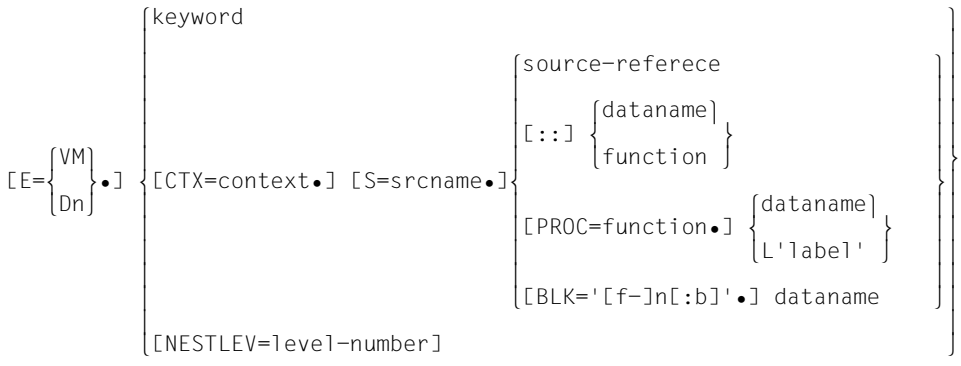
– If the required memory object is hidden by a definition with the same name.

– To designate a contiguous segment of program memory.

There are two qualifications: the base qualification, with which you define the AID work area, and the area qualification, with which you address parts of the work area. A combination of qualifications may be used to describe the path to an area or memory object.

Qualifications are separated by using periods as delimiters. A period is also required between the last qualification and the following operand part. The qualification for the superblock, in which a pair of colons precede the address operand, constitutes an exception; no additional period is inserted in this case between the :: and the address operand.

When debugging C and C++ programs, you can use the base qualification and, as area qualifications, the S, PROC and BLK qualifications. Global data and functions are addressed by means of a prepended ::. Qualifications are represented in the command syntax using the *qua* operand. The following overview shows how qualifications are used:



Base qualification

E={VM | Dn}

The base qualification determines whether the AID work area is in the loaded program (E=VM) or in a dump file (E=Dn). The base qualification is used in the same way for symbolic debugging and for machine-oriented debugging and is described in the chapter on “Addressing in AID” in the AID Core Manual [1] and under the command %BASE on page 127.

Area qualifications

These qualifications enable you to designate a part of the work area. If an address operand ends with one of these qualifications, the effect of the command will be restricted to only the part which was designated by the last qualification. In other words, an area qualification allows you restrict the scope within which a command takes effect and to thus make a data or statement name unique in the work area or to reference a name that is otherwise inaccessible at the current interruption point.

CTX=context

The CTX qualification designates a context (see also the section on “Area qualifications” in the AID Core Manual [1]). It is only in the commands %SDUMP and %QUALIFY that an address operand may end with the CTX qualification. This qualification is only required if identically named translation units are loaded in different contexts and if the desired translation unit can thus be uniquely addressed only via the CTX qualification. *context* may be the context name explicitly assigned in the `BIND` macro or the implicitly assigned name `LOCAL#DEFAULT`. The default context name `LOCAL#DEFAULT` is also assigned to programs loaded with the dynamic binder loader DBL. Further contexts of a program may occur as a result of a link to a shared-code program.
context may have a length of up to 32 positions.

Note that the CTX qualification is not included in the syntax for the address operands of the individual commands, since this would unnecessarily inflate the syntax.

Examples

```
%control1 in ctx=local#default.s=n'list.c'.proc=main
```

Here the *control-area* is not located in the current context in which the program was interrupted, but in the context `LOCAL#DEFAULT`.

```
%sdump ctx=ctxphase
```

The current interrupt point is located here in a different context of the call hierarchy. In this %SDUMP, the command is restricted to the specified context.

```
%insert ctx=local#default.s=n'list.c'.s'30'
```

The translation unit `LIST.C` occurs in both the current context as well as the context `LOCAL#DEFAULT`. The context qualification is required here so that an interrupt point can be defined.

S=srcname

The S qualification defines a translation unit.

In the case of LLMs, the name of the source file must be specified and may occupy up to 32 positions. The C/C++ compiler V3.0 only generates LLMs.

In the case of OMs, srcname is the name of the code CSECT and may thus occupy up to 8 positions.

If the name includes special characters that do not belong to the AID character set, e.g. a period or an “&”, the S qualification must be specified with `n'`*srcname*`'`. More information on constructing module names can be found in the C/C++ User Guide [8] in the section on “Standard name generation”.

AID converts *srcname* to uppercase, even if `%AID LOW[=ON]` is set.

However, if the program was compiled in the POSIX shell and the name of the relevant source program contains lowercase characters, you must set `%AID LOW=ALL`. This is the only way to ensure that uppercase/lowercase is also considered in the S qualification. Note that `LOW=ALL` is set implicitly on entering `%AID C=YES`.

You can use the S qualification to define the area in which the commands %CONTROLn, %TRACE or %SDUMP take effect.
Otherwise, you use an S qualification when you want to reference a name (function, block, data name, label or source reference) from the LSD records which is not within the current program unit.

Note for users debugging on machine-code level:

The CSECT names of LLMs generated with the C/C++ compiler as of V2.1C contain an `'&'` and must be written in `n'...'` in AID commands. More detailed information on working with CSECTs when using AID can be found in the manual “Debugging on Machine Code Level” [2].

NESTLEV=level-number

The NESTLEV qualification defines a level number.

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

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

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

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

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

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

:: You use the pair of colons to address the superblock. You can use this qualification to reference global data that is hidden at the interrupt point by a definition with the same name or to designate global data or functions which are not associated with the current program unit. In contrast to the other qualifications, no delimiting period is entered between the two colons and the following data or function name.

Example
```
%display s=n'list.c'.::name
```
The global variable `name` from the translation unit `LIST.C` is displayed.

PROC=function

The PROC qualification defines a function from the source program.
function is the name assigned in the source program to a function or `main` and can be up to 1000 positions in length.

You can use the PROC qualification to specify the area in which the commands %CONTROLn, %TRACE or %SDUMP take effect.
Otherwise, you use a PROC qualification when you want to reference a data name declared as static or a statement name (label) which is not associated with the current function. In addition, you use a PROC qualification when you want to reference a data name which is associated with the current function, but which is hidden at the interrupt point by a local definition with the same name, e.g. if a variable with the same name is defined in an inner block.

BLK='[f-]n[:b]'

The BLK qualification defines a block. As with source references, the name of a block is formed from the line number and, in some cases, the FILE number as well as the relative block number.

The outermost pair of braces in a function encloses the entire function and is not a block for AID. All definitions found there are associated with the function and are referenced with the corresponding PROC qualification. The second pair of braces in a function begins a block which you can reference with a BLK qualification.

f FILE number; it is specified only for lines which were inserted because of an `#include` or `#line` directive (see the section “Functions, labels and source references” on page 50).

n Line number in which the block begins; you can find it in the source error listing, column `SRC-LIN`; it is identical with the line number in the source file.

b Relative block number within a line; it can be found from the number of left braces in a line, where only the braces for statement blocks are considered. The braces for `struct`, `union` and `enum` declarations are not counted. The first brace in a function counts as a relative block number even if it cannot be referenced with a BLK qualification.

b is a number > 1; you specify it only if you do not want to reference the first block in a line. The *b*-th block in the line is then specified.

Using the BLK qualification, you can define the area in which the commands %CONTROLn, %TRACE and %SDUMP take effect.
Otherwise, you specify a BLK qualification when you reference a data name declared as static which is associated with a block outside the current call hierarchy. In addition, you can use a BLK qualification when you reference a data name which is associated with a block within the current call hierarchy and is hidden at the interrupt point by a definition with the same name.

keyword

The keywords are described in the AID Core Manual in the chapter on “Keywords” [1]. You will also find them in the descriptions of the commands in which they are used.

dataname

dataname is described in the section “Data names” on page 29.

{L'label' | source-reference | function}

label, source-reference and *function* are described in the section “Functions, labels and source references” on page 50.

4.1.1 Associating data with translation units, functions and blocks

The addressing methods of AID take into consideration the specifics of scope in the C or C++ programming language. Names declared extern are valid in the whole program; parameter names and labels have validity within a function. Names which were declared in a block are valid within that block only.

Global data is defined outside of all functions and are associated with the superblock . Local data is always associated with the function or the block in which they are defined.

The AID work area comprises the complete non-privileged address area that is occupied by your loaded program or the corresponding area in a memory dump and is defined by means of the base qualification. All names that lie within the AID work area defined with %BASE can be referenced without an explicit base qualification.
All names which lie in some other translation unit always require an S qualification and a PROC or BLK qualification appropriate to their scope, or a pair of colons (::) if the names of global data or functions are involved.
Names which lie within the current translation unit require a PROC and possibly a BLK qualification if they are associated with another function of the same translation unit. Names in the current function require only a BLK qualification if they are locally hidden by a definition of the same name or are associated with a function block that is not in the current call hierarchy.

All names that are valid at the interrupt point, i.e. which could also have been used within the program at that interrupt point, can be referenced without qualification; however, in the case of identical names, this is only the first definition that is found by AID within the current call hierarchy (from the innermost to outermost level).
Top-level definitions with the same name can be referenced only with qualification. You always use the qualification which corresponds to the scope of that name in C or C++.

Examples

1. Function parameters

```
C program
=========================================
 1   #include <ctype.h>
 2   ...
10   int main(int argc, char *argv[])
11   { ... }
=========================================
```

The `argc` parameter is referenced as follows, where `proc=main` in the second possibility is an overqualification which AID ignores.

```
argc
proc=main.argc
```

2. Nested blocks

```
C program
=========================================
 1   #include <stdio.h>
 2   int main(void)
 3
 4   { int a; struct s {int i;}; {
     1                          2
 5              static int b;
 6              ...
 7              b++; }
 8       printf("%d\n", a); }
=========================================
```

The current interrupt point is at source reference 8. The variable `b` is referenced with the following block qualification:

```
blk='4:2'.b
```

3. Global external declarations

```
C program – translation unit TEST2.C
=========================================
 9   extern double d;
10   int main(void) {
11   int f1(void);
12   d = PI;
13   ...
14   {
15   int d = 15;
16   ...
=========================================
```

Let us assume that the program consists of three translation units: the variable `d` is assumed to be defined in `TEST1.C`; it is simply declared in `TEST2.C`, and is not used in `TEST3.C`. Variable `d` can be referenced as follows:

– if the interrupt point is S'13':
  `%display d`

– if the interrupt point is S'16', and there is a local `d`:
  `%display ::d`

– if the interrupt point is in `TEST1.C`:
  `%display d` or `%display ::d`

– if the interrupt point is in `TEST3.C`:
  `%display s=n'test1.c'.::d`

  or
  `%display s=n'test2.c'.::d`

4.2 Data names

AID allows you to reference the following types of data:
– simple (scalar) types
– arrays and array elements
– C strings
– structures/unions and structure/union components
– enumeration (`enum`) constants
– bit-fields
– pointers

You cannot reference the following with AID:
– preprocessor constants and macros (`#define`)
– typedef names
– enumeration, structure and union types (tags)

As a rule you can reference data as in C/C++, with the following exceptions:

– Array elements can only be referenced via subscripts, not by means of pointers.

– With variables of type `long double`, AID evaluates only the first 8 bytes.

– You cannot use variables of type `char` with AID as an arithmetic type in an expression. You can only calculate with `char` variables after adding a type modification to the data name. %A converts the data type to `unsigned char` and %F to `signed char`.
A variable of type `signed char` is also handled by AID as a signed integer variable. You can use the contents of such a variable in an expression without type conversion. The same applies to variables of type `unsigned char`.

dataname
stands for all data names defined in the source program.

dataname is usually specified as in the source program. AID takes a maximum of 1000 characters into account. If `%AID LOW={ON|ALL}` is set, AID distinguishes between uppercase and lowercase. As in C/C++, C keywords such as `int`, `char`, etc., are not allowed in AID and are rejected as syntax errors.

dataname may be used in all commands for outputting and modifying data, i.e. the commands %DISPLAY, %MOVE, %SDUMP, and , %SET. In addition, *datanname* can be specified in the %FIND command (to locate strings) and in the %ON command (for write monitoring).

AID provides the following formats:
– subscript notation
– pointer notation
– structure qualification
– dereferencing

These formats can also be combined, i.e. in any of the formats, *dataname* can be replaced by any of the other formats.

4.2.1 Subscript notation

```
dataname [ [subscript] {...} ]
```

You can use subscript notation to access arrays and type-related pointers. The subscript is specified as in a C/C++ statement in brackets. The brackets used for subscripting are printed in boldface in this manual in order to distinguish them from the brackets of the metasyntax.

AID also provides the option of using the array name without a subscript, thus designating the complete array.

subscript

The subscript can have a value of between -2^{31} and $+2^{31}-1$ and comprises:

– an integer,
– a variable of type `int` or
– an arithmetic expression

The arithmetic expression can contain the arithmetic operators (+, -, /, *), integers, and numeric variables. The numeric variables used in a subscript cannot be qualified. They must therefore be visible at the interrupt point or, if *dataname* is qualified, the variables from *subscript* must be visible in the range designated by the qualification.
The variables used in the subscript can be specified in the same way as *dataname*, i.e., they can be subscripted, pointer or structure qualified, or dereferenced.
It must be noted that when a subscripted entry in a `%ON %WRITE(...)` is input, AID immediately calculates the start address and length of the range to be monitored. This means that if the contents of *subscript* change during a program run, thus changing the start address of the range designated with *dataname*`[`*subscript*`]{...}`,

the range defined when `%ON %WRITE(...)` was input is still monitored. In contrast to this, entries in subcommands of the %CONTROL*n*, %INSERT and %ON commands are only evaluated when the monitored event occurs, i.e. *dataname*`[`*subscript*`]{...}` must be visible at the point of the program run where the event occurs, e.g. when the test point is reached, but not when the command is input.

Accessing a single element of an array requires as many subscripts as have to be specified for access in a C/C++ statement.

If you specify *subscript* in the form *subscript1*`:`*subscript2*, you designate the range between *subscript1* and *subscript2*.
The following applies for *subscript1* and *subscript2*:
Both must lie within the subscript limits and *subscript1* must be less than or equal to *subscript2*.

If you use an asterisk (*) for *subscript*, you designate the complete subscript range of the dimension. For single dimension arrays, this is the same as using the array name without a subscript. This may not be followed a type or length modification.

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

Examples

1. `%DISPLAY array [*][3]`

   Outputs all elements from the first dimension of a two-dimensional array whose second dimension are 3.

2. `%DISPLAY array [1:3][*][5:15]`

   Outputs the elements from a three-dimensional array whose:
   – first dimension subscript is 1, 2 or 3,
   – second dimension subscript is anywhere within the subscript limits and
   – third dimension is 5 through 15.

Using arrays with AID

Working with arrays in AID **differs from** conventional C/C++ methods, since an additional option allows you to use an array name without a subscript:

1. Referencing an array in the scope of its definition

   In the function or block that contains the array definition, *dataname* without a subscript references the entire array. For example,
   `%FIND X'...' IN` *dataname* and
   `%ON %WRITE(`*dataname*`)` search and monitor the entire array, respectively.

   %DISPLAY and %SDUMP edit all array elements with the subscript and associated content as a table. This also applies to character arrays if `%AID C=NO` has been set. Note, however, that if you have enabled the interpretation of character arrays as C strings by specifying `%AID C=YES`, AID will display the array contents as a contiguous character string. The handling of C strings with AID is discussed in a separate section, starting on page 36.

2. Referencing an array as a transfer parameter

   When an array is passed to a function as a parameter in a function call, the array name in that function includes only a pointer to the passed array. Consequently, only the start address of the array is known in that function. This means that you can address individual array elements as usual via a subscript, but if you use the parameter name without a subscript here, you will effectively designate the start address of the array. The only way to reference the array as a whole in this case is with a following pointer operator and appropriate length modification::
   `%DISPLAY` *parametername*`->%`*type*`L`*length*
   For C strings, you can specify C for *type*, which causes the string to be output in character format. For arrays of other data types, e.g. `int`, only the value X is meaningful for *type*; this causes the array to be output in hexadecimal representation.

   The currently occupied contents of a character array can be output by using the following two commands:
   `%FIND X'00' IN` *parametername*`->%L`*n*
   `%DISPLAY` *parametername*`->%CL=(%0G -` *parametername*`)`
   The first command (%FIND) determines the address of the null byte. AID saves this address in the AID register %0G. The following %DISPLAY command outputs the string up to the null byte in character representation.

Examples

```
C program                              SOURCE: PARR.C
================================================================================
SRC
LIN
  1   void foo (char*,int*);
  2   char a[25] = "abcdefgh";
  3   char *p = &a[5];
  4   int  iv[] = {0,10,20,30,40};
  .
  .   main()
  .   {
  .
 25     foo(a, iv);
  .
  .   }
  .
  .   void foo(char* str, int* nr)
  .   {
  .
 80     printf("String str:\n%-25s\n",str);
 81     printf("Array nr:\n");
 82     for (i=0; i<5; i++) printf("%6i",nr[i]);
  .
  .   }
================================================================================
```

1. Interrupt point in `main`:

   Output of an individual array element:

```
SRC_REF: 25 SOURCE: PARR.C    PROC: main     ****************************
/%display a[6],p[1]
a( 6)           = |g|
*               = |g|
```

   Every seventh element of array `a` is output.
   The header line contains the source reference of the interrupt point and the names of the current translation unit and current function.

   Output of the entire array in dump format:

```
/%d a%x
CURRENT PC: 01000098    CSECT: PARR$O&@ ********************************
V'0100111A' = a      + #'00000000'
0100111A (00000000) 81828384 85868788 89000000 00000000    abcdefgh.....
0100117A (00000010) 00000000 00000000 00                    .........
```

Due to the type modifier %X at the end, AID outputs the entire array `a` in hexadecimal and in character representation. The hexadecimal output can be used to determine the position of the null byte.
Since AID switches to machine code level, an additional header line containing the current status of the instruction counter and the name of the associated CSECT is output.

Output of the `char` array as a C string:

```
/%aid c=yes
/%d a
SRC_REF: 25 SOURCE: PARR.C  PROC: main  ******************************
a               = "abcdefgh"
```

`%AID C=YES` enables the interpretation of `char` arrays as strings. The subsequent %DISPLAY outputs the occupied part of the string as a string literal in "...".

Output of a numeric array:

```
/%d iv
iv( 0: 4)
( 0)          0  ( 1)         10  ( 2)         20  ( 3)         30
( 4)         40
```

Since the array name was specified without a subscript, AID edits all array elements in a table and outputs them.

2. Interrupt point in the function `foo`:

   Output of the start address and a single element of the array:

```
SRC_REF: 80 SOURCE: PARR.C    PROC: foo      ***************************
/%d str,nr,str[6],nr[3]
str              = 0100111A
nr               = 01001180
*               = |g|
*               =         30
```

Unlike in `main`, specifying the unsubscripted array name causes the address of the array to be output, since the array is passed as a pointer. Subscripted specifications are also possible exactly as in `main`; however, since the element is referenced via a pointer, AID outputs an asterisk instead of the element name.

Output of the complete character array:

```
/%d str->%xl(::a)
CURRENT PC: 0100020A    CSECT: PARR$0&@  ******************************
V'0100111A' = PARR$0&# + #'0000011A'
0100111A (00000000) 81828384 85868788 89000000 00000000    abcdefgh.....
0100112A (00000010) 00000000 00000000 0000                 ........
/%f x'00' in str->%xl(::a)
PARR$0&#+00000123=01001123 : 00000000 00000000 00000000  ............
/%d str->%cl=(%0g-str)
V'0100020A' = PARR$0&# + #'0000011A'
0100111A (0000011A) abcdefgh
```

As in `main`, it is also possible in the `foo` function to output the complete character array in dump format (first %DISPLAY). If you want to output only the allocated string in character format, however, you must first use %FIND to look for the null byte. This information is used in the second %DISPLAY to calculate the length of the string.

Output of a numeric array:

```
/%d ::iv
SRC_REF: 80 SOURCE: PARR.C    PROC: foo      ****************************
iv( 0: 4)
( 0)          0  ( 1)         10  ( 2)         20  ( 3)         30
( 4)         40
/%d nr->%l(::iv)
CURRENT PC: 0100020A    CSECT: PARR$0&@  ******************************
V'01001180' = PARR$0&# + #'00000180'
01001180 (00000180) 00000000 0000000A 00000014 0000001E    ............
01001190 (00000190) 00000028                               ....
/%d nr->.8%fl4
01001188 (00000188)         +20
```

In the case of arrays with numeric elements, it is not possible to have the complete array edited via the name of the transfer parameter. You can, how-ever, always reference the whole array by means of the appropriate qualifications (i.e. with the two colons in this case, since a global data item is involved) and the name with which the array was defined. The second %DISPLAY shows how you can address the whole array on machine code level via the parameter name. The contents of the array are output in dump format (hexadecimal and character representation).
The last %DISPLAY outputs the third element of the array as an integer value.

4.2.2 C strings

Starting with Version V2.3B, AID supports the string notation of C/C++, provided you have enabled the option `%AID C=YES` (see page 115). This means that you can enter C string literals as in C/C++ within quotes ("...") and that `char` arrays are no longer treated as an array of individual `char` elements, but as strings (as in C/C++).
This functionality can also be used in older C/C++ objects that were compiled with earlier compiler versions < V3.0.

4.2.2.1 C string literals

C string literals are entered in the following form:

"x...x" Maximum length: 1000 characters on input; unrestricted for output.

x can be any printing or non-printing character. Non-printing characters must be specified with an alternate representation. The alternate representation begins with an escape character, i.e. a backslash (\), after which you can enter the value of the character in different ways:

– Hexadecimal representation `\x`*ff*:

  Character set for *f*: 0-9, a-f, A-F

  Value range from `00` to `FF`

  The hexadecimal value must always be specified with two positions.

– Octal representation `\`*ooo*:

  Character set for *o*: 0-7

  Value range from `000` to `377`

  The octal value must always be specified with three positions.

– Symbolic alternate representation:

  For some characters which cannot be printed, e.g. the bell character, specific alternate representations have been defined, so you need not know the hexadecimal or octal value of the character. Other characters such as the backslash itself (\) or double quotes ("), though printable, can only be entered in combination with a backslash.

All alternate representations are summarized in the following table:

Alternate representation	**Hexadecimal value**	**Meaning**
`\a`	X'2F'	bell character
`\b`	X'16'	backspace
`\f`	X'0C'	page feed
`\n`	X'15'	newline
`\r`	X'0D'	carriage return
`\t`	X'05'	horizontal tabulator
`\v`	X'0B'	vertical tabulator
`\\`	X'BC'	backslash
`\?`	X'6F'	question mark
`\'`	X'7D'	single quote
`\"`	X'7F'	double quote
`\x`*ff*	X'*ff*'	hexadecimal number
ooo	-	octal number

Table 1: Alternate representations and their meanings

In the output, AID always selects the printable equivalent of the character when possible, regardless of the format in which the character was entered. Non-printing characters are mapped to the alternate representation, if possible in the symbolic form. Bit combinations which represent non-printing characters and for which no symbolic representations have been defined are displayed in hexadecimal form.

C string literals can be used in the AID commands %DISPLAY and %SET and also in comparisons within a subcommand. A C string literal can only be transferred to a `char` array with %SET. If the literal is longer that the receiving field, it is truncated to the right, and AID issues a warning. Alternatively, if the literal is shorter than the receiving field, the field is padded with binary zeros. This means that you can set a `char` array to binary zero by simply transferring an empty literal ("") to it.

Comparisons with a C string literal in a subcommand are only allowed if the second relational operand is a `char` array.

Note that the handling of `char` arrays as C strings involves only a "high-level" functionality, so you cannot transfer C string literals with a %MOVE. Furthermore, the receiving field in a %SET must be designated with a symbolic memory reference, so a command such as

`%SET "abc" INTO V'...'`, for example, is not possible, and any such input is rejected as a syntax error. AID likewise rejects the use of a C string literal in all AID commands, except for %DISPLAY and %SET, as a syntax error.

If support for C string literals has been enabled with `%AID C=YES`, you must enclose comments within /*...*/. Comments within "..." are no longer recognized by AID as such and are always interpreted as C string literals, which usually results in a syntax error.

When AID commands are executed in a procedure, no parameter substitution occurs in C string literals, since the BS2000 command interpreter always interprets entries within quotes as comments, regardless of whether or not `%AID C=YES` has been set.

Example

```
/%set "\xC5\x25\x15" into cstr; %d cstr
cstr            = "E\x25\n"
```

Three characters are transferred to the `char` array `cstr` with %SET and then displayed. For the first character, which was specified with "\xC5", AID displays an "E", since "E" is represented in hexadecimal notation with C5; the second character is output as a hexadecimal number, since no corresponding printable character exists for " \x25". The third character "\x15" appears in the output as "\n", which is the symbolic alternate representation for a
newline character.

4.2.2.2 C string arrays

If `%AID C=YES` is enabled, `char` arrays are interpreted by AID as C strings. If the `char` array has only one dimension, the C string begins with the first array element and ends with the array element with the value X'00'. Multidimensional `char` arrays represent arrays of C strings, and since the last subscript is processed first, the array elements addressed via the last subscript are combined into C strings.

%DISPLAY outputs the contents of a one-dimensional `char` array as a C string literal. You specify the name of the array without a subscript. The editing of individual characters occurs as specified in the rules listed in the preceding section.
In the case of multidimensional `char` arrays, the array elements belonging to the subscript on the extreme right are combined to form a C string, i.e. an array of C strings is displayed. Following the array name, you specify one subscript less than the subscript levels contained in the definition of the array. The end criterion in each case is X'00'. If further array elements are set after X'00', these are not taken into account in the output.
When you specify a `char` array with a subscript range in the %DISPLAY command, the array is split into individual array elements in the output, even if `%AID C=YES` has been set.

C string arrays can be overwritten with %SET, where the sender can be a C string literal or another C string array. The sender is entered into the receiver up to (and including) the end criterion X'00', and the following applies:

– If the sender is longer than the receiver, it is truncated to the right, and AID issues a warning.
– If the sender is shorter than the receiver, the excess positions on the right are padded with X'00'.

In a multidimensional array, the array elements associated with the last subscript level can be transferred or overwritten as a C string.

A single `char` array element or `char` literal in the form '*x*' cannot be transferred to a C string array, but a C string literal consisting of only one character, i.e. "*x*", can naturally be transferred.

Note that the aspects applicable to the transfer of C strings must also be considered when comparing C string arrays in a subcommand. A C string array can only be compared with another C string array or with a C string literal. The comparison of a C string with an individual `char` character and the transfer of an individual character to a C string are both rejected with the following message:

```
AID0388    Types are not convertible.
```

Example

The `char` array `carray` is defined and initialized in a C program as follows:

```
char carray[3][10]={"1","22","333"};
```

```
/%aid c=yes
/%d carray
carray( 0: 2)
( 0) "1"  ( 1) "22"  ( 2) "333"
/%aid check=all
/%s "ab\n" into carray[1]
OLD CONTENT:
"22"
NEW CONTENT:
"ab\n"
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

The command `%AID C=YES` causes AID to interpret `char` arrays as C strings. Consequently, the array elements of the second subscript level are combined into C strings in the output of the following %DISPLAY. The command `%AID CHECK=ALL` then turns on the update dialog, and the following %SET overwrites the string in `carray[1]` with the character string `"ab\n"`.

4.2.3 Pointer notation

```
dataname1 -> dataname2
```

You can use pointer notation in AID only to reference structure components via pointers. *dataname1* must be a pointer type. As in a C/C++ statement, this refers to *dataname2*, which AID processes according to its type and size attributes.

Example

```
p1 -> var
```

As in C/C++, you refer - beginning with the address stored in `p1` - to the structure component `var`.

4.2.4 Structure qualification

```
top-level dataname• {...•} dataname
```

You can use structure qualification to reference components of structures as in C/C++. The first *top-level dataname* is the name of the structure. Any further *top-level datanames* are the names of structure components nested within it. The last *dataname* is the name of the structure components you want to reference. AID processes these components according to their type and size attributes. As of C/C++ V2.1C, you must specify all levels of the structure in an AID command (exactly as in a C/C++ statement) from the first *top-level dataname* down to the component that you wish to address.
For the first time, the LSD created by the C/C++ V3.0 compiler contains the relationship between base classes and derived classes, allowing AID to recreate the scope rules applicable in C++ for accessing components from class systems. You can now access data and function members from base and derived classes without qualification or with partial qualification as long as the component concerned is uniquely identified.

4.2.5 Dereferencing

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
[(]* {...} dataname[)]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

You can use dereferencing in AID on pointers only, not on arrays. The indirection operator (*) is used as in a C/C++ statement, which means that repeated use is permitted. The entire statement may be placed in parentheses. *dataname* is the name of a pointer which (perhaps by way of other pointers) points to a memory object.

Pointer arithmetic for dereferencing pointers can be expressed in AID only in subscript notation (see Example 4 below for details).

Examples

```
C-Program
================================================================================
     ...
     struct
     {   int x;
         char y;
         float *z3;
     } z, *p, p1[5];
================================================================================
```

1. `z.x`
   is the structure component `x` in the structure `z`.
2. `p->y`
   is the structure component `y` in the structure pointed to by `p`.
3. `*p`
   is the entire structure pointed to by `p`.
4. `*(p1+4)->z3`
   This C/C++-style address specification yields a syntax error in AID. To address the required memory location in an AID command you should enter the following: `*p1[4].z3`.

4.2.6 Operator precedence

The operators for symbolic addressing in AID have the same precedence levels as in C/C++. The operators ->, • and [ ] have the same precedence level, and all of them have a higher precedence level than *. AID also allows you to use parentheses to change precedence.

Be careful with the second operand of the period operator: if it is enclosed in parentheses, AID performs a byte offset (see Example 2).

Examples

These examples are related to the definition of structure `z` on the preceding page.

1.
```
*p -> z3
*(*p).z3
*p[0].z3
p[0].z3[0]
p -> z3[0]
(*p).z3[0]
```

   All notations have the same meaning: `p` is a pointer which points to a structure with the component `z3`. `z3` is itself a pointer which is addressed via pointer `p` and then referenced. The meaning is as in C/C++.

2.
```
*(*p).(z3)
```

   This expression has a different meaning than in C/C++: Since `z3` is in parentheses, AID treats the contents of `z3` as a value for a byte offset. These contents must be of type %F or %A; otherwise, AID rejects a byte offset. The result is 4 bytes of type %X.

4.2.7 The address operator & and the address selector %@(...)

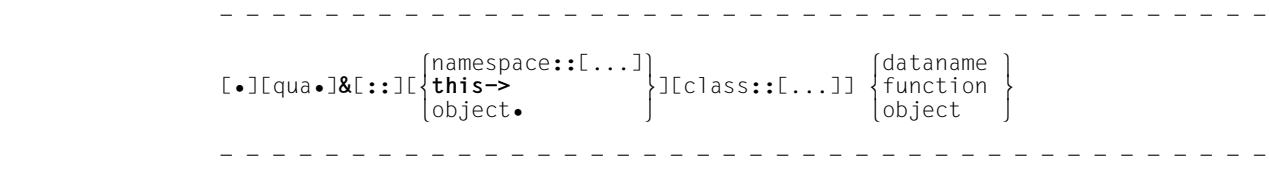
Two options are available in AID in order to access the addresses of data when debugging C/C++ programs: the address operator &, which you know from the C/C++ language, and the AID address selector %@(...), which you can use independently of the respective programming language. You can output the address of a data item with a %DISPLAY command and pass the address of a data item with a %MOVE or %SET command. Apart from the AID commands %DISPLAY, %MOVE and %SET, data addresses can be used in the comparison of a subcommand or in expressions.

The memory object addressed by adding the pointer operator, i.e. &...-> or %@(...)->, is of type %XL4 for AID, but you can also declare some other type and length with a type and length modification or use the type selector %T(...) to apply a data type defined in the source program with its associated length on the address.

The address operator & can be applied on all symbolic addresses from C/C++ programs and returns the absolute address of the accessed data item in memory. Bit-field and register variables are not allowed as an argument. The AID address selector %@(...) can be used for data names of other programming languages or complex memory references. You will find a detailed description of this in the section on “Address, type and length selector” in the AID Core Manual.

When an address determined with the address operator & is transferred with %SET, the receiving field must be of type pointer. AID does not perform any further checks, so the data type of the sender, for example, need not match the data type that is referenced by the pointer specified as the receiver. Such checks are performed by AID only when the address of a class object is set to a pointer to a class (see “object” on page 43).

Syntax of the address operator &:



qua Base or area qualification

If a base or area qualification is required, this must be entered before the address operator.

{:: | namespace:: | class::}
Qualification for the global area, namespace or class qualification

The :: qualification for the global area or a namespace or class qualification is appended to the address operator if required. The operand of the address operator must not end in *namespace* or *class*; otherwise, AID issues an error message (AID0480).

this this pointer

The this pointer saves the address of the current object associated with a member function.

It can be used in combination with a following pointer operator in the path to a component of a class.

object Name of a class object

object is used with an appended period to define the path to a component of a class.
If the operand of the address operator ends with an object name, it effectively designates the start address of the object.

If you want to use %SET to transfer the address of an object to a pointer to a class, the following must apply:

– The class to which the receiver points must match the class whose address is to be transferred

  or

– it must be the base class of the class assigned to the sender. This base class must have a unique subobject in the class of the sender.

dataname

Name of a data item

dataname is specified as in the source program. It thus designates the start address of the data item.

You can address data as in C/C++, with the following exceptions:

– An array name without a subscript returns the address of the first element in the array.

– Individual array elements can be address only via subscripts, not via pointers.

– If `%AID C=YES` is set (see page 115), AID returns the array elements of a `char` array corresponding to the last subscript level as C strings. The start address of the C string can be obtained with the appropriate subscript specification.

– The start address of array elements addressed via a subscript range cannot be determined.

– Arrays that were passed as parameters to a function serve as pointers to the arrays in the calling program; the address operator & returns the address of the pointer and not the address of the array.

For more details on working with arrays, see also the sections “Subscript notation” on page 30 and “C strings” on page 36.

dataname can be specified as follows, and the formats may also be combined (see the section “Data names” on page 29). The precedence rules of C/C++ apply:

Subscript notation:	*dataname* `[`*subscript*`] {...}`
Pointer notation:	*dataname1* `->` *dataname2*
Structure qualification:	*superordinate dataname*`.` `{...}` *dataname*
Dereferencing:	`[(]*{...}` *dataname*`[)]`
Pointer to member dereferencing:	*dataname1*`.*`*datanname2* `or` *dataname1*`->*`*datanname2*

If *dataname* is a dynamic data member of a class, AID returns either the absolute address of the data item or the relative address of the data member, i.e. the address relative to the start of the class, regardless of the interrupt point or any preceding qualification. This occurs as follows:

– The absolute address of the data item is selected if the program is interrupted outside the class and the data member is addressed via a class object or if the program is interrupted in a dynamic member function of the class and the data member can either be addressed directly of by means of an appropriate preceding class qualification which you have entered to establish uniqueness.

– The offset to the start address of the class is selected if the program is interrupted outside the class and the data member can be addressed via a class qualification. If the program is interrupted in a dynamic member function of the class, you can access the relative address only if you are addressing the associated class from an external point and via an area qualification (S, PROC or :: qualification).

Even the offset to the start of the class can only be transferred to a pointer with %SET, but not to a numeric variable.

For static data members, the address operator & always returns the absolute address.

function

Name of a function

A C function from a translation unit that was compiled with the option
`//MODIFY-SOURCE-PROPERTIES LANGUAGE=C(...)`
can be addressed in AID by name. The two trailing parentheses with the passed parameters (signature) are omitted.
More details on how to specify the names of C++ functions in AID are presented on page 58.

You can use &*function* to specify the function address in an AID command. The same address can also be accessed with *function* without an address operator; however, if you want to transfer the function address with a %SET command to a pointer, you will need to specify &*function*.

Example

```
C++ program                              SOURCE: EXADR.C
================================================================================
SRC
LIN
...
 20  class X {
 21    int a;
 22    static int c;
 23    public:
 24    void  gx(int) {...; return;}
 25  }x;
...
 68  int  main()
 69  {
 70    x.gx(3);
...
```

```
/%in s'70';%r
STOPPED AT SRC_REF: 70, SOURCE: EXADR.C , PROC: main
/%d &x.a,&x.c
010010F0
010010EC
/%d &X::a,&X::c
00000000
010010EC
```

In the first step, the program is interrupted in `main` before the `gx(int)` function call. The absolute address of the dynamic data member `a` and the static data member `c` are displayed. The second %DISPLAY, in which the data members are addressed via the associated class qualification, returns the offset to the start of the class for `a`, but the absolute address for `c`, since `c` is static.

```
/%in x.n'gx(int)';%r
STOPPED AT SRC_REF: 24, SOURCE: EXADR.C , PROC: X::gx(int)
/%d &a,&c,&X::a,&X::c
010010F0
010010EC
010010F0
010010EC
/%d &::X::a
00000000
```

In the next step, a test point is set at the start of the member function `gx(int)` and the program is executed up to that point. The data members `a` and `c` can now be addressed directly. Note, however, that the absolute addresses are now displayed even with the preceding class qualification `X::`, since the interrupt point lies in a member function of class `X` (first %DISPLAY).
If you want to determine the relative address of `a` even from this point, you will need to enter the two colons for the global block before the class qualification. This enables AID to access `X` from an external location (second %DISPLAY).

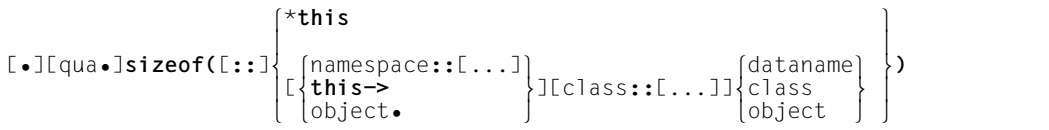
4.2.8 Length operator sizeof() and length selector %L(...)

You can use the length operator `sizeof()`, which you know from C/C++, and the AID length selector %L(...) to determine the lengths of data items. `sizeof()` can only be used when debugging C/C++ programs, whereas the length selector %L(...) can be used independently of the programming language of the program being debugged. The AID length selector is described in detail in the section on “Address, type and length selector” in the AID Core Manual [1].
The length of a data item can be output with %DISPLAY and transferred with %MOVE or %SET. You can also use the result of a length selection in a comparison of a subcommand or in an expression.
Function names are not allowed as operands, so neither the length operator nor the length selector can be used to determine the length of a function.

The length operator `sizeof()` can be applied on all symbolic addresses from C/C++ programs. It returns the length of the addressed data item. Bit-field and register variables are not allowed as arguments.

Note that `sizeof()` must be specified in lowercase as in C/C++ and that `%AID LOW={ON|ALL}` must also be enabled.

Syntax of the length operator `sizeof()`:



qua Base or area qualification

If a base or area qualification is required, it must precede the length operator.

{:: | namespace::}
Qualification for the global block and namespace qualification.

The :: qualification for the global block or a namespace qualification are appended to the address operator if required. The operand of the address operator must not end in *namespace*; otherwise, AID issues an error message (`AID0480`).

class Name of a class

A class name followed by the two appended colons can be used as a class qualification in the address path for a data member of the class. The operand of the length operator may also end with *class*. The result is the same as if `sizeof()` were applied on an object of *class*. You will receive the number of bytes occupied by an object of that class, including any fill bytes that may be required to place an object of the class in an array.

this this pointer

`this` points to the current object associated with a member function. `sizeof(*this)` thus returns the length of that object.

The `this` pointer can be used with an appended pointer operator in the path to a component of a class.

object Name of a class object

object followed by a period defines the path to a component of a class.
If the operand of the length operator ends with the name of an object, you will receive the length of the object. This length corresponds to that of `sizeof(`*class*`)` if *class* is the name of the class associated with the object (see above).

dataname
Name of a data item

dataname is specified as in the source program and effectively designates the length of the data item.

Data can be accessed as in C/C++, but with the following exceptions:
An array name without a subscript designates the total length of all array elements.
If `%AID C=YES` is set (see page 115), AID combines the array elements of a `char` array that can be addressed via the subscript on the extreme right into C strings. Note, however, that `sizeof()` does not return the length of the C string in this case, but the overall length of the underlying C string array.
In contrast to C/C++, the memory requirements for a selected subscript level cannot be determined with AID.
Individual array elements can be addressed only via subscripts, not pointers. When an array is passed to a function as a parameter in a function call, only the start address of the array is known in that function. If you use the parameter name without a subscript in the function, you will thus effectively designate that address. `sizeof()` can be applied on the passed parameter and always returns a result of 4.

dataname can be specified as follows, and the formats may also be combined. The precedence rules of C/C++ apply (see the section “Data names” on page 29):

Subscript notation:	*dataname* `[`*subscript*`] {...}`
Pointer notation:	*dataname1* `->` *dataname2*
Structure qualification:	*superordinate dataname*`.` `{...}` *dataname*
Dereferencing:	`[(]*{...}` *dataname*`[)]`
Pointer to member dereferencing:	*dataname1*`.*`*datanname2* `or` *dataname1*`->*`*datanname2*

Example

```
/%d sizeof(carray)
          30
/%aid c=yes
/%d sizeof(carray[0])
          10
```

Let us assume that `carray` is a `char` array with the following definition:

```
char carray[3][10];
```

The first %DISPLAY shows the total length of the array.

After setting `%AID C=YES`, you can use `sizeof()` to display the length of the C string (in `carray`) addressed via the second subscript. The total number of all array elements associated with the second subscript is output here, regardless of the position of the end criterion X’00’ within the underlying array of the C string.

4.3 Functions, labels and source references

Functions and labels are names from the source program under which statements can be addressed in AID. Labels and functions from C and C++ programs are stored as address constants. They hold the address of the first instruction in a function or after a label.
You should avoid assigning identical names for functions and labels, since AID cannot then tell whether the function or the label is meant.
AID interprets a maximum of 1000 characters for all names.

Source references are generated by the compiler for each executable statement. They are address constants which contain the address of the first instruction generated for a statement.

For labels and source references, the address stored in the address constant is identical to the address of the associated executable statement. In the case of functions, by contrast, the address constant holds the start address of the function prolog, which precedes the first executable statement in the function. Consequently, when localizing addresses, e.g. in `%D %HLLOC(...)`, AID cannot associate the prolog addresses with the corresponding function.

Functions, labels, and source references can only be used with a following pointer operator in the %FIND and %ON *write-event* commands. In other words, you thus designate 4 bytes of the machine code as of the address held in the address constant, i.e. the start address of the prolog in the case of functions. This defines the address in %DISPLAY, %MOVE, and %SET. In %DISASSEMBLE and %INSERT, the functions, labels, and source references always reference the first executable statement which follows the address entered in the address constant. In the commands %CONTROLn and %TRACE, you can define an area by means of two source references.

function
: is the name of a function as declared in the source program or the name of a library function. Any C function from a translation unit compiled with the option `//MODIFY-SOURCE-PROPERTIES LANGUAGE=C(...)` or a library function can be addressed in AID by name. The two trailing parentheses with the passed parameters (signature) are omitted.
For details on how to specify the names of C++ functions in AID, please refer to the description on page 58.

L'label'

label is a label declared in the source program.You can only refer to those labels to which the C/C++ program can jump from a `goto` statement. You cannot refer to `case` and `default` labels.
In the commands %DISASSEMBLE and %INSERT you can also specify *label* without `L'...'`, since in these commands confusion with a data name is not possible.

source-reference

is the statement designation generated by the compiler. It is in the following format:

S'[f-]n[:a]'

You may not include blanks within the single quotes.

f `FILE` number; only to be specified for lines inserted due to an `#include` statement, and only if the inserted lines contain executable statements or if a `#line` directive was used to explicitly specify line numbering for the following line. The `FILE` number can be obtained from the `FILE-NO` column of the source error listing.
f is a number > 0.

n Line number that is found in the column `SRC-LIN` of the source error listing; identical to the line number of the source file if the source program contains no `#include` or `#line` statements.
If a single statement extends over multiple lines, *n* is the line number of the first line in the statement.
If a source program without `#include` or `#line` statements was processed with the "Beautify" function of the C structurizer, there will only be one statement in each line, and the source references will then consist of only the line number: S'*n*'.

a Relative statement number within a line; it can be found from the number of statements in the line. *a* is a number > 1 which you specify only if you do not want to address the first statement in the line. Specifying *a* designates the *a*-th statement in the line.

If you wish to specify an area by means of two source references in the %CONTROL*n* or %TRACE command, you should note that ascending source references correspond to ascending addresses only within a function block. Furthermore, note that additional source references, which do not appear in the source error listing but are logged by %TRACE, are generated in connection with implicit constructor and destructor calls as well as conversion operations in C++ programs.

Example

```
FILE    SRC
 NO     LIN
================================================================================
0       100          i++;
                     #line 17 "incl.h"
1        17          j++; k++;
================================================================================
```

With S’100’ you address the statement `i++;`.

The line `#line 17 "incl.h"` causes the compiler numbers lines from 1 to 17, starting with the next line.
With S’1-17’ you address the statement `j++;`.
With S’1-17:2’ you address the statement `k++;`.

4.3.1 Special notes on addressing statements

A statement is understood here in the sense of ANSI C and is defined in its grammar. The following lists the AID-specific special characteristics.

1. **Definitions and declarations of data and functions**

   These are not statements, unless a definition includes initialization of the data item. Data is addressed as described in section “Data names” on page 29. Functions are addressed as described in the section “Functions, labels and source references” on page 50. Accessing the additional language constructs provided by C++ is dealt with in the chapter “C++-specific addressing” on page 57.

2. **Labeled statements**

   Labels are neither statements themselves nor components of statements, and they cannot be addressed by means of a source reference. Labels which can be jumped to with a goto statement can be addressed in AID with the statement name L’*label*’.

Example

```
SRC-LIN    Statement
-------------------------------------------------------------------------------
 99        lab5:
100                  a = b; c = d;
                     1      2
101        ...
102        case 'a': foo();
                     1
103        ...
104        default:  i = 0; break;
                     1      2
```

With `L'lab5'` you reference the address of the statement `a = b;`, i.e. the address of the first statement after the label `lab5`.

With S’100’ you reference the same statement `a = b;`.
With S’100:2’ you reference the statement `c = d;`.

You cannot address `case` and `default` labels using `L'...'`.

With S’102’ you address the function call `foo();`
With S’104’ you address the statement `i = 0;`.
With S’104:2’ you address the statement `break;`.

3. **Compound statements or blocks**

A compound statement allows you to combine a number of statements in a block and use them in places where the grammar of the C/C++ language only permits a single statement. AID considers neither the block itself nor the opening brace or closing brace to be statements. No source reference is generated for a compound statement. It can be addressed only as a block by means of the BLK qualification. The individual statements within a block can be addressed with source references.

Example

```
SRC-LIN    Statement
-------------------------------------------------------------------------------
 30        { a = b; f(a); }
             1      2
```

BLK=’30’ designates the compound statement, but that only lets you define the area for %CONTROLn or %TRACE.

S’30’ designates the assignment `a = b;`.
S’30:2’ designates the function call `f(a);`.

4. **Expression statements**

   An expression statement is only one statement. You cannot address assignments, function calls, etc., within an expression statement, because no source references are generated for them.

   Example

```
SRC-LIN    Statement
-------------------------------------------------------------------------------
 40         a = f(b);
 41         c = a > x ? b : a;
 42         a = (b = c);
 43         ...
```

   Only one source reference is generated per statement:

   With S'40' you reference the statement `a = f(b);` in line 40.
   With S'41' you reference the statement `c = a > x ? b : a;` in line 41.
   With S'42' you reference the statement `a = (b = c);` in line 42.

5. **Selection statements**

   These are the if, if else and switch statements. They are considered by AID as by C/C++ to consist of a number of statements, all of which you can reference using source references.

   Example

```
SRC-LIN    Statement
-------------------------------------------------------------------------------
 50        if (a < 0) a++; else a--;
           1          2        3
 51        if (a < 0) {a++;} else {a--;}
           1           2          3
 52        ...
 53        switch ( c = getchar()) {
           1
 54           case 'X': look('y');
                      1

 55           case 'Y': look('z'); return;
                      1          2
 56           ...
```

   With S'51' you reference the control expression of the if statement in line 51.
   With S'51:2' you reference the then branch of the if statement.
   With S'51:3' you reference the else branch of the if statement.

   With S'55' you reference the case `'Y'` (line 55).
   With S'55:2' you reference the return statement.

6. **Iteration statements**

   These are the `while`, `do` and `for` loop constructs. They are considered by AID as by C/C++ to consist of a number of statements, all of which you can address with source references. In a `for` loop, there is an additional source reference for the control expression and for the incrementation portion.

   Example

```
SRC-LIN    Statement
-------------------------------------------------------------------------------
 60        while (p->next) mknode(p->next);
           1               2
 61        ...
 62        do show() while( tick );
           1  2      3
 63        do { x++; z(x); } while(x);
           1    2    3       4
 64        ...
 65        for (i=0; i < 10; i++) j[i] = i;
               1    2       3    4
 66        a = b;
           1
 67        ...
```

   With S’65’ you reference the initialization `i=0` of the for loop in line 65,
   with S’65:2’ the control expression `i < 10`,
   with S’65:3’ the incrementation portion `i++`, and
   with S’65:4’ you reference the execution portion of the for loop `j = i;`.
   With S’66’ you reference the first statement after the `for` loop.

7. **Jump statements**

   These are the goto, continue, break and return statements.

   Example

```
SRC-LIN    Statement
-------------------------------------------------------------------------------
 70        goto label1;
           1
 71        ...
 72        if (i < 10) continue;
           1           2
 73        ...
 74        if (k >= 1) return k * 2;
           1           2
 75        ...
```

   With S’72:2’ you reference the statement `continue;` in line 72.
   With S’74:2’ you reference the statement `return k*2;` in line 74.

5 C++-specific addressing

In addition to the language elements offered by C, C++ supports namespaces, classes, templates, virtual functions, overloaded functions and operators, reference variables and other new features. This chapter describes how you can reference these elements in AID commands.

5.1 Qualifications

In C and C++ program debugging there are no differences in usage between base and area qualifications. For more details refer to section “Qualifications” on page 21.

AID provides the additional qualifications *class*`::` and *namespace*`::` for accessing data and functions from classes and namespaces.
Certain points also have to be noted with the notation of C++ function names and when specifying instances of function or class templates.

class/namespace::[...]

This qualification is used to designate a class or namespace or to specify the path to a class or namespace for derived or nested classes or nested namespaces. You append the name of the data item or function you wish to reach via this qualification directly to the last colon-pair of the qualification.
If you want to designate the instance of a class template with the *class*`::` qualification, you have to use the following syntax: `t'`*k_template*`<`*arg*`[,...]>'::`
If there is only one instance of a class template, you can access this instance with the name of the class template and omit the template argument: `t'`*k_template*`'::`

Detailed information on constructing template instance names can be found in the section “Template instantiation” on page 94. Classes are described on page 63 and namespaces on page 85.

Example
```
%DISPLAY ::A::B::i
```
In this example, `B` is a class nested in class A and `i` is a static data member in `B`. Class `A` is defined as global.
In this case, you can output `i` from any position of the translation unit with the above %DISPLAY command.

PROC qualification

With C++, you must note that functions are not assigned directly to a translation unit as in C, they can also be defined within namespaces or classes. To access these functions, you prepend the function name with the relevant *class*/*namespace*`::` qualification.
There are also overloaded functions with C++ that have the same function names but whose arguments (signature) differ and there are functions that result from instantiation from a function template. You must take all these points into account when writing a PROC qualification for a C++ function. This results in the following syntax:

```
PROC=[namespace::[...]][class::[...]]function
```

namespace::
: Namespace qualification, possibly multi-level.

class::
: Class qualification, possibly multi-level. You have to use the form described above if the class is an instance of a class template: `t'`*k_template*`<`*arg*`[,...]>'::`

function
: The following functions are discriminated in C++:
  – normal functions which correspond to those in C
  – overloaded functions
  – virtual functions
  – functions formed from a function template via instantiation

  You have to use a particular notation, depending on the type of function you want to specify in the PROC qualification:

  – With normal and overloaded functions, you have to include the signature to uniquely identify the function. The `void` signature must be omitted. In this case, you designate the function with the function name and subsequent parentheses as is also possible in C++. Since this means that the function names can also include special characters (parentheses and possibly commas) they have to be set in `n'...'`. This results in the following syntax: `n'`*function*`([`*signature*`])'`

In contrast to the functions described above, the `main` function and the compiler-generated function `__STI__` (see section “Constructors and destructors” on page 72) can be addressed in C++ with just their names. Similarly, all functions with C linkage that are called in a C++ program are addressed only via the function name, i.e. without parentheses or a `signature`.
Example: You qualify data from the `main` function as in C with `PROC=main`.

– You have to enclose the instance name of a function template in `t'...'`. You specify the template arguments in angled brackets with commas as separators, resulting in the following syntax:
`t'`*f_template*`<`*arg*`[,...]>([`*signature*`])'`

  If there is only one instance of a function template, you can access this instance with the name of the function template and omit the template arguments: `t'`*f_template*`([`*signature*`])'`.

  A detailed description of the way template names are constructed can be found in the section “Template instantiation” on page 94.

– Functions from classes addressed via pointers such as virtual functions or functions addressed via a pointer to member cannot be used in a PROC qualification; however, you can access the start address of a function that is referenced accordingly. The way to do this is described in the section “Virtual functions” on page 73 and in the section “Pointer to function member” on page 79.

The two notations `n'...'` and `t'...'` are handled differently by AID:

– If the name is enclosed in `n'...'`, AID accepts it without checking or modifying it, while retaining uppercase/lowercase. This means that no additional blanks may be inserted within `n'...'` and the name can be up to 1000 characters in length.

– If the name is enclosed in `t'...'`, AID assumes that it is a template instance and checks the syntax and semantics of the name. If AID detects that it is not a legal template instance name or if the specified instance does not exist, it outputs corresponding error message. Names enclosed in `t'...'` may be of any length.

If you want to access a function which is defined in a local class, a complete, explicit qualification comprises specification of an additional PROC qualification for the top-level function containing the definition of the local class, followed by the PROC qualification with which you designate the desired function. If the local class is in an inner block of the top-level function, you have to write one or possibly more BLK qualifications between the two PROC qualifications. How you specify a BLK qualfication is described on page 25.
You qualify a function from a local class with the following syntax:

```
PROC=top-level_fct•[BLK='[f-]n[:b]'•[...]]PROC=class::[...]function
```

You can only access functions defined in local classes from inner blocks with AID if the program was compiled with C/C++ V3.0B. In programs compiled with C/C++ V3.0A, you can neither specify these functions in a PROC qualification nor can you refer to the start address of such a function in an AID command.

Since locally defined member functions are also implemented globally in C++, this means that when working with AID, the data block containing the object definition and the object name itself do not belong to the scope of the function and you can only access the data of the block and the object name from within such a function via a corresponding area qualification. However, you can always access the object concerned via the `this` pointer (see page 64).

You will find an example of functions from local classes following this section (example 3 on page 62).

Example

Using member functions and overloaded functions in the PROC qualification

```
C++ program                          SOURCE: EXP.C
================================================================================
SRC
LIN
  1   extern "C" int printf (const char*,...);
  2
  3  int FOO(int X)  {return X++; }
  4  long FOO(long X){return X--; }
  5  class A_global
  6  {
  7    public:
  8    A_global(void)  { printf("Constructor called\n");... };
  9    ~A_global(void) { printf("Destructor called\n");... };
 10    void f(void)    { static int k; printf("f called\n");
 11                      k = 5;... return;};
 12  } a_global;
...

 68  int main(void)
 69  {
 70    FOO(1);
 71    FOO(1L);
...
120    {
121      class A_local
122      {
123        public:
124        A_local(void)  { printf("Constructor called\n");... };
125        ~A_local(void) { printf("Destructor called\n");... };
126        void f(void)   { static int k; printf("f called\n");
127                         k = 5;... return;};
128      } a_local;
...
```

1. `%trace 1 in proc=n'FOO(int)'`

   AID halts before the only statement of the function `FOO(int)` and traces it.

2. `%control1 %proc in proc=A_global::n'~A_global()'`

   The program is to halt before execution of the first and last statements in the destructor of global class `A_global`.

3. `%display proc=main.proc=A_local::n'f()'.k`

   Static variable `k` of the member function `f()` defined in local class `A_local` in the outermost block of the `main` function is output.

5.2 Data defined in the middle of a block

In contrast to C, data can also be defined in the middle of a block in C++.
As with C++, such data can also only be addressed with AID after its definition. If a qualification is required, you must specify the entire enclosing block or the associated function in a BLK or PROC qualification

Examples

```
C++ program                          SOURCE: BSPI.C
================================================================================
SRC
LIN
...
120  int main()
121  {
122     int i = 1;
123     {
124        i++;
125        int i = 3;
126        i++;
127        ...
```

1. `%insert s'124'; %resume`
   `%display i, proc=main.i`

   Due to the two commands %INSERT and %RESUME, the program is executed until source reference S'124' and then interrupted. With both `i` and `proc=main.i` from the %DISPLAY command, you designate the same `i` from the `main` function.

2. `%trace 3`
   `%display i, proc=main.i`

   %TRACE executes the program until source reference S'126'. In this case, the first operand of the %DISPLAY command designates variable `i` from block '123', which is defined in line 125. Variable `i` from `main` is addressed as above with `proc=main.i`.

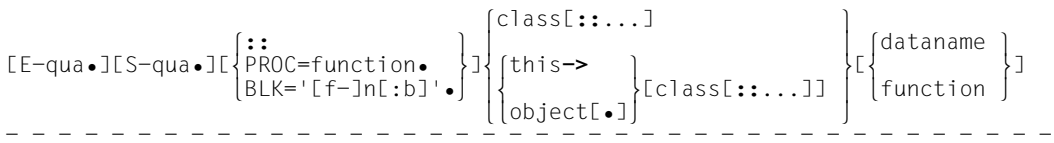
5.3 Classes

In AID, classes and the data and functions contained in them can be accessed in the usual C++ notation, which means that data members of nested or derived classes can be addressed exactly as in C++, depending on the interruption point, by specifying all the class/object names (from the outermost to the innermost) that define the path to the data item. With nested classes, all intermediate levels must always be specified as well and with derived classes only the intermediate levels required to uniquely identify the data item have to be specified.
With member functions, apart from virtual functions, you describe the path to the desired function in a prepended class qualification (*class*`::`, see the section “Qualifications” on page 57). You will find a description of how you can access virtual functions in the section “Virtual functions” on page 73.

Objects of classes are assigned to the relevant scope as with the structures in C. Therefore, objects of classes that are defined globally are accessed by prepending :: as with all other global data. Locally defined objects of classes are assigned to the function/block containing the definition.

If you specify the class name in a %DISPLAY or %SDUMP command, you will receive a listing of the static data members and of the static and dynamic member functions (excluding virtual functions) of that class and any classes nested in it (for derived classes, including the base classes). In the case of data, the content is output; for functions, the complete function name with the class names and signature in standard C++ notation and also the start address of the function prolog are listed.
The whole class, i.e. also including both dynamic data members and virtual functions, is output by AID if you specify the name of a class object in a %DISPLAY or %SDUMP command. You are also shown the complete contents of the class if the program has been interrupted in a dynamic member function of the class and you access the class with `*this` or the class name.

As far as AID is concerned, it is immaterial whether data and functions within classes are declared as public, private or protected. Access rights defined in the program have no effect during the debugging run.



class:: You specify the name of a class for *class*. You specify *class* in the first position if you want to access the complete class, a static data item or a function of the class. You must insert a colon-pair (`::`) between the class name and the data/function name. In an intermediate position, you use the class name to access a base class from a derived class, if a base class component, which is hidden by a definition of the same name in the derived class, is to be addressed.

If the class concerned is an instance of a class template, you must use the following syntax:
`t'`*k_template*`<`*arg*`[,...]>'`
If there is only one instance of the class template, `t'`*k_template*`'` suffices.
Detailed information on templates can be found on page 94.

this A C++ compiler-generated pointer that points from a dynamic member function to the associated object. The pointer operator and any other base class names (or the names of the outer classes of a nested class) that may be required in the path to the desired data name can be appended to the `this` pointer in the same way as you define the path to selected class data item, starting with the object name. However, with derived classes you only have to specify the intermediate stages if the name of the required data item is not unique.
Since the *this* pointer can only be used if the program was interrupted in a dynamic member function and since it always points to the associated object, no preceding area qualification is required.

AID shows you the pointer and its contents in the %SDUMP output for a dynamic member function.

`%DISPLAY this` outputs the address of the current object.
`%DISPLAY *this` provides you with a listing of the current object.
If an address operand ends at the `this` pointer followed by the pointer operator (->), the first 4 bytes of the current object will be addressed; storage type %X applies.

object Is the name of a class object. You specify *object* for all dynamic data members whenever the interrupt point is not in a dynamic member function of the associated class. You also use *object* to uniquely identify a data item of a class which is locally hidden by an identically named definition at the interrupt point.

5.3.1 Scope rules in classes

The scope rules known from C++ apply for accessing data and functions defined in classes.

You can access static data members via the class qualification independent of an object of the class and the current interrupt point by prepending the class name and inserting a colon-pair between the class and data names. With nested classes, you describe the path to the required data item via multiple class levels from the outermost to the innermost, using the colon-pair (`::`) to separate the class names.

Dynamic data members are accessed differently, depending on where the program was interrupted:

– If the interrupt point is in a dynamic class member function, you can access the associated dynamic data member directly. AID behaves in the manner known from the C++ scope rules. The complete object concerned is accessed via the `this` pointer with `*this` (see page 64). You can only reach the object name with the appropriate qualification.

  Since AID emulates the relationship between base and derived classes, the scope rules applicable in C++ also apply for accessing data members from derived or base classes.

  The information on base and derived classes is missing in the LSD for older objects compiled with a C/C++ compiler up to V2.2C. AID therefore does not know the relationship between base and derived classes for these objects. In this case, the method described in the previous manual for AID V2.1A must be used to access data from class systems.

– If the current interrupt point is **not** in a dynamic member function of the same class in which the data members are also described, you can only access the dynamic data via the associated object. You specify the object name as with a structure qualification in a C program (see the section “Data names” on page 29) and insert a period between the object name and data name. If you want to access a dynamic data member from a class system of nested classes, you have to include the class name of the superordinate levels in the path to the required data item, starting from the current object. You also have to add a colon-pair after a class name in this case. In contrast to this, the C++ scope rules apply within base and derived classes so that after the object name, you only need to add the class name required to uniquely identify the data item.

Examples

1. Accessing data and functions from classes from different interrupt points

```
C++ program                              SOURCE: VPTR.C
================================================================================
SRC
LIN
...
 20  class X
 21  {
 22    int        a;
 23    static int b;
 24    int        c;
 25    public:
 26    X(int x = 1) : a(x) {c=2; ...; return;}
 27    void        f()     {...; return;}
 28    static void g()     {...; return;}
 29  };
 30  int X::b = 3;
 31
 32  class Y : public X
 33  {
 34    int        a;
 35    static int b;
 36    public:
 37    Y(int x = 4) : a(x) {...; return;}
 38    void        f()     {...; return;}
 39    static void g()     {...; return;}
 40  };
 41  int Y::b = 6;
 42
 43  class Y        y;
...
 68  int  main()
 69  {
 70    y.f();
 71    Y::g();
...
```

```
/%in s'70'
/%in ::Y::n'f()'
/%in ::Y::n'g()'
/%r
```

For the sake of readability, user input in the trace is printed in bold.

First the %INSERT commands set three test points in the program:
– in `main` before the call to the member function `Y::f()`
– in the member function `Y::f()`, before the first executable statement
– in the member function `Y::g()`, before the first executable statement

%RESUME starts the program, and it runs as far as the first test point.

```
STOPPED AT SRC_REF: 70 ,  SOURCE: VPTR.C ,  PROC: main
/%d y.X::a, X::b, y.c
y.X::a          =           1
X::b            =           3
y.c             =           2
/%d X::n'f()',X::n'g()'
X::f()      = 01000628
X::g()      = 01000730
/%d y.a, Y::b, Y::n'f()',Y::n'g()'
y.a             =           4
Y::b            =           6
Y::f()      = 01000D48
Y::g()      = 01000E50
/%r
```

The interrupt point is located in `main`. The dynamic data members of object `y` of class `Y` can be addressed as in a structure qualification via the object name with a subsequent period. Data members of the base class `X` are addressed with `y.X::`*dataname*. The static variables from the base and derived classes (both called `b`) are addressed via an appropriate class qualification. The member functions are addressed by their complete names and prepended class qualification and AID displays the start address of the function prolog in each case. %RESUME continues execution until the next test point.

```
STOPPED AT SRC_REF: 38 ,  SOURCE: VPTR.C ,  PROC: Y::f()
/%d X::a, X::b, c
SRC_REF: 38 SOURCE: VPTR.C    PROC: Y::f()  ****************************************
Y.X::a          =           1
Y.X::b          =           3
Y.X.c           =           2
/%d X::n'f()',X::n'g()'
Y.X::f()    = 01000628
Y.X::g()    = 01000730
/%d a, b, n'f()', n'g()'
Y.a             =           4
b               =           6
f()         = 01000D48
g()         = 01000E50
/%r
```

In this case, the interrupt point is located in the dynamic member function `Y::f()` of class `Y`. Access to dynamic variable `a` of class `X` must be qualified as there is also an `a` in `Y`. For the same reason, access to static variable `b` of class `X` must also be qualified. `c` is, in contrast, unique and does not require qualification. `a` and `b` from `Y` can also be reached directly. However, you have to specify static data member `b` from `X` with its full name.

The `X::f()` and `X::g()` functions are not visible at the interrupt point as they are hidden locally by the functions of the same name from `Y` and must therefore be specified with the prepended class qualification. You can access functions `f()` and `g()` from `Y` directly.

```
STOPPED AT SRC_REF: 39 ,  SOURCE: VPTR.C,  PROC: Y::g()
/%d ::y.X::a,X::b,::y.c
SRC_REF: 39 SOURCE: VPTR.C    PROC: Y::g()  ****************************************
y.X::a           =           1                                        '
X::b             =           3
y.c              =           2
/%d X::n'f()',X::n'g()'
X::f()           = 01000628
X::g()           = 01000730
/%d ::y.a, b, n'f()',n'g()'
y.a              =           4
b                =           6
f()              = 01000D48
g()              = 01000E50
```

The interrupt point is now in the static member function `Y::g()`. As in the first case, where the program was interrupted in `main`, the dynamic data members of class `Y` can only be accessed via the associated object. However, as the scope of object `y` only starts after the definition of `Y::g()`, access to object name `y` (shown here with the prepended colon-pair as qualification for the superblock) must be fully qualified. AID can, however, reach static data member `b` from `Y` without qualification. Static data member `X::b` is hidden by `Y::b`.

2. The following examples demonstrate accessing variables from base and derived classes with various constructions of three classes `A`, `B` and `C`. The interrupt point is to lie in the `func_C()` function of class `C` in each case.

   First define classes `A`, `B` and `C`:

```
C++ program                              SOURCE: EX1.C
================================================================================
SRC
LIN
...
 42  class A {
 43    int i,j,l;
 44    public:
 45    A(int x=1, int y=2, int z=3) : i(x),j(y),l(z) {...}
 46    void func_A() {...}
 47  };
 48  class B {
 49    int j,k,l;
 50    public:
 51    B(int x=4, int y=5, int z=6) : j(x),k(y),l(z) {...}
 52    void func_B() {...}
 53  };
 54  class C: public A, public B {
 55    int l;
 56    public:
 57    C(int x = 7) : l(x) {...}
 58    void func_C() {...}
 59  };
```

   Debug run:

```
...
STOPPED AT SRC_REF: 58 ,  SOURCE: EX1.C ,  PROC: C::func_C()
/%d i, j, k, l
SRC_REF: 58 SOURCE: EX1.C    PROC: C::func_C()  ************************************
C.A.i                =          1
%  AID0376 Ambiguous qualification for SYMBOL j
C.B.k                =          5
C.l                  =          7
/%d A::j, B::j, A::l, B::l
C.A::j               =          2
C.B::j               =          4
C.A::l               =          3
C.B::l               =          6
```

   As it can be seen from the definition, `A` and `B` are direct base classes of `C`.
   The first %DISPLAY command initially tries to output all variables via their names, without qualification.

The `i` and `k` variables only occur once each and can therefore be accessed directly. With `l` without qualification, you reach `C::l`. The `C:l` variable belongs to class `C` and therefore hides the variables with the same name `A::l` and `B::l`. These are specified, qualified in the second %DISPLAY. AID finds two definitions on the same level for `j`, in base classes `A` and `B`, and reports the ambiguity. Both data members can be identified with an appropriate qualification, also in the second %DISPLAY.

3. In this example, the relationship between classes `A`, `B` and `C` from example 2 has been changed as follows:

```
C++ program                              SOURCE: EX2.C
================================================================================
SRC
LIN
...
 42  class A {
 43    int i,j,l;
 44    public:
 45    A(int x,int y,int z) : i(x), j(y), l(z) {...}
 46    void func_A() {...}
 47  }
 48  class B:  public A {
 49    int j,k,l;
 50    public:
 51    B(int x,int y,int z) : j(x), k(y), l(z), A(1,2,3) {...}
 52    void func_B() {...}
 53  }
 54  class C: public A, public B {
 55    int l;
 56    public:
 57    C(int x=7) : l(x), A(4,5,6), B(8,9,10) {...}
 58    void func_C() {...}
 59  }
```

Debug run:

```
...
STOPPED AT SRC REF: 58 ,  SOURCE: EX2.C ,  PROC: C::func_C()
/%d i, j, k, l
SRC_REF: 58 SOURCE: EX2.C    PROC: C::func_C()  ************************************
%  AID0376 Ambiguous qualification for SYMBOL i
%  AID0376 Ambiguous qualification for SYMBOL j
C.B.k               =           9
C.l                 =           7
/%d A::i, B::A::i, A::j, B::j, B::A::j
C.A::i              =           4
C.B::A::i           =           1
C.A::j              =           5
C.B::j              =           8
C.B::A::j           =           2
```

In this case, `A` and `B` are direct base classes of `C`, and `A` is also an indirect base class of `C`. `A` is not virtual.
Variable `i` is now ambiguous as it occurs in both the direct and the indirect base class `A`. In the second %DISPLAY, the two different `i` variables are output via the associated class qualifications `A::i` and `B::A::i`.
`j` is also ambiguous. There are three definitions on different levels, in `A`, `B` and `B::A`. These are also output with the second %DISPLAY.
`k` is unique, as in example 2 since it is only contained in `B`. AID can also identify `l` uniquely as the `l` from `C` hides the various other definitions in `A`, `B` and `B::A`.

5.3.2 Constructors and destructors

Constructors and destructors are member functions belonging to a class and in debugging with AID are thus referenced in exactly the same way as other functions in classes. If you want to specify a constructor or a destructor in a PROC qualification, you must place the function name together with its class and its signature in `n'...' or t'...'` as described on page 58. You will also find an example on page 61.
The start address of the constructor or destructor can be likewise accessed by specifying the class name in a class qualification before the function name (see example 1 at the end of this section.

In some circumstances, such as when a class contains virtual functions but no constructor is defined explicitly, the compiler generates a constructor which AID displays in the %DISPLAY or %SDUMP output for that class.

When a program is started, constructors for global objects are invoked by a compiler-generated function called `__STI__` . Symbolic debugging in `__STI__` is only possible conditionally. If you specify `__STI__` in an %AID command, this name as well as `main` must be used without a signature.
On terminating `main`, destructors of global classes are called directly by the runtime system. The names of the runtime system functions involved are displayed by AID via `%SDUMP %NEST` in the call hierarchy.

Examples

1. `%in A_global::n'A_global()'`

   A test point is set at the first executable statement of the constructor of class `A_global` from the example on page 61.

2. `%sd %nest`

   AID displays the current call hierarchy, starting with the constructor `A_global::A_global()` from the same example as above. The constructor was called from the compiler-generated function `__STI__`. The routines of the runtime system then follow.

```
SRC_REF: 7  SOURCE: BSP.C  PROC: A_global::A_global()  ******************
SRC_REF: 12  SOURCE: BSP.C  PROC:  __STI__  ****************************
ABSOLUT: V'101CF14'   SOURCE: ICPSINI@  PROC: ICPSINI@  *****************
ABSOLUT: V'101C582'   SOURCE: ICPSINI@  PROC: ICPSINI@  *****************
ABSOLUT: V'100C0FA'   SOURCE: IPPSIN@@  PROC: IPPSINI   *****************
```

5.3.3 Virtual functions

You address virtual functions in an AID command as in a C++ program:
`pointer->n'function([signature])'`

pointer
: Name of a pointer variable which points to a class object containing virtual functions.

function(signature)
: Name of a virtual function from a class. The signature must be omitted if it is `void`. Due to the special characters, you must enclose *function*`([`*signature*`])` within `n'...'`.

If you use the syntax for the virtual functions in a %DISPLAY command, AID outputs the address of the prolog of the member function to which *pointer* currently points. The prolog address of the virtual function is also accessed in a %MOVE or %SET command with the above syntax. However, in a %DISASSEMBLE or %INSERT command, you designate the first executable statement of the virtual function which is currently referenced by *pointer*. When you use an appended pointer operator (->) after the above syntax, you identify the first 4 bytes from the start address of the prolog of the current function.

Example

```
C++ program                          SOURCE: BCL1.C
================================================================================
SRC
LIN
  1  class A
  2  {
  3    public:
  4    A() { printf ("A::A called\n"); }
  5    virtual void foo1() { printf( "A::foo1 called\n" ); }
  6    virtual void foo2() { printf( "A::foo2 called\n" ); }
  7  } a;
  8
  9  class B : public A
 10  {
 11    int i;
 12    public:
 13    B(int x = 1) : i(x) { printf ("B::B called\n"); }
 14    void foo1() { printf( "B::foo1 called\n" ); }
 15    void foo2() { printf( "B::foo2 called\n" ); }
 16  } b;
...
 30  int main()
 31  {
 32    A* aptr = &a;
 33    A* bptr = &b;
 34    bptr->foo2();
...
```

```
...
STOPPED AT SRC_REF: 34, SOURCE: BCL1.C , PROC: main
/%d bptr->n'foo2()'
SRC_REF: 34    SOURCE: BCL1.C      PROC: main  ****************************
A.foo2()    = 010005E0
/%d B
01          B
 02         A
  03        A()             = 01000000
  03        foo1()          = 01000160
  03        foo2()          = 01000270
 02         B(int)          = 01000360
 02         foo1()          = 010004C0
 02         foo2()          = 010005E0
/%da 5 from bptr->n'foo2()'
BCL1$0&@+67A      MVC   20(4,R11),4(R8)        D2 03 B020 8004
BCL1$0&@+680      L     R14,20(R0,R11)         58 E0 B020
BCL1$0&@+684      ST    R14,88(R0,R13)         50 E0 D088
BCL1$0&@+688      LR    R1,R14                 18 1E
BCL1$0&@+68A      L     R15,0(R0,R9)           58 F0 9000
/%in bptr->n'foo2()'; %r
STOPPED AT SRC_REF: 15, SOURCE: BCL1.C , PROC: B::foo2()
/%d i
SRC_REF: 15 SOURCE: BCL1.C  PROC: B::foo2()  ****************************
B.i             =           1
```

The program was halted at the source reference S’35’. The first %DISPLAY shows the complete name of the virtual function to which the pointer variable `bptr` currently points. The second %DISPLAY lists class `B`. A comparison of the addresses shows that `bptr` actually points to `f002()` from `B`. The subsequent %DISASSEMBLE disassembles the first 5 assembler commands of the first executable statement of `B::foo2()`. You now use the command sequence `%INSERT...;%RESUME` to continue the program run until the first executable statement of `B::foo2()` is reached. Data of the associated object can now be addressed as usual.

5.3.4 Pointer to class member

In order to provide dynamic access to data and functions from classes at runtime, C++ offers the data type "pointer to member". A distinction is made here between a pointer to a data member" and a pointer to member functions. In AID V2.3B, you can address data and class functions using pointer to member exactly as in C++. The use of pointer to member for debugging is described in detail in the sections that follow.

5.3.4.1 Pointer to data member

Output

You can view the current contents of a pointer to a data member of a class, referred to as pointer to data member below, by using %DISPLAY or %SDUMP. The name of the data member currently referenced by the pointer to data member is shown in the output. If the data member is defined in a derived or nested class, the full class qualification is prepended to the name of the data item.

```
{%DISPLAY | %SDUMP} [qua•] pointer-to-data-member
```

qua Qualification
The pointer to data member can be addressed like any other pointer by means of an appropriate qualification if it is not visible at the interrupt point.

pointer-to-data-member
is the name of a pointer to a data member of a class.

If a pointer to data member has an invalid value, the error message AID0545 is issued.

Modification

You can use %SET to overwrite a pointer to data member. The sender may be another pointer to data member of addresses to data members of classes. In the following syntax, pointer to data member is abbreviated to *ptdm* due to the limited space available:

```
                 ⎧[object•][class::][...]ptdm⎫
%SET [qua•]      ⎨                           ⎬ INTO [qua•][object•][class::]ptdm
                 ⎩&[class::][...]dataname    ⎭
```

qua Qualification
If *ptdm*, *class* or *dataname* is not visible at the interrupt point, you must specify an appropriate qualification.

object Name of a class object
You use *object* and possibly a following class qualification to specify the address path to *ptdm*.

class:: Class qualification
A class qualification must be specified as in C++ only if *ptdm* or *dataname* is not unique within a class hierarchy or is locally hidden.

dataname
Name of a data member
The various methods available in C/C++ to access a data item are described in the section “Data names” on page 29.

ptdm Name of a pointer to a data member of a class.

The following must apply with respect to the sender and receiver:

– The class associated with the sender must match that of the receiver or must be the base class of the class to which the receiver belongs and must have a unique subobject.

– The data type referenced by the transferred pointer to data member or of the data item specified as the sender must match the type referenced by the receiver. However, if the data item referenced by the sender and receiver is of type pointer or pointer to member, no further check is performed to determine whether the types of data designated by these two pointers match.

If AID determines that the sender and receiver do not satisfy the conditions indicated above, the error message `AID0388: Types are not convertible` is issued.

Dereferencing

Dereferencing enables you to access the content of the data item currently referenced by the pointer to data member. You can display the value of the associated data item with %DISPLAY or %SDUMP and change that value with %SET. The dereferenced pointer to data member can also be used in expressions.

As in C++, two methods of dereferencing are also available to you in AID:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

```
[qua.]object.*[object.][class::][...]pointer-to-data-member
```

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

```
[qua•]pointer->*[object•][class::][...]pointer-to-data-member
```

qua Qualification
If the class object or the pointer to the object is not visible at the interrupt point, you must specify an appropriate qualification. Note that *pointer-to-data-member* must also be visible in the program area designated with *qua*.

object Name of a class object
The *object* before the •* operator designates the class object containing the desired data item.
The *object* after the •* operator or the **->*** operator, which may be followed by a class qualification, specifies the address path to *pointer-to-data-member*.

pointer Name of a pointer to a class object

class:: Class qualification
A class qualification must be specified as in C++ only if *pointer-to-data-member* is not unique within a class hierarchy or is locally hidden.

pointer-to-data-member
Name of a pointer to a data member of a class
pointer-to-data-member must be visible at the interrupt point. In other words, if a qualification was specified before the entire expression, this applies to *pointer-to-data-member* as well.

The following relationship must exist between {*object* | *pointer*} and *pointer-to-data-member*:

{*object* | *pointer*} and *pointer-to-data-member* must either refer to the same class or the class associated with *pointer-to-data-member* must be a unique base class in *object* or in the object referenced with *pointer*.

The following errors may occur when dereferencing a pointer to data member:

– The right operand is not of type pointer to member (AID0546)
– The left operand does not refer to a class object (AID0547)
– The operand refers to incompatible class types (AID0548)
– The class referred to by the pointer to member contains multiple subobjects that are not unique (AID0549)

Example

```
C++ program                              SOURCE: PTOM01.C
================================================================================
SRC
LIN
  1  class Z
  2  {
  3    public:
  4    int a;               // data member
  5    int Z::*ptr_to_dm_Z;
  6    int gz(int n)        // nonvirtual member function
  7      {
  8           a= 4;
  9           return n+a;
 10      };
 11  };
 12
 13  class X :
 14    public Z
 15  {
 16    public :
 17    int a, bx;           // data member
 18    int g1x(int n)       // nonvirtual member function
 19      {
 20          a= 8;
 21          return n+a;
 22       };
 23     };
 24
 25   class X  x;
 26   class X* px;
 27
 28   int X::* pdmX;
 29   int (X::* ptr_to_fun_mem) (int);
 30
 31   main()
 32   {
 33     px               = &x;
 34     pdmX             = &X::a;
 35     x.ptr_to_dm_Z    = &Z::a;
 36     x.*x.ptr_to_dm_Z = 99;
 37     px->*pdmX        = 2;
 38
 39     ptr_to_fun_mem   = &Z::gz;
 40     x.a              = x.gz(5);
 41     x.Z::a           = (x.*ptr_to_fun_mem)(7);
 42     x.bx             = x.g1x(9);
 43     return 0;
 44   }
```

```
...
STOPPED AT SRC_REF: 39, SOURCE: PTOM01.C , PROC: main
/%d pdmX,x.ptr_to_dm_Z
SRC_REF: 39 SOURCE: PTOM01.C      PROC: main  ****************************
pdmX            = X::a
x.ptr_to_dm_Z   = Z::a
/%d px->*pdmX,x.*x.ptr_to_dm_Z
*pdmX           =           2
*x.ptr_to_dm_Z  =          99
/%s x.ptr_to_dm_Z into pdmX
/%d px->*pdmX
*pdmX           =          99
```

The program has stopped in line 39. The first %DISPLAY shows the data members currently referenced by the pointer to member `pdmX` and the pointer to member `ptr_to_dm_Z`. The second %DISPLAY shows the contents of these data members. `Z` is then used to overwrite the pointer to member `pdmX`. The following %DISPLAY shows the contents of the data member now referenced by `pdmX`, i.e. `Z::a`.

5.3.4.2 Pointer to function member

Output

As in the case of a pointer to a data member of a class, you can also have the contents of a pointer to a member function displayed. Such pointers are known as a “pointer to function member”. AID outputs the full name of the function (with the signature) currently referenced by the pointer to function member. If the function is defined in a derived or nested class, the full class qualification is listed before the function name. If the pointer to function member references the a virtual function, the name of the currently assigned function appears in the output.

```
{%DISPLAY | %SDUMP} [qua•]pointer-to-function-member
```

qua Qualification
The pointer to function member can be addressed like any other variable via an appropriate qualification if it is not visible at the interrupt point.

pointer-to-function-member
is the name of a pointer to a member function of a class.

If a pointer to function member has an invalid value, the error message AID0545 is issued.

Modification

You can overwrite a pointer to function member with %SET. Any other pointer to function member or an address of a virtual or non-virtual class function is permissible as the sender. In the following syntax, the pointer to function member is abbreviated to *ptfm* due to space constraints:

```
%SET [qua•]{[object•][class::][...]ptfm  } INTO [qua•][object•][class::]ptfm
           {&[class::][...]function      }
```

qua Qualification
If *ptfm*, *class* or *function* is not visible at the interrupt point, you must specify an appropriate qualification.

object Name of a class object
You use *object* and possibly a following class qualification to specify the address path to *ptfm*.

class::
Class qualification
A class qualification must be specified before *ptfm* only if *ptfm* or *dataname* is not unique within a class hierarchy or is locally hidden.
The class qualification preceding *function* designates the class containing the definition of *function*. As in C/C++, only the levels of the class hierarchy that are needed to uniquely identify *function* must be specified.

function
Name of a member function
Details on how to specify the name of a C++ function when debugging with AID can be found on page 58.

ptfm Name of a pointer to a member function of a class.

The following condition must be satisfied for the sender and receiver:
The class associated with the sender must match that of the receiver or must be the base class of the class of the receiver. Otherwise, AID issues error message `AID0388`.

AID does not check whether the function types referenced by the sender and receiver match. Consequently, program errors could occur during subsequent execution if a function call in the program does not match the function accessed via the modified pointer to function member. It is therefore the responsibility of the user to ensure that the modified function calls execute without errors.

Dereferencing

In order to access a function designated via a pointer to function member in an AID command, the pointer to function member must be dereferenced.

When you specify a dereferenced pointer to function member in a %DISPLAY or %SDUMP command, the start address of the prolog of the associated member function is displayed. If you append a suitable type modification and a pointer operator (%al4->) to the dereferenced pointer to function member, you will receive the first 4 bytes of the function code in hexadecimal representation.
If desired, a test point may be specified with a dereferenced pointer to function member in %INSERT or %REMOVE. The test point is set at the first executable statement of the function.
Furthermore, a dereferenced pointer to function member can also be used to specify the area to be monitored in %CONTROL*n* or %TRACE.

As in the case of the pointer to data member, two methods of dereferencing are also available here:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

```
[qua.]object.*[object.][class::][...]pointer-to-function-member
```

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

```
[qua.]pointer->*[object.][class::][...]pointer-to-function-member
```

qua Qualification
If the class object or the pointer to the object is not visible at the interrupt point, you must specify an appropriate qualification. Note that *pointer-to-function-member* must also be visible in the program area designated with *qua*.

object Name of a class object
The *object* before the `.*` operator designates the class object containing the desired function.
The *object* after the `.*` operator or the **->*** operator, which may be followed by a class qualification, specifies the address path to *pointer-to-function-member*.

pointer Name of a pointer to a class object

class:: Class qualification
A class qualification must be specified as in C++ only if *pointer-to-function-member* is not unique within a class hierarchy or is locally hidden.

pointer-to-function-member
Name of a pointer to a member function of a class
pointer-to-function-member must be visible at the interrupt point. In other words, if a qualification was specified before the entire expression, this applies to *pointer-to-function-member* as well.

The following relationship must exist between {*object* | *pointer*} and *pointer-to-function-member*:

{*object* | *pointer*} and *pointer-to-function-member* must either refer to the same class or the class associated with *pointer-to-function-member* must be a unique base class in *object* or in the object referenced with *pointer*.

As when dereferencing a pointer to data member, errors analogous to those described on page 77 could also occur here.

Example

```
...
STOPPED AT SRC_REF: 40, SOURCE: PTOM01.C , PROC: main
/%d ptr_to_fun_mem,x.*ptr_to_fun_mem
SRC_REF: 40 SOURCE: PTOM01.C      PROC: main  ****************************
ptr_to_fun_mem  = Z::gz(int)
*ptr_to_fun_mem = 01000000
/%s &X::n'g1x(int)' into ptr_to_fun_mem
/%in x.*ptr_to_fun_mem;%r
STOPPED AT SRC_REF: 20, SOURCE: PTOM01.C , PROC: X::g1x(int)
```

The above example refers to the C++ program listed on page 78. The program run is interrupted in line 40. To begin with, the function currently referenced by the pointer to member `ptr_to_fun_mem` is displayed along with the address of the prolog of that function. The %SET command overwrites the pointer to member with the address of the function `g1x(int)`. %INSERT sets a test point in this function by addressing the function via the dereferenced pointer to member. %RESUME then continues the program execution. The function call in line 40 now activates the function `g1x(int)` as was intended in the program code instead of the function `gz(int)`.

5.3.4.3 Comparing pointers to members

Pointers to members can be compared in a subcommand in order to have a command sequence executed in accordance with the result of the comparison. Only the operators `EQ` and `NE` are allowed. You can compare a pointer to member with another pointer to member or with components of a class. The result of the comparison is `TRUE` if both relational operands refer to the same member of a class.

In order to enable a comparison, both operands must satisfy the same conditions which also apply to modification:

Both operands must either refer to same class or the class of one operand must be a unique base class in the class of the other operand.

Example

```
...
STOPPED AT SRC_REF: 33, SOURCE: PTOM01.C , PROC: main
/%in x.n'gz(int)' <(::ptr_to_fun_mem eq &X::n'gz(int)'):                -
/%d 'ptm-call'; %sd %nest; %stop>
/%r
ptm-call
SRC_REF:  8  SOURCE: PTOM01.C  PROC: Z::gz(int)  **************************
SRC_REF: 40  SOURCE: PTOM01.C  PROC: main  *********************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ********************
ABSOLUT: V'10013D0'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************
STOPPED AT SRC_REF: 8, SOURCE: PTOM01.C , PROC: Z::gz(int)
```

The program listed on page 78 is now interrupted at the start of `main`. %INSERT sets a test point at the first executable statement of the function `Z::gz(int)`. A check is then performed in the subcommand for this %INSERT to see if the function was called via the pointer to member `ptr_to_fun_mem`, and if this is true, the text `'ptm-call'` and the current call hierarchy for the start instruction of `Z::gz(int)` are displayed.

5.3.4.4 Setting a pointer to member to zero

You can use the AID command

`%SET 0 INTO` *pointer-to-member*

to set the contents of the pointer to member to binary zero. This option is essentially available for all pointers.

Example

```
/%AID CHECK=ALL
/%SET 0 INTO ptr_to_fun_mem
OLD CONTENT:
Z::gz(int)
NEW CONTENT:
00000000 00000000
%  AID0274 Change desired? Reply (Y=Yes; N=No)?n
%  AID0342 Nothing changed
```

`%AID CHECK=ALL` initiates the update dialog. Following the %SET command, AID displays the old content of the pointer to member, the reference to the function `gz(int)` of class `Z`, and the new content that would exist on executing the %SET, i.e. binary zero. AID then issues a prompt to ask whether the change is to be performed.

5.4 Namespaces

Namespaces are a new language feature in C++. They are comparable to classes and are used to prevent name collisions in the global namespace.

Accessing whole namespaces is analogous to accessing complete classes. To access namespace members, you use the namespace qualification in the same way as you use class qualification to access a class member. However, from the function of a namespace, you can only reach the data members which are defined before the function.
As with classes, namespaces can be nested, whereby the outermost namespace can only be defined globally.

The data defined in a namespace is always static.

If you specify a namespace in a %DISPLAY or %SDUMP command, the complete namespace is output together with all variables and functions defined in it. The output is listed in the same way as a structure.
The current values of variables are displayed and the functions are listed with their complete function name, signature and the start address of the prolog. Further namespaces nested in the namespace are also output completely.
You can assign a namespace an alias name with the %ALIAS command and then access it under this alias name in subsequent commands. However, the namespace is always shown under its original name in the output of a %SDUMP without operands. However, if the namespace is assigned an alias name in the C++ program, AID lists the namespace under both the original name and the alias name.
In all other AID commands which require an address operand, you can only specify a namespace as a qualification for a subsequent data or function name.

```
                                                          ⎧dataname ⎫
[E-qua•][S-qua•][::]namespace[::...][class[::...]][{     ⎨         }]
                                                          ⎩function ⎭
```

E-qua Base qualification
Specified with `E=VM` or `E=D`*n* and defines whether the AID work area is to lie in the loaded program (`E=VM`) or in a dump file (`E=D`*n*). The base qualification is used in the same way with both symbolic and machine code debugging and is described in the AID Core Manual in the section on “Addressing in AID” and in the %BASE command on page 127.

S-qua S qualification
Designates the translation unit containing the namespace. You specify the S qualification as described on page 23.

namespace
: Name of a namespace
To access an inner namespace within a nested namespace, you have to specify the names of the namespaces from the outmost to the innnermost in the path to the required namespace, each terminated with a colon-pair.

Namespaces can only be defined globally. It is therefore only possible to have a base, S or :: qualification before *namespace* for the global namespace. You can also specify the namespace qualifications of one or more superordinate namespaces to describe the path to a nested namespace in a lower level.

class
: Name of a class
If *class* designates the instance of a class template, you have to use the notation `t'`*k_template*`<`*arg*`[,...]>'`. If there is only one instance of the class template, you only have to specify `t'`*k_template*`'`.
Between the class name and subsequent function name, you insert a pair of colons as usual (see also page 63).

function
: Name of a function
Detailed information on how you specify a function name when debugging C++ programs can be found on page 58.

dataname
: Name of a data item
The various ways of accessing a data item in C/C++ are described in the section “Data names” on page 29.

5.4.1 Unnamed namespaces

You cannot access unnamed namespaces as a whole with %DISPLAY or %SDUMP. An unnamed namespace is only listed fully in the output of a %SDUMP without operands. The first line of the output, which normally contains the name of the namespace after the level number 1, only contains the level number.

Variables and functions from unnamed namespaces cannot be qualified. They are therefore only visible to AID where they can also be referenced in C++ without qualification, i.e. where the entire namespace is visible. They behave in the same way as global static variables or functions.

Example

The namespace is defined as follows:

```
namespace {int i = 1; int j = 10;}
```

The output of the unnamed namespace with the %SDUMP command is listed as follows:

```
01
 02        i               =           1
 02        j               =          10
 02        using           = // UNNAMED //
```

5.4.2 Scope rules in namespaces

AID emulates the scope rules which apply in C++ for working with namespaces. You can access variables that are not visible at the interrupt point as in C++ at any time via the associated namespace qualification *namespace*`::`.

using declaration and using directive

If the program is interrupted after a `using` declaration for a single variable from a namespace or after a `using` directive for a complete namespace, you can access the declared variable or all variables and functions from the namespace without qualification. The variable included via a `using` declaration behaves like a locally defined variable. An existing variable with the same name in the current scope area is hidden by the definition from the namespace and can only be accessed fully qualified.
In contrast to this, a namespace variable that is made known via a `using` directive has the scope of the superordinate namespace. In this case, a local variable with the same name hides the definition from the namespace.
If, while working with namespaces, duplicated names occur with functions, they are overloaded.

If ambiguities occur through using variables from namespaces, e.g. because global definitions with the same name exist or nested namespaces contain definitions with the same name on different levels, you cannot access these definitions directly. AID outputs the following message if for ambiguities:
`AID0529 Symbol` *name* `is ambiguous due to a using statement.`
You can discriminate between ambiguous definitions with AID using an appropriate qualification.

Classes in namespaces

AID still supports the C++ scope rules if namespaces contain classes.

In the following section, it is assumed that the program is interrupted in a dynamic function of a derived class which is defined in a namespace. If you access a data item directly from this interrupt point, i.e. without qualification, the following search order applies:

1. Local scope of the dynamic function in which the program is interrupted, before the interrupt point
2. Scope of the derived class containing the function
3. Scope of the base classes
4. Scope of the namespace containing the class definition
5. Scope of superordinate namespaces
6. Global scope before the interrupt point

AID searches through all relevant scope areas sequentially. If there are several hits, AID checks whether the first hides the subsequent ones, or if there is ambiguity. AID outputs an error message (`AID0376` or `AID0529`) in the second case.

Examples

1. *Namespaces and using*

   The example refers to the following program extract:

```
C++ program                              SOURCE: EXNSP1.C
================================================================================
SRC
LIN
  1 namespace PART1
  2 {
  3    int func_1(int) {...}
  4    int j = 88;
  5    int k = 99;
  6 }
  7 namespace SNI
  8 {
  9    void func_1() {...}
 10    using namespace PART1;
 11    int i = 19; int j = 20;
 12 }
 13 int k = 0;
 14
 15 int main()
 16 {
 17    k++;
 18    using SNI::i;
 19    i = k+10;
 20    using namespace PART1;
 21    j = 5; i = j + 10;
 22    ...
```

Debug run:

```
STOPPED AT SRC_REF: 19, SOURCE: EXNSP1.C , PROC: main
/%d i, k, SNI
SRC_REF: 19 SOURCE: EXNSP1.C  PROC: main  ***************************************
i               =         19
k               =          1
01          SNI
 02         func_1()        = 01000088
 02         i               =          19
 02         j               =          20
 02         using           = PART1
```

Variables i and k can be uniquely accessed at interrupt point S‘19‘. Because of the using declaration in line 18, i addresses the variable from namespace SNI; k addresses the global variable k from line 13. The output of namespace SNI includes all the components it contains and a note that namespace PART1 was registered via a using directive in namespace SNI.

```
/%in S'21';%r
STOPPED AT SRC_REF: 21, SOURCE: EXNSP1.C , PROC: main
/%d i, j, k, n'func_1(int)', PART1
SRC_REF: 21 SOURCE: EXNSP1.C  PROC: main  ***************************************
i               =         11
j               =         88
%  AID0529 Symbol k ambiguous because of using directive.
func_1(int)     = 01000000
01          PART1
 02         func_1(int)     = 01000000
 02         j               =          88
 02         k               =          99
/ךך ך┴ךךךךך⌊← ךך⌊
PART1::k        =         99
k               =          1
```

The command sequence %INSERT S‘21‘;%RESUME executes the program up to source reference S‘21‘. You can access i and j uniquely at this point. i addresses the variable from namespace SNI as above; j accesses the variable PART1::j. k is now ambiguous since, in addition to the global variable k there is also another k from namespace PART1. However you can access the two k variables with qualification.

2. *Namespaces and classes*

   Namespaces and classes are nested in each other in this example:

```
C++ program                              SOURCE: EXNSP2.C
================================================================================
SRC
LIN
  1 class A
  2 {
  3    int i,j;
  4    public:
  5    A(int x = 1) : i(x) {j=2; ...}
  6    void func_A() {i=j*i;...}
  7 }a;
  8
  9 class B
 10 {
 11    int j,l;
 12    public:
 13    B(int x = 4) : j(x) {l=6; ...}
 14    void func_B() {...}
 15 };
 16
 17 namespace M
 18 {
 19    int k=1,l=2,m=3;
 20    void func_M() {...}
 21    namespace N
 22    {
 23       int i=4,j=5,k=6;
 24       void func_N() {...}
 25       class X: public A, public B
 26       {
 27          int l;
 28          public:
 29          X(int x = 7) : l(x) {...}
 30          void func_X() {l++;...}
 31       }x;
 32    }
 33 }
 34 int main()
 35 {
 36    using namespace M;
 37    using namespace N;
 38    x.func_X();
...
```

Debug run:

```
/%t 1 in proc=main
38                        EXT.PROC START    , BLOCK START, ASSIGN
STOPPED AT SRC_REF: 38, SOURCE: EXNSP2.C , PROC: main , END OF TRACE
/%d M
SRC_REF: 38    SOURCE: EXNSP2.C      PROC: main  **********************************
 01          M
  02         N
   03        X
    04       A
     05      A(int)          = 01000000
     05      func_A()        = 01000168
    04       B
     05      B(int)          = 010001D0
     05      func_B()        = 01000338
    04       X(int)          = 01000468
    04       func_X()        = 01000600
   03        func_N()        = 01000410
   03        i               =           4
   03        j               =           5
   03        k               =           6
   03        x
    04       A
     05      i               =           1
     05      j               =           2
     05      A(int)          = 01000000
     05      func_A()        = 01000168
    04       B
     05      j               =           4
     05      l               =           6
     05      B(int)          = 010001D0
     05      func_B()        = 01000338
    04       l               =           7
    04       X(int)          = 01000468
    04       func_X()        = 01000600
  02         func_M()        = 010003B8
  02         k               =           1
  02         l               =           2
  02         m               =           3
```

The %TRACE command initially positions the program on the first executable statement in `main`. `%DISPLAY M` then lists the entire namespace `M` together with namespace `N` it contains and class `X`.

```
/%in n'func_X()'; %r
STOPPED AT SRC_REF: 30, SOURCE: EXNSP2.C , PROC: M::N::X::func_X()
/%d i, j, k,l , m
SRC_REF: 30    SOURCE: EXNSP2.C      PROC: M::N::X::func_X()  **********************
X.A.i             =           1
%  AID0376 Ambiguous qualification for SYMBOL j
k                 =           6
X.l               =           7
m                 =           3
/%d M::N::i
M::N::i           =           4
/%d A::j, B::j, M::k
X.A::j         =       2
X.B::j         =       4
M::k           =       1
```

In the next step, the program is executed up to the start of function `func_X()`, which is defined in class `X`. Variables `i`, `j`, `k`, `l` and `m` are accessed directly from this point, i.e. without explicit qualification:

- With `i` you reach variable `i` from base class `A`; the `i` from namespace `N` is hidden by this and can only be referenced with full qualification, i.e. with `M::N::i`, see next %DISPLAY.
- `j` is ambiguous, since it occurs in both base classes. The two j variables are output in the third %DISPLAY with an appropriate namespace qualification.
- `k` is unique since the `k` from namespace `N` hides the `k` from namespace `M`. Die variable from `M` is output fully qualified in the third %DISPLAY, i.e. with `M::k`.
- `l` is defined in class `X` and therefore hides all other variables with this name.
- `m` is unique since it only occurs in namespace `M`.

5.4.3 Alias names for namespaces

In C++ you can assign additional, alias names to namespaces to avoid having to type in the original name, which may be very long, each time. You can also assign several alias names to one namespace. When debugging with AID, you can access the namespace with either its original name or with any alias names assigned to it as long as the interrupt point lies within the scope area of the alias name used.

If you output a list of all the program definitions using %SDUMP without any operands, the namespace is listed fully under its original name and under all alias names.

AID provides such a mechanism with the %ALIAS command (see page 124).

Example

Alias assignment in the C++ program:

```
namespace FJ = Fujitsu;
```

You use the following %SET command to assign variable `i`, which is defined in namespace `Fujitsu`, a value using the alias name `FJ`. The subsequent %DISPLAY outputs the contents of `FJ::i`.

```
/%set 20 into FJ::i
/%display FJ::i
Fujitsu::i                =       20
```

5.5 Templates

Templates are new in C++. There are two different types, class templates and function templates.

A template is simply a pattern with which actual classes or functions, so-called template instances, are created after template arguments have been specified. This produces the following problems for debugging template instances with AID:

– Several different instances can exist for one template declaration.

– The source references of these multiple instances are derived from the template declaration and are therefore not unique.

– The template arguments are included in the formation of the template instance names. With AID, these can be literals, address constants or type parameters. The type parameter names must differ from all other definitions in the program.

5.5.1 Template instantiation

An instance of a template is always created in a program when the template is assigned concrete arguments. An instance of a **function template** is created if the function name is called from the template declaration with specific arguments. In the same way, an actual class is created from a **class template** when the place-holders in the template declaration are replaced with real values or type definitions.

If you want to produce a list of the template instances created in the program, enter the template name in a %DISPLAY command: `%DISPLAY` *template*. If only a single instance was created for a template, it can be displayed with `%DISPLAY t'`*template*`'` (see also the section “Listing template instances” on page 106). However, as in the C++ language, to access a specific instance with AID, you must add the relevant template arguments after the name defined in the template declaration, enclosed within in angle brackets as in C++. To inform AID that it is a template instance, you have to specify the instance name in `t‘...‘`.
AID checks the syntax of the template instance specification during input and rejects illegal entries with a syntax error message. AID also checks whether a template instance with the specified name has been created in the program and outputs an appropriate error message if it cannot find the template instance.

Syntax for class template instance names:

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

t'k_template<arg[,...]>'

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

Syntax for function template instance names:

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

t'f_template<arg[,...]>([signature])'

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

If there is only a single instance for a template declaration, you can access it using the short form `t'`*k_template*`'` or `t'`*f_template*`([`*signature*`])'` **without** specifying the template arguments. If several instances exist, the short form references just one of these instances and it is not possible to predict which instance will be selected.

template
: You specify the name of the template declaration with *template*.

arg
: The following entries are possible for *arg* with AID:

  – Literal

    A literal can be a number, e.g. 15, -15, 1.4, a character or a character string. Single characters or character strings must be enclosed in double quotes (`''`), e.g. `''a''`, `''ABC''`. As in C++, you may specify the numeric equivalent for a single character. For example, the same template instance can be addressed with `t'CC<''a''>'` and `t'CC<129>'`. The numeric values of characters are based on the EBCDIC table of a /390 system. An example of this concept can be found on page 100.
    The following workaround can be used to determine the numeric value of a character with AID:

`%DISPLAY '`*character*`'%X`	Displays the corresponding hexadecimal value
`%DISPLAY #'`*hexadecimal-no*`'%F`	Displays the corresponding decimal value

  – Address constant

    Address constants such as addresses of functions (`foo`), variables (`&var`), etc., are specified as in the C++ program.

– Elementary data type

The elementary C++ data types have specific template arguments assigned to them. The data type names on the left in the following table and the assigned template arguments on the right are equivalent, i.e. with AID, you can access a template instance via either the data type name or the assigned template argument:

Data type	Template argument
`char`	`char`
`unsigned char`	`unsigned char`
`signed char`	`signed char`
`bool`	`bool`
`unsigned`	`unsigned int`
`unsigned int`	`unsigned int`
`signed`	`int`
`signed int`	`int`
`int`	`int`
`unsigned short int`	`unsigned short int`
`unsigned short`	`unsigned short int`
`unsigned long int`	`unsigned long int`
`unsigned long`	`unsigned long int`
`signed long int`	`long int`
`signed long`	`long int`
`long int`	`long int`
`long`	`long int`
`signed short int`	`short int`
`signed short`	`short int`
`short int`	`short int`
`short`	`short int`
`wchar_t`	`wchar_t`
`float`	`float`
`double`	`double`
`long double`	`long double`
`void`	`void`

Table 2: Elementary data types and their assigned template arguments

– Derived data type

  Derived data types are formed from elementary types by adding the `*`, `&` or `[]` operator.

  Examples of derived type parameters are summarized in the following table:

Definition in C++ program	Type parameter
`int *pi`	`int*`
`int *p[3]`	`int* [3]`
`int (*pi) [3]`	`int (*)[3]`
`int *f()`	`int*()`
`int (*pf) (double)`	`int (*)(double)`

Table 3: Examples of derived type parameters

– User-defined data type

  You can use data types that you have defined yourself in the program with AID in the same way as in C++.
  Example
  `typedef int (*int_arr_def)[3]` defines the data type `int(*)[3]` and assigns it the name `int_arr_def`. You can use either the data type or the assigned name with AID as a template argument to reference the same template instance.

– It is illegal to specify an expression instead of a literal with AID. This would cause a syntax error. For example, in C++ you could specify `foo<40>` or `foo<20*2>` to access the same instance of function template `foo<40>`, or `foo<(1>2)>` would mean the same as `foo<(false)>`. AID rejects `foo<20*2>` or `foo<(1>2)>` with a syntax error.

Example

```
C++ program                                              SOURCE: EXTMP3.C
================================================================================
SRC
LIN
  1 template <class T, T* address> class T1
  2 {
  3    public:
  4    void foo() { (*address)++; }
  5    void bar() { (*address)(); }
  6 };
  7
  8 template <class S, class T, S* address, T (S::*ptm)()> class T2
  9 {
 10 public:
 11    void foo() { (address->*ptm)(); }
 12 };
 13
 14 template <class S, class T, S* address, T (S::*ptm)()> class T3
 15 {
 16 public:
 17    void bar() { (address->*ptm)(); }
 18 };
 19
 20 int i = 0;
 21 void f() { i--; }
 22 struct S
 23 {
 24    int i;
 25    void f() { i--; }
 26 } s;
 27 class X
 28 {
 29 public:
 30    void operator++() {};
 31    void operator++(int) {};
 32    void operator() () {};
 33 } x;
 34
 35 T1<X,&x> t1_x;
 37 T2<S, int, &s, &S::i>  t2_i;
 38 T3<S, void, &s, &S::f> t3_f;
 39 ...
```

The three %DISPLAY commands shown below list the contents of the template instances and the prolog address is output for each template function. The instances are accessed here with their full names. You could also address the instances with their short names in this case as only one instance has been created for each template in this example: `%DISPLAY t'T1', t'T2', t'T3'` would produce the same result.

```
/%d t'T1<X,&x>'
01          T1<X,&x>
 02         foo()           = 01000468
 02         bar()           = 01000578
/%d t'T2<S,int,&s,&S::i>'
01          T2<S,int,&s,&S::i>
 02         foo()           = 01000678
/%d t'T3<S,void,&s,&S::f>
01          T3<S,void,&s,&S::f>
 02         bar()           = 010006F8
```

5.5.2 Class templates

You can debug instances or objects of class template instances with AID as well as other classes or class objects. The assigned scope areas do not differ from those of other classes. Specifying the template arguments identifies the class instance uniquely and the rules for accessing data and functions described in the section “Classes” on page 63 also apply to members from class template instances. In the syntax on page 64, you must specify the class names as described above in the section “Template instantiation”.

Example

```
C++ program                                              SOURCE: EXTMP2.C
================================================================================
SRC
LIN
  1  #include <iostream.h>
  2  template <class T, int I>
  3  class XX  {
  4  public:
  5     void foo(T& t)  {
  6     --t;
  7     cout << "In XX:foo<>(t=" << t <<", I=" << I << ")\n";
  8     my_t[I-1]=t;
  9     }
 10     T my_t[I];
 11     static char *cptr;
 12  };
 13  template<> char *XX<int,10>::cptr = "XX<int,10>::cptr";
 14  template<> char *XX<int,12>::cptr = "XX<int,12>::cptr";
 15
 16  template <class W> class X {
 17  W x[5];
 18  public:
 19     int foo(int j)  {
 20     cout << "X<>:foo(" << j << ")\n";
 21     return x[j];
 22     }
 23  };
 24
 25  template <typename T> class Y {
 26  T t;
 27  public:
 28     int foo(int i)  {
 29     cout << "Y<>i::foo(" << i << ")\n";
 30     return t.foo(i);
 31     }
 32  };
 33
 34  template <char C>
 35  class CC {
 36  public:
 37     void foo() {
 38     cerr << "CC<" << C << ">::foo()\n";
 39     }
 40  };
 41
```

```
42  int main ()
43  {
44  int i = 0;
45  CC<'c'> c1;
46  CC<195> c2;
47  CC<1>   c3;
48  XX<int,10> xi10;
49  XX<int,12> xi12;
50  Y<X<int> > y;
51
52  c1.foo();
53  c2.foo();
54  c3.foo();
55  xi10.foo(i);
56  xi12.foo(i);
57  y.foo(2);
58  return(0);
59  }
```

Debug run:

```
/%t 1 in s=n'extmp2.c'
44                      EXT.PROC START    , BLOCK START, ASSIGN
STOPPED AT SRC_REF: 44, SOURCE: EXTMP2.C , PROC: main , END OF TRACE
/%d CC
01          CC
 02         CC<1>
  03        foo()           = 01005B00
 02         CC<'C'>
  03        foo()           = 010059C0
 02         CC<'c'>
  03        foo()           = 01005880
/%t 1 in t'CC<''c''>'::n'foo()'
49                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 49, SOURCE: EXTMP2.C , PROC: CC<'c'>::foo() , END OF
TRACE
/%t 1 in t'CC<195>'::n'foo()'
49                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 49, SOURCE: EXTMP2.C , PROC: CC<'C'>::foo() , END OF
TRACE
/%t 1 in t'CC<1>'::n'foo()'
49                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 49, SOURCE: EXTMP2.C , PROC: CC<1>::foo() , END OF
TRACE
```

%TRACE initially positions on the first executable statement in `main`. At this point, to access data and statements, you only need the qualification of the scope concerned and do not have to input the complete qualification starting with the S qualification each time.

The subsequent %DISPLAY command returns a list of all instances created for the class template `CC`. Note that AID internally converts 195 to the character `'C'` for the instance CC<195> and displays the instance defined with CC<195> in the program as CC<'C'>. You can nonetheless continue to address this instance with CC<195> in the subsequent test run. In the reverse case, you could also address the instance CC<'c'> with CC<131>, since 131 is the numeric equivalent for 'c'.

The next three %TRACE commands trace the first statement of each instance: this is always the statement in line 49. The reason for this is that the different instances all originate from the same template and the source references are therefore the same in each instance.

```
/%d XX
01          XX
 02         XX<int,12>
  03        foo(int &)      = 01005288
  03        cptr            = 0100624B
 02         XX<int,10>
  03        foo(int &)      = 010050E0
  03        cptr            = 0100623A
/%in t'XX<int,12>'::n'foo(int &)';%r
STOPPED AT SRC_REF: 10, SOURCE: EXTMP2.C , PROC: XX<int,12>::foo(int &)
/%sd proc=t'XX<int,12>'::n'foo(int &)'
SRC_REF: 10  SOURCE: EXTMP2.C  PROC: XX<int,12>::foo(int &)  **************
this            = 010709B0

01          XX<int,12>
 02         my_t( 0: 11)
            (  0)    16813944  (  1)           0  (  2)           0
            (  3)           0  (  4)           0  (  5)           0
            (  6)           0  (  7)           0  (  8)           0
            (  9)           0  ( 10)    17238352  ( 11)    17882652
 02         foo(int &)      = 01005288
 02         cptr            = 0100624B

t               =    17137988
```

The next step lists all instances created for template `XX`. A test point is set on function `foo(int&)` of instance `XX<int,12>` and the program is then started with %RESUME. The program is interrupted in `foo(int&)`. The subsequent %SDUMP lists all data of function `XX<int,12>::foo(int &)`.

```
/%in t'Y<X<int>>'::n'foo(int)' <%D i;%sd %nest;%d ' '>
/%in t'X<int>'::n'foo(int)' <%D j;%sd %nest;%d ' '>;%r
SRC_REF: 41 SOURCE: EXTMP2.C  PROC: Y<X<int>>::foo(int)  ******************
i               =           2
SRC_REF: 41  SOURCE: EXTMP2.C  PROC: Y<X<int>>::foo(int)  ******************
SRC_REF: 75  SOURCE: EXTMP2.C  PROC: main  ********************************
ABSOLUT: V'113C3FE'   SOURCE: IC@RT20A  PROC: IC@RT20A  ********************
ABSOLUT: V'100AF40'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************
SRC_REF: 33 SOURCE: EXTMP2.C  PROC: X<int>::foo(int)  **********************
j               =           2
SRC_REF: 33  SOURCE: EXTMP2.C  PROC: X<int>::foo(int) **********************
SRC_REF: 42  SOURCE: EXTMP2.C  PROC: Y<X<int>>::foo(int)  ******************
SRC_REF: 75  SOURCE: EXTMP2.C  PROC: main  ********************************
ABSOLUT: V'113C3FE'   SOURCE: IC@RT20A  PROC: IC@RT20A  ********************
ABSOLUT: V'100AF40'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************
```

Finally, a test point is set is in each of the `foo(int)` functions of instances `Y<X<int>>` and `X<int>` and the function parameters `i` and `j` are output together with the current call hierarchy when these test points are reached.

5.5.3 Function templates

An actual function is created from a function template via Instantiation. When you use this function in a PROC qualification to specify the definition point of a data item it contains or use the function name in an AID command to define the function start address, you access the function via the function name with the template arguments, which must be enclosed in angled brackets, and the signature. You enclose these entries in `t'...'`, as described in the section “Template instantiation” on page 94.

Example

```
C++ program                                          SOURCE: EXTEMPL1.C
================================================================================
SRC
LIN
  1 // minimum of two objects
  2 template< class T >
  3 const T& minimum( const T& a, const T& b )
  4 {
  5    const T& retval( (a<b) ? a : b );
  6    return retval;
  7 }
  8
  9 // minimum of three objects
 10 template< class T >
 11 const T& minimum( const T& a, const T& b, const T& c )
 12 {
 13    const T& retval( (a<b) ? minimum(a,c) : minimum(b,c) );
 14    return retval;
 15 }
 16
 17 int main()
 18 {
 19    int min;
 20    double xmin;
 21    min  = minimum(2,3,1);
 22    xmin = minimum(2.2,1.1,3.3);
 23 ...
```

Debug run:

```
/%in t'minimum<double>(const double &, const double &)'
/%r
STOPPED AT SRC_REF: 5, SOURCE: EXTEMPL1.C ,
 PROC: minimum<double>(const double &, const double &)
/%sd %nest
SRC_REF: 5  SOURCE: EXTEMPL1.C
            PROC: minimum<double>(const double &, const double &)  ********
SRC_REF: 13 SOURCE: EXTEMPL1.C
            PROC: minimum<double>(const double &, const double &,
                  const double &)
SRC_REF: 22  SOURCE: EXTEMPL1.C  PROC: main  ******************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ********************
ABSOLUT: V'10014A8'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************
/%da 3 from t'minimum<double>(const double &, const double &)'
          EXTEMPL1$0&@
EXTEMPL*+5C2      L     R14,0(R0,R9)           58 E0 9000
EXTEMPL*+5C6      LD    R0,0(R0,R14)           68 00 E000
EXTEMPL*+5CA      L     R15,4(R0,R9)           58 F0 9004
```

The %INSERT command sets a test point at the first executable statement of instance `minimum<double>(const double &, const double &)` of the `minimum` function template. The test point is reached with %RESUME. The call hierarchy at the interrupt point, which you can output with `%SD %NEST`, shows that the current instance of `minimum` was called from a further instance of the same template, `minimum<double>(const double &, const double &, const double &)`. Finally, the %DISASSEMBLE command disassembles the first three commands of instance `minimum<double>(const double &, const double &)`.

5.5.4 Listing template instances

You use the following %DISPLAY command to output an overview of the instances which were created for a class or function template and are visible at the interrupt point:

```
%DISPLAY  [qua]  template
```

qua Qualification
qua specifies the program section containing the template declaration.

template
Name of the class or function template
The template arguments must be omitted for AID to output the overview. You therefore do not need to enclose the template name in `t'...'`.

Note, however, that if only a single template instance exists, the following %DISPLAY command must be used in order to list the instance:

```
%DISPLAY  [qua] {t'k_template'            }
                {                         }
                {t'f_template([signature])'}
```

qua Qualification
qua specifies the program section containing the template declaration (as above).

k_template
Name of the class template
The template arguments can also be omitted here, but the template name must be enclosed in `t'...'`.

f_template([signature])
Name of the function template
The template arguments are omitted, and the template name is enclosed in `t'...'`. If the signature is not `void`, it must be specified. If the signature is `void`, only the two parentheses () must be entered.

signature
Parameters to be passed by the function
If the template involved is a function template, you must specify the signature; however, if the signature is `void`, it is omitted.

Example

The example refers to the C++ program listed on page 104.

```
/%d minimum
01          minimum
 02         minimum<double>(const double &, const double &) = 01000528
 02         minimum<int>(const int &, const int &) = 010004A0
 02         minimum<double>(const double &, const double &,
            const double &) = 01000330
 02         minimum<int>(const int &, const int &, const int &) = 010001D8
```

The %DISPLAY command lists all instances created for function template `minimum` and also outputs the associated prolog addresses.

5.5.5 Displaying template instance names

AID displays the full name of the template instance in its outputs, e.g. in the STOP message or the output of `%SDUMP %NEST`. AID uses the C++ designations for the relevant data types in the template arguments.

If a user definition exists for a derived data type, AID uses it.

Examples

1. The instance of a function template has been created via the function call `foo(15)`. AID displays `foo<int>` as the function name in a STOP message. A second instance has been created because of the `foo(&i)` call. The AID function name for this is `foo<*int>`.

2. ```
   typedef int (*int_arr_def)[3]  int_arr_def;
   int_arr_def int_arr_ptr;
   ...
   foo(int_arr_ptr);
   ```

   In this case, AID uses the function name `foo<int_arr_def>` and not `foo<int(*)[3]>`.
   ```

5.5.6 Accessing source references from template instances

Each instantiation creates an actual function from a function template or a class from a class template. If a function is defined in the class template, instantiation not only creates a new class each time, it also creates a new function. The source references to the statements of such functions are formed from the template declaration line numbers. They are therefore the same for each instance and no longer unique in the translation unit if more than one instance is created. This means that you must qualify source references from template declarations from which multiple instances have been created, with the function of the required instance to enable AID to assign them correctly. Otherwise, AID outputs an appropriate message.

Syntax for source references from template instances:

```
[E-qua•][S-qua•]PROC=[namespace::[...]][class::[...]]function•S'[f-]n[:a]'
```

E-qua Base qualification
Specified with `E=VM` or `E=D`*n* and defines whether the AID work area is to lie in the loaded program (`E=VM`) or in a dump file (`E=D`*n*). The base qualification is used for both symbolic and machine code debugging and is described in the AID Core Manual in chapter “Addressing in AID” and in the %BASE command on page 127.

S-qua S qualification
Designates the translation unit containing the template declaration. You specify the S qualification as described on page 23.

namespace
Name of the namespace containing the template declaration.

class Name of the class
If *class* designates the instance of a class template, you have to use the notation `t'`*k_template*`<`*arg*`[,...]>'`. If there is only one instance for the class template, `t'`*k_template*`'` suffices.
You insert two colons between the class name and subsequent function name as usual.

function
Name of the function
You specify *function* with `n'`*function*`([`*signature*`])'` if the source reference is in a normal C++ function which is defined in a class template, or with `t'`*f_template*`<`*arg*`[,...]>([`*signature*`])'` if it is an instance of a function template. You can abbreviate the latter to `t'`*f_template*`([`*signatur*`])'` if there is only one instance for the function template.

You insert a period between the function name and the source reference. You will find detailed information on how you specify a function name when debugging C++ programs on page 58.

S'[f-]n[:a]'
: is a source reference and designates an executable statement in a function which is either defined in a class template or contained in the declaration of a function template.

If you use an unqualified source reference from a template instance in an AID command, AID rejects it with the following error message:
`AID0376 Ambiguous qualification for SRC_REF`

Example

The example refers to the C++ program shown on page 104.

```
/%in proc=t'minimum<int>(const int &, const int &)'.s'6'
/%in proc=t'minimum<double>(const double &, const double &)'.s'6'
```

A test point is set on source reference S'6' in each of the two instances `minimum<int>(const int &, const int &)` and `minimum<double>(const double &, const double &)`.

5.6 Overloaded functions

Overloaded functions can be uniquely addressed via their signature, since the signature is included in the function designation (see the section “Qualifications” on page 57). You can obtain an overview of the overloaded functions defined in the program by using %DISPLAY to request an overview of all overloaded functions that have the same name at the interrupt point or in the specified program section:

```
%DISPLAY  [qua•]  function
```

qua Qualification of the program section in which the overloaded function is to be searched for.

function
Name of an overloaded function. If you want AID to output an overview, you must omit the signature. Alternatively, you can specify the function name as described in the section “Qualifications” on page 57, i.e. with a prepended namespace and/or class qualification if required.

Structure of the table of overloaded functions:

```
 01 function
 02 function(signature1)   = address1
 02 function(signature2)   = address2
 ...
```

The name of the desired overloaded function can be obtained from the table in standard C++ notation. For each overloaded function present in the scope area, the signature and the associated prolog address are listed.

Example

The example refers to the code fragment of the example on page 61. For the sake of readability, user commands are printed in bold.

```
/%d FOO
SRC_REF: 70 SOURCE: BSP.C   PROC: main  ***********************************
01          FOO
 02         FOO(long)       = 01000070
 02         FOO(int)        = 01000000
/%in n'FOO(int)'
/%in n'FOO(long)'
```

The %DISPLAY command lists the two prolog addresses of the overloaded functions with the name `FOO`. The %INSERT commands which follow then set a test point at the first executable statement of `FOO(int)` and `FOO(long)`, respectively.

5.7 Overloaded operators

You can use AID to debug overloaded operators in exactly the same way as any other function you have written. The desired overloaded operator can be identified to AID by using the function name from your C++ program in standard C++ notation, i.e. with a signature and, if relevant, a prepended class qualification. The complete designation must then be enclosed within `n'...'` as usual. No additional blanks may be entered within `n'...'`.

If an operator is multiply-overloaded, you should proceed as described in the section "Overloaded functions" on page 110 to ensure a unique assignment to the required function.

When debugging a program featuring overloaded operators, note that AID always works on the basis of the standard operators and thus does not allow for overloading in its own calculations.

Example

```
%insert X1::n'operator+(float&)' <%d *this>
```

A function defined in class `X1` overloads the + operator. You therefore set a test point with %INSERT at the first executable statement of `X1::operator+(float&)`. The subcommand causes the contents of the associated object to be output following each call to the function.

5.8 Reference variables

With the exception of its name, a reference variable has the same attributes as the variable it references. Thus the two variables are identical in terms of address, length, contents, storage type and return type. There are no special points to note in addressing reference variables with AID.

Example

```
C++ program                              SOURCE: EXP1.C
================================================================================
SRC
LIN
  1  int    i;
  2  int&   p = i;
  3
  4  int main(void)
  5  {
  6     i = 17;
  7     ...
```

```
/%d i, p, %@(i), %@(p), %l(i), %l(p)
SRC_REF: 7   SOURCE: BSP1.C   PROC: main  **********************************
i               =          17
p               =          17
010547C8
010547C8
          +4
          +4
```

Variables `i` and `p` differ only in their name. They have exactly the same value, address and length.

6 AID commands

%AID

The %AID command can be used to declare global settings or to change existing settings.

– The *C* operand defines how C string literals and C string arrays of the C and C++ programming languages are handled by AID.

– With the *CCS* operand, you specify a CCS for interpreting characters if no CCS is explicitly indicated in the %DISPLAY command. Unicode character sets are not allowed.

– The *CHECK* operand defines whether an update dialog is to be initiated before executing the %MOVE and %SET commands.

– The *DELIM* operand defines the delimiters for the output of alphanumeric data by AID. The vertical bar is the default delimiter.

– The *EXEC* operand defines whether debug mode is enabled after loading with `exec()`.

– The *FORK* operand defines whether a task created via `fork()` is interrupted immediately after its creation and set to debug mode.

– The *LANG* operand defines whether AID is to output %HELP information in English or German.

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

– The *LOW* operand instructs AID whether or not to convert lowercase letters of character literals and names to uppercase.

– The *OV* operand instructs AID to take the overlay structure of a program into account.

– The *REP* operand defines whether memory updates of a %MOVE command are to be stored as REPs.

– The *SYMCHARS* operand defines whether the "-" character in program, data and statement names is to be interpreted as a hyphen or as a minus sign by AID.

Command	Operand
%AID	C = {YES| NO} CCS = {<coded-character-set> | *USRDEF} CHECK [= {ALL| NO}] DELIM [= {C'x'|'x'C|'x' / '|'}] EXEC [= {OFF| ON}] FORK [= {OFF| NEXT| ALL}] LANG [= {D | E}] LEV [= {ON|OFF}] LOW [= {ON| OFF| ALL}] OV [= {YES| NO}] REP [= {YES| NO}] SYMCHARS [= {STD| NOSTD}]

The following applies for the validity period of definitions made with %AID:

– The settings made with %AID apply in the LOGON task until changed by a new %AID command or until /LOGOFF.

– All settings defined with %AID in the parent task are reset in the fork task. The only exception is FORK=ALL.

– An exec() call does not affect definitions made with %AID.

– All definitions made with %AID, apart from FORK=ALL, are reset in the POSIX shell after loading with debug *progname* (see page 291). If FORK=ALL was set in the LOGON task, it still applies. EXEC=ON is set after each loading with debug *progname*.

%AID may only be specified as a single command and may not be included in a command sequence or a subcommand.

%AID does not alter the program state.

C

YES AID supports the string notation of C/C++ and thus accepts C string literals in the form "*character string*" in %DISPLAY and %SET commands and in comparisons within subcommands. These literals are processed by AID as in C/C++, so you can now specify a comment only by enclosing it within /*...*/. In addition, AID combines the elements of a `char` array that are addressed via the last subscript into C strings, where the first array element of that subscript level with the value X'00' identifies the end of the C string. You will find more detailed information on this topic in the section "C strings" on page 36.

When AID commands are executed in a procedure, no parameter substitution occurs in C string literals even if `%AID C=YES` has been set, since the BS2000 command interpreter always interprets C string literals as comments.

Note that the `%AID C=YES` option also sets `%AID LOW=ALL` (see page 118) and `%AID SYMCHARS=NOSTD` (see page 120) at the same time.

NO Disables the interpretation of "*character string*" as a C string literal, which means that characters enclosed within "..." are seen as comments. AID treats `char` arrays as arrays of individual characters.

Note that if `%AID C=YES` was declared earlier, the implicit settings of `%AID LOW=ALL` and `%AID SYMCHARS=NOSTD` will be retained and would need to be reset independently if required.

If a *C* operand has not been entered in a debugging session, the default setting NO applies.

CCS

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

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

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

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

```
%SHOW %CCSN
```

CHECK

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

```
OLD CONTENT:
AAAAAAAA
NEW CONTENT:
BBBBBBBB
%  AID0274 Change desired? Reply (Y=Yes; N=No)?n
%  AID0342 Nothing changed
```

If `y` is entered, the old contents of memory 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 and new contents are written to SYSOUT. If SYSOUT is redirected, the above output will not appear on the terminal. The same applies if the %MOVE or %SET command and the CMD macro have been used and output to SYSOUT has been selected. Messages AID0274 and AID0342, by contrast, are always sent to the terminal.

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).

DELIM

C’x’|’x’C|’x’
With this operand the user defines a character as the left and right delimiter for AID output of symbolic data of type ’character’ (%DISPLAY and %SDUMP commands, or for update dialog in %SET command.)

| The standard delimiter is the vertical bar.

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

EXEC

OFF Programs loaded with an `exec()` call are not interrupted after loading and not set to debug mode.

ON Immediately after loading with `exec()`, the program is interrupted and set to debug mode. All settings made prior to this with AID remain intact.

`%AID EXEC` without a value is the same as `%AID EXEC=OFF` (default).

FORK

OFF Fork tasks are not interrupted after their creation and not set to debug mode (default). If %AID FORK has not been set in a task, `%SHOW %AID` displays `NOT_USED` for FORK.

NEXT All first-generation fork tasks are interrupted immediately after their creation and set to debug mode. However, `FORK=OFF` is set in these tasks, i.e. you cannot debug second and higher-generation tasks created with `fork()` without taking further steps. In this case, you can only set such a higher-generation fork task into debug mode by interrupting the fork task from the POSIX shell with `debug -p` *pid* (see page 291), or by sending a %STOP command including the appropriate *TSN* or *pid* (see page 275) from a task in the same task family to the required fork task.

ALL All fork tasks of any generation which originate from the current task are interrupted after their creation and set to debug mode. FORK=ALL is set in the fork tasks. This setting is the only AID definition that is inherited.

Changing this switch only affects the tasks which are created after the change in direct line from the task in which the switch was set.

`%AID FORK` without a value has the same effect as `%AID FORK=OFF` (default).

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, AID inserts the default (D).
You can also receive AID messages in German by using the SDF command `MODIFY-MSG-ATTRIBUTES TASK-LANGUAGE=D`.
This has no effect on the update dialog (see the CHECK operand).

LEV

ON Enable level output.

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

- A general level (counter) with a backward numeration, i.e. from the current procedure to the main procedure. This level number is applicable in the new qualification *NESTLEV*.
- A recursive level (`RLEV`) or an individual counter for each procedure with a backward numeration starting from 0. The recursive level serves as informative element.

OFF Disable level output.

LOW

ON Lowercase letters in character literals and in program, data and statement names are not converted to uppercase. When debugging C/C++ programs, you should set `%AID LOW` at the start of every debugging session; otherwise, AID cannot distinguish between upper and lowercase in C/C++. Only in S qualifications is AID not case-sensitive. Entries in the S qualification are always converted to uppercase.

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

ALL In addition to all entries affected by the `LOW=ON` setting, the distinction between uppercase/lowercase letters also being taken into account when all BLS names are entered. You always need this setting when you debug a program that was compiled in the POSIX shell and its associated source file name contains lowercase letters.

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

The following BLS names are used by AID:
- Context names of the CTX qualification
- Load unit names of the L qualification
- Link module names of the O qualification

– CSECT names of the C qualification
– COMMON names of the COM qualification
– Names of compilation units of the S qualification

Note that the `%AID C=YES` specification will implicitly set `LOW=ALL`, but this setting will not be revoked with `%AID C=NO`. `LOW=ALL` must be reset independently if required.

The presetting and default values differ for the *LOW* operand. If no *LOW* operand has been entered in a debugging session, the presetting OFF applies. However, if the *LOW* operand is input without a value specification, AID assumes the default (ON). In this case, the complete `%AID LOW=OFF` command must be entered if conversion to uppercase is to be reactivated.

OV

YES Mandatory specification if the user is debugging a program with an overlay structure. This also applies for programs which dynamically load and unload program sections (`BIND` / `UNBIND`). AID checks each time whether the program section 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 uses the one-time loaded LSD record without checking whether the addressed program section is in a dynamically loaded segment.

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

REP

YES In the event of memory updates caused by a %MOVE command, LMS correction statements (REPs) are created in SDF format. 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 be inserted in it for the LMS run. Care should therefore be taken that no other outputs are written to the file with link name F6. If no file with link name F6 is registered (see %OUTFILE), the REP record is stored in the file created by AID (AID.OUTFILE.F6).
User-specific REP files must be created with the SAM access method. REP files created by AID are likewise defined with the SAM access method, record format V and open mode EXTEND.
The file remains open until it is closed via %OUTFILE or until /LOGOFF.

NO No REPs are generated.

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

SYMCHARS

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

NOSTD

A hyphen (-) is always interpreted as a minus sign and cannot be used as a part of names. Since names in C/C++ may not contain hyphens, you should set NOSTD at the begin of each debugging session. You will then not need to be careful about when you need to enter a blank before a hyphen.

Note that the `%AID C=YES` specification will implicitly set `SYMCHARS=NOSTD`, but this setting will not be revoked with `%AID C=NO`. `SYMCHARS=NOSTD` must be reset independently if required.

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

%AINT

%AINT can be used to specify whether 24-, 31- or 32-bit addresses are to be used by AID for indirect addressing. The address preceding the pointer operator (->) will then be interpreted as a 24-, 31- or 32-bit address by AID.
This does not affect the addressing mode of the test object.

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

Command	Operand
%AINT	[aid-mode] [,...]

By default, AID interprets indirect address specifications in accordance with the current addressing mode of the test object. This automatic adaptation can be deactivated by specifying %AINT with the keyword %MODE*n*. The implicit addressing mode is 24 or 31 on /390 systems and can be retrieved with `%DISPLAY %AMODE` and altered with %MOVE (see the manual "Debugging on Machine Code Level:" [2]). The addressing mode for the current AID work area is also returned by `%SHOW %AID` or `%SHOW %BASE` along with other information.

Without a qualification, %AINT applies to AID commands that indirectly reference or use addresses of the current AID work area. If a qualification is specified, %AINT will apply only to AID commands that indirectly reference or use addresses of the qualified area.

The following applies for the validity period of the addressing mode defined with %AINT:

– The addressing mode applies in the LOGON task until default address interpretation is reverted to using %AINT without operands or %AINT with a base qualification and without %MODE*n*. Otherwise, the defined addressing mode applies until a /LOGOFF or /EXIT-JOB.

– The addressing mode is reset to default address interpretation in a task created with `fork()` and in a program loaded with `exec()`.

– The default address interpretation is always set in the POSIX shell after loading with `debug` *progname* (see page 291).

%AINT does not alter the program state.

aid-mode

Defines, for the current work area or an AID work area designated with the specified base qualification, how indirect addresses are to be interpreted in subsequent AID commands.

If you specify a keyword for the address interpretation and no qualification, the %AINT command will be valid for processing the current AID work area.

If you specify a base qualification and no keyword for the address interpretation, the default AID address interpretation will apply to the corresponding AID work area.

```
aid-mode-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                 ⎧VM⎫          ⎧%M[ODE]32⎫
[•][E=⎨  ⎬[•]] [⎨%M[ODE]31⎬]
                 ⎩Dn⎭          ⎩%M[ODE]24⎭

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

- A leading period serves as an indicator for a *prequalification*, which must be defined beforehand with the %QUALIFY command. A period must be entered between the base qualification and the keyword for the address interpretation. If you specify only a base qualification, no terminating period is permitted.

E={VM | Dn}
: Specifies that the conversion of the address interpretation is not to be applicable for the current AID work area. If you specify only a base qualification, the default address interpretation will again apply to the referenced area.

{%M[ODE]32 | %M[ODE]31 | %M[ODE]24}
: Keyword that defines how many bits are to be taken into account for indirect addressing in AID commands.

```
%M[ODE]32   32-bit addressing
%M[ODE]31   31-bit addressing
%M[ODE]24   24-bit addressing
```

Examples

Address V’100’ has the contents: 1200000C
Register 5 has the contents: 010001A0

```
1. %AINT %MODE24
   %DISPLAY V'100'->
   %MOVE %5-> INTO %5G
```

%AINT is used to switch to a 24-bit address interpretation. The change applies to the current AID work area.
%DISPLAY outputs 4 bytes as of address V’00000C’.
%MOVE transfers 4 bytes as of address V’0001A0’ to AID register 5.

```
2. %AINT %MODE31
   %DISPLAY V'100'->
   %MOVE %5-> INTO %5G
```

The address interpretation for the current AID work area is now switched to a 31-bit interpretation.

%DISPLAY outputs 4 bytes as of address V’1200000C’.
%MOVE transfers 4 bytes as of address V’010001A0’ to AID register 5.

%ALIAS

You can use %ALIAS to define short alias names for long variable names or class or namespace qualifications to avoid having to type in the long original names in subsequent commands.

– *aliasname* defines an abbreviated name that you want to use in subsequent commands instead of the original name.

– *originalname* defines the name of a data item, class, class object, namespace, template instance or function. You can prepend a class or namespace qualification to the name with or use two colons (`::`) for the global namespace.

Command	Operand
%ALIAS	aliasname [= originalname]

An *aliasname* defined with %ALIAS applies until it is deleted with a %ALIAS command without an *originalname* operand or until /LOGOFF or /EXIT-JOB.
In the POSIX shell, all alias names are deleted after a `fork()` call, i.e. also after `debug` *progname* (see the chapter “POSIX debug command” on page 291). Names defined with %ALIAS still apply in a program loaded via `exec()`.

The %ALIAS definitions are only available to subsequently input commands. A new %ALIAS has no effect on previously input subcommands in %CONTROL*n*, %INSERT and %ON, even if the subcommands are executed after the %ALIAS.

When you input an alias name, you must ensure that the uppercase/lowercase handling is correctly set (`%AID LOW={ON|OFF|ALL}`).
Several alias names can be assigned to one original name.
AID always outputs the original name in messages.
You can assign up to 40 alias names, depending on the length of the assigned original name.

%ALIAS may only be input as a single command and may not be included in a command sequence or subcommand.

%ALIAS does not alter the program state.

aliasname

Defines the name that you can use in subsequent AID commands instead of *originalname*. *aliasname* can be up to 32 characters long. The following characters can be used: a-z, A-Z, 0-9, $, #, @, underscore (_) or dash (-).
If you assign several alias names, they must all be different. AID rejects duplicated alias names with the following message:
`AID0531 Alias name is ambiguous.`

An alias name you assign should not be the same as the name of a definition in your program as this will prevent you from being able to subsequently access the definition. For the same reason, the alias name should not be the same as an AID keyword or you will no longer be able to use the keyword.

originalname

Designates the definition from the source program which is to be addressed via the shortest alias name during the subsequent debug run.

If you do not specify an *originalname* operand, *aliasname* is deleted from the list of alias names. If *aliasname* did not exist, you are informed of this with the message:
`AID0530 Alias name is undeclared.`

```
originalname-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -

= [::]name [::name...]

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

name Name of a data item, function, function template instance, class, class template instance, class object or namespace defined in the source program.

You can specify a data name as described in the section “Data names” on page 29. You can specify functions with the notation *function*, `n'`*funktion*`([`*signatur*`])'` or `t'`*f_template*`<`*arg*`[,...]>([`*signature*`])'`, depending on the type of function concerned (see page 58).
With derived or nested classes or nested namespaces, you can specify the complete path for accessing the required class or namespace for *originalname*. You specify all superordinate classes or namespaces from the outermost to the innermost and separated by a pair of colons (`::`).
You specify class template instances in the form `t'`*k_template*`<`*arg*`[,...]>'`.

If you define an alias name for an AID keyword, AID warns you of this but still accepts the assignment and replaces the AID keyword in subsequent commands with the alias name.

An alias name assigned to an original name may not be used as an original name.

Example

```
/%alias FJ=Fujitsu
/%display FJ::i
Fujitsu::i    =    32
```

The %ALIAS command is used to assign the alias name `SAG` to the namespace `Fujitsu`. A variable `i` from this namespace is referenced in the subsequent %DISPLAY with the short form `FJ::i`. AID then displays the original name in the output.

%BASE

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

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

Command	Operand
%BASE	[base]

When debugging C/C++ programs, the AID work area corresponds to the area occupied by the load unit in virtual memory or in a dump file. If you do not specify a %BASE command during a debugging session or enter %BASE without any operands, the base qualification E=VM applies by default, and the AID work area corresponds to the unprivileged portion in virtual memory that is used by the loaded program with all connected subsystems (AID default work area).

A %BASE command remains in effect until the next %BASE command or a /LOGOFF or /EXIT-JOB, is issued, or until the dump file that was declared as the base qualification is closed (see the description of the command %DUMPFILE).
Immediately after input, all memory references in a command, even within a subcommand, are supplemented with the current base qualification, i.e. a %BASE command has no effect on subcommands specified previously.

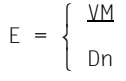
%BASE can only be entered as an individual command, it must never be part of a command sequence or subcommand.

%BASE does not alter the program state.

base

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

base–OPERAND



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

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

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

%CONTINUE

The %CONTINUE command is used to start the program which has been loaded or to continue it at the interrupt point.
The address at which program execution is continued can not be influenced with %CONTINUE. You can define such an address only by changing the program counter (%PC) using %SET (see the description of the command %SET *keyword* on page 263).

An active %TRACE command is not terminated by %CONTINUE - in contrast to %RESUME; instead, it is continued in conformance with the declarations which have been made.

Command	Operand
%CONT[INUE]	

A %TRACE is regarded as active as soon as it is entered.

In the following cases a %TRACE command is merely interrupted and can be resumed by a %CONTINUE command:

1. When a subcommand has been executed, and the subcommand contained a %STOP.
2. When the K2 key has been pressed (see the section “Commands on starting a debugging session” on page 19).

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 process monitoring functions, which then go into effect simultaneously. The seven commands are %CONTROL1 through %CONTROL7.

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

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

– By means of the *subcmd* operand you declare a command or a command sequence and possibly a condition. *subcmd* is executed if *criterion* is satisfied and any specified condition has been met. <%STOP> is used by default if *subcmd* is not explicitly specified.

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

Several %CONTROL*n* 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 %CONTROL*n* 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 %CONTROL*n* is valid until overwritten by a new specification in a later %CONTROL*n* command with the same number, until the %CONTROL*n* command is deleted or until the end of the program. In addition, all %CONTROL*n* declarations are reset in a task created with `fork()` and in a program loaded with `exec()`.

A %REMOVE command can be used to delete either a specific %CONTROL*n* or all active %CONTROL*n* declarations.

%CONTROL*n* 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.

%CONTROL*n* does not alter the program state.

criterion

Keyword defining the type of the program 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 %CONTROL*n* command is still valid.

criterion	*subcmd* **is processed prior to**
%STMT	Every statement
%ASSGN	Every assignment statement
%CALL	Every function call
%COND	Every `if` and `switch` statement, every `else` branch of an `if` statement, and every control expression of a `do`, `while`, or `for` statement
%EH %EXCEPTION	Every `throw` or `catch` statement
%GOTO	Every `goto`, `break`, and `continue` statement
%LAB	Every statement with a label; does not apply to `case` and `default` labels
%PROC	The first and the last statement of a function

Table 4: Values of the *criterion* operand and their meanings

control-area

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

A *control-area* definition is valid until the next %CONTROL*n* command with the same number is issued with a new definition, until the corresponding %REMOVE command is issued, or until the end of the program is reached. All *control-area* definitions are also reset in a task created with `fork()` and in a program loaded with `exec()`.
%CONTROL*n* without a *control-area* operand of its own results in a valid area definition

being assumed. 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 translation unit by default.

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

                ⎧S=srcname                                          ⎫
                ⎪                                                   ⎪
IN  [•][E=VM•]⎨             ⎧[qua•][PROC=]function                  ⎫⎬
                ⎪[S=srcname•]⎨BLK='[f-]n[:b]'                       ⎬⎪
                ⎩             ⎩([PROC=function•]src-ref:src-ref)    ⎭⎭

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

- If the period is in 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

Specified only if *control-area* is not to be located in the current translation unit or if a defined area restriction is no longer to apply.
If *control-area* ends with an S qualification, it includes the entire specified translation unit.
The translation unit specified using *srcname* must be loaded at the time at which the %CONTROL*n* command is issued or at which the subcommand which contains the %CONTROL*n* command is processed.

[qua•][PROC=]function

Specified if *control-area* is not to be located in the current function or if a previously valid *control-area* definition is to be overwritten.
In the case of functions from C programs, *function* is the function name declared in the source program, but without parentheses or the signature.
You must specify functions from C++ programs in the notation `n'...'` or `t'...'`, depending on the type concerned. If the function is defined in a namespace or a class, the namespace or class qualification is prepended to the function name.
The `void` signature may no longer be written. In this case, as in C++, you only enter the two parentheses after the function name.

The following syntax results for *function*:

```
                                 ⎧n'function([signature])'                ⎫
[namespace::[...]][class::[...]]⎨                                          ⎬
                                 ⎩t'f_template<arg[,...]>([signature])'   ⎭
```

The `main` and `_ _STI_ _` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

Syntax for virtual functions:

`p->n'function([signature])'`

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p*. (see the description of `this` on page 64 and the section “Virtual functions” on page 73).

If you want to specify a function addressed via a pointer to member as the program area, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

```
[qua.]object.*[object.][class::][...]pointer-to-function-member
```

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

```
[qua.]pointer->*[object.][class::][...]pointer-to-function-member
```

The class object is designated by the operand on the left of the dereferencing operator `.*` or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.

The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to member cannot be reached from the interrupt point by some other means. More details on working with a pointer to function member can be found on page 79.

qua

If the function is defined in a local class, you have to specify a PROC qualification for the superordinate function containing the definition of the local class, before the name of the required function. For functions defined in an inner block of the superordinate function, you append to the PROC qualification one or possibly several BLK qualifications, each separated by a period, to describe the path to the local class (see page 60).

Syntax for *qua*:

```
PROC=top-level_fct•[BLK='[f-]n[:b]'•[...]]
```

Accessing functions defined in local classes of inner blocks is only supported by programs that were compiled with a C/C++ compiler as of V3.0B.

BLK='[f-]n[:b]'

control-area is defined by means of a BLK qualification and contains the entire specified block. The name for a block is constructed using the line number (*n*) and possibly a FILE number (*f*) and relative block number (*b*).

The BLK qualification cannot be used in combination with the *criterion* %PROC.

([PROC=function•]src-ref : src-ref)

With source references you can define *control-area* by specifying a start and an end address. Both must lie within the same translation unit such that:
start address ≤ end address.
Note that ascending source references are only assigned ascending addresses within a function block. If the condition start address ≤ end address is not satisfied, AID rejects the command with a corresponding error message.

If *control-area* is to contain only one statement, the start and end addresses must be identical.

PROC=function•
You only have to specify the PROC qualification if the specified source references are not unique in the translation unit. This is the case if the source references are in a function resulting from instantiation of a function template or if the function containing the source references is defined in a class template and at least two instances exist for the template (see above and page 108).

src-ref
is specified with `S'[`*f*`-]`*n*`[:`*a*`]'` and identifies the address of an executable statement, where *n* is the line number, *f* is the FILE number and *a* is the relative statement number within the line (if >1).

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.
If the *subcmd* operand is not specified, AID inserts a <%STOP>.

For a complete description of *subcmd* see the “Subcommand” chapter in the AID Core Manual [1].

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

                                    ⎧AID-command    ⎫
<[subcmdname:] [(condition):] [     ⎨               ⎬ {;...}]>
                                    ⎩BS2000-command ⎭

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

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

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

In addition to the commands which are not permitted in any subcommand, the *subcmd* of a %CONTROLn must not contain the AID commands %CONTROLn, %INSERT 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 in *subcmd* will not be executed. Likewise, deletion of the current subcommand via %REMOVE is only expedient as the last command in *subcmd*.

Address operands in subcommands are not automatically supplemented on input with the qualifications that correspond to the current interrupt point. If a statement of type *criterion* occurs in the subsequent debugging run, and AID interrupts the program to process *subcmd*, only the data and functions that are visible at the address of the statement that caused the interrupt can be addressed without qualification with AID commands from *subcmd*.

Examples

1. `%control1 %call, %proc in (s'123':s'250') <%display countr; %stop>`
   `%c1 %call,%proc in (s'123':s'250') <%d countr;%stop>`

   These two AID commands differ only in their notation.
   The first example is written out in full and contains varying numbers of blanks in permissible places; the second is abbreviated.

   The %CONTROL1 command applies for the criteria %CALL and %PROC and is effective between statement lines 123 through 250.

   If a statement matching the criteria %CALL or %PROC occurs during program execution in the specified area, the %DISPLAY command from *subcmd* is executed for the `count` variable. Program execution is then interrupted by %STOP, and AID or BS2000 commands can be entered.

2. `%control1 %call <%display 'call'; %stop>`

   Before every function call, AID executes the %DISPLAY command from *subcmd* and then interrupts the program as a result of the %STOP command.

3. `%control2 %goto <%sdump %nest p=max; %remove %c1; %stop>`

   Before a goto, break or continue statement is executed, AID outputs the call hierarchy. Since `p=max` is specified, it is written to the system file SYSLST. AID then executes the %REMOVE command, which deletes the declarations of %CONTROL1. The program is stopped (%STOP).

4. `%c3 %proc in nread`

   The %C3 command causes AID to interrupt the program before the first or last statement of the function `nread` is executed.

5. `%c4 %assgn <(z1 le 10): %d tab[0])>`

   The %C4 command causes AID to output the first element of the array `tab` before every assignment statement, but only if `z1` evaluates to less than or equal to 10.

6. *Exception handling*

   In the program `EXMEM.C` shown below, an attempt is made to allocate memory for two pointers `p` and `q`. The size of the requested memory is such that the allocation triggers the exception handling.

```
C++ program                                                        EXMEM.C
================================================================================
SRC
LIN
  1  #include <iostream.h>
  2  #include <cstdlib>
  3  #include <new>
  4
  5  void main()
  6  {
  7    char* p;
  8    try
  9    {
 10      p = new char[0x1000000];
 11    }
 12    catch(bad_alloc)
 13    {
 14      cerr << "No more memory!\n";
 15      return;
 16    }
 17    char *q;
 18    try
 19    {
 20      q = new char[0x10000000];
 21    }
 22    catch(bad_alloc)
 23    {
 24      cerr << "No memory for second pointer!\n";
 25      delete[] p;
 26      try
 27      {
 28        q = new char[0x10000000];
 29      }
 30      catch(bad_alloc)
 31      {
 32        cerr << "No more memory!\n";
 33        return;
 34      }
 35    }
 36  }
```

Trace log:

```
/LOAD-PROG   *MOD(MYLIB,EXMEM,RUN-MODE=ADVANCED,PROGRAM-MODE=ANY),
TEST-OPTIONS=AID
%  BLS0523 ELEMENT 'EXMEM', VERSION '@' FROM LIBRARY '$TEST.MYLIB'
IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXMEM$', VERSION ' ' OF '1999-01-07
10:27:02' LOADED
/%c1 %eh in s=n'exmem.c' <%t 1 r %stmt>
/%r
10                        EXT.PROC START   , BLOCK START, , BLOCK START,
                          ASSIGN
20                        BLOCK END,  , BLOCK START, ASSIGN
24                         , BLOCK START, CALL
No memory for second pointer!
28                         , BLOCK START, ASSIGN
32                         , BLOCK START, CALL
No more memory!
%  CCM0998 CPU TIME USED: 0.1531 SECONDS
```

Immediately after the program is loaded, a %CONTROL1 command is input to log the exception handling process. The program is then started with %RESUME.
The first statements of the `try` and `catch` blocks are each listed in the output in the order in which they are executed by the program.

%DISASSEMBLE

%DISASSEMBLE enables memory contents to be "retranslated" into symbolic Assembler notation and displayed accordingly. Output is via SYSOUT, SYSLST, or into a cataloged file.

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

– The *start* operand defines the address where AID is to begin disassembling.

Command	Operand
{ %DISASSEMBLE | %DA }	[output-quantity] [FROM start]

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.

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

If the user has not entered a %DISASSEMBLE command during a test session or has changed the test object and does not specify current values for one or both operands in the %DISASSEMBLE command, AID works with the default values 10 for *output-quantity* and V'0' for *start*. The default value V'0' for *start* cannot be used for C/C++ programs as these programs are loaded into the upper memory address space. You therefore have to specify an explicit value for *start* with the first %DISASSEMBLE command.

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 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.

output-quantity

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

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

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

{number }
{length }
{ALL    }

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

number

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

is an integer with the value:
$1 \leq number \leq 2^{31}-1$

length

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

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

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

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

start

Defines the address at which disassembly of memory contents into Assembler instructions is to begin. If the *start* value is not specified, AID will assume the default value V’0’ for the first %DISASSEMBLE command after a program is loaded and will issue an error message if the program was not loaded at the start address V’0’.

On every subsequent %DISASSEMBLE, AID continues after the Assembler instruction last disassembled.

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

                  ⎧ function[->]       ⎫
                  ⎪ L'label'           ⎪
FROM  [•][qua•] ⎨                     ⎬
                  ⎪ S'[f-]n[:a]'       ⎪
                  ⎩ compl-memref       ⎭

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

- If the period is in a 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
: One or more qualifications may be specified here to address a function, label or source reference that cannot be reached from the current interrupt point by other means.

  E={VM | Dn}
  : Specified only if the current base qualification (see %BASE) is not to apply for a function, label or source reference.

  S=srcname
  : Specified only if a function, label or a source reference is not located in the current translation unit (see the chapter “Addressing in C and C++ programs” on page 21).

  ::
  : You specify the two colons for the global namespace if the name of a namespace, global class or function is hidden by a definition with the same name at the interrupt point. Only an E or S qualification may precede the two colons.

  namespace::
  : Specified if you wish to access a function which is defined in a namespace that has not been made known with a `using` directive before the interrupt point, if the function has not been registered with a `using` declaration or there is ambiguity at the interrupt point.
    Only an E or S qualification may precede a namespace qualification (see the section “Qualifications” on page 57).

class::
: Specified if the required function is defined in a class and the interrupt point does not belong to the function name scope. If *class* is an instance of a class template, you have to use the notation `t'`*k_template*`<`*arg*`[,...]>'` for *class* (see the section “Qualifications” on page 57).

BLK='[f-]n[:b]'
: You must specify a BLK qualification if you want to designate a function from a local class, which is defined in the specified block, in a subsequent PROC qualification (see below, PROC=*function*).
The block name is formed from the line number (*n*), a possible FILE number (*f*) and relative block number (*b*).

PROC=function
: Specified only if you want to reference a label from a function other than the current one (see the chapter “Addressing in C and C++ programs” on page 21) or if disassembly is to start with a source reference which is in a function template instance or assigned to a function which is defined in a class template instance (see the section “Templates” on page 94) and at least two instances exist.
In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.

  Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the namespace or class qualification is prepended to the function name. The `void` signature may no longer be used. In this case, you only input the two parentheses after the function name, as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated below for space reasons):

```
PROC=[namespace::[...]][class::[...]]{n'function([sign])'          }
                                     { t'f_templ<arg[,...]>([sign])'}
```

  The `main` and `__STI__` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

  If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several

BLK qualifications (see page 60) between the two PROC qualifications. Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

function[->]

Places *start* at the first executable statement in a function or at the first instruction in a library function.
function is the name of a function as declared in the source program or the name of a library function (see PROC=*function* above).

Syntax for virtual functions:

```
p->n'function([signature])'
```

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the first executable statement of the current function by using the `this` pointer instead of *p*.

If you want to specify a function addressed via a pointer to member as *start*, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

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

[qua.]object.*[object.][class::][...]pointer-to-function-member

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

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

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

[qua.]pointer->*[object.][class::][...]pointer-to-function-member

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

The class object is designated by the operand on the left of the dereferencing operator `.*` or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.
The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the

definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to member cannot be reached from the interrupt point by some other means.

Disassembly begins at the first executable instruction of the function currently associated with the pointer to function member.

If you want to add an offset to the address designated by the dereferenced pointer to function member in order to start the disassembly at a computed address somewhere in the middle of the function, note that you cannot directly append the pointer operator to one of the syntaxes above. Instead, you would have to first specify a type modification, i.e. %al4, to indicate that the address designated by the dereferenced pointer to function member should serve as a starting point from which the offset is to be calculated. This results in the following syntax:

```
dereferenced-pointer-to-function-member %al4->•offset
```

Note, however, that the address calculation does **not** begin with the address of the first executable instruction in this case, but with the prolog address of the function.

You will find more details on working with a pointer to function member on page 79.

If you use *start* to designate a library function, you must terminate *function* with the pointer operator, since there is no LSD for the library functions. Disassembly begins in these cases at the first instruction of the function prolog.

L'label'

Places *start* at the first executable statement after a label.
label is the name of a label declared in the source program. In this command you can also specify *label* without L'...', since there can be no confusion with a data name.

S'[f-]n[:a]'

Is a source reference and designates an executable statement. The source reference is constructed from the line number (*n*) and, if present, the FILE number (*f*) and the relative statement number within the line (*a*).
If the source reference is in a function created from a function template via instantiation or the function containing the source reference is defined in a class template instance, you have to add an appropriate PROC qualification before the source reference in cases of ambiguity.

compl-memref

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

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

If a *compl-memref* begins with a source reference, a label or a function name, the pointer operator must come next. Note, however, that the pointer takes you out of the symbolic level. If you put a pointer name after a function name, what you reference is not the first executable statement in the function but the first instruction in the prolog generated for the function by the compiler. If you use a label in a complex memory reference, you must always place it in L'...'.
Source references, labels and function names can be used without the pointer operator wherever hexadecimal numbers are allowed.

A type modification makes sense only if the contents of a variable can be used as an address or if the address is taken from a register.

Example: `%3g.2%a12->`
The last two bytes from AID register %3G 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.

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

The following is contained in a %DISASSEMBLE 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

Beginning with the first statement of the function facul (see the description of the %SDUMP command, Example 4 on page 248), eight machine instructions are to be disassembled.

```
/%disassemble 8 from facul
         EXAMP$O&@
 EXAMP$O*+20A       TM    O(R9),X'80'            91 80 9000
 EXAMP$O*+20E       BC    B'1000',3A(R0,R10)     47 80 A03A
 EXAMP$O*+212       L     R1,8(R0,R8)            58 10 8008
 EXAMP$O*+216       L     R15,4C(R0,R13)         58 F0 D04C
 EXAMP$O*+21A       ST    R13,18(R0,R15)         50 D0 F018
 EXAMP$O*+21E       L     R13,4(R0,R13)          58 D0 D004
 EXAMP$O*+222       ST    R13,C(R0,R15)          50 D0 F00C
 EXAMP$O*+226       L     R14,C(R0,R13)          58 E0 D00C
```

Example of line format with T=MIN

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.

```
/%out %da t=min
/%disassemble 8 from facul
 0100020A  TM    O(R9),X'80'
 0100020E  BC    B'1000',3A(R0,R10)
 01000212  L     R1,8(R0,R8)
 01000216  L     R15,4C(R0,R13)
 0100021A  ST    R13,18(R0,R15)
 0100021E  L     R13,4(R0,R13)
 01000222  ST    R13,C(R0,R15)
 01000226  L     R14,C(R0,R13)
```

Examples

1. `%disassemble from s=n'examp.c'.::facul`

   This command causes ten instructions (default value) to be disassembled, beginning at the address of the first statement in the function facul, which is contained in the translation unit `EXAMP.C`.

2. `%da 2 from e=d1.s'27'`

   In the dump file with the link name `D1`, two instructions in the program code generated for the first statement in line 27 are to be disassembled.

3. `%da from s'27:2'`

   Since no value was specified for *ausgabe-menge*, AID either sets the default value of 10 - if this is the first %DISASSEMBLE for this program - or assumes the value from the previous %DISASSEMBLE. The disassembling begins with the first instruction generated for the second statement in line 27.

4. `%disassemble from l'output'->.4`

   Sets the starting point for disassembly at a position 4 bytes from the first instruction after the label `output`.

5. `%da #'18' from facul`

   24 bytes are disassembled at the beginning of the `facul` function, that is, the instructions to the addresses 0100020A to 0100021E. The instruction to the address 0100022 is no longer displayed as it is located outside of the range.

%DISPLAY

The %DISPLAY command is used to output memory contents, addresses, lengths, system information and AID literals and to control feed to SYSLST. AID processes data in accordance with their definition in the source program, unless you select another type of output by means of type modification.
You can use %DISPLAY to call up a list of all overloaded functions that have the same names in the current scope or the scope you explicitly specify. You can also use the %DISPLAY command to list all instances of a template
Output is via SYSOUT, SYSLST or to a cataloged file.

– By means of the *data* operand you specify namespaces, templates, classes, class objects, data, their addresses and lengths, statements, registers, execution counters of subcommands, and system information. Here you also define AID literals or you control feed to SYSLST.
– By means of the *medium-a-quantity* operand you specify the output medium AID uses and whether or not additional information is to be output. This operand disables a declaration made via the %OUT command, but only for the current %DISPLAY command.

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

If you omit the qualification for *data*, you address the data of the current block (for nested blocks, also the data of the outer blocks), the data of the current function (unless it was defined after the interrupt point), and the global data of the associated translation unit.
In the case of identical names within the current call hierarchy, AID outputs the data that would also have been addressed by the program at the interrupt point.

If you do specify a qualification, you can access *data* in a dump file or in another loaded translation unit or function or in another block, provided the scope of the addressed data lies in the current call hierarchy. Outside the current call hierarchy you can address only data of storage class static or extern.

If you need to address an overloaded function, you can enter
%DISPLAY [*qua*]*function*
to display an overview of all functions with the specified name that were found by AID in the current or explicitly specified scope (see the section “Overloaded functions” on page 110). The functions are listed in standard C++ notation together with the signature and, if present, the prepended block numbers and class names.

You can also use
`%DISPLAY [`*qua*`] `*template*
to output an overview of all instances of a class or function template which are visible at the current interrupt point. AID searches for the template declarations starting from the interrupt point according to the scope rules for data or in the program section designated with *qua* (see page 106). If only a single instance was created for the template, you can have it listed with
`%DISPLAY [`*qua*`] { t'`*k_template*`'|t'`*f_template*`([`*signature*`])' }`

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

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

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

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

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

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

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

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

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

To modify the current character set use the following AID command:
`%AID CCS = {<coded-character-set>|*USRDEF}`

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 (see the AID Core Manual, chapter "Medium-a-quantity operand" [1]).

Immediately after loading the program, you can access only global and static data. AID needs the appropriate qualifications to access them.

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

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

%DISPLAY does not alter the program state.

data

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

AID processes data in accordance with the definitions in the source program, provided that you have not defined another type of output using a type modification (see also the section dealing with type modification in the AID Core Manual [1]).

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 code specifications can also be combined within a complex memory reference.

When using overloaded operators, note that AID does not emulate this process, but always uses standard operators.

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.

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

qua Specify one or more qualifications only if the interrupt point is not located within the current *data* scope or if *data* is not visible at the interrupt point. Specify only the qualifications necessary for accessing the memory object uniquely.

E={VM | Dn}
: Specified only if the current base qualification (see %BASE) is not to apply for a data name, statement name, source reference or keyword.

S=srcname
: Specified only if you are accessing a data name, a class or a class object, a statement name or a source reference which is not located in the current translation unit (see the chapter “Addressing in C and C++ programs” on page 21).

:: Use the two prepended colons to address a global data item that is locally hidden at the interrupt point by a definition of the same name. You must also place two colons before the name of a global data item or a function if either the data or the function is not in the call hierarchy or if its definition only occurs after the interrupt point. In contrast to the other qualifications, no period must be entered between the two colons and the operands which follow.

PROC=function

Specified only if you want to access a data name which is defined in the current function, but is hidden at the interrupt point by a definition with the same name. You also specify a PROC qualification when you want to address a label or a data name declared as static which is defined in a function outside the current call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21). If you specify a source reference that is located in a function template instance or assigned to a function defined in a class template instance, (see the section “Templates” on page 94), you also have to prepend the appropriate PROC qualification if ambiguity occurs.
In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.
Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the namespace or class qualification is prepended to the function name. The `void` signature may not be written, you enter just the two parentheses in this case as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated for space reasons):

```
PROC=[namespace::[...]][class::[...]]{n'function([sign])'          }
                                     {t'f_templ<arg[,...]>([sign])'}
```

The `main` and `__STI__` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications. Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK=’[f-]n[:b]’

Specified only when you want to address a data name which is assigned to a block within the current call hierarchy and is hidden at the interrupt point by a definition with the same name, or when you want to want to access a data name declared as static and assigned to a block outside the current call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21).

You must also specify a BLK qualification if you want to designate a function from a local class, which is defined in the specified block, in a subsequent PROC qualification (see PROC=*function* above).

NESTLEV= level-number

level-number A level number in the current call hierarchy

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

namespace

Name of a namespace declared in the source program.

If the *data* operand ends with the name of a namespace, AID lists all data and functions defined in it. The functions are listed in standard C++ notation and the start address of the associated prolog is output. With nested namespaces, the contents of the inner levels are also output. If a namespace contains a `using` directive to a further namespace, only the name of this is listed.

You only specify the namespace qualification in the addressing path to classes, data or functions defined in the namespace if the required namespace component is not visible at the interrupt point.

Only the E or S qualification or the two colons (`::`) for the global namespace are allowed before the namespace qualification.
You will find more information on namespaces in the section “Namespaces” on page 85.

{ class | this-> | object }

Name of a class, the `this` pointer, or the name of a class object, as declared in the source program.
You specify class names, the `this` pointer with the appended pointer operator, and the names of class objects in order to describe the address path to data members assigned to classes (see the section “Classes” on page 63).

If the current interrupt point is located in a dynamic member function, you can address the class data according to the scope rules known from C++.

If an object is in the current call hierarchy, you can access the dynamic data of that object independent of the interrupt point by means of the object name followed by a period.

Static data members can be accessed from any part of the program via the associated class names followed by the two colons. In the case of nested classes, the path to the data item includes all the class names from the outermost to the innermost level, separated by two colons each. The outermost class name requires

qualification appropriate to the scope. If the program is interrupted within a member function of the class, the class scope rules apply for accessing static data members, i.e. if the data is not hidden by a definition with the same name, it can be accessed without qualification.

If the class is a class template instance, you have to use the following notation: `t'`*k_template*`<`*arg*`[,...]>'`. If only one instance of the template exists, only `t'`*k_template*`'` is required.

If the *data* operand consists of one or more class names, and the interrupt point is outside the class system, the static data members and all non-virtual member functions are listed. In the case of data, the current content is output. Member functions are listed in standard C++ notation and the start address of the associated prolog is output. For derived classes, AID also displays the base classes. Nested classes are shown together with the contents of the inner levels.
However, if the interrupt point is located within a member function of the class, AID additionally lists the dynamic data members and the virtual functions. The prolog address of the currently valid member function is displayed with the name of the virtual function. You can get the same output with `%DISPLAY *this`.

If the *data* operand ends with an object name, AID also outputs the complete object, i.e. with the dynamic data members and complete information on the virtual functions. If required to uniquely address a data item or a base class function, append the name of the required base class to the object name, separated by a period. The scope rules known from C++ then apply within the class system.

object requires a qualification appropriate to its scope. Only a base qualification is meaningful before `this`.
If the *data* operand ends with `this->`, AID displays 4 bytes as of the starting address of the current object in dump format (hexadecimal and character).

dataname

This is the name of a data item declared in the source program. *dataname* is specified as in the source program.

You can reference data as in C/C++, but with the following exceptions:
For an array name without a subscript, AID displays all array elements.
Individual array elements can be addressed only via subscripts, not via pointers.
Subscript ranges can also be displayed.
If `%AID C=YES` is set (see page 115), AID combines the array elements of a `char` array that can be addressed via the subscript on the extreme right into C strings and displays the contents of the array in the form of C string literals.
For more information on working with arrays, see also the section “Subscript notation” on page 30.

For variables of type `long double`, AID evaluates only the first 8 bytes.
Variables of type `char` are displayed in output type %C. If desired, you can also display the appropriate numeric value by using a type modification (%A or %F; see page 29 for details). The data types `unsigned char` and `signed char`, by contrast, are treated as integer variables.
Arrays that are passed as parameters to a function and pointers are displayed as hexadecimal numbers.

If *dataname* designates a pointer to member, the name of the class member currently referenced by the pointer to member appears in the output. AID shows the contents of the current data member or the start address of the current member function if you specify the dereferenced pointer to member in the %DISPLAY command. More information on dereferencing a pointer to member can be found on page 76.

dataname can be specified as follows. The formats can also be combined (see the section “Data names” on page 29).

Subscript notation:	*dataname* `[`*subscript*`]` `{...}`
Pointer notation:	*dataname1* `->` *dataname2*
Structure qualification:	*superordinate dataname*`.` `{...}` *dataname*
Dereferencing:	`[(]*{...}` *dataname*`[)]`
Pointer to member dereferencing:	*dataname1*`.*`*datanname2* or *dataname1*`->*`*datanname2*

function

The name of a function as declared in the source program or the name of a library function (see PROC=*function* on page 152 and the chapter “Addressing in C and C++ programs” on page 21). Without the appended pointer operator, AID outputs the start address of the function prolog; with the pointer operator, the first 4 bytes as of this address are output.

The following syntax is used to address virtual functions:
`p->n'function(signature)'`

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p*. (see the description of `this` on page 64 and the section “Virtual functions” on page 73).
The instruction code as of the start address of the prolog can be accessed with `p->n‘function([signature])->.`

If you want to display the start address of the prolog of a function addressed via a pointer to member, you can dereference the pointer to member by using one of the methods below:

You designate the class object by name and enter •* as the dereferencing operator as follows:

```
[qua•]object•*[object•][cass::][...]pointer-to-function-member
```

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

```
[qua•]pointer->*[object•][cass::][...]pointer-to-function-member
```

The class object is designated by the operand on the left of the dereferencing operator •* or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.

The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to member cannot be reached from the interrupt point by some other means.

If you want to display the instruction code starting with the prolog address of the function, note that you cannot directly append the pointer operator to one of the syntaxes above. Instead, you would have to first specify a type modification, i.e. %al4, to switch to machine code level. This can be achieved with the following syntax:

`%DISPLAY` *dereferenced-pointer-to-function-member* `%al4->`

More details on working with a pointer to function member can be found on page 79.

L'label'

Designates the address of the first executable statement after a label.
label is the name of a label declared in the source program.

S'[f-]n[:a]'

Source reference which designates an executable statement. It comprises the line number (*n*) and possibly the FILE number (*f*) plus the relative statement number within the line (*a*). If the source reference is located in a function which was created from a function template via instantiation or the function containing the source reference is defined in a class template instance, you have to prepend the appropriate PROC qualification to the source reference if ambiguity occurs.

Without the subsequent pointer operator, AID outputs the address of the command code generated for the statement. With the subsequent pointer operator, AID outputs 4 bytes of the command code to this address.

keyword

Here you may specify all the keywords for program registers, AID registers, system tables and the one for the execution counter or the symbolic localization information (see the AID Core Manual, "Keywords" section [1]).
keyword can only be preceded by a base qualification.

%n	General register, 0 ≤ n ≤ 15
%nD|E	Floating point register, n = 0,2,4,6
%nQ	Floating point register, n = 0,4
%nG	AID general register, 0 ≤ n ≤ 15
%nGD	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: either 24 or 31. The addressing mode is defined when the program is loaded.
%PCB	Process control block
%PCBLST	List of all process control blocks.
%AUD1	P1 audit table starting with the latest entry: only created during system generation.
%SORTEDMAP	List of all CSECTs of the user program (sorted to names and addresses). Long names are truncated.
%MAP [(CTX=context)]	List of CSECTs and COMMONs of all contexts of the user program or of the context designated by the CTX qualification; the names are output in full, not abbreviated (for further operands see AID Core Manual, section "System information"[1])
%LINK	Name of the last dynamically loaded segment (see %ON, event %LPOV)

```
%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 the chapter on “Complex memory references” in the AID Core Manual [1]):

– byte offset (•)
– indirect addressing (->)
– type modification (%X, %C, %E, %D, %F, %A)
– length modification (%L(...), %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 the pointer operator (->). This references the first machine instruction of the prolog. Without the pointer operator, statement names and source references can be used wherever hexadecimal numbers are also allowed.

Using the type modification, data may be output in another form (see the section on type modification in the AID Core Manual [1].).

Do not confuse the AID output types with the printf conversion specifiers:

AID output type		corresponding printf conversion spec.	
%C[l-mod]	char	%c	a single character
		%s	character string
%D[l-mod]	float	%f	float, double
%F[l-mod]	signed int	%d	signed int
%A[l-mod]	unsigned int	%u	unsigned int
%X[l-mod]	hexadecimal	%x, %X	hexadecimal

With the length modification you can define the output length yourself, e.g. if you wish to output only parts of a variable or display a variable using the length of another variable. With a type or length modification the implicit area limits of an of an address can be exceeded only if you have used %@(...)-> to switch to machine code level, where the area embraces the virtual memory occupied by the loaded program.

&
: is the address operator. You can use it to display the start address of a data item, a class object or a function.

  You can also display the relative address of a dynamic data member of a class, provided you observe the following:

  If the interrupt point is located outside the class containing the data member, you should enter the appropriate class qualification after the address operator, and then the name of the data item. However, if the interrupt point is located in a dynamic member function of the class, you will need to enter a base or area qualification (S, PROC or :: qualification) before the address operator so that AID can access the class from “outside”, so to speak.

  Note that in contrast to the address selector %@(...) (see page 159), the address operator is purely a “high-level” function and thus cannot be applied on complex memory references.

  For more details on the address operator, see also the section “The address operator & and the address selector %@(...)” on page 42.

sizeof()
: is the length operator. The length of a data item or class is displayed.
  To determine the length of a class, you may specify the name of the class itself or an object of the class as operands. You will receive the number of bytes occupied by the dynamic data members of the class and by the auxiliary variables generated by the compiler (if any).
  You may specify the name of a namespace here, but only in the path to a component of the namespace.
  Bit-field and register variables are not allowed.

  The length operator is described in detail in the section “Length operator sizeof() and length selector %L(...)” on page 47.

%@(...)
: The address selector (see the AID Core Manual, section "Address, type and length selector" [1]) can be used to output the start address of a data item, a class object, or a complex memory reference. You can specify a class name only in the path for the base class of an object of a derived class to display the start address of the dynamic data members of the base class.
  You may specify the name of a namespace here, but only in the path to a component of the namespace.
  The address selector cannot be applied to constants, including labels, source references and all functions.

%L(...)

The length selector (see the AID Core Manual, section "Complex memory references" [1]) can be used to have the length of a data item or a class displayed (see the section on “Complex memory references” in the AID Core Manual [1]). If you apply the length selector to a class or a class object, the result corresponds to that of sizeof() in C++, i.e. you receive the length of the dynamic data member and of the compiler-generated auxiliary variables, if any.
You can only specify the name of a namespace here in the path to a component of the namespace.

AID always outputs the length in bytes. For bit-fields, AID outputs the number of bytes covered by the bit-field.
Example: `%l(var1)`
The length of `var1` is output.

%L=(expression)

You can use the length function to calculate a value.
expression is formed from memory references and arithmetic operators (see the chapter on “Addressing in AID” in the AID Core Manual [1]).

AID uses the standard operators for its calculations and does not emulate operator overloading.

Example: `%l=(var1)`
If `var1` is of type int (type %F), the contents of `var1` are output. Otherwise, AID outputs an error message.

literal

All AID literals described in the chapter on “AID literals” in AID Core Manual [1] may be specified in a %DISPLAY command:

```
{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                          Fixed point number
[{?}]mantissaE[{?}]exponent       Floating-point number
```

If `%AID C=YES` is set, you may also specify a C string literal ("*x...x*"). See also page 36.

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 of *n* blank lines. 1 ≤ *n* ≤ 255. The default for *n* is 1.

medium-a-quantity

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

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

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

medium-a-quantity is described in detail in the chapter "Medium-a-quantity operand" in the AID Core Manua [1].

<u>T</u>	Terminal output
H	Hardcopy output (includes terminal output and cannot be combined with *T*)
Fn	File output
P	Output to SYSLST

<u>MAX</u>	Output with additional information
MIN	Output without additional information
XMAX	In the %DISPLAY command the operand value XMAX is not currently taken into account, as a result of which the behavior is identical to the default value MAX.
XFLAT	In the %DISPLAY command the operand value XFLAT is currently not taken into account, as a result of which the behavior is identical to the default value MAX.

Examples

1.
```
%df d1=dump.test1
%base e=d1
%display s=n'test1.c'.int_var,'dump-content'
```

   Here the contents of a dump are evaluated. Besides the contents of `int_var`, AID outputs a header with the name of the dump file.

```
** D1: DUMP.TEST1 ******************************************************
int_var     =            -53
dump-content
```

2.
```
%display %l=(s'13'-s'12')
```

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

```
        +52
```

3.
```
%base
%display scanf
```

   %BASE switches back to the AID default work area. AID first outputs two header lines with the TID and TSN and the source reference showing where the program run was interrupted. AID then outputs the address of the first instruction of the function `scanf` in hexadecimal form.

```
*** TID: 00010266 *** TSN: 069R ***************************************
SRC_REF: 6 SOURCE: EXAMP.C   PROC: main  *********************************
scanf           = 01001B94
```

4.
```
%display scanf->
```

   AID outputs 4 bytes of the machine code beginning at the address of the function `scanf`. The pointer operator switches to machine code level, causing AID to display an additional header.

```
CURRENT PC: 01000098    CSECT: EXAMP$O&@  ******************************
V'01001B94' = IC@PCON  + #'00000634'
01001B94 (00000634) 58F0FF70                              .0~.
```

5. `%display var.4`
   `%display var.(z)`

In the first case AID adds four bytes to the start address of variable var and from that point outputs four bytes in dump format. In the second case, too, AID performs a byte offset, as z is in parentheses. AID adds the contents of z to the address of var and, as above, outputs four bytes starting at the address thus calculated. In C/C++, though, `var.(z)` and `var.z` are synonymous, so there the parentheses around the second operand are redundant. Hence the expression could refer to structure component z of structure var. In AID, however, the last component in a structure qualification must never be placed in parentheses.

6.

```
/LOAD-PROG *M(MYLIB,EXAMP,...),TEST-OPT = AID
%  BLS0523 ELEMENT 'EXAMP', VERSION '@' FROM LIBRARY 'MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXAMP$', VERSION ' ' OF '1999-01-07
11:47:57' LOADED
/%r
%  IDA0N51 PROGRAM INTERRUPT AT LOCATION '010000BC (EXAMP$0&), (CDUMP),
EC=58'
%  IDA0N45 DUMP DESIRED? REPLY (Y=USER/AREA DUMP; Y,SYSTEM=SYSTEM DUMP;
N = NO)? N
%  EXC0077 PROGRAM STILL LOADED AND IN 'COMMAND-MODE'. PROGRAM RUN MAY
BE CONTINUED WITH /RESUME-PROGRAM
```

Your program has encountered an error. Now you want to know which statement caused the error. To find out, enter `%DISPLAY %HLLOC` for the address at which the program was interrupted by the error. This address is contained in the program counter (%PC). You can obtain further information with `%DISPLAY %LOC`.

```
/%display %hlloc(%pc->)
*** TID: 00010266 *** TSN: 069R ****************************************
CURRENT PC: 010000BC    CSECT: EXAMP$0&@   ****************************
V'010000BC' = CONTEXT  :  LOCAL#DEFAULT
              SMOD     :  EXAMP.C
              BLOCK    :  *root*
              PROC     :  main
              SRC-REF  :  11
/%display %loc(%pc->)
V'010000BC' = CONTEXT:LOCAL#DEFAULT
              LMOD :  %UNIT
              SMOD :  EXAMP.C
              OMOD :  EXAMP$0&@
              CSECT : EXAMP$0&@ (01000000) + 000000BC (/390)
```

7. %d abc_arr

   The array abc_arr contains 27 elements of type char and is defined as follows:
   `char abc_arr[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";`

   Since the %DISPLAY command did not specify a subscript, AID outputs the entire array:

```
abc_arr( 0: 26)
(  0) |A|  (  1) |B|  (  2) |C|  (  3) |D|  (  4) |E|  (  5) |F|  (  6) |G|
(  7) |H|  (  8) |I|  (  9) |J|  ( 10) |K|  ( 11) |L|  ( 12) |M|  ( 13) |N|
( 14) |O|  ( 15) |P|  ( 16) |Q|  ( 17) |R|  ( 18) |S|  ( 19) |T|  ( 20) |U|
( 21) |V|  ( 22) |W|  ( 23) |X|  ( 24) |Y|  ( 25) |Z|  ( 26) |.|
```

   A detailed example on working with arrays when using AID can be found on page 33

8. %d abc_arr[n]

   abc_arr is defined as described in example 7. n contains the value 4. The fifth element of the array is output:

```
abc_arr( 4)     =  E
```

9. The following code fragment shows the definition of a derived class B with base class A. The class contains two virtual functions: foo1(void) and foo2(void).

```
C++ program                                                        BCL1.C
=============================================================================
SRC
LIN
  1  class A
  2  {
  3    public:
  4    A() { printf ("A::A called\n"); }
  5    virtual void foo1() { printf( "A::foo1 called\n" ); }
  6    virtual void foo2() { printf( "A::foo2 called\n" ); }
  7  } a;
  8
  9  class B : public A
 10  {
 11    int i;
 12    public:
 13    B(int x = 1) : i(x) { printf ("B::B called\n"); }
 14    void foo1() { printf( "B::foo1 called\n" ); }
 15    void foo2() { printf( "B::foo2 called\n" ); }
 16  } b;
 ...
```

```
%in s'13'; %r
%d this, *this
```

The %INSERT followed by the %RESUME interrupts program execution in the constructor for class `B` at source reference 13. The %DISPLAY command shows the contents of the `this` pointer, i.e. the address of the associated object and, by dereferencing (`*this`), the contents of that object.

```
SRC_REF: 13 SOURCE: BCL1.C PROC: B::B(int)  ****************************
this            = 01001138

01          *
 02         A
  03        A()             = 01000000
  03        foo1()          = 010004C0
  03        foo2()          = 010005E0
 02         i               =          1
 02         B(int)          = 01000360
 02         foo1()          = 010004C0
 02         foo2()          = 010005E0
```

10. The C program OUTPUT.C outputs a number of simple unstructured data types which can be defined in C.

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27     PAGE:    1
                    SOURCENAME:LIB-ELEM(MYLIB,OUTPUT.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________________

  EXP  INC  FILE    SRC BLOCK
  LIN  LEV   NO     LIN  LEV

    1    0     0      1    0  #include <stdio.h>
 1746    0     0      2    0  int main(void)
 1747    0     0      3    0  {
 1748    0     0      4    1
 1749    0     0      5    1     short int1 = -32768;
 1750    0     0      6    1     int   int2 = 234;
 1751    0     0      7    1     long  int3 = -567;
 1752    0     0      8    1
 1753    0     0      9    1     unsigned short un1 = 65535;
 1754    0     0     10    1     unsigned int   un2 = 78900;
 1755    0     0     11    1     unsigned long  un3 = 90123;
 1756    0     0     12    1
 1757    0     0     13    1     signed long long sll   = -9223372036854775808;
 1758    0     0     14    1     unsigned long long ull = 18446744073709551615;
 1759    0     0     15    1
 1760    0     0     16    1     float       fl1 = 123.456;
 1761    0     0     17    1     double      fl2 = 567.89;
 1762    0     0     18    1     long double fl3 = 333.444;
 1763    0     0     19    1
 1764    0     0     20    1     char        char1 = 'A';
 1765    0     0     21    1     signed char char2 = -63;
 1766    0     0     22    1
 1767    0     0     23    1     char *chstr     = "Character string";
 1768    0     0     24    1     char chvek[17]  = "Character array";
 1769    0     0     25    1
 1770    0     0     26    1     char *c_out  = "Corresponding C output:";
 1771    0     0     27    1
 1772    0     0     28    1     printf ("%s\n",c_out);
 1773    0     0     29    1     printf ("int1 = %d\n", int1);
 1774    0     0     30    1     printf ("int2 = %d\n", int2);
 1775    0     0     31    1     printf ("int3 = %d\n", int3);
```

```
1776    0     0     32    1
1777    0     0     33    1     printf ("%s\n",c_out);
1778    0     0     34    1     printf ("un1 = %u\n", un1);
1779    0     0     35    1     printf ("un2 = %u\n", un2);
1780    0     0     36    1     printf ("un3 = %u\n", un3);
1781    0     0     37    1
1782    0     0     38    1     printf ("%s\n",c_aus);
1783    0     0     39    1     printf ("sll = %lld\n", sll);
1784    0     0     40    1     printf ("ull = %llu\n", ull);
1785    0     0     41    1
1786    0     0     42    1     printf ("%s\n",c_out);
1787    0     0     43    1     printf ("fl1 = %f\n", fl1);
1788    0     0     44    1     printf ("fl2 = %f\n", fl2);
1789    0     0     45    1     printf ("fl3 = %f\n", fl3);
1790    0     0     46    1
1791    0     0     47    1     printf ("%s\n",c_out);
1792    0     0     48    1     printf ("char1 as character = %c and as value = %d\n",
1793    0     0     49    1                                    char1, char1);
1794    0     0     50    1     printf ("char2 as character = %c and as value = %d\n",
1795    0     0     51    1                                    char2, char2);
1796    0     0     52    1     printf ("%s\n",c_aus);
1797    0     0     53    1     printf ("*chstr = %s\n", chstr);
1798    0     0     54    1     printf ("chvek = %s\n", chvek);
1799    0     0     55    1
1800    0     0     56    1     return 0;
1801    0     0     57    1  }
```

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH.
        ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-TEST-PROPERTIES TEST-SUPPORT=YES
//MODIFY-SOURCE-PROPERTIES LANGUAGE=*C(*ANSI)
...
//MODIFY-BIND-PROPERTIES ...  RUNTIME-LANGUAGE = *C,  TEST-SUPPORT = *YES
...
//END
...
%  BLS0524 LLM '$LIB-ELEM$MYLIB$OUTPUT', VERSION ' ' OF '2015-03-05 11:15:23' LOADED
/%aid c=yes
/%in s'28' <%d 'AID-AUSGABE int1, int2 und int3', int1, int2, int3>
/%in s'33' <%d 'AID-AUSGABE un1, un2 und un3', un1, un2, un3>
/%in s'38' <%d 'AID-AUSGABE sll und ull', sll, ull>
/%in s'42' <%d 'AID-AUSGABE fl1, fl2 und fl3', fl1, fl2, fl3>
/%in s'47' <%d 'AID-AUSGABE char2 als Zeichen und als Wert', char2%c, char2>
/%in s'47' <%d 'AID-AUSGABE char1 als Zeichen und als Wert', char1, char1%a>
/%in s'52' <%d 'AID-AUSGABE *chstr und chvek', chstr->%cl16, chvek>
/%resume
```

The program OUTPUT.C was compiled without errors and linked with LSD records and loaded. %AID C=YES was set so that AID can output the char array chvek as a C string. Case sensitivity, i.e. a distinction between uppercase and lowercase, was thus enabled at the same time.
Test points were set using the %INSERT commands, causing each %DISPLAY command to be followed by the corresponding C output. To make the output easier to read, the text lines "AID-OUTPUT" and "CORRESPONDING C OUTPUT" are printed in bold.

```
AID-OUTPUT int1, int2 and int3
*** TID: 00010266 *** TSN: 069R ********************************************************
SRC_REF: 28 SOURCE: OUTPUT.C  PROC: main  ***********************************************
int1             =      -32768
int2             =         234
int3             =        -567
Corresponding C output
int1 = -32768
int2 = 234
int3 = -567

AID-OUTPUT un1, un2 and un3
SRC_REF: 33 SOURCE: OUTPUT.C  PROC: main  ***********************************************
un1              =    65535
un2              =      78900
un3              =      90123
Corresponding C output:
un1 = 65535
un2 = 78900
un3 = 90123

AID-OUTPUT sll and ull
SRC_REF: 38 SOURCE: OUTPUT.C  PROC: main  ***********************************************
sll              =   -9223372036854775808
ull              =   18446744073709551615
Corresponding C output:
sll = -9223372036854775808
ull = 18446744073709551615
```

All variables of type signed and unsigned int were displayed.
The use of printf to output data of type long long is supported in CRTE.

```
AID-OUTPUT fl1, fl2 und fl3
SRC_REF: 42 SOURCE: OUTPUT.C  PROC: main  ***********************************************
fl1              = +.1234559 E+003
fl2              = +.5678899999999999 E+003
fl3              = +.3334439999999999 E+003
Corresponding C output:
fl1 = 123.455994
fl2 = 567.890000
fl3 = 333.444000
```

Data of type float, double and long double was output. AID always outputs floating point variables in exponential notation. For single-precision variables, AID outputs 7 significant digits; for double and long double data types, 16 digits are output.

```
AID-OUTPUT char1 as character and as value
SRC_REF: 40 SOURCE: OUTPUT.C  PROC: main  ****************************************
char1           = |A|
CURRENT PC: 0100026A    CSECT: OUTPUT$O&@  ****************************************
V'010227F8' = char1    + #'00000000'
010227F8 (00000000) 193
AID-OUTPUT char2 as character and as value
V'010227F9' = char2    + #'00000000'
010227F9 (00000000) A
SRC_REF: 40 SOURCE: OUTPUT.C  PROC: main  ****************************************
char2           =  -63
Corresponding C output:
char1 as character = A and as value = 193
char2 as character = A and as value = -63
```

AID handles data type `char` differently from `signed char`. AID outputs the character value A for variable `char1`. You can only display the corresponding decimal value via the explicit type modification %A. AID outputs the decimal value -63 for `signed char` variable `char2` without type modification. You have to use the explicit type modification %C to display `char2` as a character.

```
AID-OUTPUT *chstr and chvek |
CURRENT PC: 0100035C    CSECT: OUTPUT$O&@  ****************************************
V'01001188 = OUTPUT$O&#                          + #'00000118'
001188 (00000188) Character string
SRC_REF: 52 SOURCE: OUTPUT.C  PROC: main  ****************************************
charr           = "Character array"
Corresponding C output:
*chstr = Character string
chvek = Character array
```

This output of character strings - addressed via a pointer in the first case and stored as an array of characters in the second - concludes the comparison of the treatment of the individual data types by AID and C.
The string referenced by the pointer `chstr` can be output with AID only via a following pointer operator and with a type and length modification. The array `chvek`, by contrast, is treated as a C string literal, since `%AID C=YES` was set.

11. Character output with any coded character set (CCS)

```
void main(void)
{
    char unsigned data[256];
    char ALPHA[28];
    int i;

    for (i=1; i<=255; i++)
                    data[i-1] = i;
    data[255] = 0x00;
    strncpy(ALPHA, data+64, 26);
    ALPHA[26]= 0x00;
    STOP: ;
}
```

Specify the character set for the terminal:

```
/mod-term-opt coded-character-set=edf041
```

After loading the program:

```
%AID C=YES
%INSERT STOP
%RESUME
```

Data output with the default interpretation `*USRDEF` (=EDF03IRV):

```
/%D data.32 %XL80
V'0100112C' = data     + #'00000020'
0100112C (00000020) 21222324 25262728 292A2B2C 2D2E2F30    ................
0100113C (00000030) 31323334 35363738 393A3B3C 3D3E3F40    ................
0100114C (00000040) 41424344 45464748 494A4B4C 4D4E4F50    .........`.<(+|&
0100115C (00000050) 51525354 55565758 595A5B5C 5D5E5F60    .........!$*);.-
0100116C (00000060) 61626364 65666768 696A6B6C 6D6E6F70    /........^,%_>?.
```

Change the current character set to ISO88591:

```
%AID CCS=ISO88591
```

Data output with the ISO88591 interpretation as specified:

```
/%D data.32 %XL80
V'0100112C' = data     + #'00000020'
0100112C (00000020) 21222324 25262728 292A2B2C 2D2E2F30    !"#$%&'()*+,-./0
0100113C (00000030) 31323334 35363738 393A3B3C 3D3E3F40    123456789:;<=>?@
0100114C (00000040) 41424344 45464748 494A4B4C 4D4E4F50    ABCDEFGHIJKLMNOP
0100115C (00000050) 51525354 55565758 595A5B5C 5D5E5F60    QRSTUVWXYZ[\]^_`
0100116C (00000060) 61626364 65666768 696A6B6C 6D6E6F70    abcdefghijklmnop
```

Output of the char variable ALPHA (ISO88591 interpretation):

```
/%D ALPHA
ALPHA           =        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

%DUMPFILE

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

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

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

Command	Operand
{%DUMPFILE | %DF}	[link [=file]]

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).

If a library to dynamically load LSD information was assigned with the command %SYMLIB for a file that is closed with %DUMPFILE, the library for the associated link name *Dn* is released.

%DUMPFILE may only be specified as an individual command, i.e. it may not be part of a command sequence and may not be included in a subcommand.

%DUMPFILE does not alter the program state.

link

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

file

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

Examples

1. `%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 file are closed.

%FIND

With %FIND you can search for a literal in the data section 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 or hexadecimal literal to be searched for.

– With *find-area* you specify which data or which section of the executable part of the program AID is to search for *search-criterion*. 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.

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

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

If a %FIND command is issued with a separate *search-criterion* and without any further operands, AID inserts default values for *find-area* and *alignment*, i.e. does not transfer operands from a preceding %FIND command in this case.

In the event of a hit, a maximum of 12 bytes from the hit to the end of *find-area* are output to the terminal (SYSOUT) in DUMP output format (hexadecimal and character notation). In addition to the hit itself, its address and (where possible) the name of the CSECT in which the hit was found, and the relative address of the hit with respect to the beginning of the CSECT, are output. For searches in global data, the relative address at the start of the data module is output. In all other cases the absolute address is output.

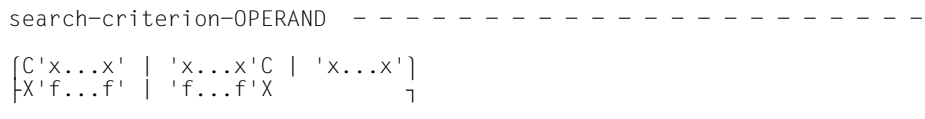
In the event of a hit, the hit address is stored in AID register %0G and the continuation address (hit address + search string length) in AID register %1G. With the *ALL* specification, the address of the last hit is stored in %OG and the continuation address of the last hit is stored in %1G. If the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

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

The %FIND command does not alter the program state.

search-criterion

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



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

Character literal

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

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

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

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

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

```
%SHOW %CCSN
```

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

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

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

```
%SHOW %AID
```

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

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

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

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

Hexadecimal literal

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

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

find-area

Defines the memory area to be searched for *search-criterion*. *find-area* can be a class object, a data item or an area in the executable part of the loaded program or of a dump file.

If no *find-area* has been specified, AID inserts the default value %CLASS6 (see the section on “Memory classes” in the AID Core Manual [1]), i.e. the class 6 memory for the currently set base qualification is searched (see %BASE). The default value for *find-area* cannot be used for C/C++ programs as these programs are loaded into the upper address space of memory. You therefore have to specify an explicit value for *find-area* with the first %FIND command (e.g. %CLASS6ABOVE).

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

                 ⎧namespace[::...]                                              ⎫
                 ⎪                                                              ⎪
                 ⎪*this                                                         ⎪
                 ⎪                                                              ⎪
                 ⎪⎧this->    ⎫                                                  ⎪
                 ⎪⎨          ⎬[class[:: ...]]                                   ⎪
                 ⎪⎩object[•] ⎭                                                  ⎪
                 ⎪                                                              ⎪
                 ⎪ ⎧[namespace::[...]]class::⎫                                  ⎪
IN [•][qua•]     ⎨┐⎨this->                   ⎬[class::[...]]]dataname           ⎬
                 ⎪ ⎩object•                  ⎭                                  ⎪
                 ⎪                                                              ⎪
                 ⎪⎧[namespace::[...]][class::[...]]function ⎫                   ⎪
                 ⎪⎨L'label'                                 ⎬->                 ⎪
                 ⎪⎩S'[f-]n[:a]'                             ⎭                   ⎪
                 ⎪                                                              ⎪
                 ⎪keyword                                                       ⎪
                 ⎪                                                              ⎪
                 ⎩compl-memref                                                  ⎭

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

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

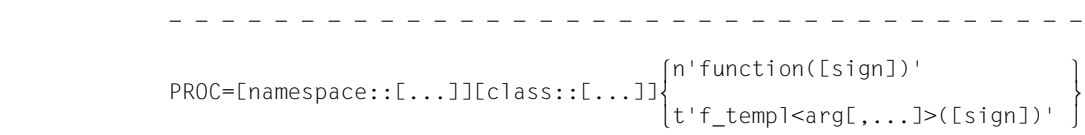
qua One or more qualifications may be specified here if *find-area* cannot be reached from the current interrupt point by other means or to address a data name that is locally hidden at the interrupt point by an identically named definition.
It is sufficient to specify only the qualifications needed for a unique address.

E={VM | Dn}
Specified only if the current base qualification (see %BASE) is not to apply to *find-area*.

S=srcname
Specified only if *find-area* is not in the current translation unit (see the chapter “Addressing in C and C++ programs” on page 21).

:: Use the two prepended colons to address a global data item that is locally hidden at the interrupt point by a definition of the same name. You must also place two colons before the name of a global data item or a function if either the data or the function is not in the call hierarchy or if its definition only occurs after the interrupt point. In contrast to the other qualifications, no period must be entered between the two colons and the operands which follow.

PROC=function
Only specified if you want to access a data name which is defined in the current function, but is hidden at the interrupt point by a definition with the same name. You also specify a PROC qualification when you want to address a label or a data name declared as static which is assigned in a function outside the current call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21). If the start address of *find-area* is designated by a source reference that is located in a function template instance or assigned to a function defined in a class template instance, (see the section “Templates” on page 94), you also have to prepend the appropriate PROC qualification if ambiguity occurs.
In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.
Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the namespace or class qualification is prepended to the function name. The `void` signature may not be written, you enter just the two parentheses in this case as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated for space reasons):



The `main` and `_ _STI_ _` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications. Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK='[f-]n[:b]'

Specified only when you want to address a data name which is assigned to a block within the current call hierarchy and is hidden at the interrupt point by a definition with the same name, or when you want to want to access a data name declared as static and assigned to a block outside the current call hierarchy (see the chapter "Addressing in C and C++ programs" on page 21).

You must also specify a BLK qualification if you want to designate a function from a local class, which is defined in the specified block, in a subsequent PROC qualification (see above, PROC=*function*).

namespace

Name of a namespace declared in the source program.

You only specify the name of a namespace if the required namespace component is not visible at the interrupt point. You use this to describe the addressing path to classes, data or functions defined in the namespace (see the section "Namespaces" on page 85).

Only the E or S qualification or the two colons (`::`) for the global namespace are allowed before the namespace qualification.

{ class | this-> | object }

Name of a class, the `this` pointer or the name of a class object as declared in the source program.
You specify class names, the `this` pointer with the appended pointer operator, and the names of class objects in order to describe the address path to data members assigned to classes (see the section “Classes” on page 63).

If the current interrupt point is located in a dynamic member function, you can address the class data according to the scope rules known from C++.

If an object is in the current call hierarchy, you can access the dynamic data of that object independent of the interrupt point by means of the object name followed by a period.

Static data members can only be addressed individually. They can be accessed from any part of the program via the associated class names followed by the two colons. In the case of nested classes, the path to the data item includes all the class names from the outermost to the innermost level, separated by two colons each. The outermost class name requires qualification appropriate to the scope. If the program is interrupted within a member function of the class, the class scope rules apply for accessing static data members, i.e. if the data is not hidden by a definition with the same name, it can be accessed without qualification.

If the class is a class template instance, you have to use the following notation: `t'`*k_template*`<`*arg*`[,...]>'`. If only one instance of the template exists, only `t'`*k_template*`'` is required.

If *find-area* ends with an object name, the dynamic data members and, if present, the compiler-generated auxiliary variables and the address of the virtual function table are referenced, regardless of the current interrupt point. In the case of derived classes, *find-area* also includes the base classes. The same area can be referenced with `*this`, if the program is interrupted in a dynamic member function of the class. To designate a base class in a derived class, you specify the name of the desired base class in the path starting from the object name or from `this->`.
If the interrupt point is not in the scope of *object*, it must be appropriately qualified. Only a base qualification is meaningful before `*this`.
If the *find-area* operand ends at `this->`, the first 4 bytes as of the start address of the current object are referenced.

dataname

This is the name of a data item declared in the source program. *dataname* is specified as in the source program.

You can reference data as in C/C++, but with the following exceptions:
An array name without a subscript addresses all array elements.
Array elements can only be referenced by means of subscripts, not pointers. For more information on working with arrays, see also the section “Subscript notation” on page 30.

You can specify *dataname* as follows. You can also combine these formats (see the section “Data names” on page 29).

Subscript notation: *dataname* `[`*subscript*`]` `{...}`
Pointer notation: *dataname1* `->` *dataname2*
Structure qualification: *superordinate dataname*`.` `{...}` *dataname*
Dereferencing: `[(]*{...}` *dataname*`[)]`

```
{function[%al4]  }
{L'label'        }->
{S'[f-]n[:a]'    }
```

Designates 4 bytes of machine code beginning at the address stored in one of the address constants. If another number of bytes is to be searched, you must specify an appropriate length modification.

function

This is the name of a function, as declared in the source program, or the name of a library function. It references the first instruction of the function prolog that is generated by the compiler (see PROC=*function* on page 176 and the chapter “Addressing in C and C++ programs” on page 21).

The following syntax is used to address virtual functions:
`p->n'function(signature)'`

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, the scope must be qualified accordingly. If the interrupt point is located in the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p*. (see the description of `this` on page 64 and section “Virtual functions” on page 73).

If you want *find-area* to be in a function addressed via a pointer to member, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

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

[qua•]object•*[object•][class::][...]pointer-to-function-member

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

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

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

[qua•]pointer->*[object•][class::][...]pointer-to-function-member

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

The class object is designated by the operand on the left of the dereferencing operator `•*` or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.
The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to member cannot be reached from the interrupt point by some other means.

Note that you cannot directly append the pointer operator to one of the syntaxes above. Instead, you would have to first specify a type modification, i.e. %al4, to switch to machine code level. This results in the following syntax:

`%DISPLAY` *dereferenced-pointer-to-function-member* `%al4->`

More details on working with a pointer to function member can be found on page 79.

L’label’
: Designates the address of the first executable statement after a label.
label is the name of a label declared in the source program.

S’[f-]n[:a]’
: Source reference which designates the address of an executable statement. It comprises the line number (*n*) and possibly the FILE number (*f*) plus the relative statement number within the line (*a*). If the source reference is located in a function which was created from a function template via instantiation or the function containing the source reference is defined in a class template instance, you have to prepend the appropriate PROC qualification to the source reference if ambiguity occurs.

keyword

Allows you to define a memory area by specifying one of the following keywords (see the chapter on “Keywords” in the AID Core Manual [1]).

When you specify one of the keywords for class 5 memory, the unprivileged areas used by your program in class-5 memory are searched.

Only a base qualification may be specified before *keyword*.

```
%CLASS6               class 6 memory below the 16MB boundary
%CLASS6BELOW          class 6 memory below the 16MB boundary
%CLASS6ABOVE          class 6 memory above the 16MB boundary

%CLASS5               class 5 memory below the 16MB boundary
%CLASS5BELOW          class 5 memory below the 16MB boundary
%CLASS5ABOVE          class 5 memory above the 16MB boundary

%n                    general purpose registers, 0 ≤ n ≤ 15
%nD|E                 floating point registers, n = 0,2,4,6
%nQ                   floating point registers, n = 0,4
%nG                   AID general purpose registers, 0 ≤ n ≤ 15
%nGD                  AID floating point registers, n = 0,2,4,6
%MR                   All 16 general purpose registers in tabular form
%PC                   Instruction counter (program counter)
```

compl-memref

The following operations may occur in *compl-memref* (see also the section on “Complex memory references” in the AID Core Manual [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 a source reference, the pointer operator (->) must come next. Without the pointer operator, statement names and source references can be used wherever hexadecimal numbers are also allowed. Labels in *compl-memref* must always be placed within `L'...'`.
compl-memref designates an area of 4 bytes, starting with the calculated address. If a different number of bytes is to be searched, *compl-memref* must terminate with the appropriate length modification. When modifying the length of data items, you must pay attention to area limits or switch to machine code level using *%@(dataname)*->. A maximum of 65 535 bytes can be declared with a length modification.

alignment

Restricts the search for *search-criterion* to some aligned addresses.

```
alignment-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

ALIGN [=] { 1 }
          { 2 }
          { 4 }
          { 8 }

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

search-criterion is searched for only at the:

1 byte boundary (default)

2 halfword boundary

4 word boundary

8 doubleword boundary

Examples

1. `%find x'f0' in arr1`

   The hexadecimal literal X'F0' is searched for in array `arr1`. Hits are output to SYSOUT.

2. `%f x'd2' in s'12'->%l=(s'13'-s'12') align=2`

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

3. `%f`

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

%HELP

You use %HELP to 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.

– By means of the *info-target* operand you specify the command on which you need further information.

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

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

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

The format for the message key for AID messages is AID0*n*; the format for AIDSYS messages is IDA0*n*. Both can be requested with `/HELP-MSG-INFORMATION`.

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

The %HELP command does not alter the program state.

info-target

Designates a command about which information is to be output.
If the *info-target* operand is omitted, the command outputs an overview of the AID commands with a brief description of each command.

AID responds to a %HELP command containing an invalid *info-target* operand by issuing an error message. This is followed by the same overview described above. This overview can also be requested by specifying %?, %H? or %H %?.

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

{ %AID | %AINT | %ALIAS | %BASE | %CONT[INUE] | %C[ONTROL]   }
| %DISASSEMBLE | %DA | %D[ISPLAY] | %DUMPFILE | %DF          |
| %F[IND] | %H[ELP] | %IN[SERT] | %JUMP | %M[OVE]          |
{ %ON | %OUT | %OUTFILE | %Q[UALIFY]                       }
| %REM[OVE] | %R[ESUME] | %SD[UMP] | %T[RACE]              |
{ %SH[OW] | %STOP | %SYMLIB | %TITLE                       }

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

The AID command names may be abbreviated as shown above.

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=MAX. The specification {MIN | MAX | XMAX | XFLAT} has no effect with %HELP, but the syntax requires one of these two specifications.

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

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

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

For more details on *medium-a-quantity,* see the chapter “Medium-a-quantity operand” in the AID Core Manual [1].

T Terminal output

H Hardcopy output (includes terminal output and cannot be combined with *T*)

Fn File output

P Output to SYSLST

%INSERT

You use %INSERT to specify a test point and define a subcommand. Once the program sequence reaches the test point, AID processes the associated subcommand.

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

– By means of the *subcmd* operand you 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.

Command	Operand
%IN[SERT]	test-point [<subcmd>]

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

1. When the end of the program is reached.
2. If the *test-point* is deleted with %REMOVE.

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 loaded program, therefore the base qualification E=VM must have been set (see %BASE) or must be specified explicitly.

You can only set a test point on the `throw` and `catch` statements of a C++ program if the program has been initialized, i.e. after it has run up to the first executable statement. The program is interrupted immediately after initialization if you input the following command after loading:

```
/%trace 1 in s=srcname
```

When you debug a program containing classes with virtual functions or constructors, the program halts in a compiler-generated function called `_ _STI_ _` instead of in `main`. This function calls the constructors and generates the virtual function tables. AID outputs the name `_ _STI_ _` in the STOP message.

%INSERT does not alter the program state.

test-point

Must be the address of an executable machine instruction that was either generated for a C/C++ statement or, if `throw`/`catch` statements are to be monitored, is located in a runtime system routine which is called with each `throw` or `catch` 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 or the subcommand containing %INSERT is processed. Since entering *test-point* modifies the program code, 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 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

 ⎧            ⎧function[->]    ⎫ ⎫
 ⎪            ⎪L'label'        ⎪ ⎪
 ⎪[•][qua•]   ⎨                ⎬ ⎪
 ⎪            ⎪S'[f-]n[:a]'    ⎪ ⎪
 ⎨            ⎩compl-memref    ⎭ ⎬
 ⎪                               ⎪
 ⎪⎧%EXCEPTION⎫ ⎧throw⎫           ⎪
 ⎪⎨          ⎬(⎨     ⎬)          ⎪
 ⎩⎩%EH       ⎭ ⎩catch⎭           ⎭
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

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

qua	One or more qualifications may be specified here to address a function, label or source reference that cannot be reached from the current interrupt point by other means.
It is sufficient to specify only the qualifications needed for a unique address.

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

S=srcname
Specified only if you wish to address a statement name or source reference that is not in the current translation unit (see the chapter “Addressing in C and C++ programs” on page 21).

BLK='[f-]n[:b]'

You must specify a BLK qualification if you want to reference a function from a local class in a subsequent PROC qualification and the local class is defined in the specified block (see below PROC=*function*).
The block name is constructed from the line number (*n*), a possible FILE number (f) and relative block number (*b*).

PROC=function

Specified only if you want to reference a label from a function other than the current one (see the chapter "Addressing in C and C++ programs" on page 21) or if *test-point* is to be set to a source reference that is ambiguous in the translation unit because it is located in a function template instance or assigned to a function which is defined in a class template instance (see the section "Templates" on page 94).
In the case of functions from C programs, *function* is the function name declared in the source program, but without parentheses or the signature.
Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the function name is prepended with the namespace or class qualification. The `void` signature may not be written, you enter just the two parentheses in this case as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated for space reasons):

```
PROC=[namespace::[...]][class::[...]]{n'function([sign])'          }
                                     {t'f_templ<arg[,...]>([sign])'}
```

The `main` and `__STI__` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications. Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

function[->]

Places *test-point* at the first executable statement in a function or at the first instruction in a library function.
function is the name of a function as declared in the source program or the name of a library function (see PROC=*function* above and the chapter “C++-specific addressing” on page 57).

Syntax for virtual functions:

```
p->n'function([signature])'
```

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the first executable statement of the current function by using the `this` pointer instead of *p*.

If you want to set a test point at the first executable statement of a function addressed via a pointer to member, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

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

[qua.]object.*[object.][class::][...]pointer-to-function-member

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

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

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

[qua.]pointer->*[object.][class::][...]pointer-to-function-member

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

The class object is designated by the operand on the left of the dereferencing operator `.*` or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.
The name of the pointer to function member must be entered on the right of the dereferencing operator. Note, however, that this may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to function member cannot be reached from the interrupt point by some other means. More details on working with a pointer to function member can be found on page 79.

If you use *test-point* to designate a library function, you must terminate *function* with the pointer operator. In this case, the test point is set on the first command of the function prolog. Note, however, that if you %SDUMP %NEST to view the call hierarchy from this point, you may not see the direct caller of the library function, since AID cannot determine the full call hierarchy until the prolog is traversed and the first executable statement of the function is reached. If an LSD is available for the function, you can use %TRACE 1 %STMT to position onto the first executable statement of the function.

Note that AID cannot usually associate the prolog address with the corresponding function.. The same effect must be noted if function addresses listed by AID for `%DISPLAY{`*namespace*`|`*object*`|`*class*`}` or in the %SDUMP output are used in an %INSERT for the sake of simplicity in the case of functions from C++ programs, which usually have very long names. Consequently if you use `%SHOW %INSERT` later in the test run to obtain information on all the test points that have been set thus far, AID displays the name of an earlier function for the prolog address if virtual functions are involved.

L'label'

Places *test-point* at the first executable statement after a label.
label is the name of a label declared in the source program. In this command you can also specify *label* without `L'...'`, since there can be no confusion with a data name.

S'[f-]n[:a]'

Is a source reference. It places *test-point* at an executable statement. The source reference is constructed from the line number (*n*) and, if present, the FILE number (*f*) and the relative statement number within the line (*a*).
If the source reference is located in a function which was created from a function template via instantiation or the function containing the source reference is defined in a class template instance, you have to prepend the appropriate PROC qualification to the source reference if ambiguity occurs.

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 the section on "Complex memory references" in the AID Core Manual [1]):

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

If a *compl-memref* begins with a statement name or a source reference, the pointer operator must come next. Labels in *compl-memref* must always be placed within L'...'. Note, however, that the pointer takes you out of the symbolic level. If you put a pointer name after a function name, what you reference is not the first executable statement in the function but the first instruction in the prolog generated for the function by the compiler.
Statement names and source references can be used without the pointer operator wherever hexadecimal numbers are allowed.

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: `%3g.2 %al2 ->`
The last two bytes from AID register %3G are used as the address.

{%EXCEPTION | %EH} ({throw | catch})

Can be used to monitor the `throw` or `catch` statements of a C++ program. To do this, a test point is set in a runtime system routine and called each time a `throw` or `catch` statement is executed. The result of this is that the program is not interrupted at the associated statement in the user program when a `throw` or `catch` statement is executed, but rather before the first instruction in this runtime routine. If you want to examine the local conditions which triggered the exception handling from this point, you must note that with an interruption in the runtime system, full qualification is required to access data and statements in your program.
To determine which statement in your program triggered the interrupt, you have to proceed differently for `throw` and `catch` statements:

– `throw` (triggers exception handling):

  Output the call hierarchy at the interrupt point with `%SDUMP %NEST`. You can write the `%SDUMP %NEST` command directly in the subcommand of the %INSERT and AID then automatically outputs the call hierarchy with each `throw`.

– `catch` (processes the exception):

  AID provides the prolog address of the `catch` handler of your program in register 15. The following command sequence
  `%INSERT %15-> <%TRACE 1 %STMT>; %RESUME`
  executes the program up to the first statement of the `catch` handler, the statement is logged and the program is halted.

To input `%INSERT %EH({throw|catch})`, you must enable uppercase/lowercase discrimination (`%AID LOW={ON|ALL}` command). Otherwise, AID reports a syntax error.

Example 5 on page 193 illustrates the use of `%INSERT %EH(...)`.

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 chapter “Subcommand” of the AID Core Manual [1].

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

                                    ⎧AID command    ⎫
<[subcmdname:] [(condition):] [     ⎨               ⎬ {;...}]>
                                    ⎩BS2000 command ⎭

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

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

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

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

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

Address operands in subcommands are not automatically supplemented on input with the qualifications that correspond to the current interrupt point. When *test-point* is reached in the subsequent debugging run and AID interrupts the program to process *subcmd*, only the data and functions that are visible at the test point can be addressed without qualification with AID commands from *subcmd* .

Examples

1. `%in s'48'`

   test-point is specified with a source reference and is set to the memory location of the instruction code generated for the first statement in line 48.

2. `%in facul <%display %.,n>`

   test-point is designated by a function name. Since `facul` is a C program, the function name is specified without the signature or parentheses. Whenever the program run arrives at the first statement in the function `facul`, the %DISPLAY command from *subcmd* is executed.

3. ```
   %in n'A::out1(int)' <%display var1, 'function A::out1'>
   %in n'A::out2(int)' <%display 'INSERT1', var1; %in n'A::out3(int)' -
       <%d 'INSERT2',i,j,k, i_arr[i]; %in s'172' <%d 'INSERT3' ,i,j; -
       %remove n'A::out3(int)'>>>
   ```

   With the first %INSERT command, the first statement of the member function `A::out1(int)` is set as the *test-point*. If, after the end of command input, the program execution reaches this address, the subcommand is executed. It consists of a %DISPLAY command (for data name `var1`) and the literal `'function A::out1'`. The program is then continued.

   The second %INSERT command declares *test-point* `A::n'out2(int)'`. 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 the first executable statement in member function `A::out2(int)`, the corresponding *subcmd* is executed, i.e. the %DISPLAY command for the literal `'INSERT1'` and the data name `var1` is executed and the *test-point* `A::n'out3(int)'` is set.
   The *subcmd* for *test-point* `A::n'out3(int)'` is still inactive. Thus, in the program to be debugged, the following three *test-points* have been set at this stage in the program run: the first executable statement in each of the member functions `A::out1(int)`, `A::out2(int)` and `A::out3(int)`.

   Since the *subcmd* for *test-point* `A::n'out2(int)'` does not contain a %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 first statement in `A::out3(int)`, then `%D 'INSERT2', i, j, k, i_arr[i]` is executed. Furthermore, *subcmd* contains a further %INSERT command, whose *test-point* this time is specified with source reference S'172'.
   ```

If the statement in line 172 is reached during further program execution, AID executes the %DISPLAY command for the literal 'INSERT3' and the contents of variables `i` and `j`. By means of the second command in this *subcmd*, the `%REMOVE A::n'out3(int)'` command, *test-point* `A::n'out3(int)'` is deleted. This is necessary, for instance, if a *test-point* is located in a loop and this would lead to an undesired chaining of nested subcommands. Without the %REMOVE command, the following *subcmd* would be created for *test-point* S'172' during the second pass of `A::n'out3(int)'`:
`<%d 'insert3',i,j; %d 'insert3',i,j>`

4.
```
%out %display p=max
%in s'73' <%d 'i GE 10',i,c_str[i],k,num[i][k]>
%in s'73' <(i lt 10): %d 'i LT 10',i,c_str[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'73':

```
<(i lt 10): %d 'I LT 10',i,c_str[i]; %cont; %d 'i GE 10',i,c_str[i],
-k,num[i][k]>
```

Every time program execution reaches the statement in line 73, 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 `c_str[i]` to SYSLST and, as a result of %CONTINUE, continues the program (with tracing, if the subcommand interrupted a %TRACE).
If the value of i is ≥ 10, AID writes the comment `'i GE 10'` and, in addition to `i` and `c_str[i]`, also writes the values of `k` and the array element `num[i][k]` to SYSLST and likewise continues the program. In this case, too, any active %TRACE is continued.

5. *Exception handling*

The example program for the following trace log is on page 137.

```
/LOAD-PROG   *MOD(LIB.23A,EXMEM,RUN-MODE=ADVANCED,PROGRAM-MODE=ANY),
TEST-OPTIONS=AID
%  BLS0523 ELEMENT 'EXMEM', VERSION '@' FROM LIBRARY '$TEST.MYLIB' IN
PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXMEM$', VERSION ' ' OF '1999-01-07
10:27:02' LOADED
/%aid low
/%t 1 in s=n'exmem.c'
10                        EXT.PROC START   , BLOCK START,  , BLOCK START,
                          ASSIGN
STOPPED AT SRC_REF: 10, SOURCE: EXMEM.C , BLK: 8 , END OF TRACE
/%in %eh(throw) <%sd %nest; %stop>
/%in %eh(catch) <sub1: %in %15-> <%t 1 %stmt>; %r>
```

After the program is loaded the %AID command is called first to enable uppercase/lowercase discrimination and the first executable statement is found with the %TRACE. The two %INSERTs set test points for the `throw` and `catch` case. The call hierarchy is requested at the `throw` test point with `%SD %NEST`, to indicate the point in the program where the exception handling was triggered. The `sub1` subcommand at the `catch` test point causes the program to be executed up to the first statement of the `catch` handler in the user program after reaching the test point in the runtime routine `AIDITO@`.

```
/%r
SRC_REF:  1354  SOURCE: AIDITO@  PROC: AIDITO@  ************************
ABSOLUT: V'1029282'   SOURCE: THROW&@  PROC: unwind_stack  *************
ABSOLUT: V'102CDCA'   SOURCE: THROW&@  PROC: __throw  ******************
ABSOLUT: V'1038DE6'   SOURCE: NEW&@  PROC: operator new  ***************
ABSOLUT: V'1025BFE'   SOURCE: ARRAY_NEW&@  PROC: operator new[]  *******
SRC_REF: 20  SOURCE: EXMEM.C  BLK : 18  ********************************
SRC_REF: 36  SOURCE: EXMEM.C  PROC: main  ******************************
ABSOLUT: V'7873C3FE'  SOURCE: IC@RT20A  PROC: IC@RT20A  ****************
ABSOLUT: V'1009970'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ****************
STOPPED AT LABEL: CPPTHROW , SRC_REF: 1354, SOURCE: AIDITO@ ,
PROC: AIDITO@
```

The program is started with %RESUME and runs until the exception handling is triggered. From the call hierarchy you can see that the statement in program line 20 (`q = new char[0x10000000];`) could not be executed, thus triggering the exception handling.

```
/%resume
 24                        , BLOCK START, CALL
STOPPED AT SRC_REF: 24, SOURCE: EXMEM.C , BLK: 22 , END OF TRACE
/%sh %in
> CTX: LOCAL#DEFAULT  SRC-REF: 20  SOURCE: EXMEM.C  PROC: main
  ( V'0100504C' )
> CTX: LOCAL#DEFAULT  SRC-REF: 1354  SOURCE: AIDITO@  PROC: AIDITO@
  LABEL: CPPTHROW
> CTX: LOCAL#DEFAULT  SRC-REF: 1363  SOURCE: AIDITO@  PROC: AIDITO@
  LABEL: CPPCATCH
/%rem V'0100504C'
```

Program execution is continued up to the interruption in the `catch` handler, with %RESUME. To ensure that the %INSERT on the catch handler prolog address stored in register 15 does not cause unwanted interruptions during the further program run, it is advisable to delete this test point with %REMOVE. The `%SHOW %INSERT` command provides you with the appropriate address for this.

%MOVE

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

– With the *sender* operand you designate a variable, a class object or one of its components, a structure or a structure component, an array or an array element, a length, an address, an execution counter, a register, a complex memory reference or an AID literal.
sender can be located in virtual memory of the loaded program or in a dump file.

– With the *receiver* operand you designate a variable, a class object or one of its components, a structure or a structure component, an array or an array element, a complex memory reference, an execution counter or a register which is to be overwritten. *receiver* can only be located in virtual memory of the loaded program.

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

Command	Operand
%M[OVE]	sender INTO receiver [REP]

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

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.

When *receiver.* is an object of a class, %MOVE overwrites auxiliary variables which have possibly been generated by the compiler. This leads to an inconsistent status of the *receiver*-object.

Immediately after loading the program you can access only global and static data. AID needs the appropriate qualifications to access them.

In addition to the operand values described here, the values described in the manual for debugging on machine code level [2] can also be employed.

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

The %MOVE command does not alter the program state.

sender	INTO	receiver

For *sender* or *receiver* you can specify a variable, a class object or one of its components, a structure or a structure component, an array or an array element, an execution counter, a register or a complex memory reference. Constants, addresses, lengths and AID literals can only be employed as *sender*.

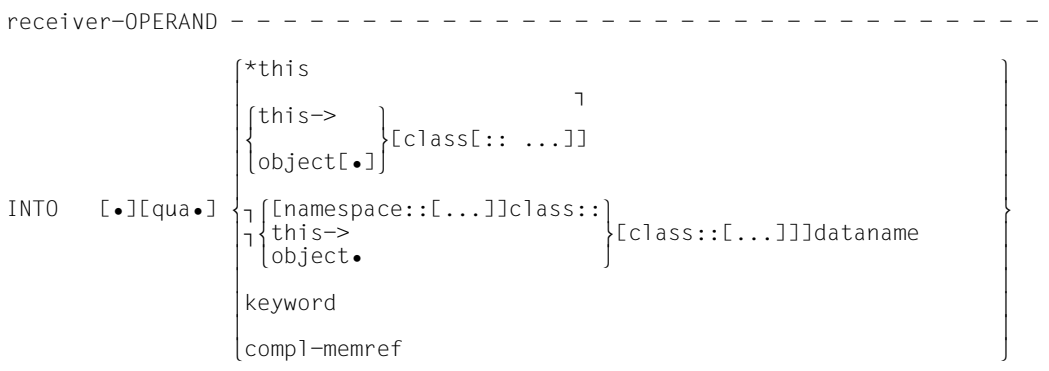
sender may be either in the virtual memory area of the program which has been loaded or in a dump file; *receiver*, on the other hand, can only be within the virtual memory of the loaded program. Moving program segments or overwriting them with instruction code may have unwanted side-effects if this affects addresses which are associated with a *control-area* or a *trace-area* or addresses at which a *test-point* has been set with %INSERT (see the section on “Interactions” in the AID Core Manual [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.

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

{            {*this                                                  }    }
|            |                                                       |    |
|            |{this->   }                                            |    |
|            |{         }[class[:: ...]]                             |    |
|            |{object[•]}                                            |    |
|            |                                                       |    |
|            |  {[namespace::[...]]}                {dataname }      |    |
|          ┐ |{ {this->            }[class::[...]]]{function } }    |    |
| [•][qua•] { |{ {object•           }                {object   } }    |    |
|            |                                                       |    |
|            |L'label'                                               |    |
{            |S'[f-]n[:a]'                                           }    }
|            |keyword                                                |    |
|            |compl-memref                                           |    |
|            |                                                       |    |
|            |&...                                                   |    |
|            |sizeof(...)                                            |    |
|            |                                                       |    |
|  %@(...)                                                           |    |
|  %L(...)                                                           |    |
|  %L=(expression)                                                   |    |
|                                                                    |    |
{  AID-literal                                                       }    }

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



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

qua One or more qualifications may be specified here if *sender* or *receiver* cannot be reached from the current interrupt point by other means or to address a data name that is locally hidden at the interrupt point by an identically named definition.
It is sufficient to specify only the qualifications needed for a unique address.

{E={VM | Dn} for *sender* | E=VM for *receiver*}
Specified only if the current base qualification (see %BASE) is not to apply for a data name. class, class object, statement name, source reference or keyword (see %BASE). *sender* can be located in virtual memory or in a dump file. *receiver*, on the other hand, can only be in the virtual memory.

S=srcname
Specified only if you are accessing a data name, a class or a class object, a statement name or a source reference which is not located in the current translation unit (see the chapter “Addressing in C and C++ programs” on page 21).

:: Use the two prepended colons to address a global data item that is locally hidden at the interrupt point by a definition of the same name. You must also place two colons before the name of a global data item or a function if either the data or the function is not in the call hierarchy or if its definition only occurs after the interrupt point. In contrast to the other qualifications, no period must be entered between the two colons and the operands which follow.

PROC=function

Specified if you want to access a class or class object or a data name which is defined in the current function, but is hidden at the interrupt point by a definition with the same name. You also specify a PROC qualification when you want to address a label or a data name declared as static which is assigned in a function outside the current call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21). If you want to specify a source reference that is located in a function template instance or assigned in a function defined in a class template instance, (see the section “Templates” on page 94), you also have to prepend the appropriate PROC qualification if ambiguity occurs.

In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.

Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the namespace or class qualification is prepended to the function name. The `void` signature may not be written, you enter just the two parentheses in this case as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated for space reasons):

```
PROC=[namespace::[...]][class::[...]]{n'function([sign])'          }
                                     {                             }
                                     {t'f_templ<arg[,...]>([sign])'}
```

The `main` and `__STI__` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications. Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK=’[f-]n[:b]’

Specified when you want to address a data name which is assigned to a block within the current call hierarchy and is hidden at the interrupt point by a definition with the same name, or when you want to want to access a data name declared as static and assigned to a block outside the current call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21).

You must also specify a BLK qualification if you want to designate a function from a local class, which is defined in the specified block, in a subsequent PROC qualification (see above, PROC=*function*).

The block name is formed from the line number (*n*), a possible FILE number (*f*) and relative block number (*b*).

NESTLEV= level-number

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

namespace

Name of a namespace declared in the source program.

You only specify the name of a namespace if the required namespace component is not visible at the interrupt point. You use this to describe the addressing path to classes, data or functions defined in the namespace (see the section “Namespaces” on page 85).

Only the E or S qualification or the two colons (`::`) for the global namespace are allowed before the namespace qualification.

{ class | this-> | object }

Name of a class, the `this` pointer, or a class object declared in the source program. You specify class names, the `this` pointer with the appended pointer operator, and the names of class objects in order to describe the address path to data members assigned to classes (see the section “Classes” on page 63).

If the current interrupt point is located in a dynamic member function, you can address dynamic data members in exactly the same way as in C++, i.e. if the data item is visible at the interrupt point you can access it directly with AID, without qualification. As in C++, locally hidden data requires appropriate class qualification. You can also access dynamic data members by using the `this` pointer followed by the pointer operator. This is equivalent to using the object name followed by a period.

Static data members can only be addressed individually. They can be reached via the associated class name from any part of the program by means of the two subsequent colons. In the case of nested classes, the path to the data item includes all class names from the outer to inner levels, each separated by two colons. The outermost class name requires qualification corresponding to the scope. If the

program is interrupted in a member function of the class, the class scope rules apply for accessing static data members, i.e. if the data is not hidden by a definition with the same name, it can be accessed without qualification.

If the class is a class template instance, you have to use the following notation: `t'`*k_template*`<`*arg*`[,...]>'`. If only one instance of the template exists, only `t'`*k_template*`'` is required.

If *sender* or *receiver* ends with an object name, the dynamic data and, if present, the compiler-generated auxiliary variables and the address of the virtual function table are referenced, regardless of the current interrupt point. For derived classes, *sender* or *receiver* also includes the base classes. You can reference the same area using `*this`, if the program was interrupted in a dynamic member function of the class. To identify a base class in a derived class, you specify the name of the desired base class in the path, with the object name or `this->` as the starting point. The area identified by *sender* or *receiver* are moved or are overwritten for the length of sender in left-justified binary, with the division into components being ignored.
If *object* is not in the scope of the interrupt point, an appropriate qualification is required. Only a base qualification is meaningful before `*this`.

If *sender* or *receiver* ends on `this->`, it designates the first 4 bytes from the start address of the current object.

dataname

This is a data name declared in the source program. *dataname* is specified as in the source program.
You can reference data as in C/C++, but with the following exceptions:
An array name without a subscript addresses all array elements.
Array elements can only be referenced by means of subscripts, not pointers. For more information on working with arrays, see the section “Subscript notation” on page 30.

You can specify *dataname* as follows. You can also combine these formats (see the section “Data names” on page 29):

Subscript notation:	*dataname* `[`*subscript*`]` `{...}`
Pointer notation:	*dataname1* `->` *dataname2*
Structure qualification:	*superordinate dataname*`.` `{...}` *dataname*
Dereferencing:	`[(]*{...}` *dataname*`[)]`

You can transfer an entire structure by specifying the name of the structure as *sender*.
You can transfer an entire array by specifying the name of the array without subscript as *sender*. There is one exception: the names of formal array parameters. With them you reference not the array, but just its address.

If you specify the name of a structure or array as *receiver*, the structure or array will be overwritten in a length equal to that of *sender*, beginning with the start address, without reflecting a division into components or elements.

```
{function    }
{L'label'    }
{S'[f-]n[:a]'}
```

Statement names and source references are address constants. They can only be specified as *sender*. The address held in the address constant is transferred.
A following pointer operator (->) designates 4 bytes of the machine code located at the corresponding address. You can use %DISASSEMBLE to output the machine instructions in preparation for a length modification, should one be necessary.
function`[%al4]->`, `L'`*label*`'->` and `S'[`*f*`-]`*n*`[:`*a*`]'->` can be used as *sender* and as *receiver* (see Examples 4 on page 207 and 6 on page 207).

function

This is the name of a function, as declared in the source program, or the name of a library function. It references the start address of the function prolog that is generated by the compiler (see PROC=*function* on page 198 and the chapter "C++-specific addressing" on page 57).

Virtual functions can be addressed with the following syntax:
`p->n'function(signature)'`

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p* (see the description of `this` on page 64 and section "Virtual functions" on page 73).

If you want to access a function addressed via a pointer to member, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

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

[qua.]object.*[object.][class::][...]pointer-to-function-member

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

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

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

[qua•]pointer->*[object•][class::][...]pointer-to-function-member

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

The class object is designated by the operand on the left of the dereferencing operator `•*` or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.
The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to function member cannot be reached from the interrupt point by some other means.

If you want to transfer or overwrite the instruction code of a function addressed via a pointer to member with %MOVE, note that you cannot directly append the pointer operator to one of the syntaxes above. You would have to first switch to machine code level with a type modification, i.e. %al4, by using the following syntax:

dereferenced-pointer-to-function-member `%al4->`

This designates the first 4 bytes of the instruction code located at the prolog address.

More details on working with a pointer to function member can be found on page 79.

L'label'
: This designates the address of the first executable statement after a label.
label is the name of a label as declared in the source program.

S'[f-]n[:a]'
: is a source reference and designates the address of an executable statement. It is constructed from the line number (*n*) and, if present, the FILE number (*f*) and the relative statement number within the line (*a*).
If the source reference is located in a function which was created from a function template via instantiation or the function containing the source reference is defined in a class template instance, you have to prepend the appropriate PROC qualification to the source reference if ambiguity occurs.

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
%nGD                AID floating point register, n = 0,2,4,6
```

compl-memref

compl-memref may contain the following operations (see the section on “Complex memory references” in the AID Core Manual [1]):

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

If *compl-memref* begins with an address constant (such as a source reference or a label), the pointer operator (->) must come next. A label must always be placed within `L'...'` in such cases. Without the pointer operator, address constants can be used in *compl-memref* wherever hexadecimal numbers are also allowed.

After byte offset (•) or a 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 to the sender, if no value for type and length has been explicitly specified by the user. If you specify a complex memory reference as the receiver, the area which can be overwritten extends from the start address of *compl-memref* to the end of the memory occupied by your program. However, the most you can move with one %MOVE command is 3900 bytes. A type modification at the end of *compl-memref* is pointless, since transfers with %MOVE are always in binary form. However, a type modification may be necessary before a pointer operation (->).

Example: `%3g.2%al2->`
The last two bytes of the AID register %3G are used as an address.

The assigned memory area for any operand in a complex memory reference must not be exceeded as the result of byte offset or length modification; otherwise AID does not execute the command and writes an error message. By combining the address selection (%@) with the pointer operator (->) you can exit from the symbolic level. You may then use the address of a data item without having to take note of its area limits.

Example:
The arrays `name` and `name1` are of type char and occupy 5 bytes each. The last 2 bytes of name as well as the following 3 bytes are to be transferred to `name1`.

```
%move name.3%l5 into name1
```

This command is rejected by AID because the memory area of name was violated. The command should read:

```
%move %@(name)->.3%l5 into name1
```

& is the address operator. You can use it to define the start address of a data item, a class object or a function as the *sender*.

The address operator & can also be used to determine the relative address of a dynamic data member of a class, provided you observe the following:

– If the interrupt point is located outside the class containing the data member, you should enter the appropriate class qualification after the address operator, and then the name of the data item.

– If the interrupt point is located in a dynamic member function of the class, you will need to enter a base or area qualification (S, PROC or :: qualification) before the address operator so that AID can access the class from “outside”, so to speak.

Note that in contrast to the address selector %@(...) (see page 159), the address operator is purely a “high-level” function and thus cannot be applied on complex memory references.

For more details on the address operator, see also the section “The address operator & and the address selector %@(...)” on page 42.

sizeof()
is the length operator. The length of a data item or class is transferred.
To determine the length of a class, you may specify the name of the class itself or an object of the class as operands. You will receive the number of bytes occupied by the dynamic data members of the class and by the auxiliary variables generated by the compiler (if any).
You may specify the name of a namespace here, but only in the path to a component of the namespace.
Bit-field and register variables are not allowed here.

The length operator is described in detail in the section “Length operator sizeof() and length selector %L(...)” on page 47.

%@(...)

The address selector enables you to use the start address of a data item, of a class object, or of a complex memory reference as *sender*. You can specify a class name only in the path for the base class of an object of a derived class; this identifies the start address of the dynamic data of the base class.
You can only specify the name of a namespace here in the path to a component of the namespace.
The address selector returns an address constant as its result (see the section on “Address, type, and length selectors” in the AID Core Manual [1]).
The address selector cannot be applied to constants, including labels, source references and functions.

%L(...)

The length selector allows you to use the length of a data item or of a class as *sender*. If you apply the length selector to a class or a class object, the result corresponds to that of sizeof() in C++, i.e. you receive the length of the dynamic data and of the compiler-generated auxiliary variables, if any.
You can only specify the name of a namespace here in the path to a component of the namespace.

The length selector returns an integer as its result (see the section on “Complex memory references” in the AID Core Manual [1]). For bit-fields, the number of bytes over which the bit-field extends is calculated and returned as the length.
Example: `%move %l(var1) into %3g`
The length of `var1` will be transferred.

%L=(expression)

The length function enables you to calculate a value and store it in *receiver*. *expression* is formed from the contents of memory, constants and integers together with the arithmetic operators (+,-,*,/). Only memory reference contents which are integers (type %F or %A) are permitted. The length function returns an integer (see the AID Core Manual, section "Length modification" [1]).

When using overloaded operators, note that AID does not emulate this process, but always uses standard operators.

Example: `%move %l=(var1) into %3g`
The content of `var1` is transferred, provided it is an integer (data type int); otherwise AID issues an error message.

AID literal

The following AID literals (see the chapter on “AID literals” in the AID Core Manual [1]) 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 a declaration made with the %AID command. If *REP* is not specified and there is no valid declaration in the %AID command, no REP record is created.

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

REP = {Y[ES] | NO}

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

REP=Y[ES]

LMS correction statements (REPs) are created in SDF format for the update caused by the current %MOVE command. If the object structure list is not available, no REP records are generated, and AID will output an error message.

If *receiver* is not located completely within one CSECT or *sender* exceeds a length of 3900 bytes, AID will also output an error message and not write a REP record. To obtain REP records despite this, you will have to spread the transfer operation over several %MOVE commands (see also the manual “Debugging on Machine Code Level” [2]).

AID stores the corrections in a file with the link name F6. The MODIFY-ELEMENT statement must then be inserted in it for the LMS run. Care should therefore be taken that no other outputs are written to the file with link name F6.

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

REP=NO

No REPs are created for the current %MOVE command.

Examples

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

```
C program
==========================================
      int           i_arr_1[10];
      int           i_arr_2[20];
      unsigned long number;
      float         x_arr[10];
==========================================
```

1. `%move i_arr_1 into i_arr_2`

   No subscript is given for either array; AID therefore transfers the entire array `i_arr_1` left-justified in hexadecimal form to `i_arr_2` without reflecting a division into individual elements.

2. `%move 20 into number`

   Into the variable number, which also occupies 4 bytes in the C program, AID writes a word containing the value 20 (X'00000014').

3. `%move 20 into x_arr[5]`

   Note: As in example 2, a word with the contents X'00000014' is written to `x_arr[5]`, which of course makes no sense when an array element of type float is involved. To transfer the value 20 to `x_arr[5]`, you would have to enter a %SET command (see %SET on page 254) to perform conversion prior to the transfer.

4. `%move x'58f0c160' into s'21:2'->.16 rep=yes`

   The code generated for statement S'21:2' is changed: Beginning with the 16th position after the start address of S'21:2', 4 bytes are overwritten with the hexadecimal literal X'58F0C160'. A REP record is created for the correction and written to the file AID.OUTFILE.F6 or the file assigned to the link name F6.

5. `%move s'12' into %2g`

   The address of the first statement in line 12 is written into the AID register %2G.

6. `%move s'12'->%l=(s'13'-s'12') into Y::n'f()'->.#'20'`

   The machine code generated for statement S'12' is moved. The length of this statement is defined by the entry `%l=(s'13'-s'12')`. This length of machine code is moved into member function `Y::f()`, starting at the address determined by the prolog address and the byte offset (#'20' = 32 Bytes).

7. `%move ::a into *this.4`

   The program was interrupted in a dynamic function of a class. The %MOVE command transfers the contents of global variable `a` into the current object associated with the function. The object is overwritten starting with the fifth byte.

8. `%move ::A::j into z.X.4`

   The %MOVE transfers static variable `j` from global class `A` into base class `X` of object `z` in left-justified binary format starting with the fifth position.

%ON

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

– With *write-event* you define the event associated with write access to an area of memory. Whenever the program modifies the specified area of memory, AID is to interrupt program execution in order to process *subcmd*.

– With *event* you define one of the other events (normal or abnormal program termination, supervisor call (SVC), program error, etc.) for which AID is to interrupt program execution in order to process *subcmd*.

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

Command	Operand	
%ON	{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*. Each new *write-event* overwrites the previous one.

A defined event remains in effect until it is deleted with %REMOVE or the program terminates. In addition, all definitions made with %ON are reset after a `fork()` call and in a program invoked with `exec()`.

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

The %ON command does not alter the program state.

write-event

The keyword %WRITE activates write monitoring. The keyword is followed in parentheses by the area of memory to be monitored. If the program modifies a byte within the specified area, program execution is interrupted after the instruction which modified the memory location and *subcmd* is executed.

Only one *write-event* may be defined at any one time. Each new *write-event* you define overwrites the previous one. There may, however, be other types of event defined at the same time. If an *event* is reported at the same time as *write-event*, AID processes the subcommand for *write-event* first.
You delete *write-event* by issuing %REMOVE %WRITE without specifying the memory reference.

%ON *write-event* and other AID commands interact in the following ways:

– If a %CONTROLn or a %TRACE with a machine-oriented *criterion* is in progress, any attempt to enter %ON *write-event* will be rejected with an error message.

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

– If a machine instruction has been overwritten with the AID-internal marker by the test point defined in an %INSERT command, AID likewise does not recognize this instruction's write access.

To ensure the continuity of write monitoring it is advisable to kill all %CONTROLn and %INSERT commands with %REMOVE and cancel any %TRACE which is still in progress by specifying %RESUME after %ON.

The memory area to be monitored can be any memory object, no matter how it is addressed. The area is defined by its start address and an implicit or explicit length specification. The length must be not exceed 64KB; if it does, an error message is issued.

If the address of the specified memory object is overloaded in a program with an overlay structure, the corresponding area of the last segment loaded is monitored.

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

qua One or more qualifications may be specified here if the area to be monitored cannot be reached from the current interrupt point by other means or to address a data name that is locally hidden at the interrupt point by an identically named definition. It is sufficient to specify only the qualifications needed to provide a unique address for the memory object.

S=srcname
: Specified only if the memory area you want to monitor is not located in the current translation unit (see the section “Qualifications” on page 21).

:: Use the two prepended colons to address a global data item that is locally hidden at the interrupt point by a definition of the same name. You must also place two colons before the name of a global data item or a function if either the data or the function is not in the call hierarchy or if its definition only occurs after the interrupt point. In contrast to the other qualifications, no period must be entered between the two colons and the operands which follow.

PROC=function
: Specified only if you want to access a data name which is defined in the current function, but is hidden at the interrupt point by a definition with the same name. You also specify a PROC qualification when you want to address a label or a data name declared as static which is assigned in a function outside the current

call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21). If you specify a source reference that is located in a function template instance or assigned to a function defined in a class template instance, (see the section “Templates” on page 94), you also have to prepend the appropriate PROC qualification if ambiguity occurs.
In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.
Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the namespace or class qualification is prepended to the function name. The `void` signature may not be written, you enter just the two parentheses after the function name in this case as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated for space reasons):

```
PROC=[namespace::[...]][class::[...]]{n'function([sign])'          }
                                     {t'f_templ<arg[,...]>([sign])'}
```

The `main` and `__STI__` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications. Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK=’[f-]n[:b]’

Specified only when you want to reference a data name which is assigned to a block within the current call hierarchy and is hidden at the interrupt point by a definition with the same name, or when you want to want to reference a data name declared as static and assigned to a block outside the current call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21).

You must also specify a BLK qualification if you want to designate a function from a local class, which is defined in the specified block, in a subsequent PROC qualification (see above, PROC=*function*).

The block name is formed from the line number (*n*), a possible FILE number (*f*) and relative block number (*b*).

namespace

Name of a namespace declared in the source program.

You only specify the name of a namespace if the required namespace component is not visible at the interrupt point. You use this to describe the addressing path to classes, data or functions defined in the namespace (see the section “Namespaces” on page 85).

Only the E or S qualification or the two colons (`::`) for the global namespace are allowed before the namespace qualification.

{ class | this-> | object }

Name of a class, the `this` pointer, or a class object, declared in the source program.
You specify class names, the `this` pointer with the appended pointer operator, and the names of class objects in order to describe the address path to data members assigned to classes (see the section “Classes” on page 63).

If the current interrupt point is located in a dynamic member function, you can address the class data according to the scope rules known from C++.

You can reach the dynamic data of an object independent of the interrupt point via the object name followed by a period, if the object is located in the current call hierarchy.

Static data members can only be addressed individually. They can be reached via the associated class name from any part of the program by means of the two subsequent colons. In the case of nested classes, the path to the data item includes all class names from the outer to inner levels, each separated by two colons. The outermost class name requires qualification corresponding to the scope. If the program is interrupted in a member function of the class, the class scope rules apply for accessing static data members, i.e. if the data is not hidden by a definition with the same name, it can be accessed without qualification.

If the class is a class template instance, you have to use the following notation: `t'`*k_template*`<`*arg*`[,...]>'`. If only one instance of the template exists, only `t'`*k_template*`'` is required.

If the area operand ends with an object name, the dynamic data and, if present, the compiler-generated auxiliary variables and the address of the virtual function table are referenced, regardless of the current interrupt point. For derived classes, the area to be monitored also includes the base classes. You can reference the same area using `*this`, if the program was interrupted in a dynamic member function of the class. To identify a base class in a derived class, you specify the name of the desired base class in the path, with the object name or `this->` as the starting point. You only have to specify the class names required to uniquely identify the desired member.

If *object* is not visible at the interrupt point, an appropriate qualification is required. Only a base qualification is meaningful before `this`.
If the area operand ends on `this->`, you designate 4 bytes as of the start address of the current object.

dataname

This is a data name declared in the source program. *dataname* is specified as in the source program.

You can reference data as in C/C++, but with the following exceptions,:
An array name without a subscript addresses all array elements.
Array elements can only be referenced by means of subscripts, not pointers.
It is not possible to monitor a range of subscripts.
Note that when a subscripted entry is specified in an `%ON %WRITE(...)`, the start address and length of the area to be monitored is calculated by AID as soon as the input is received. Consequently, even if the value of the subscript changes during the program run and thus results in a different start address for the area specified with *dataname*`[`*subscript*`]{...}`, `%ON %WRITE(...)` will continue to monitor only the area that was applicable on entering the command.
If `%AID C=YES` is set (see page 115), AID combines the array elements of a `char` array that can be addressed via the subscript on the extreme right into C strings. When such a C string is specified, `%ON %WRITE(...)` monitors the entire array underlying the C string and not just the C string up to the end criterion X'00'.
Arrays that are passed as parameters to a function are implemented therein as pointers to the array in the calling program. In such cases, `%ON %WRITE(...)` monitors these pointers, but not the associated array.
For more information on working with arrays, see also the section "Subscript notation" on page 30.

If you specify a data member that is referenced via a pointer to member in %ON %WRITE(...), AID will always monitor the memory area designated on input even if the address entered in the pointer to member changes in the course of the program and the pointer to member subsequently references some other data member.AID combines the array elements of a `char` array that can be addressed via the subscript on the extreme right into C strings.

You can specify *dataname* as follows. You can also combine these formats (see the section "Data names" on page 29):

Subscript notation:	*dataname* `[`*subscript*`]{...}`
Pointer notation:	*dataname1* `->` *dataname2*
Structure qualification:	*superordinate dataname*`.{...}` *dataname*
Dereferencing:	`[(]*{...}` *dataname*`[)]`
Pointer to member dereferencing:	*dataname1*`.*`*datanname2* `or` *dataname1*`->*`*datanname2*

```
⎧function[%al4]⎫
⎨L'label'      ⎬->
⎩S'[f-]n[:a]'  ⎭
```

This designates 4 bytes of machine code starting at the address stored in one of the address constants. To have a different number of bytes monitored you should specify a suitable length modification.

function

This is the name of a function, as declared in the source program, or the name of a library function. It references the start address of the function prolog that is generated by the compiler (see PROC=*function* on page 211 and the chapter "C++-specific addressing" on page 57).

Virtual functions can be addressed with the following syntax:
`p->n'function(signature)'`

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p* (see the description of `this` on page 64 and the section "Virtual functions" on page 73).

If you want to access a function addressed via a pointer to member, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

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

[qua.]object.*[object.][class::][...]pointer-to-function-member

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

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

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

[qua.]pointer->*[object.][class::][...]pointer-to-function-member

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

The class object is designated by the operand on the left of the dereferencing operator `.*` or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.

The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to member cannot be reached from the interrupt point by some other means.

If you want to monitor the instruction code of a function addressed via a pointer to member with %ON %WRITE(...), note that you cannot directly append the pointer operator to one of the syntaxes above. You would have to first switch to machine code level with a type modification, i.e. %al4, by using the following syntax:

dereferenced-pointer-to-function-member `%al4->`

This designates the first 4 bytes of the instruction code located at the prolog address.

More details on working with a pointer to function member can be found on page 79.

L'label'
: This designates the address of the first executable statement after a label. *label* is the name of a label as declared in the source program.

S'[f-]n[:a]'
: is a source reference and designates the address of an executable statement. It is constructed from the line number (*n*) and, if present, the FILE number (*f*) and the relative statement number within the line (*a*).
If the source reference is located in a function which was created from a function template via instantiation or the function containing the source reference is defined in a class template instance, you have to prepend the appropriate PROC qualification to the source reference if ambiguity occurs.

compl-memref
: The following operations may occur in *compl-memref* (see also the section on "Complex memory references" in the AID Core Manual [1]):
– byte offset (•)
– indirect addressing (->)
– type modification (%A, %S, %SX)
– length modification (%L(...), %L=(expression), %Ln)
– address selection (%@(...))

compl-memref designates an area of 4 bytes, starting with the calculated address. If a different number of bytes is to be monitored, *compl-memref* must end with the appropriate length modification. When modifying the length of data items, you must pay attention to area limits or switch to machine code level using %@(...)->.
If *compl-memref* begins with a function name, label or source reference, the pointer

operator (->) must come next. Labels in *compl-memref* must then be specified within `L'...'`. Without the pointer operator, function names, labels and source references can be used wherever hexadecimal numbers are also allowed.

event

A keyword is used to specify an event (program error, program termination, supervisor call, etc.) upon which AID is to process the *subcmd* specified. An event code which has been processed by an STXIT routine can no longer be responded to with a *subcmd* assigned to the *event*.
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* is reported at the same time as some other *event*, AID processes the subcommand for *write-event* first.
For information on selection of the SVC numbers and event codes see the "Executive Macros" manual (page 341).

It must be noted that an `exec()` call cannot be monitored with `%ON %TERM`. The program is overloaded and thereby terminated by the `exec()` call, but this is not a program termination in the sense of `%ON %TERM`. You have to set `%AID EXEC=ON` to monitor an `exec()` call. This causes the program to be halted immediately after the `exec()`.

event	*subcmd* **is processed:**	
%ERRFLG (z)	after before	the occurrence of an error with the specified event code and abortion of the program.
%INSTCHK	after before	the occurrence of an addressing error, an impermissible supervisor call (SVC), an operation code which cannot be decoded, a paging error or a privileged operation and abortion of the program.
%ARTHCHK	after before	the occurrence of a data error, divide error, exponent overflow or a zero mantissa and abortion of the program.
%ABNORM	after	the occurrence of one of the errors covered by the previously described events or a DMS error (as of BS2000 V10).

Table 5: %ON command events and their meanings

event	*subcmd* **is processed:**	
%ERRFLG	after	the occurrence of an error with any event code.
%SVC(z)	before	execution of the supervisor call (SVC) with the specified number.
%SVC	before	execution of any supervisor call (SVC).
%LPOV(name)	after	loading of the segment with the specified name.
%LPOV	after	loading of any segment (to list the name enter %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. An exec() call cannot be monitored with this.
%ANY	before	termination of a program due to a program error or the %TERM events described above or a DMS error (as of BS2000 V10).

Table 5: %ON command events and their meanings

z Is an integer, where $1 \le z \le 255$. *z* can be specified as an unsigned decimal number of up to three digits or as a two-digit hexadecimal number (#'ff'). No check is made whether the specified event code or the SVC number is meaningful or permissible.

subcmd

Is processed whenever the specified *event* occurs in the course of program execution. If the *subcmd* operand is omitted, AID inserts a <%STOP>.
A detailed description of *subcmd* can be found in the "Subcommand" chapter of the AID Core Manual [1].

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

                                          ⎧AID-command    ⎫
<[subcmdname:] [(condition):] [           ⎨               ⎬ {;...}]>
                                          ⎩BS2000-command ⎭

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

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

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

subcmd does not overwrite an existing subcommand for the same *event*. Instead, the new subcommand is prefixed to the existing one. It is not possible to concatenate write events in this way.

The commands %CONTROL*n*, %INSERT, and %ON can be used in *subcmd* in combination with %ON. In other words, several monitoring commands may be nested. For an example refer to the description of %INSERT.

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 in *subcmd* will not be executed. For the same reason, a %REMOVE for the current subcommand or for the %ON command itself or for the associated *event* makes sense only as the last command in *subcmd*.

Address operands in subcommands are not automatically supplemented on input with the qualifications that correspond to the current interrupt point. If the defined *event* occurs in the subsequent debugging run and AID interrupts the program to process *subcmd*, only the data and functions that belong to the same scope as the address at which the *event* occurred can be referenced without qualification with AID commands from *subcmd* . The interrupt point can also be in a routine of the runtime system, which means that in this case all user program data and functions can only be accessed with qualification.

Examples

1.
```
%on %errflg (108)
%on %errflg (#'6c')
```

Each specification designates the same program error (mantissa equals zero).

2.
```
%on %errflg (107) <%d 'error'>
```

This event code does not exist; the *subcmd* defined for this *event* will therefore never be started.

3.

```
/%on %any
/%resume
STOPPED AT SRC REF: 59, SOURCE: VPTR.C , PROC: g(void) ,
EVENT: ADDRESS ERROR
```

Your program was interrupted due to an error. In addition to the kind or error, AID outputs the source reference as well as the names of the translation unit and the function in which the error that caused the interrupt occurred.

4.

```
/%on %write(page) <wr1: %d 'Write access to page'>
/%r

Write access to page
Write access to page
Write access to page
Write access to page
Write access to page
```

Five write-access operations on `page` were performed by the program.

5.

```
/%on %write(page) <wr1:>
/%on %term <%display %.wr1; %stop>
/%r
%  CCM0998 CPU TIME USED: 0.0323 SECONDS
CURRENT PC: 01015846    CSECT: ITOTRM@@  ******************************
%.WR1                          =              5
STOPPED AT V'1015846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,
NODUMP)
```

This time only an execution counter `%.WR1` is set up in a %WRITE subcommand. The counter is incremented by one on each pass. On program termination, event %TERM, the program is halted and the execution counter and a STOP message are displayed.

%OUT

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

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

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

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

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

Before selecting a file as the output medium via %OUT, you must issue the %OUTFILE command to assign the file to a link name and open it; otherwise AID creates a default output file with the name AID.OUTFILE.Fn.
The declarations made with the %OUT command remain in effect until they are overwritten by a new %OUT command or until a /LOGOFF or /EXIT-JOB is issued.

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

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

target-cmd

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

```
{%D[IS]A[SSEMBLE]}
 %D[ISPLAY]      
{%H[ELP]         }
 %SD[UMP]        
 %T[RACE]        
```

medium-a-quantity

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

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

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

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

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

For more details on *medium-a-quantity,* see the chapter “Medium-a-quantity operand” in the AID Core Manual [1].

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

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

MAX Output with additional information

MIN Output without additional information

XMAX Definition of XMAX mode for the corresponding command %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP or %TRACE.

XFLAT Definition of XFLAT mode for the corresponding command %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP or %TRACE.

Examples

1. ```
   %outfile f1=output
   %out %sdump t=min,f1=max
   ```

   Data output of the %SDUMP command is to be output to the terminal in abbreviated form, and parallel to this also, with additional information to the file with link name `F1`. Prior to this the file `OUTPUT` was assigned to the link name `F1`.

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 an output file.

Command	Operand
%OUTFILE	[link [= file]]

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

An %OUTFILE without operands closes all open AID output files. If you have not explicitly closed an AID output file using the %OUTFILE command, the file will remain open until LOGOFF or EXIT-JOB. The AID output files are also closed by a `fork()` or `exec()` call.

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

1. Enter an `ADD-FILE-LINK` command for a link name Fn which has not yet been reserved. Then AID opens this file when the first output command for this link name is issued.
2. Leave the creation, assignment and opening of files to AID. AID then uses default file names with the format `AID.OUTFILE.F`*n* 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 F*n*, where *n* is a number with a value $0 \leq n \leq 7$.

The REP records for the %MOVE command are written to the output file with link name F6 (see also the %AID and %MOVE commands); so make sure that you do not write any other output to the file with the link name F6.

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 require the same qualification.

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

Command	Operand
%Q[UALIFY]	[prequalification]

A *prequalification* specified with the aid of the %QUALIFY command applies until it is overwritten by a %QUALIFY with a new *prequalification* or revoked by a %QUALIFY without operands, or until /LOGOFF or /EXIT-JOB. %QUALIFY specifications are also reset by a `fork()` or `exec()` call.

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 translation unit has been loaded or included in the LSD records is not checked until subsequent commands are executed and the information from *prequalification* is actually used in addressing.

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

When entering the %QUALIFY command, the current setting for handling uppercase/lowercase characters (`%AID LOW={ON|OFF|ALL}`) must be taken into account to ensure that the *prequalification* produces the correct extensions in the address operands of subsequent commands.

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

The %QUALIFY command does not alter the program state.

prequalification

Designates a base qualification or one or more area qualifications. Multiple qualifications must be delimited by a period.

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

```
prequalification-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                  ⎧VM⎫                  ⎧[[[qua]•]PROC=function]  ⎫
[E=⎨  ⎬] [[•]S=srcname] ⎨                                        ⎬
                  ⎩Dn⎭                  ⎩[[•]BLK='[f-]n[:b]']     ⎭

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

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
: Designates a translation unit (see the section “Qualifications” on page 21).

[qua•]PROC=function
: Designates a function.

  In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.

  Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the namespace or class qualification is prepended to the function name. The `void` signature may no longer be written. You enter just the two parentheses after the function name as in C++. This results in the following syntax (*f_template* and *signature* are abbreviated below for space reasons):

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

                                                ⎧n'function([sign])'          ⎫
PROC=[namespace::[...]][class::[...]]           ⎨                             ⎬
                                                ⎩t'f_templ<arg[,...]>([sign])'⎭

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

  The `main` and `_ _STI_ _` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

qua

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications.

Syntax for *qua*:

```
PROC=superordinate_function[BLK='[f-]n[:b]'•[...]]
```

Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK='[f-]n[:b]'

Designates a block. As in the case of names for source references, block names are constructed from the line number (*n*) and, if appropriate, the FILE number (*f*) and the relative block number (*b*).

Examples

1.
```
%qualify e=d1.s=n'parr.c'.proc=foo
%d .str[1]
```

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

```
%d e=d1.s=n'parr.c'.proc=foo.str[1]
```

2.
```
%qualify s=n'bcl.c'
%set .::a into b
```

   This qualification designates the translation unit `BCL.C`. Because of the *prequalification*, the %SET command has the same effect as the following %SET command in full format:

```
%set s=n'bcl.c'.::a into b
```

   The global variable `a` of the compilation unit `BCL.C` is transferred into `b`.

3.
```
%qualify s=n'examp.c'.proc=main
%d .x_arr[i]
```

   As in examples 1 and 2, the prequalification from the %QUALIFY command is written before the period in the %DISPLAY command.
   The prequalification applies here not only to the array `x_arr`, but also to the specified subscript `i`. You thus address the `i`-th element of the array `x_arr` from the function `main` in the translation unit `EXAMP.C`.

%REMOVE

With the %REMOVE command you revoke the debug declarations for the %CONTROL*n*, %INSERT and %ON commands.

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

Command	Operand
%REM[OVE]	target

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

%REMOVE 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 a subcommand to be deleted. If *target* is within a nested subcommand and therefore has not yet been entered, it cannot be deleted.

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

{ %C[ONTROL] | %C[ONTROL]n    }
{ %IN[SERT] | test-point      }
{ %ON | %WRITE | event        }
{ %•[subcmdname]              }

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

%C[ONTROL]
: The declarations for all %CONTROLn commands entered are deleted.

%C[ONTROL]n
: The %CONTROLn command with the specified number ($1 \leq n \leq 7$) is deleted.

%IN[SERT]
: All test points which have been entered are deleted.

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

%ON All events which have been entered are deleted.

%WRITE
: The *write-event* is deleted.

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

The following applies for the events %ERRFLG(*z*), %SVC(*z*) and %LPOV(*name*):

– %REMOVE *event(z* | *name)* deletes only the event with the specified number/name.

– %REMOVE *event* without specification of a number/name deletes all events of the corresponding group.

%•[subcmdname]
: deletes the subcommand with the name *subcmdname* in a %CONTROL*n* 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 %CONTROL*n* cannot be chained, the associated %CONTROL*n* will be deleted as well. Deleting the subcommand therefore has the same effect as deleting the %CONTROL*n* 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 the “Subcommand” chapter in the AID Core Manual [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'58' <sub1: %d char1, char2, numb>
%in s'58' <sub2: %d result; %rem %.>
%r
...
%rem s'58'
```

   When the test point S’58’ is reached, `result` is output, and the subcommand `SUB2` is deleted. This subcommand is thus executed only once. `char1, char2` and `numb` are then output, and the program is continued. Whenever the program run reaches the test point S’58’, subcommand `SUB1` is executed. `%rem s'58'` deletes the test point later. `%rem %.sub1` would have the same effect, since this subcommand is the only remaining entry for test point S’58’

%RESUME

With %RESUME you start the loaded program or continue it at the interrupt point. The program executes without tracing.
The continuation address for the program run cannot be influenced with %RESUME. You can define another continuation point only by using %SET to change the program counter (%PC) (see the description of the command %SET *keyword* on page 203).

%RESUME terminates all active %TRACE commands, but %CONTINUE has no effect on %TRACE.

Command	Operand
%R[ESUME]	

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

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

The %RESUME command alters the program state.

%SDUMP

With %SDUMP you output a symbolic dump; you can dump individual data, all data of the current call hierarchy, or the current call hierarchy itself. The current call hierarchy extends from the function or block where the program was interrupted to the main program. Output is via SYSOUT or SYSLST or into a cataloged file.

– With *dump-area* you designate the data or data areas which AID is to output, or you specify that AID is to output 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 for the current %SDUMP command a declaration made using the %OUT command.

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

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

%SDUMP without operands outputs all data of the current call hierarchy to the extent that AID can access the associated LSD records.
Data which has multiple definitions is also output multiple times. Data that was assigned an alias name with %ALIAS is listed under its original name.

%SDUMP %NEST outputs the current call hierarchy, i.e. the source reference of the interrupt point, the numbers of the blocks, and the names of the functions and of the program that are active at the interrupt point. AID also outputs the current call hierarchy for programs for which no LSD records were generated (see also Example 3 on page 247).

If the current interrupt point is located in a recursive function, %SDUMP counts each call of this function, i.e. the data of the function is output for each call, and for %SDUMP %NEST, the function is listed as many times as it was called up to the interrupt point.

You cannot use %SDUMP immediately after the program is loaded. Only when the program is before the first executable statement does a call hierarchy exist, and AID can then allocate the appropriate name areas. You interrupt the program with one of the following commands before the first executable statement:

```
/%insert main;%r
```
or
```
/%trace 1 in s=srcname
```

If you use %TRACE when debugging a C++ program that includes classes with virtual functions or constructors, the program will not halt in `main`, but in a compiler-generated function with the name `_ _STI_ _`. This function invokes the constructors of global objects and sets up the virtual function tables. AID indicates the function name `_ _STI_ _` in the STOP message and in the current call hierarchy.
The destructors of global objects are called by routines of the runtime system on completion of `main`. Their names do not appear in the call hierarchy.

dump-area can be repeated up to 7 times. 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.

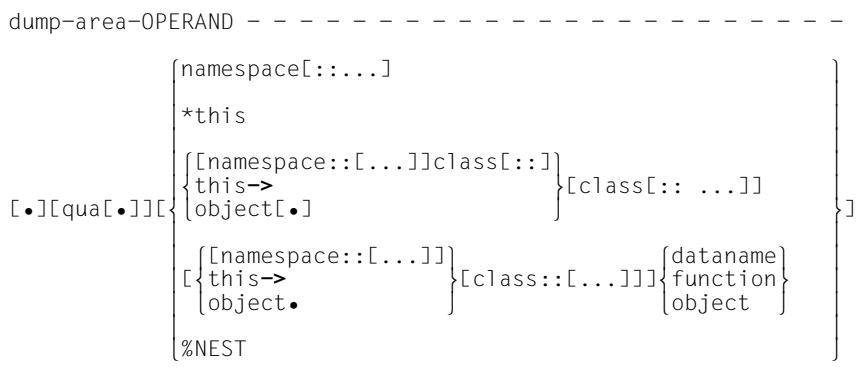
dump-area

Describes which information AID is to output. AID can output the current call hierarchy, all data of the current call hierarchy, all data of a translation unit, function or block, or individual data items.

AID edits the data items in accordance with their definition in the source program.

If a *dataname*, *class*, *object* or *function* has multiple definitions in the current call hierarchy, it is output more than once, unless *dump-area* has been restricted by a qualification. Note that the global area is not treated as part of the current call hierarchy. To output a global data item of the same name, you must specify `%DISPLAY [S=`*srcname*`•]::`*dataname*. This also applies to *class*, *object* or *function*.

In an `%SDUMP S=`*srcname* for all data items of a translation unit, by contrast, AID includes all global data items and the addresses of all functions defined or declared and used in that translation unit. You cannot use %SDUMP with a :: qualification to explicitly address the global area.



- 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. If the addressing ends in a qualification, do not enter the terminating period.

qua Specify one or more qualifications if the interrupt point is not within the scope of the addressed object or if the memory object is not visible at the interrupt point. Only enter the qualification required for unique addressing.
The :: qualification cannot be used in this case. You can therefore only output data and functions which need the :: qualification for access with %DISPLAY, but not with %SDUMP.
If *dump-area* ends with a qualification, all data and functions of the program section designated by the qualification are listed.

E ={VM | Dn}
If the addressing ends with the base qualification, you will receive all namespaces, classes, class objects, data and functions of the relevant call hierarchy.
Otherwise, specify a base qualification only if the current base qualification is not to apply to *dump-area*.
Only a base qualification is possible before the %NEST keyword.

S=srcname
If the addressing ends with the S qualification, you will get all namespaces, classes, class objects, data and functions of the corresponding translation unit. The translation unit must be located in the current call hierarchy.
Otherwise, you specify an S qualification only to reference a class object, a data name, a function or a block from another translation unit.

PROC=function

Restricts *dump-area* to the specified function. *function* must be located in the call hierarchy.

If the addressing ends with the PROC qualification, you will get all classes, class objects, data and functions assigned to the name space of the corresponding function.
Otherwise, you specify a PROC qualification only to reference a class, class object, data name or function uniquely.

In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.

Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the function name is prepended with the namespace or class qualification. The `void` signature may no longer be used. In this case, you only input the two parentheses after the function name, as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated below for space reasons):

```
                                                 ⎧n'function([sign])'          ⎫
PROC=[namespace::[...]][class::[...]]            ⎨                             ⎬
                                                 ⎩t'f_templ<arg[,...]>([sign])'⎭
```

The `main` and `_ _STI_ _` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications.

Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK='[f-]n[:b]'

If the addressing ends with the BLK qualification, you will get all class objects and data of the corresponding block. The block must be located in the current call hierarchy.
Otherwise, you specify a BLK qualification only to reference a class, class object or data name uniquely.

If you want to use a PROC qualification to designate a function from a local class, which is defined in an inner block, you have to prepend the corresponding BLK qualification to the PROC qualification (see above, PROC=*function*).

NESTLEV= level-number

level-number A level number in the current call hierarchy

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

namespace

Name of a namespace declared in the source program.

If the *dump-area* operand ends with the name of a namespace, AID lists all data and functions defined in it. The functions are listed in standard C++ notation and the start address of the associated prolog is output. With nested namespaces, the contents of the inner levels are also output. If a namespace contains a `using` directive to a further namespace, only the name of this is listed.

You only specify the namespace qualification in the addressing path to classes, data or functions defined in the namespace if the required namespace component is not visible at the interrupt point.

Only the E or S qualification are allowed before the namespace qualification.
You will find further information on namespaces in section “Namespaces” on page 85.

{ class | this-> | object }

Name of a class, the `this` pointer, or the name of a class object, as declared in the source program.
You specify class names, the `this` pointer with the appended pointer operator, and the names of class objects in order to describe the address path to data members (see the section “Classes” on page 63).

If the current interrupt point is located in a dynamic member function, you can address the class data according to the normal C++ scope rules.

If an object is in the current call hierarchy, you can access the dynamic data of that object independent of the interrupt point by means of the object name followed by a period.

Static data members can be accessed at any point in the program via the associated class names followed by the two colons. In the case of nested classes, the path to the data item includes all the class names from the outermost to the innermost level, separated by two colons each. The outermost class name requires

qualification appropriate to the scope. If the program is interrupted in a member function of the class, the class scope rules apply for accessing static data members, i.e. if the data member is not hidden by a definition with the same name, it can be accessed without qualification.

If the *dump-area* operand consists of one or more class names, the static data members and all non-virtual member functions are listed. In the case of data, the current content is output. Member functions are listed in standard C++ notation and the start address of the associated prolog is shown. For derived classes, AID also displays the base classes. Nested classes are shown together with the contents of the inner levels.
If the *dump-area* operand ends with an object name, the dynamic data members are also displayed by AID. The currently valid function is assigned to a virtual function name. The same output can be obtained with `%SDUMP *this`, if the program is interrupted in a dynamic member function of the class. To designate a base class in a derived class, you specify the name of the desired base class in the path starting from the object name or from `this->`.
object requires a qualification appropriate to its scope. Only a base qualification is meaningful before `this`.

dataname

This is a data name declared in the source program. *dataname* is specified as in the source program.

You can reference data as in C/C++, but with the following exceptions:
An array name without a subscript addresses all array elements.
Individual array elements can be addressed only via subscripts, not via pointers.
Subscript ranges can also be displayed.
If `%AID C=YES` is set (see page 115), AID combines the array elements of a `char` array that can be addressed via the subscript on the extreme right into C strings and displays the contents of the array in the form of C string literals. For more information on working with arrays, see also the section “Subscript notation” on page 30.

For variables of type `long double`, AID evaluates only the first 8 bytes.
Variables of type `char` are displayed in output type %C. If desired, you can also display the corresponding numeric value by using a type modification (%A or %F; see page 29 for details). The data types `unsigned char` and `signed char`, by contrast, are treated as small integer variables.

If *dataname* designates a pointer to member, the name of the class member currently referenced by the pointer to member appears in the output. Note that you cannot specify a dereferenced pointer to member with %SDUMP. In this case, you must address the desired data item directly via the associated object.

dataname can be specified as follows. The formats can also be combined (see the section “Data names” on page 29)

Subscript notation:	*dataname* [*subscript*] {...}
Pointer notation:	*dataname1* –> *dataname2*
Structure qualification:	*superordinate dataname*• {...} *dataname*
Dereferencing:	[(]*{...} *dataname*[)]

function

This is the name of a function as declared in the source program or the name of a library function. The address of the first instruction of the compiler-generated function prolog is output (see PROC=*function* on page 237 and the chapter "C++-specific addressing" on page 57).

The following syntax is used to address virtual functions:

```
p->n'function(signature)'
```

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p*. (see the description of `this` on page 64 and the section "Virtual functions" on page 73).

%NEST

An AID keyword which outputs the current call hierarchy.
For the lowest hierarchical level AID outputs the source reference of the interrupt point and the number of the block or the name of the function. For higher hierarchical levels, for blocks AID outputs the source reference of the first executable statement after exit from the block and the name of the function containing the block, or the block number of the superordinate block. For functions it outputs the source reference of the function call and the name of the calling function.

AID outputs the current call hierarchy even for programs for which no LSD records exist. Note, however, that if the program is interrupted in the prolog of a library function (e.g. after `%INSERT` *library-function*`->;%R`), the call hierarchy may not include the direct caller of the library function, since AID cannot determine the full call hierarchy until the prolog is traversed and the first executable statement of the function is reached. If an LSD is available for the function, you can use `%TRACE 1 %STMT` to position onto the first executable statement of the function and you can then display the complete call hierarchy with `%SDUMP %NEST`.

Note for users debugging on machine code level:
For C and C++ programs without LSD records, %SDUMP %NEST outputs the addresses of the interrupt point, the CSECT names and the compiler-generated

entry names of the functions that are part of the call hierarchy (see Example 3 on page 247). Addresses, CSECT and entry names can be found in the object listing of your program.

You can only specify a base qualification before %NEST.

medium-a-quantity

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

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

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

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

For more details on *medium-a-quantity,* see the chapter “Medium-a-quantity operand” in the AID Core Manual [1].

TTerminal output
HHardcopy output
(includes terminal output and cannot be combined with *T*)
FnFile output
POutput to SYSLST

MAX	Output with additional information
MIN	Output without additional information
XMAX	Output as with MAX, but extended by the type information: In addition, each data element is preceded by a type tag which defines the type, size and output format of this data element. Syntax of the type tag: `<data-type(memory-size-in-bytes),output-format>`
XFLAT	Output as with XMAX, but with the following restrictions: Only the topmost structure level is output for structured data types. In the case of long data (e.g. long strings or arrays), the first elements are output.

Data types

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

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

size	Storage length in bytes.
int-name	Specifies an element of the type integer.
int-value	Decimal value (D); value of *int-name*.

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

size	Storage length in bytes.
pointer-name	Specifies an element of the type pointer.
pointer-value	Hexadecimal number (X); value of *pointer-name*.

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

size	Storage length in bytes.
float-name	Specifies an element of the type floating point number.
float-value	Floating point number displayed as a decimal fraction with exponent (E); value of *float-name.*

```
<CHARS(1),C>
char-name = |character|
```

char-name	Specifies an element of the type character.
character	The character as a printable character (C); value of *char-name*. A non-printable character is displayed as `|.|`.

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

size	Storage length in bytes.
chars-name	Specifies an element of the type string, in other words an array of the type character.
string	String of printable characters (C); value of *chars-name;* Non-printable characters are displayed as a hexadecimal value. If *string* is longer than 80 characters, with XFLAT only the first 72 characters are output, followed by three periods ... in order to display the incompleteness of the output. See also note 1 at the end of the list.

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

size	Storage length in bytes.

unsign-name	Specifies an element of the type integer without a sign (unsigned).
unsign-value	Decimal value (D); value of *unsign-name*.

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

size	Storage length in bytes.
addr-name	Specifies an element of a relative or absolute storage address.
addr-value	Hexadecimal number (X); value of *addr-name*.

```
<CLASS(size),S>
class-name = class-value
```

size	Storage length in bytes.
class-name	Specifies an element of the type enum.
class-value	Symbolic constant (S), value of *class-name*.

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

size	Primary memory length in bytes.
type	Data type (CHARS, INT, FLOAT,...) if the array consists of a particular data type.
STRUCT	The array has a complex structure consisting of various data types.
array-name	Specifies an element of the type array.
dimension	The dimensions of the array.
(*a1*) *value1* (*a2*) *value2* (*a3*) *value3* ...	*a1*, *a2*, *a3*, ... specifies the subelements of the array, *value1*, *value2*, *value3*, ... and their values. The display of the values depends on the particular data type. With XMAX, all subelements are output. With XFLAT, no subelements are output, see also note 1. For details on array areas, see note 2.

```
<STRUCT(size)>
level struct-name
sub-elements
```

size	Storage length in bytes.
level	Level of embedding of the structure or of a structure element (01, 02, 03, etc.). 01 stands for the topmost level.
struct-name	Specifies an element of the type structure.

sub-elements	Further elements which are contained in the structure. With XMAX, all elements are output. With XFLAT, only some of the elements, see section „Structures with XFLAT“. See also note 1 at the end of the list.

```
<SET(size)>
01 set-name
<UNSIGN(size),D>
 02 class-name1 = class-value1
<UNSIGN(size),D>
 02 class-name2 = class-value2
...
```

size	Particular storage length in bytes.
set-name	Specifies an element of the type set.
class-name1/2... *class-value1/2...*	Names of the symbolic constants. Values of the symbolic constants. With XMAX and XFLAT, in the case of *set-name* all elements are output, with XFLAT without *set-name*, only level 01 is output.

Notes

1. Use the following syntax to query the entire content of a string, structure or array distributed over several lines:

   ```
   %SDUMP name {T | H | Fn | P} = {XMAX | MAX}
   ```

2. Use the following syntax to query the content of the array elements within the particular area:

   ```
   %SDUMP name [from:to] {T | H | Fn | P} = {XMAX | XFLAT | MAX}
   ```

Structures with XFLAT

For structures, AID generates various XFLAT data outputs depending on whether or not the %SDUMP command contains data operands.

- %SDUMP without data operand

  ```
  %SDUMP {T | H | Fn | P} = XFLAT
  ```

  Only the type tag and the name are output (level 01). The output of the structure elements is omitted.

- %SDUMP with a structure as operand

  ```
  %SDUMP structure-name {T | H | Fn | P} = XFLAT
  ```

The structure name and the structure elements are output (level 02). Elements with elementary types are normally output, elements with array type with their name, and elements with structure type only with their name. Each element is preceded by a type tag. The name is extended by a number, the level of embedding.

- %SDUMP with a substructure as operand

  ```
  %SDUMP structure-name.substruct-name {T | H | Fn |P} = XFLAT
  ```

  Also outputs the structure elements of the substructure (level 03)

  Further levels of embedding can also be specified by the other substructure names being chained by a period:

  ```
  structure-name.substruct1-name.substruct2-name.substruct3-name. ....
  ```

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

Examples

1. ```
   %aid c=yes
   %sdump
   ```

   This command requests a symbolic dump of all data in the current call hierarchy. The value for *medium-a-quantity* is T=MAX. The compiler listing for this SDUMP output is given in Example 10 of the %DISPLAY description on page 165.

```
SRC_REF: 46  SOURCE: OUTPUT.C  PROC: main  ****************************
int1            =      -32768

int2            =         234

int3            =        -567

un1             =    65535

un2             =      78900

un3             =      90123

sll             =   -9223372036854775808

ull             =   18446744073709551615

fl1             = +.1234559 E+003

fl2             = +.5678899999999999 E+003

fl3             = +.3334439999999999 E+003
```
```

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 translation unit. This is followed by the variables of type `signed` and `unsigned int` as well as `float` with name and contents.

```
char1           =  |A|

char2           =     -63

chstr           = 01001188

chvek          = "Character array"

c_out           = 01001248
```

AID outputs the char variables `char1` and `char2` in character format, which means that the character corresponding to the contents is output. If you have not used the DELIM operand of the %AID command to define another delimiter, the output uses the vertical bar (for variables of `char` type) or double quotes (for strings).

For the pointer variables `chstr` and `c_out`, the address to which they point is output. Since `%AID C=YES` was set, AID displays the `char` array `chvek` as a C string literal.

```
01          ::std
 02         printf          = 01001E5C

::main            = 01000000
```

The prolog addresses of the `printf` and `main` functions are output.

2. `%sd %nest`

   The current call hierarchy from example 1 is to be output.

```
SRC_REF: 55  SOURCE: OUTPUT.C  PROC: main  ****************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ****************
ABSOLUT: V'10015E8'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ****************
```

   The program was interrupted before the execution of the statement in line 55. The interrupt point lies in the `main` function; the translation unit is `OUTPUT.C`.

3. Comparison of the call hierarchies for symbolic debugging and for debugging on machine code level:

   This example is based on the program `STRING.C` from the section “Sample C++ application in BS2000” on page 308. The program was interrupted at source reference S’50’.

   – Call hierarchy with LSD

```
/%symlib mylib
/%sdump %nest
SRC_REF: 50  SOURCE: STRING.C
                PROC: string::operator const char *() const  ********
SRC_REF: 30  SOURCE: STRING.C
                PROC: string::string(const string &)  ****************
SRC_REF: 72  SOURCE: STRING.C  PROC: main  ************************
ABSOLUT: V'113CF88'     SOURCE: IC@RT20A  PROC: IC@RT20A  ************
ABSOLUT: V'1002A70'     SOURCE: IC@MAIN@  PROC: IC@MAIN@  ************
```

   – Call hierarchy without LSD

```
/%symlib
/%sdump %nest
ABSOLUT: V'10002FE'       SOURCE: STRING$0&@
                PROC: operator const char *  ***********************
ABSOLUT: V'1000816'       SOURCE: STRING$0&@   PROC: string  *********
ABSOLUT: V'1000B1C'       SOURCE: STRING$0&@   PROC: main  ***********
ABSOLUT: V'113CF88'       SOURCE: IC@RT20A  PROC: IC@RT20A  **********
ABSOLUT: V'1002A70'       SOURCE: IC@MAIN@  PROC: IC@MAIN@  **********
```

   In the call hierarchy on machine code level, the CSECT name is inserted as the source instead of the name of the translation unit. AID shows the compiler-generated entry name of the function under PROC instead of the function name from the source program.

4. The following program example contains the recursive function `facul`.

   With `%in s'16:3' <%sd; %sd %nest>` you cause AID to output all data and the functions of the current call hierarchy before leaving `facul` for the last time. First, here is the source error listing:

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27     PAGE:    1
             SOURCENAME:*LIB-ELEM(MYLIB,EXAMP.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________

   EXP   INC    FILE   SRC BLOCK
   LIN   LEV     NO    LIN  LEV

     1     0      0      1     0  #include <stdio.h>
  1746     0      0      2     0  int facul(int n);
  1747     0      0      3     0  int main()
  1748     0      0      4     0  {
  1749     0      0      5     1     unsigned n;
  1750     0      0      6     1     printf("n? : ");
  1751     0      0      7     1     scanf("%d",&n);
  1752     0      0      8     1     if (n > 16) return 0;
  1753     0      0      9     1     printf("%d! : %d\n", n, facul(n));
  1754     0      0     10     1     return 0;
  1755     0      0     11     0  }
  1756     0      0     12     0
  1757     0      0     13     0  int facul(int n)
  1758     0      0     14     0  {
  1759     0      0     15     1     if (n < 0) return (-1);
  1760     0      0     16     1     if (n == 0 || n == 1) return (1);
  1761     0      0     17     1     else return (n * facul(n-1));
  1762     0      0     18     1  }
```

What follows is the command entry (in boldface) and the corresponding output of %SDUMP:

```
/LOAD-PROG *M(MYLIB,EXAMP,RUN-MODE=ADVANCED,PROG-MODE=ANY),TEST-OPT=AID
%  BLS0523 ELEMENT 'EXAMP', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB'
IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXAMP$', VERSION ' ' OF '1999-01-11
12:47:37' LOADED
/%aid low
/%in s'16:3' <%sd; %sd %nest>
/%r
 n? : 4
*** TID: 000401F9 *** TSN: 88G5 ****************************************
SRC_REF: 16:3  SOURCE: EXAMP.C       PROC: facul  **********************
n               =          1

SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  **********************
n               =          2

SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  **********************
n               =          3

SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  **********************
n               =          4

SRC_REF: 9  SOURCE: EXAMP.C       PROC: main  ***************************
n               =          4
```

```
::printf        = 01001B14

::scanf         = 01001B4C

::main          = 01000000
::facul         = 01000180
SRC_REF: 16:3  SOURCE: EXAMP.C       PROC: facul  **********************
SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  **********************
SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  **********************
SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  **********************
SRC_REF: 9  SOURCE: EXAMP.C       PROC: main  ***************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT2OA  PROC: IC@RT2OA  *****************
ABSOLUT: V'10012A8'   SOURCE: IC@MAIN@  PROC: IC@MAIN   *****************
4! : 24
%  CCM0998  CPU TIME USED: 0.0269 SECONDS
```

First, the source reference and the corresponding contents of `n` are output for each call to `facul`. For the last call to `facul`, corresponding here to the highest hierarchical level, AID also outputs for the source reference and for `n` the addresses of the functions `scanf`, `printf` and `main`.
In the output of the current call hierarchy which then follows, you also see for each time `facul` calls itself a line with the same source reference up to the last call of `facul`, the superordinate main function, and the runtime system `IC@RT2OA` and `IC@MAIN@` routines.

5. Example for XMAX and XFLAT

   The following C program is to be debugged:

```
#include <stdio.h>
#include <string.h>
struct Universe
{
  int id;
  struct Galaxy
  {
    int id;
    struct Planet
    {
      int id;
      struct Continent
      {
        int id;
        struct Country
        {
          int id;
          struct Region
          {
            int id;
            struct Dense
            {
              int id;
              struct Forest
              {
                int id;
                struct Anthill
                {
                  int id;
                  struct Ant
                  { float x,y;
                  } ant;
                } anthill;
              } forest;
            } dense;
          } region;
        } country;
      } continent;
    } planet;
  } galaxy;
} universe;

void main(void)
{
```

```
  struct {
     struct {int x; char y;} inner[5][5];
     struct {float u, v;} *pointer;
     union {
            char c100[100];
            char               chx12345678901234567890123456789;
            char unsigned      uch12345678901234567890123456789;
            short              isx12345678901234567890123456789;
            short unsigned     isu12345678901234567890123456789;
            int                ixx12345678901234567890123456789;
            int unsigned       iux12345678901234567890123456789;
            long               ilx12345678901234567890123456789;
            long unsigned      ilu12345678901234567890123456789;
            float              flx12345678901234567890123456789;
            double             dbx12345678901234567890123456789;
           } databox;
     } outer [2];

strcpy(outer[0].databox.c100,
"12345678901234567890123456789012345678901234567890123456789012345678901234567890");

STOP: ;
}
```

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

```
%AID C=YES
%INSERT STOP
%RESUME
```

The following variants show the effect of various specifications for XFLAT and XMAX:

– XFLAT without data operand
– XFLAT for the array element outer
– XFLAT for the databox structure of the array element outer
– XMAX for a sub-array element
– XFLAT for a deeply nested element

XFLAT without data operand

When you specify XFLAT without an operand, only the topmost level 01 is output.

```
/%SD T=XFLAT
SRC_REF: 59  SOURCE: X-C2  PROC: main  *******
<ADDR(4),X>
STOP          = 010000B6

<ARRAY(624),STRUCT>
```

```
outer( 0: 1)

<ADDR(4),X>
::strcpy      = 7DB2BEC8

<ADDR(4),X>
::main        = 01000000

<STRUCT(44)>
01         ::universe
```

XFLAT for the array element outer

```
/%SD outer[0] T=XFLAT
SRC_REF: 59  SOURCE: X-C2  PROC: main  ******
<STRUCT(312)>
01          outer( 0)
<ARRAY(200),STRUCT>
 02          inner( 0: 4, 0: 4)
<POINTER(4),X>
 02          pointer  = 00000000
<STRUCT(104)>
 02          databox
```

XFLAT for the databox structure of the array element outer

Long strings are displayed truncated.

```
/%SD outer[0].databox T=XFLAT
SRC_REF: 59  SOURCE: X-C2  PROC: main
**************************************
<STRUCT(104)>
02          outer.databox( 0)
<CHARS(71),C>
 03         c100  =
"12345678901234567890123456789012345678901234567890123456789012345678901234567890" ...
<CHARS(1),C>
 03         chx12345678901234567890123456789 = |1|
<UNSIGN(1),D>
 03         uch12345678901234567890123456789 = 241
<INT(2),D>
 03         isx12345678901234567890123456789 =  -3598
<UNSIGN(2),D>
 03         isu12345678901234567890123456789 =  61938
<INT(4),D>
 03         ixx12345678901234567890123456789 =  -235736076
<UNSIGN(4),D>
```

```
 03         iux12345678901234567890123456789 = 4059231220
<INT(4),D>
 03         ilx12345678901234567890123456789 =  -235736076
<UNSIGN(4),D>
 03         ilu12345678901234567890123456789 = 4059231220
<FLOAT(4),E>
 03         flx12345678901234567890123456789 = -.9531502 E+059
<FLOAT(8),E>
 03         dbx12345678901234567890123456789 = -.9531502657561182 E+059
```

XMAX for a sub-array element

The element *inner* is itself once again part of an array element.

```
/%SD outer[0].inner[2][2] T=XMAX
SRC_REF: 59  SOURCE: X-C2  PROC: main  ******
<STRUCT(8)>
02         outer.inner( 0, 2, 2)
<INT(4),D>
 03         x         =  0
<CHARS(1),C>
 03         y         = |.|
```

XFLAT for a deeply nested element

```
/%SD universe.galaxy.planet.continent.country.region.dense.forest.anthill.ant T=XMAX
SRC_REF: 59  SOURCE: X-C2  PROC: main  ****************************************
<STRUCT(8)>
02  ::universe.galaxy.planet.continent.country.region.dense.forest.anth ill.ant
<FLOAT(4),E>
 03  x         = +.0000000 E+000
<FLOAT(4),E>
 03  y         = +.0000000 E+000
```

%SET

With the %SET command you transfer the memory contents or AID literals to memory positions in the program which has been loaded. Before transfer, the storage types *sender* and *receiver* are checked for compatibility. The contents of *sender* are matched to the storage type of *receiver*. AID always transfers bytewise.

– With *sender* you designate a variable, a class object or one of its components, a structure or a component of a structure, an array element, a constant, a length, an address, an execution counter, a register or an AID literal.
sender can be located either in the virtual memory of the loaded program or in a dump file.

– With *receiver* you designate a variable, a class object or one of its components, a structure or a component of a structure, an array element, an execution counter or a register which is to be overwritten. *receiver* may only be located in the virtual memory of the program which has been loaded.

Command	Operand
%S[ET]	sender INTO receiver

In contrast to the %MOVE command, for the %SET command AID checks (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.

The following applies if *sender* and *receiver* are pointers to class objects:

– *sender* points to an object of a derived class,

– *receiver* points to an object of a base class of this derived class,

and only the base class entries are transferred. All other entries in the derived class are ignored. AID checks the transfer permissibility in the same way as C++:
if the base class cannot be reached uniquely from the derived class, e.g. if the class referenced via *receiver* is both a direct and indirect base class of the derived class addressed with *sender*, AID rejects the transfer and outputs an error message.

The information on base and derived classes is missing from the LSD for older objects that were compiled with a C/C++ compiler up to V2.2C. With these objects, AID therefore does not know the relationship between the base and derived classes for these objects. Transferring derived classes into base classes via pointer as described above is then not possible.

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.

Immediately after loading, you can reference only global and static data. AID requires the appropriate qualification for access.

In addition to the operand values described here, you can also use those described in the manual for debugging on machine code level [2].

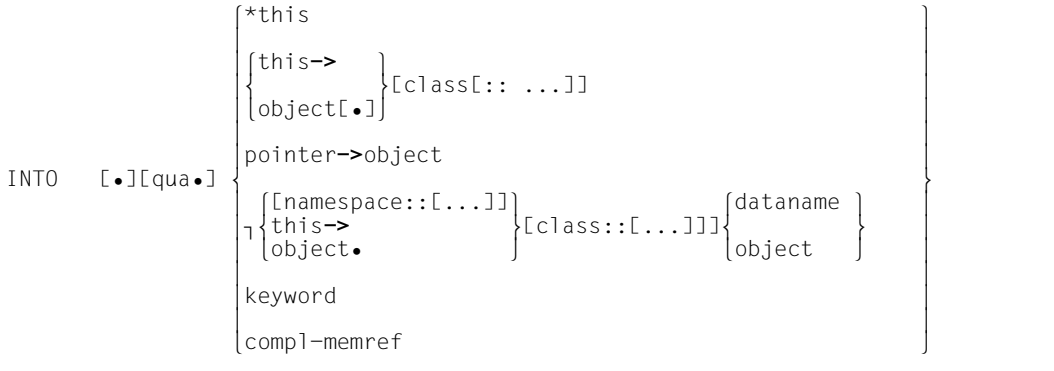
With `%AID CHECK=ALL` you can activate an update dialog; this dialog shows you the old and new contents of *receiver* prior to transfer and offers the option of aborting the %SET command.

The %SET command does not alter the program state.

sender	INTO	receiver

For *sender* or *receiver* you can specify a variable, a class object or one of its components, a structure, a structure component, an array element, a C string (if `%AID C=YES` is set), an execution counter, a register or a complex memory reference. You can use addresses, lengths, constants and literals only as the *sender*.
sender may be located in the virtual memory area of the loaded program or in a dump file. *receiver*, on the other hand, can be only in the virtual memory area of the loaded program. Moving program segments or overwriting them with instruction code may have unwanted side-effects if this affects addresses which are associated with a *control-area* or a *trace-area* or addresses at which a *test-point* has been set with %INSERT (see the section on “Interactions" in the AID Core Manual [1]).


sender-OPERAND
*this
this->
object[•]
[class[:: ...]]
pointer->object
[•][qua•]
[namespace::[...]]
this->
object•
[class::[...]]]
dataname
function
object
L'label'
S'[f-]n[:a]'
keyword
compl-memref
&...
sizeof(...)
%@(...)
%L(...)
%L=(expression)
literal


receiver-OPERAND
INTO
[•][qua•]
*this
this->
object[•]
[class[:: ...]]
pointer->object
[namespace::[...]]
this->
object•
[class::[...]]]
dataname
object
keyword
compl-memref

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

qua One or more qualifications may be specified here if *sender* or *receiver* cannot be reached from the current interrupt point by other means or to address a data name that is locally hidden at the interrupt point by an identically named definition.
It is sufficient to specify only the qualifications needed for a unique address.

{E={VM | Dn} for *sender* | E=VM for *receiver*}
Specified only if the current base qualification (see %BASE) is not to apply to a data name, class, class object, statement name, source reference or keyword. *sender* can be located in virtual memory or in a dump file. *receiver*, on the other hand, can only be in the virtual memory.

S=srcname
Specified only if you are accessing a data name, a namespace, a class or a class object, a statement name or a source reference which is not located in the current translation unit (see the section “Qualifications” on page 21).

:: Use the two prepended colons to address a global data item that is locally hidden at the interrupt point by a definition of the same name. You must also place two colons before the name of a global data item or a function if either the data or the function is not in the call hierarchy or if its definition only occurs after the interrupt point. In contrast to the other qualifications, no period must be entered between the two colons and the operands which follow.

PROC=function
Specified only if you want to address a data name or class object which is defined in the current function, but is hidden at the interrupt point by a definition with the same name. You also give a PROC qualification when you want to reference a label or a data name declared as static which is assigned to a function outside the current call hierarchy (see the chapter “Addressing in C and C++ programs” on page 21). If you specify a source reference which is located in a function template instance or assigned to a function which is defined in a class template instance (see the section “Templates” on page 94), you have to prepend the appropriate PROC qualification if ambiguity occurs.
In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.
Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the function name is prepended with the namespace or class qualification. The `void` signature may no longer be used. In this case, you only input

the two parentheses after the function name, as is also possible in C++. The following syntax results (*f_template* and *signature* are abbreviated below for space reasons):

```
PROC=[namespace::[...]][class::[...]]{n'function([sign])'          }
                                     {t'f_templ<arg[,...]>([sign])'}
```

The `main` and `__STI__` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to insert one or possibly several BLK qualifications (see page 60) between the two PROC qualifications. Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK='[f-]n[:b]'

Specified only when you want to reference a data name which is assigned to a block within the current call hierarchy and is hidden at the interrupt point by a definition with the same name, or when you want to want to reference a data name declared as static and assigned to a block outside the current call hierarchy (see the chapter "Addressing in C and C++ programs" on page 21).

You must also specify a BLK qualification if you want to designate a function from a local class, which is defined in the specified block, in a subsequent PROC qualification (see above, PROC=*function*).

The block name is formed from the line number (*n*), a possible FILE number (*f*) and relative block number (*b*).

NESTLEV= level-number

level-number A level number in the current call hierarchy

level-nummer has to be followed by *dataname*.

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

namespace

Name of a namespace declared in the source program.

You specify the name of a namespace to describe the address path to classes, data or functions defined in the namespace (see the section “Namespaces” on page 85) if the required namespace component is not visible at the interrupt point.

Only the E or S qualification or the two colons (`::`) for the global namespace are allowed before the namespace qualification.

{ class | this-> | object | pointer->object}

Name of a class, the `this` pointer the name of a class object or a pointer to a class object as declared in the source program.
You specify class names, the `this` pointer with the appended pointer operator, and the names of class objects in order to describe the address path to data members (see the section “Classes” on page 63).
If the class is a class template instance, you have to use the following notation: `t'`*k_template*`<`*arg*`[,...]>'`. If only one instance of the template exists, only `t'`*k_template*`'` is required.

You use the names of objects, or pointers to class objects to transfer class objects as a whole. This is only possible under the following conditions:

1. *sender* and *receiver* reference objects of the same class. If the program is interrupted in a dynamic member of the class, you can also access the complete class object with `*this`.
2. *sender* is a pointer to a derived class object and *receiver* points to an object of an associated base class. This transfer therefore corresponds to the C++ assignment statement:
   `pointer_a = pointer_b;`
   if `pointer_a` is a pointer to a base class object and `pointer_b` is a pointer to an object of a derived class of this base class.

AID transfers the dynamic data part of the class object and, if available, compiler-generated auxiliary variables and the address of the virtual functions table. Each component is transferred according to its memory type. Appropriate qualification is required if *object* or *pointer* are not visible at the interrupt point. Only a base qualification is meaningful before `this`.
With derived classes, *sender* or *receiver* also includes the base classes as well as the inner levels of nested classes. If *receiver* refers to a base class object of *sender*, only the base class data is transferred.

Static data members can only be addressed individually. They can be reached via the associated class name with the two subsequent colons from any part of the program. In the case of nested classes, the path to the data item includes all class names from the outer to inner levels, each separated by two colons. The outermost

class name requires qualification corresponding to the scope. If the program is interrupted in a member function of the class, the class scope rules apply for accessing static data members, i.e. if the data is not hidden by a definition with the same name, it can be accessed without qualification.

If the current interrupt point is located in a dynamic member function, you can access the data member in exactly the same way as in C++, i.e. if the data item is visible at the interrupt point, you can access it with AID directly without qualification. As in C++, locally hidden data requires appropriate class qualification. You can also access dynamic data members via the `this` pointer with a subsequent pointer operator. This is the same as using the object name followed by a period.

You can access dynamic data of an object, independent of the interrupt point, via the object name and a subsequent period if the object is located in the current call hierarchy.

If *sender* or *receiver* ends on `this`, the %SET will transfer or overwrite the start address of the object that `this` points to, as the start address is recorded in `this`. With a following pointer operator, i.e. with `this->`, you designate the first 4 bytes from the start address of the current object in storage type %X.

dataname

This is a data name declared in the source program. *dataname* is specified as in the source program.

You can reference data as in C/C++, but with the following exceptions:
Array elements can be referenced only via subscripts, not via pointers.
The specification of a subscript range is not allowed.
C strings are recognized by AID only if `%AID C=YES` is set (see page 115). You can then modify C string arrays as you normally would in the C/C++ language (see also the section on "C strings" on page 36).
If `%AID C=NO` is set or if arrays of some other data types are involved instead of C strings, then you cannot transfer or overwrite arrays as a whole. If you specify an array name without a subscript, AID will reject the transfer.
One exception is when using the names of arrays as passed parameters. Note, however, that the name of the passed parameter does not refer to the array itself, but only its address.
For more information on working with arrays, see also the section "Subscript notation" on page 30.

For variables of type `long double`, AID evaluates only the first 8 bytes.
Variables of type `char` are always treated as characters. If desired, you may also use the corresponding numeric value, but only after a type modification with %F (`signed char`) or %A (`unsigned char`). The data types `unsigned char` and `signed char`, by contrast, are treated as integer variables.

If you specify a data name of type pointer to member as the *receiver*, then the *sender* could also be a pointer to member, or you could specify the address of a data member or a member function of a class as the *sender*. The following condition must be satisfied when modifying a pointer to member:
The class associated with the sender must match the class referenced by the receiver or must be the unique base class in the class of the receiver (see also the section “Pointer to class member” on page 74).

You can specify *dataname* as follows and can also combine these formats (see the section “Data names” on page 29):

Subscript notation:	*dataname* `[`*subscript*`] {...}`
Pointer notation:	*dataname1* `->` *dataname2*
Structure qualification:	*superordinate dataname*`.` `{...}` *dataname*
Dereferencing:	`[(]*{...}` *dataname*`[)]`
Pointer to member dereferencing:	*dataname1*`.*`*datannname2* `or` *dataname1*`->*`*datannname2*

Structures can be transferred using %SET only if *sender* and *receiver* are defined as structures and the definitions of the components match.

Pointers can be set to binary zero with `%SET 0 INTO` *pointer*.

```
{function    }
{L'label'    }
{S'[f-]n[:a]'}
```

Statement names and source references are address constants. They can only be specified as *sender*. The address held in the address constant is transferred.
A following pointer operator (->) designates 4 bytes of the machine code located at the corresponding address. You can use %DISASSEMBLE to output the machine instructions in preparation for a length modification, should one be necessary.
funktion`[%a14]->`, `L'`*label*`'->` and `S'`*[f-]n[:a]*`'->` can be used as *sender* and as *receiver* (see Example 9 on page 271 following the %SET description).

function

This is the name of a function, as declared in the source program, or the name of a library function. It references the start address of the function prolog that is generated by the compiler (see PROC=*function* on page 257 and the chapter “C++-specific addressing” on page 57).

Virtual functions are addressed with the following syntax:
`p->n'function(signature)'`

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in

the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p* (see the description of `this` on page 64 and the section “Virtual functions” on page 73).

If you want to access a function addressed via a pointer to member, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

```
[qua.] object.*[object.][class::][...]pointer-to-function-member
```

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows**:**

```
[qua.] pointer->*[object.][class::][...]pointer-to-function-member
```

The class object is designated by the operand on the left of the dereferencing operator `.*` or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.
The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to function member cannot be reached from the interrupt point by some other means.

If you want to transfer or overwrite the instruction code of a function addressed via a pointer to member with %SET, note that you cannot directly append the pointer operator to one of the syntaxes above. You would have to first switch to machine code level with a type modification, i.e. %al4, by using the following syntax:

dereferenced-pointer-to-function-member `%al4->`

This designates the first 4 bytes of the instruction code located at the prolog address.

More details on working with a pointer to function member can be found on page 79.

L'label'
: This designates the address of the first executable statement after a label. *label* is the name of a label as declared in the source program.

S'[f-]n[:a]'
: is a source reference and designates the address of an executable statement. It is constructed from the line number (*n*) and, if present, the FILE number (*f*) and the relative statement number (*a*).
If the source reference is located in a function which was created from a function template via instantiation or the function containing the source reference is defined in a class template instance, you have to prepend the appropriate PROC qualification to the source reference if ambiguity occurs.

keyword
: is an execution counter, the program counter or a register. Only a base qualification can be specified before *keyword*.
The implicit storage types of the keywords are listed in the chapter on "Keywords" in the AID Core Manual [1].

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

The program counter holds the address at which the program is to continue with %CONTINUE, %RESUME or %TRACE. You can define another continuation address by overwriting the program counter (%PC). However, you must then make sure yourself that register contents, file status, contents of subscripts and so on are appropriate for the new continuation address so that the program can be continued without error.

compl-memref
: The following operations may occur in *compl-memref* (see the section on "Complex memory references" in the AID Core Manual [1]):
– byte offset (•)
– indirect addressing (->)
– type modification (%T(dataname), %X, %C, %E, %D, %F, %A, %S, %SX)
– length modification (%L(...), %L=(expression), %Ln)
– address selection (%@(...))

The storage types of *sender* and *receiver* can be matched by means of an explicit type or length modification. Note, however, that if the storage type conflicts with the memory contents, it will be rejected by AID even if a type modification is specified.

If *compl-memref* begins with an address constant (such as a source reference or a label), the pointer operator (->) must come next. Labels must always be placed within `L'...'` in such cases. Without the pointer operator, address constants can be used in *compl-memref* wherever hexadecimal numbers are also allowed.
With an explicit type or length modification you can match the storage type for *sender* to that of *receiver*. Memory contents which are incompatible with the storage type will be rejected by AID even if a type modification is performed.

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.
The assigned memory area for any operand in a complex memory reference must not be exceeded as the result of byte offset or length modification; otherwise AID will reject the command and issue an error message. By combining address selection (%@) and pointer operator (->) you may exit from the symbolic level. You can then use the address of a data item without considering its area limits.

& is the address operator. You can use it to define the start address of a data item, class object or function as the *sender*. You can transfer an address only to a *receiver* of type pointer, but the types associated with the *sender* and *receiver* need not match.

Note, however, that if the address of a class object is to be transferred to a pointer to a class, AID performs the following check:

– The class to which the receiver points must match the class whose address is to be transferred

  or

– it must be the base class of the class assigned to the sender. This base class must have a unique subobject in the class of the sender.

The address operator & can also be used to determine the relative address of a dynamic data member of a class, provided you observe the following:

– If the interrupt point is located outside the class containing the data member, you should enter the appropriate class qualification after the address operator, and then the name of the data item.

– If the interrupt point is located in a dynamic member function of the class, you will need to enter a base or area qualification (S, PROC or :: qualification) before the address operator so that AID can access the class from “outside”, so to speak.

Even the relative address, i.e., the offset of the data member to the start of the class in bytes, can only be transferred to a pointer.
Note that in contrast to the address selector %@(...) (see page 265), the address operator is purely a “high-level” function and thus cannot be applied on complex memory references.

More detailed information on the address operator can be found in the section “The address operator & and the address selector %@(...)” on page 42.

sizeof()

is the length operator. The length of a data item or class is transferred.
To determine the length of a class, you may specify the name of the class itself or the name of a class object as operands. You will receive the number of bytes occupied by the dynamic data members of the class and by the auxiliary variables generated by the compiler (if any).
You may specify the name of a namespace here, but only in the path to a component of the namespace.
Bit-field and register variables are not allowed.

The length operator is described in detail in the section “Length operator sizeof() and length selector %L(...)” on page 47.

%@(...)

The address selector (see the section on “Address, type, and length selectors” in the AID Core Manual [1]) enables you to use the start address of a data item, of a class object, or of a complex memory reference as *sender*. You can only specify a class name in the path to the base class of an object of a derived class to identify the start address of the dynamic data of the base class.
You can only specify the name of a namespace here in the path to a component of the namespace.

The address selector cannot be applied to constants, including labels, source references and functions.

%L(...)

The length selector allows you to use the length of a data item or of a class as *sender* (see the section on “Complex memory references” in the AID Core Manual [1]). If you apply the length selector to a class or a class object, the result corresponds to that of sizeof() in C++, i.e. you receive the length of the dynamic data and of the compiler-generated auxiliary variables, if any.
You can only specify the name of a namespace here in the path to a component of the namespace.

AID always outputs the length in bytes. For bit-fields, the number of bytes over which the bit-field extends is returned as the length.
Example: `%set %l(var1) into %3g`
The length of `var1` will be transferred.

%L=(expression)

The length function enables you to calculate a value and store it in *receiver*. *expression* is formed from the contents of memory, constants and integers together with the arithmetic operators (+,-,*,/). Memory references must be of type %F or %A (integers).

The length function returns an integer (see the section on “Address, type and length selectors” in the AID Core Manual [1]).

When using overloaded operators, note that AID does not emulate this process, but always uses standard operators.

Example: `%set %l=(var1) into %3g`
The content of `var1` is transferred if it is a whole number (data type int). Otherwise, AID issues an error message.

literal

All AID literals described in the chapter on “AID literals” in the AID Core Manual [1] may be specified with %SET. Take note of the possibilities described in that chapter for converting AID literals to the respective receiver type:

```
{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                          Fixed point number
[{?}]mantissaE[{?}]exponent       Floating-point number
```

If `%AID C=YES` is set, you can also transfer a C string literal ("*x...x*") to a character array (see Example 10 on page 271 following the %SET description).

%SET table

The following table shows an overview of the combinations allowed for sender and receiver types in conjunction with the %SET command.

Sender	Receiver				
	int, float %F %A %D	char %C	%X	Pointer	C strings (%AID C=YES)
int, float %F %A %D {±}n	num	–	bin	–	–
#'f...f'	num	–	bin	bin	–
{±}n.m {±}mantE{±}exp	num	–	–	–	–
char %C C'x...x'	num(1)	char	bin	–	–
%X X'f...f' B'b...b'	bin	bin	bin	bin	bin
pointer, address	–	–	bin	bin	–
C strings, C string literal (%AID C=YES)	–	–	–	–	char(1)

Table 6: Permitted combinations of sender and receiver types

bin Binary transfer; left-justified;
sender < *receiver*: padding with binary zeros on the right.
sender > *receiver*: truncation on the right.

For transfer to storage type %X, a numeric literal (only integer permitted) corresponds to a signed integer with length of 4 bytes (%FL4) transferred in binary form.

char Character transfer; left-justified;
sender < *receiver*: padded on the right with blanks (X'40').
sender > *receiver*: truncated on the right.

char(1) Character transfer; left-justified;
sender < *receiver*: padding with binary zeros on the right.
sender > *receiver*: truncation on the right.

num Numeric transfer; retains value;
sender is converted where necessary to the storage type of *receiver*.

num(1) If a character literal which contains only numbers and is at most 18 digits long is specified as the sender, and if the receiver is of type numeric, AID performs a numeric conversion. All other senders of type character cannot be converted to numeric receivers.

\- No transfer;
AID reports the incompatibility of the storage types.

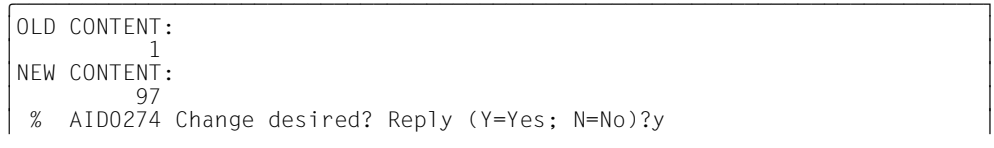
Examples

In a C program the following variables, arrays and structures are defined:

```
C program
=========================================
         unsigned short  count;
         float           x_arr[16];
         struct tele {
            char         name[10];
            unsigned     number;
            }
        struct tele      person_1;
        struct tele      person_2;
        char             c_arr[10];
        int              i;
=========================================
```

In the following examples, `%aid check=all` was used to enable the update dialog. You are shown the content of the receiver field before and after %SET is executed:

1. `%set #'61' into count`



The following command has the same result:
`%set 97 into count`

2. `%qualify proc=main`
   `%set .count into .x_arr[15]`

```
OLD CONTENT:
+.1234499 E+003
NEW CONTENT:
+.9700000 E+002
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

3. `%s 'A' into person_1.name[0]`

```
OLD CONTENT:
|T|
NEW CONTENT:
|A|
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

4. `%s 'ABCDEF' into %@(person-1.name)->%cl6`

```
OLD CONTENT:
|uvwxyz|
NEW CONTENT:
|ABCDEF|
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

5. `%set person_1 into person_2`

```
OLD CONTENT:
01          person_2
 02         name( 0: 9)
            ( 0) |H|  ( 1) |u|  ( 2) |b|  ( 3) |e|  ( 4) |r|  ( 5) |.|
            ( 6) |.|  ( 7) |.|  ( 8) |.|  ( 9) |.|
 02         number          =       4444
NEW CONTENT:
01          person_2
 02         name( 0: 9)
            ( 0) |M|  ( 1) |a|  ( 2) |i|  ( 3) |e|  ( 4) |r|  ( 5) |.|
            ( 6) |.|  ( 7) |.|  ( 8) |.|  ( 9) |.|
 02         number          =      12345
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

6. `%set 123.45 into count`

```
I390 WARNING: SOURCE TRUNCATED
OLD CONTENT:
    9876
NEW CONTENT:
     123
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

7. Interpreting a `char` as an `unsigned` and `signed integer`

```
%s x'ff' into c_arr[0]
%s c_arr[0]%a into i
%s c_arr[0]%f into i
```

– First %SET command

```
OLD CONTENT:
00   .
NEW CONTENT:
FF   ~
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

– Second %SET command

```
OLD CONTENT:
          0
NEW CONTENT:
        255
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

– Third %SET command

```
OLD CONTENT:
        255
NEW CONTENT:
         -1
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

8. `%set Y::n'f()' into %2g`

The prolog address of member function `Y::f()` is written to AID register %2G.

9.
```
%da 5 from s'12'->
%set s'12'->%l2 into %3g
```

   With the %DISASSEMBLE command you have two machine instructions disassembled beginning at the source reference S’12’ . The first instruction is a 2-byte instruction. This first instruction is transferred with %SET to AID register %3G.

10.
```
%aid c=yes
%set "HELLO" into c_arr
```

    The command `%AID C=YES` enables the handling of `char` arrays as C strings. You can then overwrite the `char` array `c_arr` with the C string literal `"HELLO"`. The remaining bytes of `c_arr` are filled with binary zeros.

%SHOW

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.

Command	Operand
%SH[OW]	[show-target]

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.

show-target

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*.

Command or subcommand	Information displayed
%AID	The currently valid settings for the %AID, %AINT and %BASE commands and the version of AID loaded.
%ALIAS	List of all defined alias names and their original names.
%BASE	The current settings for %BASE, %AINT and %SYMLIB, the TSN, TID and the version of the operating system and type of computer.
%C[ONTROL]	The input string for each registered %CONTROLn.

Table 7: Operand values of the %SHOW command and the information displayed

Command or subcommand	Information displayed
%D[IS]A[SSEMBLE]	The current *number* and *start* address (V'...').
%F[IND]	The entered command and, if appropriate, the virtual address of the last hit.
%IN[SERT] [test-point]	Without the *test-point* entry, all active test points are output. Otherwise, AID shows the entered command in which *test-point* was declared. Note that AID cannot usually associate the prolog address with the corresponding function. Consequently, AID may display the name of an earlier function if function addresses which are listed by AID for a %DISPLAY {*namespace* | *object*| *class*} or in an %SDUMP output are used in an %INSERT for the sake of simplicity in the case of functions from C++ programs, which usually have very long names.
%ON	The input string for each active %ON command.
%OUT	The valid *medium-a-quantity* values for the commands that can be controlled via %OUT.
%OUTFILE	All implicitly or explicitly registered output files, with their link names.
%QUALIFY	The last %QUALIFY command entered.
%SYMLIB	The registered libraries with the associated base qualifications and the TSN.
%TRACE	The %TRACE parameters set on activation of the %TRACE are output. Default values are supplied by AID for the operands not explicitly specified. The translation unit in which the %TRACE is executed is specified, and account is taken of whether the last %TRACE was symbolic or on machine code level. In trailing lines AID shows how many instructions or statements have already been processed with the current %TRACE and what the input string of the last %TRACE command looked like.
%.*	The names of all active subcommands with the type of the AID command in which they were defined.
%.subcmdname	The command in which subcmdname was defined.

Table 7: Operand values of the %SHOW command and the information displayed

Example

```
$debug examp
%  AID0348 Program stopped due to EXEC event (PID=0000000891)
%0000000891/%aid c=yes
%0000000891/%show %aid
A I D   V03.4B11 OF  2016-03-17
Copyright (C) Fujitsu Technology Solutions 2016
All Rights Reserved

E=VM : %AINT = %MODE31

%AID CHECK      =   NO
%AID REP        =   NO
%AID SYMCHARS   =   NOSTD
%AID OV         =   NO
%AID LOW        =   ALL
%AID DELIM      =   '|'
%AID LANG       =   D
%AID FORK       =   NOT_USED
%AID EXEC       =   ON
%AID C          =   YES
%AID EBCDIC     =   EDF03IRV
%AID CCS        =   EDF03IRV
```

After loading the program under POSIX with `debug`, the command %AID was first issued with the option `C=YES`. This causes AID to accept C string literals enclosed within "" and to interpret `char` arrays as C strings. In addition, the conversion of lowercase letters to uppercase is disabled even for entries in the S qualification. The next command, `%SHOW %AID`, requests a listing of the currently applicable settings for %AID. The output of the %SHOW command shows the default settings of the operand values for %AID, with the following
exceptions:

– SYMCHARS was set implicitly to NOSTD on setting `C=YES`, which means that the hyphen (-) will always be interpreted as a minus character, since hyphens are not allowed in names in C/C++ programs in any case.

– LOW was also set implicitly to ALL by `%AID C=YES`, which means that the conversion of lowercase letters to uppercase is disabled even for entries in the S qualification.

– `EXEC` is always enabled in the POSIX shell on loading a program with `debug`.

– `C=YES` was set explicitly.

In the case of `%AID FORK`, the value `NOT_USED` is equivalent to the setting `OFF`; `NOT_USED` simply indicates that the `FORK` option was not set in this task.

%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 at what statement, in which translation unit, and in which function or block 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.

Under POSIX, you can use %STOP to interrupt a task created via `fork()` to check the further progress of this task using AID commands. A task created via `fork()` can be interrupted with the T=*tsn* (Task Sequence Number) and PID=*pid* (Process Identification) operands. AID reports the process number (*pid)* of the interrupted task and you can then check the further progress of the task using AID commands.

– *T=tsn* designates the task sequence number (TSN) of the task which AID is to interrupt.
– *PID=pid* designates the process identification (pid) of the task which AID is to interrupt.

Command	Operand
%STOP	[{ T=tsn | PID=pid }]

If the %STOP command is contained in a command sequence or subcommand, any commands following it will not be executed.
If you used %BASE to set a dump file as the base qualification and then enter a %STOP command, AID outputs a STOP message containing localization information on the address at which the program was interrupted as the dump was written.
If the program has been interrupted by pressing the K2 key or via a %STOP command, the program interrupt point need not necessarily be within the user program; it may also be located in the runtime system routines. To access programs functions and variables without having to specify the complete qualification each time, it is advisable to initially let the program continue running to the next executable statement with `%TRACE 1 IN S=`*srcname*.

The %STOP command alters the program state.

A %STOP in a subcommand always refers to the loaded program.

T

tsn The fork task, which is to be set to debug mode, is addressed via its TSN.

PID

pid The fork task, which is to be set to debug mode, is addressed via its PID.

Examples

1.

```
/%in s'10' <%display abc_arr; %stop>
/%resume

abc_arr( 0: 26)
(  0) |A|  (  1) |B|  (  2) |C|  (  3) |D|  (  4) |E|  (  5) |F|  (  6) |G|
(  7) |H|  (  8) |I|  (  9) |J|  ( 10) |K|  ( 11) |L|  ( 12) |M|  ( 13) |N|
( 14) |O|  ( 15) |P|  ( 16) |Q|  ( 17) |R|  ( 18) |S|  ( 19) |T|  ( 20) |U|
( 21) |V|  ( 22) |W|  ( 23) |X|  ( 24) |Y|  ( 25) |Z|  ( 26) |.|
STOPPED AT SRC REF: 10 ,  SOURCE: EXAMP.C ,  PROC: main
```

%INSERT sets a test point on the first statement in line 10. The subcommand contains the %DISPLAY and %STOP commands. After `abc_arr` has been output, AID halts the program and writes a STOP message indicating the source reference, translation unit and function of the current interrupt point.

2.

```
$debug exstop
%  AID0348 Program stopped due to EXEC event (PID=0000000876)
%0000000876/%aid fork=next
%0000000876/%aid low=all
%0000000876/...
%0000000876/%resume
%  AID0348 Program stopped due to FORK event (PID=0000000877)
%0000000877/...
%0000000877/%stop pid=876
%  AID0492 %STOP was sent to fork task (PID=0000000876)
%0000000877/<EM><DÜ>
%  AID0348 Program stopped due to STOP event (PID=0000000876)
%0000000876/%trace 1 in s=n'exstop.c'
%0000000877/<EM><DÜ>
45                      BLOCK END, LOOP END
STOPPED AT SRC_REF: 45, SOURCE: exstop.c , BLK: 39 , END OF TRACE
%0000000876/%display count
%0000000877/<EM><DÜ>
*** TID: 003400D1 *** TSN: OEUV ***************************************
SRC_REF: 45 SOURCE: exstop.c  BLK : 39  *******************************
count           =         933
```

After loading the program with the POSIX `debug` command, `%AID FORK=NEXT` is used to specify that the fork task created by `exstop` is also to run in debug mode. `%AID LOW=ALL` is also set, otherwise the name of the source file `exstop.c` would be converted into uppercase in the S qualification.

The parent task runs under `pid` 876 and the child task under `pid` 877. The parent task is interrupted with the `%STOP PID=876` command. AID reports back with the prompt `%0000000876/`. You stop the parent task before the next executable statement with the subsequent %TRACE command. You can now access the parent task variables without qualification. Since both tasks are now competing for the terminal, you have to respond to the prompt of the unwanted task with EM DÜ to allow the task you want to debug to report on the terminal.

%SYMLIB

With the %SYMLIB command you direct AID to open or close PLAM libraries. AID accesses open PLAM libraries if in a command you address symbolic memory references located in a translation unit for which no LSD records have been loaded.

– By means of *qualification-a-lib* you open or close one or more libraries in which object modules and their associated LSD records are stored. In order to dynamically load LSD records, any library can be assigned to the current program or to a dump file by specifying the appropriate base qualification.

Command	Operand
%SYMLIB	[qualification-a-lib][,...]

When this command is executed, AID checks only whether the specified library can be opened; it does not check whether the contents of the library match the program being processed. Thus it is possible to initially open all libraries which you might need later during a debug run. AID does not check whether the object module (OM) or the link and load module (LLM) of the program which has been addressed matches that of the PLAM library until the dynamically loaded LSD records are accessed.
If several libraries have been opened for a base qualification, AID scans them in the order in which they were specified in the %SYMLIB command.
If the AID search is not successful or if no library is open, you may assign the correct library by way of a new %SYMLIB command after the corresponding message has been issued. You then repeat the command for whose execution the LSD records were lacking.

A library remains open until it is released by:
– a new %SYMLIB command for the same base
– a %SYMLIB command without an operand
– a %DUMPFILE command with which the dump file assigned to *Dn* is closed,
– a /LOGOFF or /EXIT-JOB.

You can also not access a library registered in the parent task, in a task created via a `fork()` call.

If a new command contains new file names, these libraries are assigned and opened.

The %SYMLIB command does not alter the program state.

qualification-a-lib

is a base qualification and/or the file name of a PLAM library.

– If you enter a base qualification and a file name, AID assigns the specified library for this base qualification and opens it. Previously assigned libraries for the same base qualification are closed.
– If you specify a file name only, AID assigns the library for the base qualification which is currently applicable (see %BASE command) and opens it. All libraries previously assigned for the current base qualification will be closed.
– If you specify a base qualification only, all open libraries for this qualification will be closed.

AID can handle up to 15 library assignments. A library which is concurrently assigned for several base qualifications is counted as often as it is specified.

```
qualification-a-lib-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                 ⎧VM ⎫
[•][E=           ⎨   ⎬•][filename]
                 ⎩Dn ⎭

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

- If the period is in the leading position it denotes a *prequalification*, which must have been defined with a preceding %QUALIFY command and can only stand for a base qualification.

E=VM
: %SYMLIB applies for the loaded program (see also %BASE command).

E=Dn
: %SYMLIB applies for a memory dump in a dump file with the link name *Dn* (see %BASE, %DUMPFILE).

filename
: 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.mylib,out.cpp
```

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 library `MYLIB`.
The library `OUT.CPP` is assigned for the currently set base qualification. If no %BASE command has been issued, AID uses this library to dynamically load LSD records for the program being executed.

%TITLE

With the %TITLE command you define the text of your own page header. AID uses this text when the %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP and %TRACE commands write to the system file SYSLST.

– By means of the *page-header* operand you specify the text of the header and direct AID to set the page counter to 1 and to position SYSLST to the top of the page before the next line to be printed.

Command	Operand
%TITLE	[page-header]

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

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

The %TITLE command does not alter the program state.

page-header

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

page-header

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

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

%TRACE

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

– By means of the *number* operand you can specify the maximum number of statements to be traced, i.e. logged prior to execution.

– By means of the *continue* operand you control whether the program halts after the %TRACE terminates (default) or continues running without logging.

– By means of the *criterion* operand you select different types of program statements which AID is to log. Logging takes place prior to execution of the statements selected.

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

Command	Operand
%T[RACE]	[number] [continue] [criterion][,...] [IN trace-area]

A %TRACE cannot be active in conjunction with a *write-event* of %ON.

If program execution is interrupted while a %TRACE is running, the %TRACE can be reactivated with %CONTINUE. This applies in the following situations:

– A subcommand containing a %STOP command has been executed.

– The K2 key has been pressed (see page 19).

A %TRACE command is terminated by any of 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.

– A %RESUME command is issued after one of the above program interrupts.

– A `fork()` or `exec()` call was executed.

– The end of the program was reached.

The operand values of a %TRACE command apply until they are overwritten by the entries in a subsequent %TRACE command, until a `fork()` or `exec()` call is executed 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 uses the default values:
– 10 is set for *number*
– `S` is set for *continue*
– `%STMT` is inserted for *criterion*
– the translation unit containing the current interrupt point is set 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.

If you activate tracing in a C++ program prolog immediately after program loading or in the epilog after `main` has terminated, it may not always be possible to assign the source references and the associated statement types in the %TRACE log to the corresponding source program statements in the source error listing.

trace-area can only be located in the loaded program, therefore the base qualification E=VM must have been set (see the description of the command %BASE) or must be specified explicitly.

The %TRACE command alters the program state.

number

Specifies the maximum number of program 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, the program is either halted or continued without logging, depending on the value of the *continue* operand. If `S` is set for *continue*, AID outputs a message to SYSOUT containing information on which statement, in which translation unit, and in which function or block the program was halted.

continue

Defines whether AID is to halt or continue program execution after the %TRACE terminates. 'continue' applies until a different operand value for it is entered in a new %TRACE or until the program terminates.

```
continue=OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{S | R}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

S The program is halted. AID issues a STOP message containing the localization information about the interrupt point. S is the default value.

R The program is continued without a message being issued.

criterion

Keyword which defines the type of statements to be traced during program execution. Several keywords can be specified at a time; they take effect simultaneously. A comma must be used to separate any two keywords.
If no *criterion* is declared, AID uses the default value %STMT unless a *criterion* declaration from an earlier %TRACE command is still valid.

criterion	**Output of log prior to**
%STMT	Every statement that is executed
%ASSGN	Every assignment statement
%CALL	Every function call
%COND	Every `if` and `switch` statement, every `else` branch of the `if` statement and every control expression of the `do`, `while` or `for` statement
%EH %EXCEPTION	Every catch and throw statement
%GOTO	Every goto, break and continue statement
%LAB	Every statement with a label, but does not apply to `case` and `default` labels
%PROC	The first and the last statement of a function

Table 8: Values of the *criterion* operand and their meanings

trace-area

Defines the program area in which tracing is to take place, i.e. only within this area can monitoring and logging of the statements selected by means of the *criterion* operand be effected. The %TRACE command is inactive outside of this area and is activated again only on returning to this area. *trace-area* is only valid if it is located in the loaded program; and if you specify a translation unit, that unit must be loaded at the time when you enter the %TRACE command or when the subcommand containing the %TRACE command is executed.

A *trace-area* definition 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, until a `fork()` or `exec()` call is executed 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 translation unit containing the current interrupt point.

The continuation address for program execution cannot be influenced by the %TRACE command. You can define another continuation address only by using %SET to change the program counter (%PC) (see the description of the command %SET *keyword on* page 263).

```
trace-area-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                  ⎧S=srcname                                                ⎫
                  ⎪                                                         ⎪
IN  [•][E=VM•] ⎨                ⎧[qua•][PROC=]function                    ⎫⎬
                  ⎪[S=srcname•] ⎨BLK='[f-]n[:b]'                          ⎬⎪
                  ⎪                ⎩([PROC=function•]src-ref:src-ref)      ⎭⎪
                  ⎩                                                         ⎭

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

- 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* 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 %BASE).

S=srcname

Specified only if *trace-area* is not to be in the current translation unit or if a defined area limit is no longer to apply. If *trace-area* ends with an `S` qualification, it encompasses the complete specified translation unit.

[qua•][PROC=]function

trace-area is defined by a PROC qualification and includes the entire specified function.
In the case of functions from C programs, *function* is the function name declared in the source program, but without the parentheses or signature.
Functions from C++ programs must be specified in `n'...'` or `t'...'` notation, depending on their type. If the function is defined in a namespace or class, the function name is prepended with the namespace or class qualification.
The `void` signature may no longer be used. In this case, you only enter the two parentheses after the function name, as is also possible in C++. The following syntax results for *function*:

```
[namespace::[...]][class::[...]] { n'function([signature])'                  }
                                 { t'f_template<arg[,...]>([signature])'     }
```

The `main` and `__STI__` functions and all functions with C linkage constitute an exception; these functions are identified only by the function name even when debugging within C++ programs (see page 58).

Syntax for virtual functions:

`p->n'function([signature])'`

p is a pointer variable that points to the class object containing the desired member function. If *p* cannot be accessed from the current interrupt point, it must be qualified in accordance with its scope. If the interrupt point is located in the virtual function itself, you can reference the prolog address of the current function by using the `this` pointer instead of *p*. (see the description of `this` on page 64 and the section "Virtual functions" on page 73).

If you want to specify a function via a pointer to member as the *trace-area*, you can use one of the following two methods:

You designate the class object by name and enter `.*` as the dereferencing operator as follows:

```
[qua.]object.*[object.][class::][...]pointer-to-function-member
```

You address the class object via a pointer and enter **->*** as the dereferencing operator as follows:

```
[qua•]pointer->*[object•][class::][...]pointer-to-function-member
```

The class object is designated by the operand on the left of the dereferencing operator •* or **->***:
object designates the class object by name;
pointer addresses the object via a pointer.
The name of the pointer to function member must be entered on the right of the dereferencing operator. This may need to be preceded by the object containing the definition of the pointer to function member and the class qualification needed for unique addressing within the object if the pointer to member cannot be reached from the interrupt point by some other means. More details on working with a pointer to function member can be found on page 79.

qua

If the function is defined in a local class, before the PROC qualification you have to add an additional PROC qualification for the superordinate function containing the definition of the local class. With functions defined in an inner block of the superordinate function, you have to append one or possibly several BLK qualifications, each separated by a period, to the superordinate function PROC qualification to describe the path to the local class (see page 60).

Syntax for *qua*:

```
PROC=superordinate_function[BLK='[f-]n[:b]'•[...]]
```

Accessing functions defined in inner blocks of local classes is only supported with programs that were compiled with a C/C++ compiler as of V3.0B.

BLK='[f-]n[:b]'

trace-area is defined by a BLK qualification and includes the entire specified block. Block names are constructed from the line number (*n*) and, if appropriate, the FILE number (*f*) and the relative block number (*b*).

The BLK qualification cannot be used together with the %PROC *criterion*.

([PROC=function•]src-ref : src-ref)

Source references let you define *trace-area* by specifying a start and end address. Both must lie within the same translation unit, and start address must be ≤ end address.
Note that ascending source references are only assigned ascending addresses within a function block. If the condition start address ≤ end address is not satisfied, AID rejects the command with a corresponding error message.

Furthermore, note that additional source references, which do not appear in the source error listing but are logged by %TRACE, are generated in connection with implicit constructor and destructor calls as well as conversion operations in C++ programs.

If *trace-area* is to include only one statement, the start and end addresses must be identical.

PROC=function•

You only have to write the PROC qualification if the specified source references in the translation unit are not unique. This is the case if the source references are located in a function that was created from a function template via instantiation or if the function containing the source references is defined in a class template and at least two instances exist for the template (see above and page 108).

src-ref

Designates the address of an executable statement and is specified in the form S'[*f*–]*n*[:*a*]', where *n* is the line number, *u* is the FILE number if it is > 0, and *a* is the relative statement number within the line if it is > 1.

Output of the %TRACE log

The %TRACE log 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 the chapter “Medium-a-quantity operand” in the AID Core Manual [1]).

A %TRACE listing with additional information (T=MAX) contains the number and type of the statement that was executed.

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

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

Examples

1.
```
/%out %trace    t=max
/%t 3
15                          EXT.PROC START    , BLOCK START, IF
16                          IF
17:2                          THEN/ELSE, END
STOPPED AT SRC_REF: 17:2 ,  SOURCE: EXAMP.C ,  PROC: facul ,
END OF TRACE
```

Using the %OUT command, output is switched back to the terminal and the maximum amount of information is selected for output.
The %TRACE command is to trace three C statements. After the third statement follows the termination message for this %TRACE command. Program control is before the second statement in line 17, which is assigned to the function `facul` in the translation unit `EXAMP.C`.

2.
```
/%out %t t=min
/%t 3
15
16
17:2
STOPPED AT SRC_REF: 17:2 ,  SOURCE: EXAMP.C ,  PROC: facul ,
END OF TRACE
```

The %OUT command reduces the range of information for the %TRACE command. The next %TRACE entered outputs the log with less information.

3. `%trace 5 r %instr`

5 program statements are executed and logged. Then the program is continued without logging.

4. `%c1 %call in s=testprog <%trace 1 r>`

All subroutine calls of program unit `TESTPROG` are logged. The program is continued after each execution and logging of the CALL statement.

7 POSIX debug command

The `debug` command allows debugging of POSIX programs which were started in the POSIX shell. With `debug`, you can load a program with LSD in the POSIX shell or interrupt a running process and set it to debug mode.

`debug` is not allowed in POSIX sessions opened via `rlogin` for system security reasons.

Syntax

```
debug { [␣-e]␣progname[␣argument]... }
      { ␣-p␣pid                      }
```

debug [␣**-e**] ␣progname [␣argument]...

Program *progname* is loaded in a task created by the shell via `fork()` and set to debug mode. AID then responds with a prompt formed from the task process number (pid) and you can input AID commands for debugging. You can use the `-e` option to control whether the LSD is to be loaded for symbolic debugging (without `-e`) or not (with `-e`). The `debug` *progname* command in the POSIX shell thereby corresponds to the BS2000 `LOAD-PROGRAM` *progname* command with the operand `TEST-OPTIONS=YES` in the BS2000 environment.

-e *progname* is loaded without the LSD.

progname

Name of the program to be debugged.

argument

Argument of *progname*.

debug ␣**-p**␣pid

The process with the specified *pid* is taken over by AID and interrupted if the process designated by *pid* belongs to its own task family. The POSIX shell is thereby the parent task for all processes started in the shell.
`debug -p` *pid* in the POSIX shell corresponds to the AID command `%STOP PID=`*pid* (see page 275), which you can enter BS2000 command mode or in debug mode in a task.

-p The program is taken over via the associated *pid*.

pid Process number of the task to be taken over by AID and interrupted.

Example

The example shows a running program being taken over by AID:

```
$ ps -ef
      UID   PID  PPID  C    STIME TTY       TIME CMD
   ¬┐≠┐│≠   ┐≠≏   ┌┐┐┤  ≏ ┐≏┌┐┐┐│┐ ┌┐┐┐┤↓≏≏│  ≏┐≏┐ ┌┐┐]
   D89239   888   824  0 10:22:27 term/003  0:00 [pexec]
   D89239   889   888  0 10:22:28 term/003  0:00 [pexec]
   D89239   830     1  0 09:35:13 term/004  0:04 [sh]
   D89239   824     1  0 09:31:22 term/003  0:06 [sh]
$ debug -p 888
%  AID0492 %STOP was sent to fork task (PID=0000000888).
%  AID0348 Program stopped due to STOP event (PID=0000000888)
```

The POSIX `ps -ef` command is used first to request a list of all running processes. You can select the PID of the process to be examined by AID (888) from this list. This process is the parent task for the fork task with PID 889. The parent task is interrupted and set to debug mode with `debug -p 888`.

```
%0000000888/%stop pid=889
%  AID0492 %STOP was sent to fork task (PID=0000000889).
```

The child task is also interrupted. Both tasks then report alternately with their prompts.

```
%0000000888/%aid low=all
%0000000888/%symlib test.lib
%  AID0348 Program stopped due to STOP event (PID=0000000889)
%0000000889/<EM><DÜ>
%0000000888/%trace 1 in s=n'pexec.c'
%0000000889/<EM><DÜ>
%0000000889/<EM><DÜ>
38                        BLOCK END, LOOP END
STOPPED AT SRC_REF: 38, SOURCE: pexec.c , PROC: main , END OF TRACE
```

In the next step, the parent task is to be executed until the first statement after the interrupt point. To enable AID to process the %TRACE 1 IN S=*srcname* command, the case sensitivity must be enabled for the S qualification with %AID LOW=ALL and the PLAM library containing the LSD for the pexec program must be registered with %SYMLIB.
Since the parent and child tasks are running in parallel, it is advisable to improve legibility by responding to the prompt of the other task each time with EM DÜ until the %TRACE command output is complete.

```
%0000000889/%aid low=all
%0000000888/<EM><DÜ>
%0000000889/%symlib test.lib
%0000000888/<EM><DÜ>
%0000000889/%trace 1 in s=n'pexec.c'
%0000000888/<EM><DÜ>
27                        BLOCK END, LOOP END
STOPPED AT SRC_REF: 27, SOURCE: pexec.c , BLK: 17 , END OF TRACE
%0000000888/...
```

The same process as above is repeated for the child task.

8 Special notes on debugging under POSIX

In addition to information on debug context inheritance with `fork()` or `exec()` calls and dump processing when debugging with POSIX, this chapter also contains information on the strategies most likely to lead to success when debugging fork tasks and programs loaded with an `exec()` call.

8.1 Inheriting the debug context

The only setting which remains applicable in a task created via `fork()` is `%AID FORK=ALL`, if it was set in the parent task. All other settings such as:

– settings defined with %AID
– set test points
– events traced with %ON
– PLAM libraries registered with %SYMLIB, etc.

are reset in the fork task.

However, settings made with AID and definitions made with %SYMLIB remain effective in a program loaded with `exec()`. All other definitions are reset, as in a fork task.

8.2 Debug strategies

If you only have a BS2000 terminal or appropriate emulation available for debugging fork tasks, you may run into problems when debugging several tasks concurrently as these tasks have to compete for the terminal.

This section contains instructions on the most expedient procedures for successfully debugging fork tasks and programs loaded with an `exec()` call.

One suitable strategy is to initially debug each program section, i.e. parent task, tasks created via `fork()` and programs loaded with `exec()` completely separately. This also has another advantage for programs which are to be loaded later with `exec()` calls. If the program is loaded with `exec()`, the LSD cannot be loaded and must be explicitly assigned with %SYMLIB. However, if you load the program directly with the POSIX `debug` command, you can also load the LSD with it.

You should debug the call context separately and only proceed with debugging the entire program after ensuring that all program sections and the call context are free of errors. One way of doing this is to use successive `fork()` and `exec()` calls while the superordinate task is quiescent in each case, which you can achieve by temporarily inserting a loop of sufficient length or with a suitable `wait()` call.

Each program section should mark its outputs during the debug phase to allow them to be properly allocated. More detailed information on allocating the inputs/outputs while debugging multiple fork tasks is provided in the section “Allocation” on page 299.

You should initially debug the program in the POSIX shell. The various fork tasks run with the same priority in the shell, i.e. each fork task has the same priority for accessing the terminal to request inputs or output information. If the program is started in the LOGON task, the BS2000 command mode has a higher priority than the fork task debug mode. This may result in the parent tasks blocking the terminal, thus preventing the fork tasks from accessing the terminal for input/output purposes.
A table in the following form can be a useful aid when simultaneously debugging several fork tasks. You can use it to make a list of each fork task number together with its process number and TSN, the source references of the `fork()` call and the current interrupt point:

Fork number	pid	TSN	Source reference	
			Fork start source code	Current interrupt
F1	929	OND1		168
F11	930	OND2	110	124, 128
...	...	...	...	...

Table 9: Overview of active fork tasks

You should also note that the program is interrupted in the runtime system immediately after a `fork()` or `exec()` call. You can only access data, functions and source references from this interrupt point with full qualification. To save yourself from having to write too much, it is advisable to initially use `%TRACE 1 IN S=`*srcname* to advance to the next executable statement in the user program.

You cannot use the K2 key under POSIX. To terminate a POSIX process, you have to input the string `@@c`. The POSIX shell then responds with its prompt, generally $. You can terminate a task in debug mode with the BS2000 EXIT-JOB or LOGOFF command or you can input the CANCEL-JOB command with the TSN of the task to be terminated, from another task (see “BS000 User Commands (SDF Format)”).

8.3 Input/output

When debugging fork tasks, the separate tasks have to compete for the terminal. The outputs of the separate fork tasks are initially put into a queue and then processed in sequence. There are therefore specific rules that have to be noted for input/output in debug mode which may differ from those for “normal” debugging in BS2000 command mode. The following sections handle possible inputs in debug mode, problems with allocating inputs/outputs to the various tasks and possible errors.

8.3.1 Possible inputs

You can input all AID commands and most BS2000 commands in debug mode. All BS2000 commands are allowed that you can also specify in command sequences and subcommands (see the AID Core Manual [1]). Guided SDF dialog is not possible.

Command sequences comprising AID and BS2000 commands separated by semicolons (;) can also be input. The restrictions described in the AID Core Manual in the section “Command sequences and subcommands” [1] also apply in this case. These restrictions also apply if BS2000 commands are input in debug mode, even separately.
As in BS2000 command mode, you can also only input EM DÜ in debug mode. This “dummy” input is required in debug mode to allow the required task to report to the terminal with its prompt from among several in one fork family. You may have to input EM DÜ several times as tasks report to the terminal in the same order as their associated inputs/outputs are entered in the queue.

Example

```
$ debug ex1fork
%  AID0348 Program stopped due to EXEC event (PID=0000002893)
%0000002893/%on %svc(44) <%trace 1 %instr>
%0000002893/%aid fork=next
```

The program is first loaded with the POSIX `debug` command. Program `ex1fork` contains a `fork()` call.
Tracing of SVC number 44 is activated with the AID %ON command. The associated subcommand ensures that the SVC is executed and that the program is halted immediately after this.
You then use `%AID FORK=NEXT` to enable debug mode for the first-generation fork tasks.

```
%0000002893/%resume
ICXSVCTU+7C46     SVC   44                      1 FCT=POSPWENT IR1=010BCA40
                                                PAR=00E4B601 00000000 00000000
STOPPED AT V'10646D0' = ICXSVCTU + #'7C48' , END OF TRACE
%0000002893/%r
ICXSVCTU+7C46     SVC   44                      1 FCT=POSLDENV IR1=010BCA40
                                                PAR=00E34101 00000000 00000000
STOPPED AT V'10646D0' = ICXSVCTU + #'7C48' , END OF TRACE
%0000002893/%r
...
%0000002893/%r
ICXSVCTU+7C46     SVC   44                      1 FCT=POSFORK  IR1=010BCEB0
                                                PAR=00E30201 00000000 00000000
STOPPED AT V'10646D0' = ICXSVCTU + #'7C48' , END OF TRACE
```

The subsequent %RESUME commands execute the program up to the deciding SVC with number 44 (FCT=POSFORK). This SVC starts the fork task creation.

```
%0000002893/<EM><DÜ>
%  AID0348 Program stopped due to FORK event (PID=0000002897)
%0000002893/<EM><DÜ>
```

You now respond to the parent task prompt with a dummy input (EM DÜ). AID outputs message AID0348 which confirms that the fork task has been created. You then again respond to the subsequent parent task prompt with EM DÜ (forced task change). AID now prompts you input a command for the fork task.

```
%0000002897/%show %base
%0000002893/<EM><DÜ>
%BASE E=VM
TSN: OJO5    TID: 0091017D
%AINT = %MODE31
BS:  V12.0   HW:  CFCS V3
%0000002897/
```

You have to respond to the parent task prompt once more with a dummy input to output the information of the %SHOW %BASE command to the terminal.

8.3.2 Allocation

As mentioned several times, it is generally advisable to debug each task separately and put other tasks created by the program into a quiescent state in the meantime. However, should several tasks run simultaneously and output to the terminal, the following applies:

– Only the inputs can be uniquely allocated, i.e. each input always goes to the tasks whose prompt was used to input it.

– Outputs cannot generally be allocated, except for the %TRACE logs which can be allocated via the source references.
If several tasks try to output to the terminal simultaneously, the output order is more or less random. If you are waiting for the output from a specific task, you should always respond to the prompts from other tasks with a dummy input until the awaited output is complete (see the example above).

– To ensure correct allocation during the debug phase, you should either redirect program outputs into a file or identify them temporarily with a prepended program abbreviation.

8.3.3 Errors

A fork task cannot input or output to the terminal in the following cases:

– No program is loaded in the LOGON task.

– A program other than that from which the fork task was created (either directly or indirectly) is loaded in the LOGON task.

– The LOGON task has been terminated.

This also applies analogously for programs that were started in the POSIX shell.

In this case, the fork task is aborted without an error message when it tries to output to or input from the terminal. This also applies if the input/output request has already been entered into the queue.

The fork task is not aborted while the input/output is redirected into files.

8.4 Dump processing

You can process dumps (memory areas) from fork tasks and programs loaded via an `exec()` call in the normal manner. Dumps are generally stored in BS2000, even if the program that generated the dump was started in the POSIX shell. If AID has to load LSD via the AID %SYMLIB command to dump a POSIX program, you must note that %SYMLIB cannot access POSIX files. The file concerned must first be copied with the POSIX `bs2cp` command as an L member into a PLAM library in BS2000 and can then be assigned with %SYMLIB (see also the section “Loading the LSD dynamically” on page 18).
The %DUMPFILE command, which directs AID to open the dump file, and the %BASE command, with which you specify to AID that the trace is to take place in the order of the memory area stored in this dump file, are described in the chapter “AID commands” on page 113.

In the POSIX shell, a user dump is always written for programs which were aborted because of an error. The “IDA0N45 Dump desired?” query you know from BS2000 is not output and the program is unloaded.
It is therefore advisable to trap program errors with %ON %ANY during debugging. When an error occurs, AID then reports the address of the interrupt point at which the error occurred and the event which caused the error. The program remains loaded and you can inspect the error context immediately. If the error can be eliminated with AID commands, you can continue program execution with %RESUME. However, if it is not possible to continue program execution, you can terminate the task with EXIT-JOB or LOGOFF and subsequently analyze the program error with further debugging.

9 Sample applications

This chapter presents three AID debugging sessions for short C and C++ programs. After working through these sessions, you will understand the use and effects of a number of AID commands better. The programs have intentionally been kept simple.

The examples in this chapter were executed on a specific BS2000 system. A test run on some other machine may differ marginally from the descriptions here, depending on the installed operating system and the other system components.

9.1 Sample C application in BS2000

The program is intended to read in a maximum of 5 names and telephone numbers and the names are then sorted and output in list form together with the telephone numbers.
The source error listing of the C program is reprinted first, followed by the debug run.
To improve legibility, the user inputs in the run logs are printed in **boldface**. Data and function names are shown in continuous text in `typewriter font`.

9.1.1 Source error listing

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27  PAGE:    1
                    SOURCENAME:*LIB-ELEM(MYLIB,NLIST.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________________

  EXP     SRC  BLOCK
  LIN ... LIN   LEV

    1       1     0  #include <stdio.h>
 1662       2     0  #include <stdlib.h>
 2128       3     0  #include <string.h>
 2414       4     0  #define MAX 5
 2415       5     0
 2416       6     0  struct tele {                 /* Structure tele */
 2417       7     0     char           name[15];
 2418       8     0     unsigned long  number;
 2419       9     0     };
 2420      10     0
 2421      11     0  void nlist(struct tele array[], int count);  /* Function nlist */
 2422      12     0  int nread(struct tele array[]);              /* Function nread */
 2423      13     0
 2424      14     0  int main(void)
 2425      15     0  {
 2426      16     1     struct tele arrp[MAX];     /* Phone list */
 2427      17     1     int nentry;                /* Number of entries */
 2428      18     1
 2429      19     1     nentry = nread(arrp);      /* Read in */
 2430      20     1     qsort (arrp, nentry, sizeof(struct tele),
 2431      21     1        (int(*)(const void*, const void*))strcmp);  /* Sort list */
 2432      22     1     nlist (arrp, nentry);      /* Output list */
 2433      23     1     return 0;
 2434      24     1  }
 2435      25     0
 2436      26     0  int nread(struct tele f[])        /* Read in names and numbers */
 2437      27     0  {
 2438      28     1     int i;
 2439      29     1
 2440      30     1     for (i=0; i<=MAX; i++)
 2441      31     1     {   printf ("Enter a name (end = 9): ");
 2442      32     2         if ((scanf ("%s", f[i].name) != 1) ||
 2443      33     2               (f[i].name[0] == '9')) break;
 2444      34     2     do {
 2445      35     3         fflush(stdin);
 2446      36     3         printf ("Now enter the phone number: ");
 2447      37     3        } while (scanf ("%lu", f[i].number) != 1);
 2448      38     2     }
 2449      39     1     return 0;
 2450      40     1  }
 2451      41     0
 2452      42     0  void nlist (struct tele f[], int n)             /* Output list */
 2453      43     0  {
 2454      44     1     int i;
 2455      45     1
 2456      46     1     printf ("%-25s | %10s\n", "Name", "Number");      /* Header */
 2457      47     1     printf ("-----------------------------------\n");
 2458      48     1     for (i=0; i<n; i++)
 2459      49     1        printf ("%-25s | %10lu\n", f[i].name, f[i].number);
 2460      50     1     printf ("\n\n");
 2461      51     1     return;
 2462      52     1  }
```

9.1.2 Debug run

Step 1

The C source program in the file NLIST.C is compiled by the C compiler. Specifying the SDF option MODIFY-TEST-PROPERTIES TEST-SUPPORT=*YES causes C to generate the LSD records which make symbolic debugging possible. Optimization is suppressed during compilation by means of the MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW option (see section “Compiling in BS2000” on page 14). The compilation does not produce any errors.

```
/START-CPLUS-COMPILER
 %  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
         ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
 %  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
 %  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
 %  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*C(MODE=*ANSI)",DEFINE=..."
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW
...
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES  INCLUDE = *LIB-ELEM(L=MYLIB,E=NLIST.O)
//BIND                    OUTPUT  = *LIB-ELEM(L=MYLIB,E=NLIST)
...
//END
%  CDR9908 : END   C TIME USED = 3.5300 SEC
%  CCM0998 CPU TIME USED: 3.6159 SECONDS
```

Step 2

The program is loaded. The command %AID C=YES causes AID to accept C string literals and enables the processing of char arrays as C strings. C=YES implicitly enables case sensitivity, i.e. a distinction between uppercase and lowercase, as well as the interpretation of hyphens as minus characters.
The command %ON %ANY ensures that the program will not be unloaded if an error is encountered and that AID will issue an error message giving the address of the interrupt point and the event which caused the interrupt.

On starting the program with %RESUME, it initially runs correctly and requests the input of names and numbers; however, the nread function does not stop reading after the 5-th number and requests a further name instead. Reading the 6-th name results in a page error. The program can therefore now be examined at the interrupt point to determine what caused the error.
The following two %DISPLAYs show the subscript and the associated array element f[i].name. AID reports an invalid address for f[i].name: the array has only 5 elements, so the highest permissible value for i is 4. On examining the start statement of the loop in line 30, you learn that the query at the end of the loop is formulated incorrectly: instead of testing for i<=MAX, a test for i<MAX should be performed.

In order to continue in the program despite the error that occurred, you set the instruction counter to the address of the `return` statement that was stored for source reference S'39' and start the program with %RESUME at that address. Note, however, that it is advisable to be cautious here, especially if large jumps are to be performed within the program by overwriting the instruction counter. There is no guarantee that the register states, file status, contents of subscripts, etc., will always be suitable at the destination address to continue the program normally.
The program now executes up to the end, but the function `nlist` displays only the title.

```
/LOAD-PROG *M(MYLIB,NLIST,RUN-MODE=ADV,PROGRAM-MODE=ANY), TEST-OPT=AID
%  BLS0523 ELEMENT 'NLIST', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$NLIST', VERSION ' ' OF '2015-01-14 12:24:58'  LOADED
/%aid c=yes
/%on %any
/%resume
Enter a name (end = 9): Joe
Now enter the phone number: 123
Enter a name (end = 9): Weber
Now enter the phone number: 456
Enter a name (end = 9): Blob
Now enter the phone number: 789
Enter a name (end = 9): Peter
Now enter the phone number: 101112
Enter a name (end = 9): Williams
Now enter the phone number: 131415
Enter a name (end = 9): Smith
STOPPED AT SRC_REF: 32, SOURCE: NLIST.C , BLK: 31 , EVENT: PAGING ERROR
/%d i
*** TID: 00420333 *** TSN: 8MEH ************************************************************
SRC_REF: 32 SOURCE: NLIST.C   BLK: 31  ***********************************************
i               =           5
/%d f[i].name
*.name( 0: 14)
%  AID0396 Invalid address for name
/%s s'39' into %pc
/%r
Name                        |     Number
------------------------------------


 %  CCM0998 CPU TIME USED: 0.0540 SECONDS
 STOPPED AT V'1018846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

Step 3

The program is reloaded. In order to bypass the error with the help of AID, the reading of names and numbers is stopped via the subcommand in %INSERT S'31' after five iterations. The next command %CONTROL1 causes the program to stop on starting the `nlist` function and just before it is exited, thus allowing you to examine why only the title of the phone list was output by `nlist`.

On restarting the program with %RESUME, exactly 5 names and numbers are now read in correctly. Due to the %CONTROL, the program stops at the first executable statement of `nlist`. You now use the %DISPLAY command to examine the contents of the passed parameter `n`, which has a value of 0. The variable `nentry`, which accepts the result of

`nread` and passes it on to `nlist`, is obviously not set correctly, as confirmed by the following command `%DISPLAY proc=main.nentry`. A quick look at the source program indicates that the function `nread` is passing the value 0 instead of `i`, so the `return` statement in line 39 must be replaced by `return i;`.

With the next %SET command, you set `n` to 5, which causes `nlist` to output the list of names and phone numbers. However, the names are not sorted, and this, in turn, is due to the fact that the value of the `nentry` variable was still 0 when the library function `qsort:` was called.
Furthermore, even the numbers are not accurate, and this is presumably because an address operator was forgotten in the function call to `scanf`. This would mean that `scanf` is probably using the content of `f[0].number` as an address and has entered the number at that address. The subsequent %DISPLAY proves the invalid addressing. The first phone number, “123”, can actually be found there.
The correct way call to `scanf` in line 37 would be:
`scanf ("%lu", &f[i].number).`

```
/LOAD-PROG *M(MYLIB,NLIST,RUN-MODE=ADV,PROGRAM-MODE=ANY), TEST-OPT=AID
%  BLS0523 ELEMENT 'NLIST', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$NLIST', VERSION ' ' OF '2015-01-14 12:24:58'  LOADED
/%on %any
/%in s'31' <s31:(i eq 5): %s s'39' into %pc>
/%c1 %proc in proc=nlist
/%r
Enter a name (end = 9): Joe
Now enter the phone number: 123
Enter a name (end = 9): Weber
Now enter the phone number: 456
Enter a name (end = 9): Blob
Now enter the phone number: 789
Enter a name (end = 9): Peter
Now enter the phone number: 101112
Enter a name (end = 9): Williams
Now enter the phone number: 131415
STOPPED AT SRC_REF: 46, SOURCE: NLIST.C , PROC: nlist
/%d n
*** TID: 00420333 *** TSN: 8MEH **********************************************************
SRC_REF: 46 SOURCE: NLIST.C  PROC: nlist  **********************************************
n           =          0
/%d proc=main.nentry
nentry      =          0
/%s 5 into n
/%r
Name                      |     Number
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
Joe                       |   17988680
Weber                     |   17989292
Blob                      |   17989136
Peter                     |          0
Williams                  |          0


STOPPED AT SRC_REF: 51, SOURCE: NLIST.C , PROC: nlist
/%d f[0].number->%f
CURRENT PC: 010004E2    CSECT: NLIST$0&@  ********************************************
V'01127C48' = IC@RT20A + #'00027C48'
01127C48 (00027C48)        +123
```

Step 4

The program is reloaded, and the detected errors are eliminated with %INSERT commands:

– The first %INSERT places the missing address operator before `f[i].number` in the call to `scanf`.

– The second %INSERT is used to verify the entered names and numbers and also to display each subscript.

– The third %INSERT breaks the loop in `nread` as above.

– Finally, the fourth %INSERT corrects the value of `nentry`.

Start the program with %RESUME.

The names and numbers are now read in correctly, and the phone list is sorted and output in alphabetical order.
Due to the command `%ON %ANY`, the program stops before being finally unloaded even on executing correctly, and a corresponding STOP message is issued.

```
/LOAD-PROG *M(MYLIB,NLIST,RUN-MODE=ADV,PROGRAM-MODE=ANY), TEST-OPT=AID
%  BLS0523 ELEMENT 'NLIST', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$NLIST', VERSION ' ' OF '2015-01-14 12:24:58'  LOADED
/%on %any
/%in s'32' <s32: %set %@(f[i].number) into f[i].number>
/%in s'38' <s38: %d i, f[i].number, f[i].name>
/%in s'31' <s31: (i eq 5): %s s'39' into %pc>
/%in s'20' <s20: %set 5 into nentry>
/%r
Enter a name (end = 9): Joe
Now enter the phone number: 123
*** TID: 00420333 *** TSN: 8MEH ***************************************************
SRC_REF: 38 SOURCE: NLIST.C  PROC: nread  ******************************************
i               =           0
*.number        =         123
*.name          = "Joe"
Enter a name (end = 9): Weber
Now enter the phone number: 456
i               =           1
*.number        =         456
*.name          = "Weber"
Enter a name (end = 9): Blob
Now enter the phone number: 789
i               =           2
*.number        =         789
*.name          = "Blob"
Enter a name (end = 9): Peter
Now enter the phone number: 101112
i               =           3
*.number        =      101112
*.name          = "Peter"
Enter a name (end = 9): Williams
Now enter the phone number: 131415
i               =           4
*.number        =      131415
*.name          = "Williams"
```

Continued...

Continued...

```
Name                        |     Number
------------------------------------------
Blob                        |        789
Williams                    |     131415
Joe                         |        123
Weber                       |        456
Peter                       |     101112


 %  CCM0998 CPU TIME USED: 0.1435 SECONDS
 STOPPED AT V'1018846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

9.2 Sample C++ application in BS2000

This program is intended to display two lines of text, using the class `string`.
To enhance the readability of this example, data and function names are printed in `typewriter font` in the body of the text, while user input is printed in **boldface**.

9.2.1 Source error listing

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27  PAGE:    1
                                SOURCENAME: *LIB-ELEM(MYLIB,STRING.C(*HIGHEST-EXISTING),S
_____________________________________________________________________________________________
EXP   INC    FILE   SRC
LIN   LEV     NO    LIN

  1    0       0      1
  2    0       0      2  extern "C" void* malloc(unsigned);
  3    0       0      3  extern "C" void  free(void*);
  4    0       0      4
  5    0       0      5  extern "C" int   strlen( const char* );
  6    0       0      6  extern "C" char* strcpy( char*, const char* );
  7    0       0      7  extern "C" char* strcat( char*, const char* );
  8    0       0      8
  9    0       0      9  extern "C" int   printf( const char*, ... );
 10    0       0     10
 11    0       0     11  class string
 12    0       0     12  {
 13    0       0     13          int length;
 14    0       0     14          char* start;
 15    0       0     15  public:
 16    0       0     16          /*
 17    0       0     17           * constructors
 18    0       0     18           */
 19    0       0     19          string() : length(0), start(0) {};
 20    0       0     20          string( const char *s ) {
 21    0       0     21                length = strlen(s) + 1;
 22    0       0     22                start = new char[length];
 23    0       0     23                strcpy( start, s );
 24    0       0     24          };
 25    0       0     25          string( const string &s ) {
 26    0       0     26                length = s.length;
 27    0       0     27                start  = new char[length];
 28    0       0     28                strcpy( start, s );
 29    0       0     29          };
 30    0       0     30          const string& operator=( const string& s )
 31    0       0     31          {
 32    0       0     32                length = s.length;
 33    0       0     33                start  = new char[length];
 34    0       0     34                strcpy( start, s );
 35    0       0     35                return *this;
 36    0       0     36          };
 37    0       0     37          /*
 38    0       0     38           * destructor
 39    0       0     39           */
 40    0       0     40          ~string() {
 41    0       0     41                delete start;
 42    0       0     42          };
```

```
43    0       0     43          /*
44    0       0     44           * conversion
45    0       0     45           */
46    0       0     46          operator char*() const {
47    0       0     47                return start;
48    0       0     48          };
49    0       0     49  };
50    0       0     50  /*
51    0       0     51   * string concatenation
52    0       0     52   */
53    0       0     53  string& operator + ( const string& p, const string& q )
54    0       0     54  {
55    0       0     55        static string s = p;     // copy first string
56    0       0     56        s = strcat(s,q);         // cat second string
57    0       0     57        return s;
58    0       0     58  }
59    0       0     59
60    0       0     60  string s = "Hello";
61    0       0     61
62    0       0     62  int main(void)
63    0       0     63  {
64    0       0     64        string p(s);             // p is "Hello"
65    0       0     65
66    0       0     66        string q("World\n");     // q is "World\n"
67    0       0     67
68    0       0     68        printf(p + " C++ " + q); // should print "Hello C++ World\n"
69    0       0     69
70    0       0     70        p = "Goodbye";           // p is now "Goodbye"
71    0       0     71
72    0       0     72        q = "C " + q;            // q is now "C World\n"
73    0       0     73
74    0       0     74        printf(p + q);           // should print "Goodbye C World\n"
75    0       0     75
76    0       0     76        return 0;
77    0       0     77  }
```

9.2.2 Debug run

Step 1

The C++ source program in the file STRING.C is compiled by the C++ compiler. Since the SDF option MODIFY-TEST-PROPERTIES TEST-SUPPORT=*YES is set, C++ generates the LSD records which support symbolic debugging. Optimization is suppressed during compilation by the option MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW, and the expansion of inline functions is suppressed by BUILTIN-FUNCTIONS=*NONE (see the section “Compiling in BS2000” on page 14). No errors are reported during compilation. The program is subsequently to be linked with the C/C++ compiler BIND statement. To ensure that the LSD information is included in the linked module, you have to first set the TEST-SUPPORT=*YES option in the MODIFY-BIND-PROPERTIES statement.

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*CPLUSPLUS()
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW, BUILTIN-FUNCTIONS=*NONE
//COMPILE SOURCE=*LIB(MYLIB,STRING.C),MODULE-OUTPUT=*LIB(MYLIB,STRING.O)
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES START-LLM-CREATION = *YES,         -
//INCLUDE = *LIB-ELEM(L=MYLIB,E=STRING.O),                 -
//STDLIB=*STATIC,                                          -
//RUNTIME-LANGUAGE =*CPLUSPLUS(MODE=ANSI), TEST-SUPPORT = *YES
//BIND OUTPUT = *LIB-ELEM(LIB=MYLIB,ELEM=STRING)
%  BND1501 LLM FORMAT: '1'
%  BND1101 BINDER NORMALLY TERMINATED.
//END
%  CDR9908 : END   C TIME USED = 4.7600 SEC
%  CCM0998 CPU TIME USED: 4.8956
```

Step 2

The program is loaded. The %AID LOW command causes AID to be case-sensitive. The command %ON %ANY ensures that the program will not be unloaded in the event of an error and that AID will issue an error message giving the address of the interrupt point and the event which caused the interrupt. When the program is started with %RESUME, execution continues without errors until the end of the program is reached, but the result is false.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB'
IN PROCESSING
%  BLS0524 LLM '$LIB-ELEM$MYLIB$STRING$', VERSION ' ' OF '2015-03-06 13:26:24' LOADED
/%aid c=yes
/%on %any
/%resume
Hello C+World
Hello C+World
World
Hell
%  CCM0998 CPU TIME USED: 0.0047 SECONDS
STOPPED AT V'101B846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

Step 3

The outputs indicate that the contents of the `string` object have been overwritten. To find out which statement in the program has caused the error, write monitoring is activated for the allocated memory area at each dynamic memory request and associated initialization. The program is started with %RESUME and runs up to the first interruption caused by the activated write monitoring. You now display the call hierarchy with `%SDUMP %NEST` to check which statement overwrote the contents of `start`.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB'
IN PROCESSING
%  BLS0524 LLM '$LIB-ELEM$MYLIB$STRING$', VERSION ' ' OF '2015-03-06 13:26:24' LOADED
/%in s'24' <%on %write(start-> %l=(length))>
/%in s'29' <%on %write(start-> %l=(length))>
/%in s'35' <%on %write(start-> %l=(length))>
/%r
%  AID0496 Warning: previously defined event %WRITE is replaced
%  AID0496 Warning: previously defined event %WRITE is replaced
%  AID0496 Warning: previously defined event %WRITE is replaced
STOPPED AT V'1001E98' = IC@STRG@ + #'258' , EVENT: WRITE
/%sd %nest
*** TID: 010802A9 *** TSN: 6EJP ************************************************************
ABSOLUT: V'1001E98'   SOURCE: IC@STRG@  PROC: STRCAT  ***********************************
SRC_REF: 56  SOURCE: STRING.C  PROC: operator+(const string &, const string &)  ********
SRC_REF: 68  SOURCE: STRING.C  PROC: main  ********************************************
ABSOLUT: V'10237B8'   SOURCE: ICS$MAI@  PROC: ICS$MAI@  ***********************************
ABSOLUT: V'10019C8'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ***********************************
```

Step 4

The `strcat()` function, which caused the erroneous overwriting, is called in S'56'. This is analyzed in detail below. Because the statement in line 56 uses the `string` object `s`, we suspect that its contents were modified and this is confirmed by the two %DISPLAYs. However, a prequalification is first defined with %QUALIFY to avoid having to write the complete qualification in each %DISPLAY.

```
/%q s=n'STRING.C'.proc=n'operator+(const string &, const string &)'
/%d .s.start->%l20
V'010E75C8' = ABSOLUT  + #'010E75C8'
010E75C8 (010E75C8) C8859393 9640C34E 4E400000 00000000    Hello C++ ......
010E75D8 (010E75D8) 00000000                               ....
/%d .s.length
ABSOLUT: V'01001E98' SOURCE: IC@STRG@  PROC: STRCAT  ***********************************
s.length        =           6
```

Step 5

So the error is caused by statement S'56', `s = strcat(s,q);`. The `strcat` call modifies and extends its first argument, as can be seen from the fact that the first argument in its declaration is specified as type `char*` rather than `const char*`.
The function `operator+(const string &,const string &)` is now modified such that it creates a string with a length which is the sum of the lengths of strings `p` and `q`. For this purpose it has to access the `start` and `length` components of class `string` and therefore has to be declared as a friend function of that class. Then a number of other adjustments need to be made, and the resulting program is as follows:

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27  PAGE:    1
                                SOURCENAME: *LIB-ELEM(MYLIB,STRING.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________________________

EXP   INC    FILE   SRC
LIN   LEV     NO    LIN

  1    0      0      1
  2    0      0      2  extern "C" void* malloc(unsigned);
  3    0      0      3  extern "C" void  free(void*);
  4    0      0      4
  5    0      0      5  extern "C" int   strlen( const char* );
  6    0      0      6  extern "C" char* strcpy( char*, const char* );
  7    0      0      7  extern "C" char* strcat( char*, const char* );
  8    0      0      8
  9    0      0      9  extern "C" int   printf( const char*, ... );
 10    0      0     10
 11    0      0     11  class string
 12    0      0     12  {
 13    0      0     13        int length;
 14    0      0     14        char* start;
 15    0      0     15  public:
 16    0      0     16        /*
 17    0      0     17         * constructors
 18    0      0     18         */
 19    0      0     19        string(int n=0) : length(n){
 20    0      0     20              start = new char[length];
 21    0      0     21        };
 22    0      0     22        string( const char *s ) {
 23    0      0     23              length = strlen(s) + 1;
 24    0      0     24              start = new char[length];
 25    0      0     25              strcpy( start, s );
 26    0      0     26        };
 27    0      0     27        string( const string &s ) {
 28    0      0     28              length = s.length;
 29    0      0     29              start  = new char[length];
 30    0      0     30              strcpy( start, s );
 31    0      0     31        };
 32    0      0     32        const string& operator=( const string& s )
 33    0      0     33        {
 34    0      0     34              delete start;
 35    0      0     35              length = s.length;
 36    0      0     36              start  = new char[length];
 37    0      0     37              strcpy( start, s );
 38    0      0     38              return *this;
 39    0      0     39        };
 40    0      0     40        /*
 41    0      0     41         * destructor
 42    0      0     42         */
 43    0      0     43        ~string() {
 44    0      0     44              delete start;
 45    0      0     45        };
```

```
46    0       0     46        /*
47    0       0     47         * conversion
48    0       0     48         */
49    0       0     49        operator const char*() const {
50    0       0     50              return start;
51    0       0     51        };
52    0       0     52  friend string& operator+(const string&,const string&);
53    0       0     53  };
54    0       0     54
55    0       0     55  /*
56    0       0     56   * string concatenation
57    0       0     57   */
58    0       0     58  string& operator + ( const string& p, const string& q )
59    0       0     59  {
60    0       0     60        static string s;
61    0       0     61        s = p.length + q.length -1;
62    0       0     62                                        //allocate right length
63    0       0     63        s.start = strcpy(s.start,p);    //copy first string
64    0       0     64        s.start = strcat(s.start,q);    //copy second string
65    0       0     65        return s;
66    0       0     66  }
67    0       0     67
68    0       0     68  string s = "Hello";
69    0       0     69
70    0       0     70  int main(void)
71    0       0     71  {
72    0       0     72        string p(s);              // p is "Hello"
73    0       0     73
74    0       0     74        string q("World\n");      // q is "World\n"
75    0       0     75
76    0       0     76        printf(p + " C++ " + q); // should print "Hello C++ World\n"
77    0       0     77
78    0       0     78        p = "Goodbye";            // p is now "Goodbye"
79    0       0     79
80    0       0     80        q = " C " + q;            // q is now "C World\n"
81    0       0     81
82    0       0     82        printf(p + q);            // should print "Goodbye C World\n"
83    0       0     83
84    0       0     84        return 0;
85    0       0     85  }
```

Step 6

The modified program is compiled, loaded and started. It runs through to the end but produces the wrong result again.

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*CPLUSPLUS()
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW, BUILTIN-FUNCTIONS=*NONE
//COMPILE SOURCE=*LIB(MYLIB,STRING.C),MODULE-OUTPUT=*LIB(MYLIB,STRING.O)
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES START-LLM-CREATION = *YES,         -
//INCLUDE = *LIB-ELEM(L=MYLIB,E=STRING.O),                 -
//STDLIB=*STATIC,                                          -
//RUNTIME-LANGUAGE =*CPLUSPLUS(MODE=ANSI), TEST-SUPPORT = *YES
//BIND OUTPUT = *LIB-ELEM(LIB=MYLIB,ELEM=STRING)
%  BND1501 LLM FORMAT: '1'
%  BND1101 BINDER NORMALLY TERMINATED.
%//END
%  CDR9908 : END   C TIME USED = 5.6300 SEC
%  CCM0998 CPU TIME USED: 5.7149
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$STRING', VERSION ' ' OF '2015-01-25 10:57:18' LOADED
/%on %any
/%r
World
Goodbye C World
%  CCM0998 CPU TIME USED: 0.0027 SECONDS
STOPPED AT V'101C846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

Step 7

The first text line ("Hello C++ World") has not been put together correctly. The error is probably located in the function `operator+(const string &,const string &)`.
Therefore once the program has been reloaded, several %CONTROL commands are entered to cause the contents of `s`, `p` and `q` to be displayed before each of the statements S'61' through S'63' are executed. The %INSERT S'78' which follows causes the function `operator+(const string &,const string &)` to be monitored only until the first text line is displayed.
The Stop message at the end of the log is displayed as a result of the `%ON %ANY` command, which halts the program before it is finally unloaded even after a normal program run, and outputs the current status of the instruction counter and the name of the runtime routine responsible for termination handling.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$STRING', VERSION ' ' OF '2015-01-25 10:57:18' LOADED
/%on %any
/%c1 %stmt in (s'61':s'63') <(s.length eq 0): %d s.start>
/%c2 %stmt in (s'61':s'63') <(s.length ne 0): %d ' ',s.length, -
/ 's.start->:',s.start->%l=(s.length)>
/%c3 %stmt in (s'61':s'63') <%d p.length,'p.start->:' -
/ ,p.start->%l=(p.length)>
/%c4 %stmt in (s'61':s'63') <%d q.length,'q.start->:' -
/ ,q.start->%l=(q.length)>
/%in s'78' <%rem %c>
/%r
*** TID: 010802A9 *** TSN: 6EJP ********************************************************
SRC_REF: 61 SOURCE: STRING.C    PROC: operator+(const string &, const string &)
s.start         = 010E85C8
p.length        =          6
p.start->:
CURRENT PC: 01000588    CSECT: STRING$0&@  *******************************************
V'010E8598' = ABSOLUT  + #'010E8598'
010E8598 (010E8598) C8859393 9600                          Hello.
SRC_REF: 61 SOURCE: STRING.C    PROC: operator+(const string &, const string &)
q.length        =          6
q.start->:
CURRENT PC: 01000588    CSECT: STRING$0&@  *******************************************
V'010E85B8' = ABSOLUT  + #'010E85B8'
010E85B8 (010E85B8) 40C34E4E 4000                           C++ .

SRC_REF: 63 SOURCE: STRING.C    PROC: operator+(const string &, const string &)  *******
s.length        =         11
s.start->:
CURRENT PC: 01000988    CSECT: STRING$0&@  *******************************************
'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00000000 00000000 000000                ...........
SRC_REF: 63 SOURCE: STRING.C    PROC: operator+(const string &, const string &)  *******
p.length        =          6
p.start->:
CURRENT PC: 010005EC    CSECT: STRING$0&@  *******************************************
V'010E8598' = ABSOLUT  + #'010E8598'
010E8598 (010E8598) C8859393 9600                          Hello.
```

Continued...

Continued...

```
SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
q.length        =           6
q.start->:
CURRENT PC: 010005EC    CSECT: STRING$0&@  ******************************************
V'010E6598' = ABSOLUT  + #'010E6598'
0010E6598 (010E6598) 40C34E4E 4000                          C++ .

SRC_REF: 61 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
s.length        =          11
s.start->:
CURRENT PC: 01000924    CSECT: STRING$0&@  ******************************************
V'010E65D8' = ABSOLUT  + #'010E65D8'
010E85F8 (010E85F8) C8859393 9640C34E 4E4000              Hello C++ .
SRC_REF: 61 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
p.length        =          11
p.start->:
CURRENT PC: 01000588    CSECT: STRING$0&@  ******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) C8859393 9640C34E 4E4000              Hello C++ .
SRC_REF: 61 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
q.length        =           7
q.start->:
CURRENT PC: 01000588    CSECT: STRING$0&@  ******************************************
V'010E85A8' = ABSOLUT  + #'010E85A8'
010E85A8 (010E85A8) E6969993 841500                       World..

SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &) ********
s.length        =          17
s.start->:
CURRENT PC: 010005EC    CSECT: STRING$0&@  ******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00859393 9640C34E 4E400000 00000000    .ello C++ ......
010E8608 (010E8608) 00                                      .
SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &) ********
p.length        =          17
p.start->:
CURRENT PC: 010005EC    CSECT: STRING$0&@  ******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00859393 9640C34E 4E400000 00000000    .ello C++ ......
010E8608 (010E8608) 00                                      .
SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &) ********
q.length        =           7
q.start->:
CURRENT PC:  010005EC    CSECT: STRING$0&@  ******************************************
V'010E85A8' = ABSOLUT  + #'010E85A8'
010E85A8 (010E85A8) E6969993 841500                       World..
World
Goodbye C World
%  CCM0998 CPU TIME USED: 0.1770 SECONDS
STOPPED AT V'101C846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

Step 8

As suspected, the string "World\n" is not appended to the old text the second time *operator+(const string &,const string &)* is called, but is again stored in *s.start->* from the start. To find out with AID exactly what is happening in statement S'61', *s = p.length + q.length -1;* (which implicitly calls constructor *string::string(int)* and function *operator=(const string &)*), two %INSERT commands are issued on reloading the program to have the contents of the *length* and *start* components output at various test points in *string::string(int)* and *operator=(const string &)*.
The last %INSERT command again tells the program to run through to the end without monitoring once it has output the first text line.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$STRING', VERSION ' ' OF '2015-01-25 10:57:18' LOADED
/%in s'61' <(%. eq 2): %in s'20' <%d length,'start->:',start->%l=(length)>>
/%in s'61' <(%. eq 2): %c1 %proc in proc=string::n'operator=(const string &)'-
/<%d ' ',length,s.length;%d 'start->:',start->%l20; %d 's.start->:',s.start->%l20>>
/%in s'78' <%rem %c; %rem %in>
/%r
*** TID: 010802A9 *** TSN: 6EJP ************************************************************
SRC_REF: 20 SOURCE: STRING.C  PROC: string::string(int)  ******************************
string.length   =          17
start->:
CURRENT PC: 01000236    CSECT: STRING$0&@  *******************************************
V'010E85D8' = ABSOLUT  + #'010E85D8'
010E85D8 (010E85D8) 00000000 00000000 00000000 00000000    ................
010E85E8 (010E85E8) 00                                     .

SRC_REF: 34 SOURCE: STRING.C  PROC: string::operator=(const string &)  *****************
string.length   =          11
s.length        =          17
start->:
CURRENT PC: 010003CA    CSECT: STRING$0&@  *******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) C8859393 9640C34E 4E400000 00000000    Hello C++ ......
010E8608 (010E8608) 00000000                               ....
s.start->:
V'010E85D8' = ABSOLUT  + #'010E85D8'
010E85D8 (010E85D8) 00000000 00000000 00000000 00000000    ................
010E85E8 (010E85E8) 00000000                               ....

SRC_REF: 38 SOURCE: STRING.C  PROC: string::operator=(const string &)  *****************
string.length   =          17
s.length        =          17
start->:
CURRENT PC: 0100046C    CSECT: STRING$0&@  *******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00859393 9640C34E 4E400000 00000000    .ello C++ ......
010E8608 (010E8608) 00000000                               ....
s.start->:
V'010E85D8' = ABSOLUT  + #'010E85D8'
010E85D8 (010E85D8) 00000000 00000000 00000000 00000000    ................
010E85E8 (010E85E8) 00000000                               ....
World
Goodbye C World
%  CCM0998       CPU TIME USED: 0.0920 seconds
```

Step 9

From the information supplied by AID we can infer the following situation:
when *operator+(const string &, const string &)* is called for the second time and the third part of the text ("World\n") is about to be appended to the results of the first pass, parameter *p* is equal to static object *s*, in which the results of the first pass through function *operator+(const string &, const string &)* have been stored. Thus after the second call to the function, *s.start* and *p.start* both contain the same address.
Statement S'61' invokes constructor *string::string(int)*, which allocates a new area of memory with the length calculated for the whole string *(p.length + q.length -1)* to hold the result line. Here in the example this area contains binary zeros. The function *operator=(const string &)* copies the requested area of memory to *s*. This overwrites the first byte of *s.start->* with X'00' and, because *s* and *p* are identical on the second pass, also destroys *p.start->*. The call to *strcat* in statement S'63' thus combines *q* with the now empty string *s.start->*. To eliminate this error, *operator+(const string &,const string &)* must cause each new piece of text to be buffered in a dynamic object of class *string*. With this enhancement the final code for the function reads as follows:

```
string& operator + ( const string& p, const string& q )
{
   static string s;
   string s1 = p.length + q.length -1;         //allocate right length
   s1.start = strcpy(s1.start,p);              //copy first string
   s1.start = strcat(s1.start,q);              //copy second string
   s = s1;
   return s;
}
```

Step 10

The program is recompiled, loaded and started. The text is now output correctly:

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*CPLUSPLUS()
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW, BUILTIN-FUNCTIONS=*NONE
//COMPILE SOURCE=*LIB(MYLIB,STRING.C),MODULE-OUTPUT=*LIB(MYLIB,STRING.O)
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES START-LLM-CREATION = *YES,         -
//INCLUDE = *LIB-ELEM(L=MYLIB,E=STRING.O),                 -
//STDLIB=*STATIC,                                          -
//RUNTIME-LANGUAGE =*CPLUSPLUS(MODE=ANSI), TEST-SUPPORT = *YES
//BIND OUTPUT = *LIB-ELEM(LIB=MYLIB,ELEM=STRING)
%  BND1501 LLM FORMAT: '1'
%  BND1101 BINDER NORMALLY TERMINATED.
%//END
%  CDR9908 : END   C TIME USED = 5.6300 SEC
%  CCM0998 CPU TIME USED: 5.7149
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$STRING$', VERSION ' ' OF '2015-01-26 10:46:33' LOADED
/%r
Hello C++ World
Goodbye C World
%  CCM0998 CPU TIME USED: 0.0026 SECONDS
```

9.3 Sample C application under POSIX

You can find a sample application under POSIX in the manual „POSIX Commands“ [11].

10 Appendix

10.1 Comparison: debugging older objects / C++ V3.0 objects

C++ programs that were compiled with earlier versions of the C/C++ compiler up to V2.2C are subject to the same rules as those described in the previous manual for AID V2.1A (“Debugging C/C++ Programs“) even if you are debugging with AID as of V3.4B. This is due to the different LSD structure. There are also some deviations in connection with accessing data and functions of classes and transferring derived classes to base classes.

Objects compiled with C/C++ up to V2.2C	Objects compiled with C/C++ V3.0
Dynamic data of a class can only be accessed from within a dynamic member function via the `this` pointer.	Dynamic data of a class can be accessed from within a dynamic member function according to the same scope rules as apply in C++.
The class qualification of a member function is included in the function designation: `n'`*class*`::[...]`*function*`(`*signature*`)'`	The class qualification of a member function is in front of the function designation: *class*`::[...]n'`*function*`([`*signature*`])'`
The signature in a function designation must always be specified, even if it is `void`.	If the signature in a function designation is `void`, it must be omitted. However, the two parentheses must be written, as in C++.
The assignment “pointer to base class = pointer to derived class” cannot be executed with the %SET command. All dynamic data must be transferred individually instead.	The assignment “pointer to base class = pointer to derived class” can be executed with the %SET command.

Table 10: Differences in the debugging of older objects and objects of the C++ V3.0 compiler

Glossary

/390

Designation of a computer in the 7,500 (CISC) series.

addressing mode

The addressing mode determines how addresses are to be converted for the execution of machine instructions. By default, AID assumes the addressing mode of the object being debugged. This applies to the address length (24 or 31 bits) for programs (%AMODE) and also to the addressing of data spaces (%ASC).
System information on the address length can be referenced with the keyword %AMODE. This setting can be checked with %DISPLAY and modified with `%MOVE %MODE{24|31} INTO %AMODE`.
The keyword %ASC (access space control mode) references the system information for the AR mode (access register mode). It returns information on whether access registers for addressing data spaces are included in the address conversion. This setting can also be checked with %DISPLAY.

address operand

This is an operand used to address a memory location or memory area. The operand may specify virtual addresses, data names, statement names, source references, keywords, complex memory references, or an S, PROC or BLK qualification. The memory location or area is located either in the program which has been loaded or in a memory dump in a dump file. If you have assigned a name more than once in your program and thus no unambiguous address reference is possible, you can use area qualifications or a structure qualification to associate 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 (OMs) or link and load modules (LLMs). If the library has been assigned with the %SYMLIB command, LSD records can be dynamically loaded by AID.

AID literals

AID provides the user with both alphanumeric and numeric literals (see the chapter on “AID literals” in the AID Core Manual [1]):

```
{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
```

AID output files

AID output files are files to which the user can direct output of the %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP and %TRACE commands. The files are addressed via their link names (F0 through F7) in the output commands (see %OUT and %OUTFILE).
The REP records are written to the file assigned to link name F6 (see %AID REP=YES and %MOVE).

There are three ways of creating an output file, or of assigning an output file:

1. %OUTFILE command with link name and file name
2. ADD-FILE-LINK 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 uses the SAM access method, record format V, and is opened with MODE=EXTEND.

AID standard address interpretation

Indirect addresses, i.e. addresses which precede a pointer operator, are interpreted by default in accordance with the currently valid addressing mode of the debugged object. The %AINT command allows you to deviate from the default address interpretation of AID, i.e. to define whether AID is to work with 24-bit or 31-bit addresses in the case of indirect addressing.

AID standard work area

This is the non-privileged segment of virtual memory in your task that is occupied by the program and all 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 space in which memory references can be accessed without the need for a base qualification.
It includes the non-privileged segment of virtual memory in your task that is occupied by the program and all connected subsystems or the corresponding area in a memory dump.
You may deviate from the AID work area in a command by specifying a base qualification in the address operand. Using the %BASE command, you can shift the AID work area from the loaded program to a memory dump, or vice versa.

area checking

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').
The area limits for a CSECT or a COMMON as a memory object are determined by the start and end addresses of the CSECT/COMMON (see the section on "Machine code memory references" in the AID Core Manual [1]).

area qualifications

The S, PROC, BLK, and :: qualifications are called area qualifications. The S qualification designates a translation unit and is used to describe the path to a memory object which is not located in the current translation unit. The PROC qualification designates a function; the BLK qualification designates a block.
The :: qualification for the superblock is used to address global data or to designate functions that are not visible at the current interrupt point because their definition occurs later.
Area qualifications are used to describe the path to memory objects that are not in the scope of the interrupt point or are locally hidden at the interrupt point by identically-named definitions.

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. Using the name, AID can find all the associated attributes in the LSD records to be able to work with them.
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.

byte offset

In AID, this is an operation which allows address calculations, enabling you to move forward or backward from an address in steps measured in bytes.

character format

Output type for %DISPLAY (see also the section on “General storage types” in the AID Core Manual). If a storage area is edited with the following type modification %C, AID will output its contents in character notation.

child task

A task created via a `fork()` call.

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 prompts for command input with "/".

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, %ALIAS, %BASE, %DUMPFILE, %HELP, %OUT and %QUALIFY as well as the BS2000 commands listed in the appendix of the AID Core Manual.
If a command sequence contains one of the commands for runtime control, the command sequence is aborted at that point and the program is started (%CONTINUE, %RESUME, %TRACE) or halted (%STOP). As a result, any commands which follow as part of the command sequence are not executed.

constant

A constant represents a value which cannot be accessed via an address stored in program memory.
Constants include the results of length selections, length functions and address selections, as well as statement names and source references.
You can determine the length of a memory reference by using either the AID length selector %L(...) or the length operator `sizeof(...)`, which corresponds to the sizeof operator in the C/C++ language. As in C/C++, `sizeof` must be entered in lowercase letters (make sure that you set `%AID C=YES` or `%AID LOW={ON|ALL}`).
Similarly, there are two operators with which you can specify the address of a memory reference: the AID address selector %@(...) and the address operator &, which you should know from C/C++.
An address constant represents an address. Address constants include statement names, source references and the result of an address selection with the address selector %@(...) or the address operator &. 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 block and function nesting at the interrupt point. It extends from the block or function in which the program was interrupted, to the middle hierarchical levels (i.e. the superblocks or functions in which the corresponding function call is located), to the main function.
The current call hierarchy is output using %SDUMP %NEST.
For a recursive function, each call of itself is also output.

current program

The current program is the one loaded in the task in which the user enters AID commands.

current translation unit

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

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.
You specify a data name as in C, with the following exceptions:
You reference array elements only via subscript, not by way of a pointer.
For variables of type long double, AID evaluates only the first 8 bytes.
Unlike C/C++, AID does not treat a variable of type char as an arithmetic type (see data type). You can calculate with such a variable only after a type modification to %A (`unsigned char`) or %F (`signed char`).
For structures you may use pointer notation and structure qualification.
For arrays you can only use subscript notation.
For pointers you can use subscript notation, pointer notation, and dereferencing.

data type

In accordance with the data type declared in the source program, AID assigns an AID storage type to all data:
– binary string (≙ %X)
– character (≙ %C)
– numeric (≙ %D, %F, %A)

The data type `char`, which is treated as numeric in C/C++, is not numeric for AID. You can calculate with such a variable only after a type modification to %A (`unsigned char`) or %F (`signed char`). However, data of type `signed char` and `unsigned char` are also handled by AID as small integers with and without a leading sign, respectively.

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

debug mode

Designates the state of a task in which you can input AID commands for debugging. Debug mode in the LOGON task is identical to the BS2000 command mode. With fork tasks, AID handles the dialog between the user and the task. AID displays a command input prompt which is formed from the process number of the fork task.
Debug mode has a lower priority than the LOGON task command mode, i.e. the fork task does not have the same priority for using the terminal as the LOGON task.

dump format

Output type for %DISPLAY; corresponds to storage type %X (see the section on “General storage types” in the AID Core Manual). If a storage area is edited with the following type modification %X, AID will output its contents in both hexadecimal and character notation.

DMS file

BS2000 data management system file.
These can be single files or modules stored in PLAM libraries. Files can be copied between POSIX and BS2000 with the POSIX `bs2cp` command (see also UFS file).

ESD/ESV

The ESD for OMs/ESV for LLMs 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 link editor accesses this information when creating the object structure list (OMs) or the external symbol dictionary (LLMs).

exec()

Designates a function group to which the following functions belong: `execl()`, `execv()`, `execle()`, `execve()`, `execlp()`, `execvp()`.
An `exec()` call causes the program specified in the call to overlay the calling program.

external symbol dictionary

If the generation of an external symbol dictionary has not been suppressed, the link editor BINDER will create one on the basis of the ESV (External Symbols Vector).
If the external symbol dictionary was subsequently loaded, you can use %SDUMP %NEST to output the current call hierarchy even if the LSD information was not loaded at the same time.
Instead of the source references and the names of the translation units and functions, AID outputs the absolute addresses, CSECT names and the compiler-generated entry names of the functions.

fork()

System call which creates a copy of the process containing the `fork()` call. After the `fork()` call, an additional, identical process exists in the system.

fork task

A task created by a `fork()` call.

global settings

AID offers commands facilitating addressing, saving input efforts and enabling the behavior of AID to be adapted to individual requirements. The presettings specified in these commands continue to apply throughout the debugging session if not explicitly modified (see %AID, %AINT, %BASE, %OUT and %QUALIFY).

input buffer

AID has an internal input buffer. If this buffer is not large enough to accommodate a command input, the command is rejected with an error message identifying it as too long. You will then need to abbreviate the command or command sequence or distribute the function over multiple commands.

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 translation unit in which the interrupt point is located. The program is continued at this point.

LIFO

Stands for the **L**ast **I**n **F**irst **O**ut 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 the section on “Chaining” in the AID Core Manual).

localization information

Static program nesting for a given memory location is output by AID with %DISPLAY %HLLOC(*memref*) for the symbolic level and
%DISPLAY %LOC(*memref*) for the machine code level.
Conversely, %SDUMP %NEST outputs the **dynamic** program nesting, i.e. the call hierarchy for the current program interrupt point.

LOGON task

Task which is started with the SDF /SET-LOGON-PARAMETERS command.
The LOGON task command mode has a higher priority than the debug mode of a task created via fork(), which can cause problems when simultaneously debugging parent and child tasks.

LSD

The **L**ist for **S**ymbolic **D**ebugging 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, statement names, source references, the results of address selection, length selection and length function, and the AID literals do not constitute memory objects because they represent a value that cannot be changed.

memory reference

A memory reference addresses a memory object. Memory references can either be simple or complex.
Simple memory references include virtual addresses, a closing C or COM qualification, keywords, and names for which AID can obtain the address from the LSD information. Statement names and source references are allowed as memory references in the AID commands %CONTROL*n*, %DISASSEMBLE, %INSERT, %REMOVE and %TRACE, even though they are merely address constants.
Complex memory references instruct AID how to calculate a particular address and which type and length are to apply.

The following operations are possible here:
– byte offset
– indirect addressing
– type modification
– length modification
– address selection

monitoring

%CONTROL*n*, %INSERT and %ON are monitoring commands. When the program reaches a statement of the selected group (%CONTROL*n*) or the defined program address (%INSERT), or if the declared event occurs (%ON), program execution is interrupted and AID processes the specified subcommand.

namespace

This comprises all names assigned in the LSD records to a program unit, a function, or a block. It corresponds to scope in C/C++. You specify the name range via %SDUMP, specifying the appropriate qualification.

numeric output types

%F, %A and %D are the numeric output types (see the section on “General storage types” in the AID Core Manual [1]) . When a memory area is output with %DISPLAY and edited with one of the numeric output types, the following assignments apply:

%F	signed integer (equivalent to `int` in C/C++)
%A	unsigned integer (equivalent to `unsigned int` in C/C++); %A is also used in AID to interpret the contents of a memory location as an address before a pointer operator.
%D	Floating point number (equivalent to `float` in C/C++)

object structure list

If the object structure list was subsequently loaded, you can use %SDUMP %NEST to output the current call hierarchy even if the LSD information was not loaded at the same time.
Instead of the source references and the names of the translation units and functions, AID outputs absolute addresses, CSECT names and the compiler-generated entry names of the functions.

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. A list of all AID-specific storage types together with their output types can be found in the section on “general storage types” in the AID Core Manual. This assignment also applies for the data types used in C/C++.
A type modification in %DISPLAY and %SDUMP causes the output type to be changed as well.

parent task

The first task in the hierarchy of a task family.

pointer operator

This is the string ->, which you enter in an address operand when the contents of a memory object or the value of a constant is used for indirect addressing (see the section on “Indirect addressing” in the AID Core Manual [1]). The addressing mode is also taken into account for indirect addressing.

POSIX shell

A ported UNIX system program which provides communication between the user and system. The POSIX shell is a command interpreter which interprets the input commands into a language that the system can process.

process

A term from the UNIX world which is also used under POSIX. A process corresponds to a task at the BS2000 level. Process is used to designate the address space and the program executed in it as well as the required system resources. A process is created by another process by calling the `fork()` function. The process which calls `fork()` is known as the parent process (parent task in BS2000) and the new process created by `fork()` is known as the child process (child task in BS2000).

process number (pid)

A number assigned by the system to uniquely identify a process. AID forms the prompt output by a fork task as an input prompt, from the process number (Process Identification/pid).

program state

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

1. The program has stopped.
   %STOP or the K2 key interrupt a running program. The program is also interrupted when a %TRACE is completed and the `continue` operand is set to `S`. The task is in command mode. The user may enter commands.
2. The program is running **without** tracing.
   The program was loaded and started with `START-EXECUTABLE-PROGRAM` or started or continued with %RESUME. If no %TRACE has been defined, %CONTINUE can be used for the same purpose.
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.

qualification

A qualification allows you to reference a memory object which is not located in the AID work area or which has a name which is not unambiguous there.
The **base qualification** specifies whether the memory object is located in the loaded program or in a memory dump.
The **S qualification** specifies in which translation unit the memory object is located.
The **PROC** or **BLK qualification** specifies in which function or block the memory object is located. Both of these qualifications are used to reference data names declared as static which are located in a function or block outside the current call hierarchy, or to reference data names located in a function or block within the current call hierarchy, but which are hidden at the interrupt point by a definition with the same name.
The **:: qualification** designates a global data item or a function. Global data and functions can be referenced by the two prepended colons from the interrupt point even if they are defined after the interrupt point.

scope

The scope of local data extends from the point at which it is defined to the end of the block containing the definition and includes any blocks nested within that block. If the definition comes at the beginning of the function, or if it is in the form of a passed parameter, the scope extends over the entire function. In the case of external variables and functions, the scope extends from the point of declaration to the end of the translation unit. The scope of a label is the whole of the function in which it is defined.
AID cannot address data, function names or labels without qualifications, unless the current interrupt point is within the scope of the corresponding name. In the case of data names this applies only if the name is not hidden locally by a definition with the same name.

signature

The parenthesized type specifications for the transfer parameters of a function are called its signature. In the case of functions from C++ programs, the signature is included in the function name. Due to the special characters (parentheses and possibly commas), the name must be specified within `n'...'`. No additional blanks may be inserted. If the signature is `void`, only the parentheses are written. The exact function name in standard C++ notation can be determined from the %SDUMP output. If the function is defined in a class, AID displays the exact function name in the %DISPLAY output for that class.

source reference

A source reference designates a name generated by the compiler with which you can reference any executable statement. The name consists of the line number, which may have a `FILE` number prefixed or a line-specific statement number appended to it: `S'[`*f*`-]`*n*`[:`*a*`]'`. The LSD records hold an address constant associated with the source reference which contains the address of the statement. More precisely, it contains the address of the first instruction generated for the statement.

statement name

This designates a name declared in the source program which can be used in AID to reference an executable statement. The ones relevant for debugging C/C++ programs are labels and function names. An address constant is stored in the LSD records for this purpose; in the case of labels, the address of the first statement after the label is stored.
When you use function names in the %DISASSEMBLE and %INSERT commands, you designate the first executable statement of the corresponding function; in all other commands, you reference the prolog address of that function.

storage type

This is either the data type defined in the source program or the one selected by way of type modification. AID recognizes the general storage types %X, %C, %E, %P, %D, %F and %A and the special storage types %SX and %S for the interpretation of machine instructions (see %SET and the chapters on “addressing in AID” and “Keywords” in the AID Core Manual).

subcommand

A subcommand is an operand of the monitoring commands %CONTROL*n*, %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. 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 can be found in the chapter “Subcommand” of the AID Core Manual [1].
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.

subscript

Subscripts are used to address the elements of an array. In AID as in C/C++, subscript notation can be used both for arrays and for pointers. As in C/C++, a subscript can be specified in square brackets.

superblock

Designates the outermost block in a translation unit. This corresponds to the global namespace in C++.
All global data and all functions are assigned to the superblock.
Namespaces can only be defined in the superblock.

task family

All tasks of all generations created from one task with `fork()`.

tracing

%TRACE is a tracing command. You use it to define the type and number of statements (symbolic debugging) or instructions (machine code level) to be logged.
Program execution is normally traced at the screen, but %OUT %TRACE may be specified to redirect the output to some other output medium.

translation unit

The part of a C/C++ program which is compiled as a unit. It can be referenced via the S qualification.

UFS file

UNIX file system file.
As with UNIX, the files are also stored under POSIX in hierarchically organized directories. The C/C++ compiler can process both UFS and DMS files (see DMS file). However, you can only use the AID %SYMLIB command on PLAM libraries in BS2000.

update dialog

The update dialog is initiated by means of the %AID CHECK=ALL command. It goes into effect when a %MOVE or %SET command is executed. During the dialog, AID asks whether updating of the memory contents is to take place. If N is entered in response, no modification is carried out; if Y is entered, AID will execute the transfer.

user area

This is the area in virtual memory which is occupied by the loaded program and all its connected subsystems. It corresponds to the area represented by the keyword %CLASS6 or %CLASS6ABOVE and %CLASS6BELOW.

Related publications

You will find the manuals on the internet at *http://manuals.ts.fujitsu.com*. You can order printed copies of those manuals which are displayed with an order number.

AID

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

[2] **AID** (BS2000)
Debugging on Machine Code Level
User Guide

[3] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of COBOL Programs
User Guide

[4] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of FORTRAN Programs
User Guide

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

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

[7] **AID** (BS2000)
Advanced Interactive Debugger
Ready Reference
User Guide

C/C++

[8] **C/C++** (BS2000)
C/C++ Compiler
User Guide

[9] **C/C++** (BS2000)
POSIX Commands of the C/C++ Compiler
User Guide

POSIX

[10] **POSIX** (BS2000)
POSIX Basics for Users and System Administrators
User Guide

[11] **POSIX** (BS2000)
Commands
User Guide

BS2000

[12] **CRTE** (BS2000)
Common RunTime Environment
User Guide

[13] **BINDER** (BS2000)
User Guide

[14] **BS2000 OSD/BC**
Dynamic Binder Loader / Starter
User Guide

[15] **BS2000 OSD/BC**
Executive Macros
User Guide

[16] **BS2000 OSD/BC**
Commands
User Guide

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

Index