English

FUJITSU Software BS2000

AID V3.4B

Debugging under POSIX

Supplement

Edition November 2015

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

Certified documentation according to DIN EN ISO 9001:2008

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 9001:2008.

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

Copyright and Trademarks

Contents

1 Preface

The interactive debugger AID can be used to test not only pure BS2000 programs, but also mixed-mode programs. Pure POSIX programs are executed entirely in the POSIX shell. Mixed-mode programs, by contrast, are BS2000 programs that use POSIX interfaces. AID can be used to debug pure BS2000 programs.

1.1 Structure of the AID documentation

The AID documentation consists of an 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 (see page 42), which shows the syntax of all commands and operands together with brief explanatory notes. It also includes the %SET tables and a comparison of AID and IDA. 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 (see page 41).

AID Core Manual

The core manual provides an overview of AID and deals with the topics and operands that are common to all programming languages. The AID overview describes the BS2000 environment, explains basic concepts and presents the AID command set. The other chapters discuss preparations for testing; command input; the operands *subcmd*, *compl-memref* and *medium-a-quantity*; AID literals and keywords. The manual also includes messages, the BS2000 commands that are invalid in command sequences, and a comparison of AID and IDA.

AID User Guides

The User Guides deal with the commands in alphabetical order and describe all simple memory references. Apart from the manual,
AID - Debugging of C and C++ programs,
the available User Guides are:

AID - Debugging of COBOL Programs
AID - Debugging of FORTRAN Programs
AID - Debugging of PL/I Programs
AID - Debugging of ASSEMBH Programs
AID - Debugging on Machine Code Level

In the language-specific manuals, the description of operands has been customized to each respective programming language. It is assumed that the user is familiar with the scope of the language involved and the operation of the relevant compiler.

The manual for debugging on machine code level can be used for programs for which no LSD records exist or for which the information obtained from symbolic debugging is not sufficient for error diagnostics. Debugging on machine code level means that you can issue AID commands regardless of the language in which the program was written.

1.2 Summary of contents

This supplement describes the use of the advanced interactive debugger AID under POSIX. It provides, together with the core manual and the manual “Debugging of C/C++ Programs”, all the information that you will need to debug any C or C++ program in the POSIX environment. Each section in this supplement begins with a list of the manuals and the sections that are affected by the changes described therein.

Following the preface, this supplement contains the sections listed below:

Metasyntax
: Contains a list of notational conventions used in this manual.

Prerequisites for debugging
: Describes the operands that are needed when compiling, linking and loading in order to pass the LSD to the loaded program and the operands and options that enable the program to be executed under POSIX.

Extended AID commands
: Describes the extensions to the AID commands %AID, %SHOW and %STOP.

The POSIX debug command
: Describes how programs that are loaded and started in the POSIX shell can be debugged.

Special debugging aspects
: This section contains basic information on inheriting the debugging context and on processing dumps together with some notes on specific strategies that are effective when debugging fork tasks and programs loaded with `exec()` calls. It also includes an application example.

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

The Readme files for AID V3.0 and AID V3.4A have been incorporated in the manual:

- Information on debugging COBOL programs under POSIX.

1.4 Notational conventions

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

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

2 Notational conventions

The following notational conventions are used in this manual to represent commands. The individual symbols and their meanings are listed below.

UPPERCASE
: String that must be accepted exactly as shown when selecting a function.

boldprint
: Lowercase letters and special characters to be entered exactly as shown. Boldprint is also used to indicate user inputs in examples.

lowercase letters
: String that identifies a variable, which must be replaced by one of the permitted operand values.

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

`{alternative | ... | alternative}`
: A list of alternatives, from which one must be selected. The above two formats are equivalent.

`[optional]`
: Entries enclosed within square brackets are optional and may be dropped.
In the case of AID command names, the part shown in square brackets may only be dropped in its entirety; other abbreviations will result in a syntax error.

`[...]` Indicates that an optional syntax unit may be repeated. In cases where each repetition must be preceded by a delimiter (such as a comma), the appropriate delimiter is shown before the ellipses.

`{...}` Indicates that a syntax unit, which must be specified once, may be repeated. In cases where each repetition must be preceded by a delimiter (such as a comma), the appropriate delimiter is shown before the ellipses.

Underlining
: The default values used by AID in cases where no values are specified for operands are shown underlined.

Notational conventions used in explanatory text

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

This symbol is used to mark text locations with important information that must be carefully observed.

3 Prerequisites for debugging

Supplements the chapter of the same name in the manual
“AID - Debugging of C/C++ Programs“ (see page 41).

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.
The control statements required for the C/C++ compiler to generate the LSD on BS2000 and POSIX levels are described in the first two sections of this chapter. 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.
The sections that follow also list the operands that must be specified when compiling, linking and loading in order to generate and execute a POSIX- capable program.

If the program was loaded without the LSD, AID allows you to load the LSD dynamically if required. As a prerequisite, the LSD and the associated program must be placed in a PLAM library. This may have been done directly by the compiler at compilation. If not, you may also copy the entire program with the LSD from the POSIX file system to a PLAM library by using the POSIX command `bs2cp`. A description of how the LSD can be loaded dynamically with the AID command %SYMLIB and a brief explanation of the `bs2cp` command can be found in the last section of this chapter.

Information on debugging COBOL programs under POSIX

AID enables you to debug a COBOL Program compiled in the POSIX environment using the POSIX `COBOL` command in exactly the same way as described for C/C++ Programs in the "Debugging under POSIX" supplement.

You cannot use the `%TRACE 1` command specified in the section "Commands at the start of a debugging session" of the manual "Debugging COBOL Programs" to position the program to the first executable statement of the main program after loading it in the POSIX environment. The following command sequence must be used instead

```
%INSERT 'prog-id'; %RESUME
```

For `'prog-id'`, enter the name of the main program.

3.1 Compiling, linking and loading in BS2000

Compiling

The generation of LSD information can be controlled with the START command for the C/C++ compiler by means of the following option:

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

NO No LSD information is generated by the compiler with the default setting NO. It is possible to retrace the call hierarchy (%SDUMP %NEST) even without this LSD information, but only on machine code level.

YES The compiler generates LSD information.

LSD information can be generated only for unoptimized programs. If optimization has been turned on in any case (cf. MODIFY-OPTIMIZATION-PROPERTIES statement), the compiler will reset the optimization level to LOW and issue a corresponding message. The generation of LSD information also has an effect on inlining in C++ programs, i.e. inline functions are generated as “outline” functions.

The following two options must be specified to enable the use of POSIX interfaces by the program:

– The `_OSD_POSIX` define must be set before the first `#include` statement in the program. The easiest way to achieve this is to specify the following option at compilation.
```
//MODIFY-SOURCE-PROPERTIES DEFINE=_OSD_POSIX
```

– To enable the search for standard headers, the library containing the standard include elements for POSIX functions, i.e. `SYSLIB.POSIX-HEADER`, must be specified at compilation in addition to the CRTE library `SYSLNK.CRTE`. This can be done by using the option:
```
//MODIFY-INCLUDE-LIBRARIES STD-INCLUDE-LIBRARY=
(*STANDARD-LIBRARY,$.SYSLIB.POSIX-HEADER)
```

If parameters for the `main` function are to be read by the program (as is usual in UNIX), the following option must be set at compilation:

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

This causes the program to halt immediately after it is started and to wait for the input of parameters for the `main` function or for redirections of `stdin/stdout` or `stderr`. If the program was started in the POSIX shell, entering this option will have no effect, since parameters and redirections are specified directly on the command line as in UNIX.

A detailed description of the operands that control compilation can be found in the C and C++ User Guide (see page 42).

Linking, loading and starting

Compiled programs can be linked, loaded and started by using the SDF commands which apply to all languages that are described in the chapter on “Prerequisites for debugging with AID” in the AID Core Manual. The same manual also describes the respective parameters that instruct the compiler to pass the LSD information which it generates to the linkage editor (BINDER), the dynamic binder loader DBL or the static starter (ELDE) so that you can perform symbolic debugging.

If you wish to use the POSIX functions of the C runtime system, you must specify the “linkage option” library SYSLNK.CRTE.POSIX at link time. The module contained in it must be given precedence over the modules of other CRTE libraries to be linked. The library SYSLNK.CRTE.POSIX must therefore be assigned a lower link name BLSLIB*nn* than any other CRTE libraries that follow if it is to be used for dynamic linkage with DBL.

Example

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

If you are using BINDER for static linkage and link the SYSLNK.CRTE.POSIX library by means of an INCLUDE-MODULES or INCLUDE statement, the module from the “linkage option” library is guaranteed to be linked before the modules of the runtime system.

– BINDER statement:

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

More information on the common runtime environment CRTE can be found in the manual “CRTE - Common RunTime Environment” (see page 42).

3.2 Compiling, linking and loading under POSIX

Compiling and linking

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

cc-Xt Invokes the compiler in Kernighan-Ritchie mode

c89 Invokes the compiler in ANSI/ISO mode

CC Invokes the compiler in C++ mode

The C/C++ compiler generates LSD information if you specify the `-g` option:

If `-g` is not specified, you will not be able to debug the program symbolically. However, it may still be debugged on machine code level.

If the `-O` or `-F` option has been set to turn on optimization, the generated program can be debugged only to a restricted extent, i.e.:

– Breakpoints can be set only at function entry points.

– It is only at these breakpoints that function parameters and global data can be definitely displayed.

A detailed description of the commands `cc`, `c89` and `CC` can be found in the manual “POSIX Commands of the C and C++ Compilers” (see page 42).

Loading and starting

The POSIX command `debug`, which is described in detail in chapter 5, can be used to load the program together with the associated LSD information. When loading has completed, AID issues the message AID0348, from which you can obtain the process ID (pid) of the generated process, and the debugging mode prompt. You may then enter AID commands for debugging and start the program with %RESUME.

If you load and start the program in the POSIX shell directly, i.e. without using the `debug` command, the program will be unloaded if it aborts with an error. In contrast to the BS2000 level, you will then have no chance to immediately examine the error environment and the cause of the error and to potentially recover from the error and continue with the program.

3.3 Dynamic loading of the LSD

Programs which are already in productive use are usually loaded without the LSD. This is also advisable for large programs in which only a few individual modules are to be debugged symbolically. In such cases, AID can access the associated LSD dynamically at a later stage if the module was stored together with the LSD in a PLAM library. To enable this, you must specify the PLAM library containing the program with the LSD information in the %SYMLIB command. This PLAM library will be subsequently opened by AID to search for the required information whenever you address a symbolic memory reference in an AID command. The same approach can also be used if the program is running in the POSIX shell. Since %SYMLIB does not support access to UFS files, the program with the LSD must be stored in a BS2000 PLAM library in this case as well. If the program was compiled in the POSIX shell, you will need to copy the generated object to BS2000 with the POSIX command `bs2cp` and store it there as a type-L member in a PLAM library.

The LSD cannot be loaded together with programs that are loaded with `exec()`. The procedure described above must therefore always be used here to enable symbolic debugging.

A detailed description of the AID command %SYMLIB can be found in the manual “AID - Debugging of C/C++ Programs” (see page 41).

Brief description of bs2cp

`bs2cp` copies files from the POSIX file system to BS2000 and vice versa. Such BS2000 files may be DMS files or members of BS2000 PLAM libraries. A detailed description of `bs2cp` can be found in the manual “POSIX Commands“ (see page 42).

```
Syntax- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

bs2cp[␣-k][␣-h][␣-f]␣[bs2:]file␣[bs2:]filecopy

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

-k The file contents are converted on copying:

- from ASCII to EBCDIC if *file* is a file from the POSIX file system.
- from EBCDIC to ASCII if *file* is a BS2000 file.

This option is required only when copying files from UNIX systems. Programs that are compiled in the POSIX shell are saved by the compiler in EBCDIC.

-h Shows the command syntax with an explanation of the options.

-f If a file or library member named *filecopy* already exists in BS2000, it is overwritten without confirmation.

bs2: Indicates that the following *file* or *filecopy* is a BS2000 file.

A DMS file is addressed by name.

A library member is identified as follows:
`'lib(elem[,type[,vers]])'`

lib Name of the PLAM library in BS2000

elem Name of the library member

type Type of member. The default value is `S`.

vers Version of the member. The default value is `*HIGH`.

file Name of the file to be copied.

filecopy

Filename of the copy

Only one of the files (*file* or *filecopy*) must be a BS2000 file.

Example

```
$bs2cp prog1 bs2:'test.lib(prog1,1)'
$debug sym_test prog1
%  AID0348 Program stopped due to EXEC event (PID=0000000224)
%0000000224/...
...
%0000000224/%resume
%  AID0348 Program stopped due to EXEC event (PID=0000000224)
%0000000224/%symlib test.lib
...
```

The object module `prog1` is copied from the POSIX file system to BS2000 and entered there as an LLM named `PROG1` in the library `TEST.LIB`. The POSIX `debug` command loads the program `sym_test`, which in turn includes an `exec()` call to load the program `prog1`. Following the successful `exec()`, the library containing the LSD for `prog1` is reported with %SYMLIB, which means that `prog1` can now also be debugged symbolically.

4 Enhanced AID commands

4.1 %AID

Supplements the “Administration functions” section in the Core Manual and the “AID commands” chapters in the language-specific manuals, the manual for debugging on machine code level, and the Ready Reference (see page 41).

The LOW operand in the %AID command can now assume the additional value ALL, which prevents lowercase letters from being converted to uppercase even in the S qualification Two new operands, which enable the debugging of fork tasks and programs loaded by `exec()`, were introduced.

– *LOW* defines whether or not lowercase letters of AID from character literals and names are to be converted into uppercase.

– *FORK* defines whether a task generated by `fork()` is to be interrupted immediately after its creation and placed in debugging mode.

– *EXEC* defines whether debugging mode is activated after loading with `exec()`.

Command	Operand
%AID	{ LOW [= {ON | OFF | ALL}] | FORK [= {OFF| NEXT| ALL}] | EXEC [= {OFF| ON}] }

The following applies with respect to the scope and validity period of declarations made with %AID:

– Settings made with %AID remain valid in the LOGON task until they are modified by a new %AID command or until /LOGOFF.

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

– `exec()` calls do not affect declarations made with %AID.

– In the POSIX shell, all declarations made with %AID except for FORK=ALL are reset after loading with `debug` *progname* (see page 23). If FORK=ALL was set in the LOGON task, it will remain in effect. EXEC=ON is set (i.e. turned on) whenever a program is loaded with `debug` *progname*.

%AID may be entered only as a single command; it must not appear in a command sequence or a subcommand.

LOW

ON Lowercase letters in character literals and in program, data, and statement names are not converted into uppercase. If you are debugging C/C++ programs, you should enter %AID LOW at the start of each debugging session. It is only then that a distinction between uppercase and lowercase can be made by AID in C/C++. AID does not distinguish between uppercase and lowercase only for the S qualification. Entries in the S qualification are always converted into uppercase letters.

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

ALL In addition to all inputs that are affected by the LOW=ON setting, LOW=ALL causes even the lowercase/uppercase notation in an S qualification to be retained. This setting is needed if the program to be debugged was compiled in the POSIX shell and if the source file associated with it includes lowercase letters in its name.

Note that the preset and default values for the *LOW* operand do not match. If no *LOW* operand has been specified in a debugging session as yet, the preset value is OFF. However, if the *LOW* operand is entered without a value, the default value is ON. If you want to reactivate the conversion to uppercase letters, you will have to enter the complete command %AID LOW=OFF.

FORK

OFF Fork tasks are not interrupted after being created and are not placed in debugging mode. This is the default value. If %AID FORK has not yet been set for a task, the %SHOW %AID command will indicate the value NOT_USED for FORK.

NEXT All fork tasks of the first generation are interrupted immediately after they are created and automatically placed in debugging mode. Note, however, that FORK=OFF is set in these tasks, which means that tasks of the second and higher generations created by `fork()` cannot be debugged without additional measures. In this case, you can place such a fork task of a higher generation in debugging mode only by interrupting it from the POSIX shell with `debug -p` *pid* (see page 23) or by sending a %STOP command to it with the appropriate *TSN* or *pid* (see page 21) from another task of the same task family.

ALL All fork tasks of any generation that originate from the current task are interrupted after creation and placed in debugging mode. FORK=ALL is set in all such fork tasks. This setting is the only declaration made with %AID that is inherited.

Changing this option will only affect tasks which are created after the change in direct lineage from the task in which the option was set.

%AID *FORK* without a value is equivalent to %AID FORK=OFF (the default).

EXEC

OFF Programs that are loaded with an `exec()` call are not interrupted after loading and not placed in debugging mode.

ON The program is interrupted immediately after it is loaded with `exec()` and is placed in debugging mode. All earlier settings made with %AID are retained.

%AID *EXEC* without a value is equivalent to %AID EXEC=OFF (the default).

4.2 %SHOW

Supplements the “AID commands” chapters in the language-specific manuals, the manual for debugging on machine code level, and the Ready Reference (see page 41).

The new AID options were added to the output of the %SHOW %AID command.

This command can be used in command sequences and subcommands.

Example

```
$debug examp
%  AID0348 Program stopped due to EXEC event (PID=0000000891)
%0000000891/%aid low=all
%0000000891/%show %aid
A I D V03.4B OF  2015-02-25
Copyright (C) 2015 Fujitsu Technology Solutions
All Rights Reserved

E=VM : %AINT = %MODE31

%AID CHECK      =   NO
%AID REP        =   NO
%AID SYMCHARS   =   STD
%AID OV         =   NO
%AID LOW        =   ALL
%AID DELIM      =   '|'
%AID LANG       =   D
%AID FORK       =   NOT_USED
%AID EXEC       =   ON
%AID C          =   NO
%AID EBCDIC     =   EDF03IRV
```

After the program is loaded with `debug`, all conversions from lowercase to uppercase letters are initially suppressed, even for entries in the S qualification. The currently applicable settings for %AID are then requested with %SHOW %AID. The output of the %SHOW command lists the preset values for all operand values of %AID, except for %AID LOW and %AID EXEC. LOW was explicitly set to ALL, and EXEC is always turned on in the POSIX shell on loading a program with `debug`. In the case of %AID FORK, the entry NOT_USED corresponds to the OFF setting; NOT_USED merely indicates that the FORK option was not yet set in this task.

4.3 %STOP

Supplements the “Administration functions” section in the Core Manual and the “AID commands” chapters in the language-specific manuals, the manual for debugging on machine code level, and the Ready Reference (see page 41).

The AID command %STOP was extended by adding two new operands: T=*tsn* (task sequence number) and PID=*pid* (process identification), which can be used to interrupt a task generated by `fork()`. AID reports the process ID (*pid)* of the interrupted task. You can then control its subsequent execution by means of AID commands.

Command	Operand
%STOP	[{ T=tsn | PID=pid }]

If %STOP is entered in a command sequence or a subcommand, the commands that follow will no longer be executed.

When a fork task is interrupted by a %STOP command, the interrupt point may not always be in the user program; it could also lie in the routines of the runtime system. In order to avoid having to address the functions and variables of the program with a full qualification every time, it is advisable to first run the program with `%TRACE 1 IN S=`*srcname* until the next executable statement.

T

tsn The fork task to be placed in debugging mode is addressed via its TSN (task sequence number).

PID

pid The fork task to be placed in debugging mode is addressed via its process ID.

Example

The program `exstop` is a C program that contains a `fork()` call. When the fork task has been created, the parent task is to be halted with a %STOP command:

```
$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: 0EUV *********************************
SRC_REF: 45 SOURCE: exstop.c  BLK : 39  *************************
count           =         933
```

After the program is loaded with the POSIX command `debug`, `%AID FORK=NEXT` is used to specify that even the fork task created by `exstop` is to be run in debugging mode. In addition, `%AID LOW=ALL` is set, since the name of the source file `exstop.c` in the S qualification would be otherwise converted to uppercase letters.
The parent task runs under the `pid` 876, and the child task is assigned `pid` 877. The command `%STOP PID=876` is then used to interrupt the parent task. AID responds with the prompt `%0000000876/`. The next %TRACE command causes the parent task to halt before the next executable statement, which means that you can now address the variables of the parent task without a qualification. However, since both tasks are now competing for the terminal, you must respond to the prompt of the task that is not needed with EM DÜ so that the task which you want to debug has a chance to access the terminal.

5 The POSIX debug command

Extension to the manual “AID - Debugging of C/C++ Programs“ (see page 41).

The `debug` command enables you to debug POSIX programs that were started in the POSIX shell. You can use `debug` to load a program together with the LSD in the POSIX shell or to interrupt a running process and place it in debugging mode.

The use of `debug` not allowed in POSIX sessions that were opened via `rlogin` for system security reasons.

Syntax

```
debug { [␣-e]␣progname[␣argument]... }
      { ␣-p␣pid                      }
```

debug [␣**-e**] ␣progname [␣argument]...

The program *progname* is loaded into a task generated by the shell with `fork()` and placed in debugging mode; AID responds with a prompt that is constructed from the process ID (pid) of the task. You can then enter AID commands for debugging. The `-e` option controls whether (without `-e`) or not (with `-e`) the LSD is to be loaded for symbolic debugging. The command `debug` *progname* in the POSIX shell thus corresponds to the BS2000 command `LOAD-PROGRAM` *progname* with the operand `TEST-OPTIONS=YES` in the BS2000 environment if you do not specify `-e`. If you specify `-e` it corresponds to the `LOAD-PROGRAM` command with `TEST-OPTIONS=NO`.

-e *progname* is loaded without the LSD.

progname
: Name of the program to be debugged.

argument
: Argument for *progname*.

debug ␣**-p**␣pid

The process with the specified *pid* is taken over by AID and interrupted if the process designated by that *pid* belongs to the same task family as AID. Note that the POSIX shell is 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 21), which you can enter in BS2000 command mode or in debugging mode.

-p The program was taken over via the associated *pid*.

pid Process ID of the task that is to be taken over by AID and interrupted.

Example

This example illustrates how a running program is taken over by AID:

```
$ ps -ef ------------------------------------------------------------- (1)
      UID   PID  PPID  C    STIME TTY       TIME CMD
   D89239   890   824  0 10:22:38 term/003  0:01 [ps]
   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)
%0000000888/%stop pid=889 ------------------------------------------- (2)
%  AID0492 %STOP was sent to fork task (PID=0000000889).
%0000000888/%aid low=all -------------------------------------------- (3)
%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
%0000000889/%aid low=all -------------------------------------------- (4)
%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/...
```

(1) To begin with, a list of all running processes is requested with the POSIX command `ps -ef`. You can use this list to obtain the PID of the process to be examined with AID (888). This process is the parent task for the fork task with PID 889. The command `debug -p 888` interrupts the parent task and places it in debugging mode.

(2) The child task is also interrupted. Both tasks subsequently report with their prompts.

(3) In the next step, the parent task is to be continued until the next statement following the breakpoint. To enable AID to process the command %TRACE 1 IN S=*srcname*, case sensitivity for the S qualifier must be turned on with %AID LOW=ALL, and the PLAM library containing the LSD for the program `pexec` must be reported with %SYMLIB.
Since the parent and child tasks are executing concurrently, it is advisable in the interest of clarity to respond to the prompt of the other respective task with EM DÜ until the output of the %TRACE command is complete.

(4) The same procedure as described under point (3) is also performed for the child task.

6 Special debugging aspects

Supplements the manual “AID - Debugging of C/C++ Programs” (see page 41).

6.1 Inheriting the debugging context

The only setting that is inherited by a task created by `fork()` is %AID FORK=ALL, assuming that this was set in the parent task. All other declarations such as:

– settings defined with %AID,
– the set breakpoints,
– events monitored with %ON,
– PLAM libraries reported with %SYMLIB, etc.,

are reset in the fork task.

Programs loaded with `exec()`, by contrast, inherit the settings and declarations made with %AID and %SYMLIB, respectively, but all other declarations are also reset as in a fork task.

6.2 Debugging strategies

If you have only a BS2000 terminal or a corresponding emulation available for debugging fork tasks, you may face a problem when debugging multiple fork tasks that are executed concurrently, since these tasks will compete for the terminal.

This section contains some tips that will help you establish the best approach to guarantee quick success when debugging fork tasks and programs loaded via an `exec()` call.

One suitable strategy is to first test every program section, i.e. the parent task, the task created by `fork()`, and the programs loaded by `exec()` independently of one another and as thoroughly as possible. This is even more advantageous for programs that are to be loaded later via `exec()` calls. When the program is loaded by means of an `exec()` call, the LSD cannot be loaded with it and must be explicitly assigned using %SYMLIB. However, if you load the program with the POSIX command `debug`, the LSD information can also be loaded with it.

The context of the call should be tested separately. It is only when all program segments execute without errors and the call context has also been debugged that you should attempt to test the entire program structure. This is best accomplished by adding `fork()` and `exec()` calls in succession while the respective parent task is sleeping, a state that can be easily achieved by temporarily inserting a sufficiently long loop or a suitable `wait()` call.

Every program segment should identify its outputs in the debugging phase so that they can be correctly associated with their source. More details on the association of inputs/outputs when debugging multiple fork tasks concurrently can be found in section 6.3.2.

In general, you should first debug the program in the the POSIX shell, since the various fork tasks are executed here with the same priority, i.e. every fork task has an equal opportunity to access the terminal in order to request inputs or display outputs. If the program is started in the LOGON task, by contrast, the BS2000 command mode will have a higher priority than the debugging mode of the fork tasks. This could result in the parent task blocking the terminal and thus preventing the fork tasks from completing the required I/O at the terminal.

If you are debugging multiple fork tasks, it is generally useful to create a table in the format below so that you can record the number of each respective fork task, the associated process ID and TSN, the source references of the `fork()` call, and the location of the current breakpoint::

Fork number	**pid**	**TSN**	**Source reference**	
			Startup source code of the fork	**Current breakpoint**
F1	929	0ND1		168
F11	930	0ND2	110	124, 128
...	...	...	...	...

Table 1: Overview of active fork tasks

Note that the program will also be halted in the runtime system immediately after a `fork()` or `exec()`. You can access data, functions and source references from this breakpoint, but only with full qualifications. Consequently, you can save yourself some typing by first moving to the next executable statement of the user program by entering the command `%TRACE 1 IN S=`*srcname*.

The K2 key cannot be used under POSIX. To interrupt a POSIX process, you must enter the string “@@c”. The POSIX shell will then respond with its prompt, which is usually “$”. A task in debugging mode can be interrupted with the BS2000 command EXIT-JOB or LOGOFF. Another possibility is to enter the command CANCEL-JOB with the TSN of the task to be halted. This must be done from some other task (see “BS000 - User Commands(SDF Format)”).

6.3 Inputs/Outputs

When multiple fork tasks are debugged, the individual tasks compete for the terminal. The outputs of the various fork tasks are initially placed in a queue and then processed in order. Consequently, some of the rules that apply to inputs/outputs in debugging mode may deviate from those used during “normal” debugging in the BS2000 command. The following sections deal with the possible inputs in debugging mode, problems in associating I/O with various tasks, and the possibility of erratic behavior.

6.3.1 Possible inputs

All AID commands and most of the BS2000 commands can be entered in debugging mode. This includes all BS2000 commands that may also be specified in a command sequence and in subcommands (see the “AID Core Manual”). No guided SDF dialog is possible.

Command sequences consisting of AID and BS2000 commands delimited by a semicolon (;) may also be entered. This is likewise subject to the restrictions described in the section “Command sequences and subcommands” in the AID Core Manual (see page 41). The same restrictions also apply if only BS2000 commands are entered in debugging mode, i.e. even for individual commands.
As in BS2000 command mode, you may enter only EM DÜ in debugging mode as well. This “null” input is required in debugging mode to provide the desired task, which may be one of many in a fork family, with an opportunity to respond at the terminal with its prompt. You may, of course, have to enter EM DÜ several times, since the tasks respond at the terminal in the order in which the associated I/O requests were placed on the queue.

Example

```
$ debug ex1fork ——————————————————————————————————————————— (1)
%  AID0348 Program stopped due to EXEC event (PID=0000002893)
%0000002893/%on %svc(44) <%trace 1 %instr>
%0000002893/%aid fork=next
%0000002893/%resume ——————————————————————————————————————— (2)
ICXSVCTU+D6AA     SVC   44                      2 FCT=POSIX    IR1=01023A40
STOPPED AT V‘EC10AC‘ = ICXSVCTU + #‘D6AC‘ , END OF TRACE
%0000002893/%r
ICXSVCTU+71B2     SVC   44                      1 FCT=POSIX    IR1=01023A40
STOPPED AT V‘EBABB4‘ = ICXSVCTU + #‘71B4‘ , END OF TRACE
%0000002893/%r
ICXSVCTU+D2DA     SVC   44                      0 FCT=POSCONIN IR1=01023A00
                                                 PAR=00E4B401 00000000 00000000
STOPPED AT V‘ECOCDC‘ = ICXSVCTU + #‘D2DC‘ , END OF TRACE
%0000002893/%r
ICXSVCTU+9B9A     SVC   44                      1 FCT=POSSPEND IR1=01023BC8
```

```
                                                PAR=00E36301 00000000 00000000
STOPPED AT V'EBD59C' = ICXSVCTU + #'9B9C' , END OF TRACE
%0000002893/%r
ICXSVCTU+93FA     SVC   44                     1 FCT=POSSPMSK IR1=01023BC8
                                                PAR=00E35F01 00000000 00000000
STOPPED AT V'EBCDFC' = ICXSVCTU + #'93FC' , END OF TRACE
%0000002893/%r
ICXSVCTU+318      SVC   44                     1 FCT=POSFORK  IR1=01023950
                                                PAR=00E30201 00000000 00000000
STOPPED AT V'EB3D1A' = ICXSVCTU + #'31A' , END OF TRACE
%0000002893/[EM] [DÜ] ------------------------------------------------------------ (3)
%  AID0348 Program stopped due to FORK event (PID=0000002897)
%0000002893/[EM] [DÜ]
%0000002897/%show %base ---------------------------------------------------------- (4)
%0000002893/[EM] [DÜ]
%BASE E=VM
TSN: 0J05    TID: 0091017D
%AINT = %MODE31
BS:  V12.0   HW:  CFCS V3
%0000002897/
```

(1) The program is first loaded with the POSIX command `debug`. The program `ex1fork` contains a `fork()` call.
The AID command %ON then activates monitoring of the SVC with number 44. The associated subcommand causes the SVC to be executed and the program to be halted immediately thereafter.
The `%AID FORK=NEXT` command which you enter then activates debugging mode for fork tasks of the first generation.

(2) The subsequent %RESUME commands execute the program up to the decisive SVC with the number 44 (`FCT=POSFORK`). This SVC initiates the creation of the fork task.

(3) You now respond to the prompt of the parent task with a null input ([EM] [DÜ]). AID issues the message AID0348 and confirms that the fork task was created. You then respond to the next prompt of the parent task with [EM] [DÜ] as well (to force a change of task). This causes AID to issue the prompt of the fork task and wait for command input.

(4) To enable information from the `%SHOW %BASE` command to be output at the terminal, the prompt of the parent task must be answered again with a null input.

6.3.2 Associating I/O with tasks

As already indicated earlier, it is generally advantageous to debug only one task at a time and to let the other tasks created by the program sleep until that task completes. On some occasions, however, multiple tasks may execute concurrently and send output to the terminal. The following applies in such cases:

– Only the inputs can be uniquely assigned, i.e. each input is always assigned to the task identified by the prompt with which it was sent.

– In general, outputs cannot be associated with the originating task. The only exception is the logs of the %TRACE command, which can be allocated on the basis of the source references.
When multiple tasks attempt to send output to the terminal at the same time, the outputs generally appear in an essentially random order. In other words, if you are waiting for the output of a task, you should always send off the prompt of any other task with a null input until the expected output has been fully received (see the example above).

– During the debugging phase, it is advisable to redirect the outputs of programs to a file or to temporarily identify them with a preceding program code so that a unique association is possible.

6.3.3 Erratic behavior

A fork task cannot perform I/O at the terminal in the following cases:

– No program is loaded in the LOGON task.

– The program loaded in the LOGON task is not the same as the one from which the fork task was (directly or indirectly) created.

– The LOGON task was terminated.

This is also applicable to programs that were started in the POSIX shell.

In all the above cases, the fork task is aborted without a further error message on attempting to read from or write to the terminal. This also applies if the I/O request has already been entered in the queue.

The fork task is not aborted so long as the I/O is redirected to files.

6.4 Dump processing

Memory dumps of fork tasks and of programs that were loaded via an `exec()` call can be processed as usual. Dumps are always stored in BS2000 even if the program that generated the dump was started in the POSIX shell. If LSD information is to be loaded dynamically by AID for a memory dump of a POSIX program via the AID command %SYMLIB, you should bear in mind that %SYMLIB cannot access POSIX files.
The appropriate file must therefore be first copied to a PLAM library in BS2000 as a member of type L by using the POSIX command `bs2cp` and then assigned with %SYMLIB (see also section “Dynamic loading of the LSD” on page 15).
The %DUMPFILE command, which causes AID to open the dump file, and the %BASE command, which instructs AID that the memory dump in that dump file is to be then examined, are both explained in detail in the manual “Debugging of C/C++ Programs”.

In the POSIX shell, a user dump is always written for programs which are aborted due to an error. The query “IDA0N45 Dump desired?”, which you should know from BS2000, is suppressed in the POSIX shell, and the program is unloaded.
It is therefore advisable to trap program errors with %ON %ANY when debugging. AID will then report the address of the breakpoint at which an error occurred as well as the event that caused the error. The program remains loaded, so you can immediately examine the error context. If the error can be recovered with AID commands, you can then continue the program execution with %RESUME. Alternatively, if no continuation of the program is possible, you can exit the task with EXIT-JOB or LOGOFF in order to analyze the program error with other tests.

6.5 Application example

The programming example presented below illustrates a simple case in which `fork()` and `exec()` are used. The runtime log reflects an actual test run and thus serves as a concrete example for the debugging approach under POSIX.
The program `exfork` first generates a fork task. The parent task waits because of the `wait()` call until the child task has completed. The child task is overlaid by another program (`facul`) as a result of the `execvp()` call in line 21. The program name `facul` must be passed as a parameter to the `main` function of `exfork` on loading the program. `facul` calculates the factorial of an integer that is read and outputs the result on the screen.

For the sake of clarity, data and function names are represented in the explanatory text below with a `proportional` font. User inputs in runtime logs are shown in **boldprint.**

6.5.1 Source files

exfork.c

```
 1   #include <stdio.h>
 2   main (int argc, char* argv[])
 3   {
 4       char* my_name = argv[0];
 5       char* prog = argv[1];
 6       int pid;
 7       if (argc < 2)
 8       {
 9          fprintf (stderr,
10          "usage: %s subprogram [options]\n", my_name);
11          exit(1);
12       }
13       pid = fork();
14       if (pid < 0)
15       {
16          fprintf (stderr, "fork failed\n");
17          exit(1);
18       }
19       if (pid == 0)        /*   child   */
20       {
21          execvp (prog, &argv[1]);
22          fprintf (stderr, "execvp failed\n");
23       }
24       else /*   parent   */
25       {
26          wait ((int*) 0);
27          printf ("\n%s : %s has finished\n",
28                                  my_name, prog);
29          exit (0);
30       }
31   }
```

facul.c

```
 1   #include <stdio.h>
 2   int facul(int n)
 3   {
 4      return (n>1 ? n * facul(n-1) : 1);
 5   }
 6
 7   int main(void)
 8   {
 9      unsigned n;
10      printf("\nn? : ");
11      scanf("%d",&n);
12      if (n>16) return 0;
13      printf("%d! : %d\n",n,facul(n));
14      return 0;
15   }
```

6.5.2 Debugging run

Step 1

The programs `exfork.c` and `facul.c` are compiled with the C compiler in the POSIX shell. The `-g` option causes the compiler to generate LSD information at compilation. Since `facul` is to be loaded via `execvp()`, and since no LSD can be loaded by this method at the same time, the program is copied to BS2000 with `bs2cp`. This is required in order to enable the LSD to be assigned via %SYMLIB at a later stage in the debugging run.

```
$ c89 -g -o exfork exfork.c
%  NMH1102 MESSAGE OUTPUT FILE ':1OST:$TSOS.SYSMES.BINDER.013', ACCESS=
ISAM, ACTION=ADD
$ c89 -g -o facul facul.c
%  NMH1102 MESSAGE OUTPUT FILE ':1OST:$TSOS.SYSMES.BINDER.013', ACCESS=
ISAM, ACTION=ADD
$ bs2cp facul 'bs2:test.lib(facul,1)'
%  NMH1102 MESSAGE OUTPUT FILE ':1OST:$TSOS.SYSMES.LMS.031', ACCESS=ISAM,
ACTION=ADD
```

Step 2

The program is loaded with the POSIX command `debug` and placed in debugging mode. The name of the program that is to overlay the later fork task is passed as a parameter to `exfork`. The parent task is assigned the process ID 5421. To ensure that AID stops the fork task immediately after it is created, the appropriate switch is set with the %AID command. The parent task is also to be halted before the end of the program, so the command %INSERT S'28' is used for this purpose. It is advisable to set this breakpoint now, since the program will continue in the fork task after the `fork()`, while the parent task will be waiting for the fork task to terminate due to the `wait()` call. You could, of course, regain control over the parent task by entering a %STOP PID=*pid*. This would cause the parent task to immediately respond with its prompt as soon as the child task has completed.
The final %RESUME command starts the program. The parent task is now executed up to the `wait()` call, and the child task is halted immediately after it is created. AID issues a corresponding message and the process ID of the child task.

```
$ debug exfork facul
%  AID0348 Program stopped due to EXEC event (PID=0000005241)
%0000005241/%aid fork=next
%0000005241/%insert s'28'
%0000005241/%resume
%  AID0348 Program stopped due to FORK event (PID=0000005242)
```

Step 3

The currently applicable settings are output in the child task via %SHOW %AID. All operand values are reset; since %AID FORK=NEXT was set in the parent task, the FORK switch in the child task is set to NOT_USED. %AID EXEC=ON is entered to enable debugging of the `facul` program that is loaded with `execvp()`. %AID LOW=ALL is required to enable the use of lowercase letters in the S qualification. An attempt is then made to output the variable `pid`. However, `pid` cannot be addressed without a qualification in this case, since the breakpoint is located immediately after `fork()` in the runtime CRTE, as verified with DISPLAY %LOC(*address*). The following %TRACE command executes the program up to the next statement of the child task. `pid` can be addressed here without a qualification. The content of `pid` is 0 in the child task.
%RESUME starts the program again. The `execvp()` call is executed, and the `facul` program that is loaded with it is halted and placed in debugging mode.

```
%0000005242/%show %aid
A I D     V03.4B OF  2015-01-30
Copyright (C) 2015 Fujitsu Technology Solutions
All Rights Reserved

E=VM : %AINT = %MODE31
%AID CHECK      =   NO
%AID REP        =   NO
%AID SYMCHARS   =   STD
%AID OV         =   NO
%AID LOW        =   OFF
%AID DELIM      =  '|'
%AID LANG       =   D
%AID FORK       =   NOT_USED
%AID EXEC       =   OFF
%AID C          =   NO
%AID EBCDIC     =   EDF03IRV
%0000005242/%aid exec=on
%0000005242/%aid low=all
%0000005242/%display pid
*** TID: 006F01EE *** TSN: 1A4I ********************************************
ABSOLUT: V'00EB0D1A' SOURCE: ICXSVCTU  PROC: ICXSVCTU  ********************
%  AID0378 Symbolic information missing
%0000005242/%display %loc(%pc->)
CURRENT PC: 00EB0D1A    CSECT: ICXSVCTU  ***********************************
V'00EB0D1A' = CONTEXT:$CRTEC@@@02@0@@@
              LMOD :  IC@RTSXS
              OMOD :  IC@RTSXS
              CSECT : ICXSVCTU (00EB0A00) + 0000031A
%0000005242/%trace 1 in s=n'exfork.c'
14                  ┐┐
STOPPED AT SRC_REF: 14, SOURCE: exfork.c , PROC: main , END OF TRACE
%0000005242/%display pid
SRC_REF: 14 SOURCE: exfork.c  PROC: main ***********************************
pid             =           0
%0000005242/%resume
%  AID0348 Program stopped due to EXEC event (PID=0000005242)
```

Step 4

Since the `facul` program was loaded with the `execvp()` call, the LSD must be explicitly assigned via %SYMLIB. %TRACE 1 is then used to proceed to the start of the program `facul`. The subcommand %INSERT S'4' is intended to output the current value *n* in each case and the call hierarchy of the associated recursion level. The program is then started with %RESUME. `facul` runs up to the end of the program; the `printf()` call in line 13 of `facul` outputs the correct result: the value of 4! is 24.
The breakpoint that was set at S'28' in the parent task at the start of this example now becomes active. In order to enable the output of the `pid` variable of the parent task, case sensitivity must now also be turned on for the parent task. `pid` contains the process ID of the child task, i.e. 5242, in the parent task. The closing %RESUME command causes the last two statements of the parent task to be executed.

```
%0000005242/%symlib test.lib
%0000005242/%trace 1 in s=n'facul.c'
10                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 10, SOURCE: facul.c , PROC: main , END OF TRACE
%0000005242/%insert s'4' <%display n;%sdump %nest>
%0000005242/%resume
n? : 4
*** TID: 006F01EE *** TSN: 1A4I ****************************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
n              =        4
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: V'E60432'    SOURCE: ICS$MAI@  PROC: _main  ************************
ABSOLUT: V'1002234'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ***********************

n              =        3
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: V'E60432'    SOURCE: ICS$MAI@  PROC: _main  ************************
ABSOLUT: V'1002234'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ***********************

n              =        2
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: V'E60432'    SOURCE: ICS$MAI@  PROC: _main  ************************
ABSOLUT: V'1002234'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ***********************
n              =        1
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: V'E60432'    SOURCE: ICS$MAI@  PROC: _main  ************************
ABSOLUT: V'1002234'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ***********************

4! : 24

STOPPED AT SRC_REF: 28, SOURCE: exfork.c , BLK: 25
%0000005241/%aid low
%0000005241/%display pid
*** TID: 00EB169E *** TSN: 1A4G ****************************************
SRC_REF: 28 SOURCE: exfork.c  BLK : 25  ************************************
pid            =        5242
%0000005241/%resume
exfork: facul has finished
```

Glossary

child task

Task created by a `fork()` call.

command mode

More precisely: BS2000 command mode.
In the AID manuals this refers to expert mode of the SDF command language. In expert mode, the system prompts you with a slash (/) to enter commands.

debugging mode

This denotes the status of a task in which you can enter AID commands for debugging. In the LOGON task, debugging mode is identical with BS2000 command mode. In the case of fork tasks, AID handles the dialog between user and task. AID issues a prompt character formed from the process identification of the fork task; this prompt requests you to enter commands.
Debugging mode has a lower priority than the command mode of the LOGON task. In other words, the fork task is not authorized to use the terminal on the same level as the LOGON task.

DMS file

File of the BS2000 Data Management System.
This may be either individual files or modules which are stored in PLAM libraries. Files can be copied between POSIX and BS2000 with the aid of the POSIX command `bs2cp` (see *UFS file*).

exec()

This identifies a function group which includes the following functions: `execl()`, `execv()`, `execle()`, `execve()`, `execlp()`, `execvp()`.
An `exec()` call causes the program specified in the call to be loaded on top of the calling program.

fork()

System call that creates a copy of the process containing the `fork()` call. After completion of `fork()` the system contains an additional identical process.

fork task

Task created by a `fork()` call.

LOGON task

Task started by means of the SDF command SET-LOGON-PARAMETERS. The command mode of the LOGON task has a higher priority than the debugging mode of a task created by `fork()`, and this may lead to problems in conjunction with the simultaneous debugging of the parent and child tasks.

parent task

First task in the hierarchy of a task family.

POSIX shell

A ported UNIX system program that handles communication between the user and the system. The POSIX shell is a command interpreter; it compiles the entered POSIX commands into a language which the system can work with.

process

Term from the UNIX world which is also used under POSIX. A process corresponds to a task on the BS2000 level. It is also used to refer to the address space and the program executed therein, as well as to the required system resources.
A process is created by another process by calling the `fork()` function. The function that calls `fork()` is known as the parent process (or parent task in BS2000); the new process created by `fork()` is known as the child process (or child task in BS2000).

process identification (pid)

A number assigned by the system in order to identify the system unambiguously. On the basis of the process identification (pid), AID constructs the prompt output by a fork task to request input.

task family

All tasks produced by a single task, regardless of the generations, by means of `fork()`.

UFS file

File of the UNIX file system.
In the same way as with UNIX, files in POSIX are stored in hierarchically organized directories. The C/C++ compiler can process both UFS files and DMS files (q.v.). The %AID command %SYMLIB, on the other hand, can only be used in conjunction with PLAM libraries in BS2000.

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

AID (BS2000)
Advanced Interactive Debugger
Core Manual
User Guide

AID (BS2000)
Advanced Interactive Debugger
Debugging of C/C++ Programs
User Guide

AID (BS2000)
Advanced Interactive Debugger
Debugging of COBOL Programs
User Guide

AID (BS2000)
Advanced Interactive Debugger
Debugging of FORTRAN Programs
User Guide

AID (BS2000)
Advanced Interactive Debugger
Debugging of ASSEMBH Programs
User Guide

AID (BS2000)
Debugging on Machine Code Level
User Guide

AID (BS2000)
Advanced Interactive Debugger
Ready Reference
User Guide

C/C++

C (BS2000)
C Compiler
User Guide

C++ (BS2000)
C++ Compiler
User Guide

POSIX

POSIX (BS2000)
POSIX Basics for Users and System Administrators
User Guide

POSIX (BS2000)
Commands
User Guide

C/C++ (BS2000)
POSIX Commands of the C and C++ Compiler
User Guide

BS2000

CRTE (BS2000)
Common RunTime Environment
User Guide

BS2000 OSD/BC
Commands
User Guide

Index