Abstract

This document describes an architecture to support the debugging of parallel and distributed (PVM) applications. It also presents the set of system calls supported by the debugging engine.

1 System architecture

The figure presents the architecture and interaction(s) between the GRAPNEL Execution Environment and the debugging engine.

The client process uses the debugging library$^1$ to make requests to the main daemon$^2$.

1.1 System initialization

To initialize communications (and the debugging engine), the client application(s) will make a system call to initialize the library (see \texttt{dbg\_init()}). If there is no main daemon running at the time the initialization is requested, it will be started automatically.

1.2 Communication between clients and the debugging engine

All the services provided by the debugging engine are classified as belonging to one of to classes:

\footnotesize
$^1$\texttt{\$DDBG\_ROOT/lib/\$DDBG\_ARCH/libdbg.a}

\footnotesize
$^2$\texttt{\$DDBG\_ROOT/bin/\$DDBG\_ARCH/ddbgd}
• **Immediate answer services.** The services belonging to this class will either fail or be executed immediately and return the relevant data as return parameter(s) to the system call function.

• **Delayed answer services.** The services belonging to this class—e.g. `dbg.next()`—will either fail or be executed immediately but, because they can take a (very) long time until its execution is finished, the relevant returning data will be sent later to the client.

1.3 Getting the delayed answers

- `int dbg_get_special_info(char *procid, struct code_info *info)`
  If there is returning data available (from a delayed answer command), the function will return `DBG_OK` with the GRAPNEL Process ID in `procid` and the data in `info`. Otherwise it will return `DBG_NO_DATA`.

1.4 Conversion between PVM tid’s and GRAPNEL procid’s

PVM uses `task ID`’s (integers) to identify the processes, GRAPNEL uses `Process ID`’s (strings) to do the same. In order to support the mapping between the GRAPNEL symbolic name and the PVM naming scheme, the following function is provided.

- `int dbg_set_tid_proc(int tid, char *procID)`
  This function should be called once for each process, specifying its PVM task ID and the GRAPNEL symbolic name.

2 Data types used

The library uses 9 different structured types:

- `struct code_info` To reference a location in the program.
- `struct news_info` To indicate that there is something “new” in the system.
- `struct news_new_proc` The description of a new process in the system (to be used inside `struct news_info`).
- `struct brkpt_info` A reference to a breakpoint in the system.
- `struct stack_info` Data about stack frames.
- `struct var_info` A description of a variable.
- `struct locals_info` Data about (all) the local variables.
- `struct args_info` Data about the program arguments.
- `struct proc_info` Data relevant to a running process.

All these structures will be described in the following sections when they are used the first time, except `struct code_info` that will be presented in the next section.

2.1 Code information structure

The references to the program code can be described by the following data type:

```c
struct code_info {
    int line_no;
    char function_name[];
    char source_file[];
};
```

When used as a normal parameter, the following remarks apply:

- The field `line_no` is considered empty if it contains a value ≤ 0;
- The fields `function_name` and `source_file` are considered empty if they contain an empty (zero length) string;
- Only one of `line_no` or `function_name` may be set (the other one must be empty);
- If field `source_file` is empty, the data refers to the current source file.

When used as a return parameter, the following remarks apply:

- In case of success the fields `line_no`, `function_name` and `source_file` contain the line number, the function name and the source file name (e.g. of a breakpoint);
- If an error occurred, only the fields that are not empty have relevant data.

---

3 The client should ask explicitly for the delayed (see function `dbg_get_special_info()`).
3 Library functions

For all the library functions presented in the next sections, when it is said that the function will “Return the operation status”, this means that the return value is:

DBG_OK If the call had success.
DBG_UNKNOWN When is was not possible to parse the information returned by the underlying debugger (the GNU gdb debugger).

3.1 Starting/Ending the debugger

- int dbg_init(void)
  Initialize the communications and the main deamon if needed. Return the operation status.

- int dbg_end(void)
  Cleanup the communications before finish. If it's the last call, will kill the main deamon as well. Return the operation status.

- int dbg_attach(char *procid, struct code_info *info)
  Start a new debugger and attach it to the process specified in procid. The process will be stopped and will become under control of the debugger. Return in info the relevant data to determine where the processes were stopped. Return the operation status.

  The function can also return one of the possible error conditions presented below:
  DBG_INEXISTENT_PROCESS
  DBG_ALREADY_ON_DEBUG
  DBG_ATTACHER_ERROR

- int dbg_detach(char *procid)
  Kill the debugger associated with process procid and leave the process running free. Return the operation status.

  The function can also return one of the possible error conditions presented below:
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

- int dbg_kill(char *procid)
  Kill process procid and its debugger. Return the operation status.

  The function can also return one of the possible error conditions presented below:
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

- int dbg_get_news(struct news_info *info)
  Return in info data about news in the system. Currently, only information on new processes on the system is available. If it is possible to identify the process with the GRAPNEL symbolic name, the type will be DBG_GRAPNEL_PROC and the news field will contain the GRAPNEL process ID. Otherwise the field type will have the value DBG_PVM_PROC and the news field will contain the PVM task ID (tid) of the process. Return 1 if there are news, 0 otherwise.

  struct news_new_grapnel_proc {
    char *procid;
  };

  struct news_new_pvm_proc {
    int tid;
  };

  struct news_info {
    int type;
    union {
      struct news_new_grapnel_proc proc;
      struct news_new_pvm_proc proc;
    } news;
  };

3.2 Managing breakpoints

- **int dbg_set_break(char *procid, struct code_info *info)**
  Set a breakpoint on process `procid` in the line/function specified in `info`. Return the unique breakpoint id.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_ON_DEBUG**
  **DBG_NOT_READY**

- **int dbg_set_cond_break(char *procid, struct code_info *info, char *exp)**
  If the expression `exp` evaluates to `TRUE`, set a breakpoint in the process specified in `procid` in the line/function specified in `info` (conditional breakpoint). The expression is evaluated every time the breakpoint is reached. Return the unique breakpoint id.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_ON_DEBUG**
  **DBG_NOT_READY**

- **int dbg_set_temp_break(char *procid, struct code_info *info)**
  Set a temporary breakpoint (one time only) on process `procid` in the line/function specified in `info`. Return the unique breakpoint id.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_ON_DEBUG**
  **DBG_NOT_READY**

- **int dbg_set_watch(char *procid, char *exp)**
  Set a watchpoint on process `procid`. The process will stop when the condition in `exp` will become `TRUE`. Return the unique watchpoint id.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_ON_DEBUG**
  **DBG_NOT_READY**

- **int dbg_disable_break(int brkptid)**
  Disable the breakpoint specified in `brkptid`. This breakpoint will become inactive until enabled again. Return the operation status.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_READY**
  **DBG_INEXISTENT_BREAK**

- **int dbg_enable_break(int brkptid)**
  Enable the previously disabled breakpoint specified in `brkptid`. This breakpoint will become active again. Return the operation status.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_READY**
  **DBG_INEXISTENT_BREAK**

- **int dbg_ignore_break(int brkptid, int counter)**
  Disable the breakpoint specified in `brkptid` for `count` times. Return the operation status.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_READY**
  **DBG_INEXISTENT_BREAK**

- **int dbg_clear_break(int brkptid)**
  Remove the breakpoint specified in `brkptid`. Return the operation status.
  
  The function can also return one of the possible error conditions presented below:
  
  **DBG_NOT_READY**
  **DBG_INEXISTENT_BREAK**
3.3 Managing the program stack

- **int dbg_select_frame(char *procid, int count)**
  Select a new current frame for process procid. If count > 0, select a frame count frames up referring to the current frame. If count < 0, select count frames down referring to the current frame. If count = 0, does nothing. Return the distance between the previous and the new current frame.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

3.4 Controlling the execution of the (debugged) processes

- **int dbg_run(char *procid)**
  Run a (previously spawned) program from the beginning, until a breakpoint is found or the expression of a watchpoint is true (or until the end, if none above conditions are true). Return the operation status.

  When the execution of this command is finished, the contents of the corresponding struct code_info will be written to the communication socket.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

- **int dbg_continue(char *procid)**
  Continue the execution of a process from the point where it was stopped, until a breakpoint is found or the expression of a watchpoint is true (or until the end, if none above conditions are true). Return the operation status.

  When the execution of this command is finished, the contents of the corresponding struct code_info will be written to the communication socket.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

- **int dbg_next(char *procid)**
  Execute the code until the next instruction. Subroutines are executed as one instruction only. Return the operation status.

  When the execution of this command is finished, the contents of the corresponding struct code_info will be written to the communication socket.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

- **int dbg_step(char *procid)**
  Execute the code until the next instruction. Subroutines are executed as normal code. Return the operation status.

  When the execution of this command is finished, the contents of the corresponding struct code_info will be written to the communication socket.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

- **int dbg_finish(char *procid)**
  Run until the selected stack frame returns. Return the operation status.

  When the execution of this command is finished, the contents of the corresponding struct code_info will be written to the communication socket.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

\(^4\)See `dbg_select_frame()`. 
• int dbg_return(char *procid, char *exp, struct code_info *info)
  Pop selected stack frame without executing. Set the return value to exp (a NULL pointer means no return value). Return in info the relevant data to determine where the processes stopped. Return the operation status.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

• int dbg_interrupt(char *procid, struct code_info *info)
  Interrupts the execution of the process procid. Return in info the relevant data to determine where the processes were stopped. Return 1 if the interrupt was sent to the process, 0 otherwise.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG

3.5 Information

• int dbg_print(char *procid, char *fmt, char *exp, char *value)
  Return in value the result of the evaluation of expression exp in the context of process procid, according to format fmt (see table ?? for the allowed formats).

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

<table>
<thead>
<tr>
<th>x</th>
<th>hexadecimal</th>
<th>t</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>signed decimal</td>
<td>a</td>
<td>address</td>
</tr>
<tr>
<td>u</td>
<td>unsigned decimal</td>
<td>c</td>
<td>character</td>
</tr>
<tr>
<td>o</td>
<td>octal</td>
<td>f</td>
<td>floating point</td>
</tr>
</tbody>
</table>

Table 1: Formats allowed in expression evaluation

• int dbg_get_var(char *procid, struct var_info *var)
  Return in value the value (as a string) of the variable name of process procid. Return the operation status.

  struct var_info {
    char name[];
    char value[];
  };

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

  Any (simple type) variable with a textual representation of more than 256 characters will be truncated and three dots (…) will be added at the end.

• int dbg_set_var(char *procid, struct var_info *var)
  Set the value of the variable name of process procid with the value specified in value. Return the operation status.

  The function can also return one of the possible error conditions presented below:
  
  DBG_NOT_ON_DEBUG
  DBG_NOT_READY

• int dbg_info_break(int brkptid, struct brkpt_info *info)
  Return in info information about the breakpoint specified, e.g., number, type (simple or conditional), line number or expression associated. Return the operation status.
struct brkpt_info {
    int type;
    char process[];
    struct code_info where;
    char condition[];
    int state;
    int hits;
};

The function can also return one of the possible error conditions presented below:

**DBG_NOT_ON_DEBUG**

**DBG_NOT_READY**

The following text could be printed based on a set of return values of this function:

<table>
<thead>
<tr>
<th>Num</th>
<th>Type</th>
<th>Process</th>
<th>Filename</th>
<th>Where</th>
<th>Condition</th>
<th>State</th>
<th>Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Breakpoint</td>
<td>0x0a5f8abc</td>
<td>main.c</td>
<td>f:main()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TempBrkpt</td>
<td>0xa8762345</td>
<td>main.c</td>
<td>f:event()</td>
<td>disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CondBrkpt</td>
<td>0x00546543</td>
<td>util.c</td>
<td>l:123</td>
<td>n==5</td>
<td>ignored</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Watchpoint</td>
<td>0x4a7f6bc3</td>
<td>main.c</td>
<td>event==32</td>
<td>enabled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **int dbg_info_stack(char *procid, int count, struct stack_info *info)**

  Return in `info` a trace of the last `count` stack frames of process `procid`. Return the number of frames found.

  ```
  struct stack_info {
    char source_file[];
    char function_name[];
    int nvars;
    struct var_info vars[];
  };
  ```

  The function can also return one of the possible error conditions presented below:

  **DBG_NOT_ON_DEBUG**

  **DBG_NOT_READY**

  If the function `main()` calls a function `f1()` with one integer as argument, and this function calls the function `f2()` with one integer and one float as arguments, the following text could be printed based on the data returned by this function:

  ```
  Level | File | Function | Args |
  -----|------|----------|------|
  0    | a.c  | main     | (argc=2, argv=0x7b033320) |
  1    | a.c  | f1       | (myarg=3) |
  2    | b.c  | f2       | (thearg=1, theotherarg=3.1415) |
  ```

- **int dbg_info_locals(char *procid, struct locals_info *info)**

  Return in `info` information about local variables on the selected frame of process `procid`. Return the operation status.

  ```
  struct var_info {
    char name[];
    char value[];
  };

  struct locals_info {
    int nvars;
    struct var_info vars[];
  };
  ```

  The function can also return one of the possible error conditions presented below:

  **DBG_NOT_ON_DEBUG**

  **DBG_NOT_READY**
If the function `main()` calls a function `f()` with one integer and one float as arguments, the following text could be printed based on the data returned by this function:

\[
\begin{align*}
\text{pi} & = 3.1416 \\
\text{name} & = \text{(char *) } 0x7b03333a
\end{align*}
\]

- **int dbg_info_args(char *procid, struct args_info *info)**
  
  Return in `info` information about the (function) arguments on the selected frame of process `procid`. Return the operation status.

  ```c
  struct var_info {
    char name[];
    char value[];
  };

  struct args_info {
    int nargs;
    struct var_info args[];
  };
  ```

  The function can also return one of the possible error conditions presented below:

  - **DBG_NOT_ON_DEBUG**
  - **DBG_NOT_READY**

  If the function `main()` receive one argument, the following text could be printed based on the data returned by this function:

  \[
  \begin{align*}
  \text{argc} & = 2 \\
  \text{argv} & = \text{(char **) } 0x7b033364
  \end{align*}
  \]

- **int dbg_info_process(char *procid, struct proc_info *info)**

  Return in `info` data about the status of process `procid`. Return the operation status.

  ```c
  struct proc_info {
    int status;
    int bwpid;
    struct code_info cinfo;
    char symbols[];
  };
  ```

  The function can also return one of the possible error conditions presented below:

  - **DBG_NOT_ON_DEBUG**
  - **DBG_NOT_READY**

  If process `0x45677654` is running, and process `0x56784321` is stopped in a breakpoint, then the following information could be reported:

  ```plaintext
  Symbols from "/home/guest/myprogram"
<table>
<thead>
<tr>
<th>TaskId</th>
<th>Status</th>
<th>File</th>
<th>Line</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x45677654</td>
<td>running</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x6798ab3c</td>
<td>ready</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x56794321</td>
<td>stopped</td>
<td>main.c</td>
<td>123</td>
<td>fibb</td>
</tr>
</tbody>
</table>
  ```