Libgdb

Version 0.3
Oct 1993

Thomas Lord

1 Overview

Function and Purpose

Libgdb is a package which provides an API to the functionality of GDB, the GNU symbolic debugger. It is specifically intended to support the development of a symbolic debugger with a graphic interface.

This Document

This document is a specification of the libgdb API. It is written in the form of a programmer's manual. So the goal of this document is to explain what functions make up the API, and how they can be used in a running application.

Terminology

In this document, *libgdb* refers to a library containing the functions defined herein, *application* refers to any program built with that library.

Dependencies

Programs which are linked with libgdb must be linked with libbfd, libopcodes, libiberty, and libmmalloc.

Acknowledgments

Essential contributions to this design were made by Stu Grossman, Jim Kingdon, and Rich Pixley.

2 Libgdb is an Interpreter Based Server

To understand libgdb, it is necessary to understand how the library is structured. Historically, GDB is written as a small interpreter for a simple command language. The commands of the language perform useful debugging functions.

Libgdb is built from GDB by turning the interpreter into a debugging server. The server reads debugging commands from any source and interprets them, directing the output arbitrarily.

In addition to changing GDB from a tty-based program to a server, a number of new GDB commands have been added to make the server more useful for a program with a graphic interface.

Finally, libgdb includes provisions for asynchronous processing within the application.

Most operations that can be carried out with libgdb involve the GDB command interpreter. The usual mode of operation is that the operation is expressed as a string of GDB commands, which the interpreter is then invoked to carry out. The output from commands executed in this manner can be redirected in a variety of useful ways for further processing by the application.

The command interpreter provides an extensive system of hooks so an application can monitor any aspect of the debugging library's state. An application can set its own breakpoints and attach commands and conditions to those. It is possible to attach hooks to any debugger command; the hooks are invoked whenever that command is about to be invoked. By means of these, the displays of a graphical interface can be kept fully up to date at all times.

We show you how to define new primitives in the command language. By defining new primitives and using them in breakpoint scripts and command hooks, an application can schedule the execution of arbitrary C-code at almost any point of interest in the operation of libgdb.

We show you how to define new GDB convenience variables for which your code computes a value on demand. Referring to such variables in a breakpoint condition is a convenient way to conditionalize breakpoints in novel ways.

To summarize: in libgdb, the gdb command language is turned into a debugging server. The server takes commands as input, and the server's output is redirectable. An application uses libgdb by formatting debugging commands and invoking the interpreter. The application might maintain breakpoints, watchpoints and many kinds of hooks. An application can define new primitives for the interpreter.

3 You Provide the Top Level for the Libgdb Command Interpreter

When you use libgdb, your code is providing a *top level* for the command language interpreter. The top level is significant because it provides commands for the the interpreter to execute. In addition, the top level is responsible for handling some kinds of errors, and performing certain cleanup operations on behalf of the interpreter.

Initialization

Before calling any other libgdb functions, call this:

`void` **gdb_init** (`void`) Function

Perform one-time initialization for libgdb.

An application may wish to evaluate specific gdb commands as part of its own initialization. The details of how this can be accomplished are explained below.

The Top-Level Loop

There is a strong presumption in libgdb that the application has the form of a loop. Here is what such a loop might look like:

```
while (gdb_still_going ())
  {
    if (!GDB_TOP_LEVEL ())
      {
        char * command;
        gdb_start_top_loop ();
        command = process_events ();
        gdb_execute_command (command);
        gdb_finish_top_loop ();
      }
    }
```

The function `gdb_still_going` returns 1 until the gdb command 'quit' is run.

The macro `GDB_TOP_LEVEL` invokes setjmp to set the top level error handler. When a command results in an error, the interpreter exits with a longjmp. There is nothing special libgdb requires of the top level error handler other than it be present and that it restart the top level loop. Errors are explained in detail in a later chapter.

Each time through the top level loop two important things happen: a debugger command is constructed on the basis of user input, and the interpreter is invoked to execute that command. In the sample code, the call to the imaginary function `process_events` represents the point at which a graphical interface should read input events until ready to execute a debugger command. The call to `gdb_execute_command` invokes the command interpreter (what happens to the output from the command will be explained later).

Libgdb manages some resources using the top-level loop. The primary reason for this is error-handling: even if a command terminates with an error, it may already have allocated resources which need to be freed. The freeing of such resources takes place at the top-level, regardless of how the the command exits. The calls to `gdb_start_top_loop` and `gdb_finish_top_loop` let libgdb know when it is safe to perform operations associated with these resources.

Breakpoint Commands

Breakpoint commands are scripts of GDB operations associated with particular breakpoints. When a breakpoint is reached, its associated commands are executed.

Breakpoint commands are invoked by the libgdb function `gdb_finish_top_loop`.

Notice that if control returns to the top-level error handler, the execution of breakpoint commands is bypassed. This can happen as a result of errors during either `gdb_execute_command` or `gdb_finish_top_loop`.

Application Initialization

Sometimes it is inconvenient to execute commands via a command loop for example, the commands an application uses to initialize itself. An alternative to `execute_command` is `execute_catching_errors`. When `execute_catching_errors` is used, no top level error handler need be in effect, and it is not necessary to call `gdb_start_top_loop` or `gdb_finish_top_loop`.

Cleanup

The debugger command "quit" performs all necessary cleanup for libgdb. After it has done so, it changes the return value of `gdb_still_going` to 0 and returns to the top level error handler.

4 How the Server's I/O Can be Used

In the last chapter it was pointed out that a libgdb application is responsible for providing commands for the interpreter to execute. However some commands require further input (for example, the "quit" command might ask for confirmation). Almost all commands produce output of some kind. The purpose of this section is to explain how libgdb performs its I/O, and how an application can take advantage of this.

I/O Vectors

Libgdb has no fixed strategy for I/O. Instead, all operations are performed by functions called via structures of function pointers. Applications supply theses structures and can change them at any time.

struct gdb_input_vector Type
struct gdb_output_vector Type

These structures contain a set of function pointers. Each function determines how a particular type of i/o is performed. The details of these strucutres are explained below.

The application allocates these structures, initializes them to all bits zero, fills in the function pointers, and then registers names for them them with libgdb.

`void` **gdb_name_input_vector** (*name*, *vec*) Function
`void` **gdb_remove_input_vector** (*name*, *vec*) Function
`void` **gdb_name_output_vector** (*name*, *vec*) Function
`void` **gdb_remove_input_vector** (*name*, *vec*) Function

```
char * name;
struct gdb_output_vector * vec;
```

These functions are used to give and remove names to i/o vectors. Note that if a name is used twice, the most recent definition applies.

Output

An output vector is a structure with at least these fields:

```
struct gdb_output_vector
{
  /* output */
  void (*put_string) (struct gdb_output_vector *, char * str);
}
```

Use the function `memset` or something equivalent to initialize an output vector to all bits zero. Then fill in the function pointer with your function.

A debugger command can produce three kinds of output: error messages (such as when trying to delete a non-existent breakpoint), informational messages (such as the notification printed when a breakpoint is hit), and the output specifically requested by a command (for example, the value printed by the "print" command). At any given time, then, libgdb has three output vectors. These are called the *error*, *info*, *value* vector respectively.

Input

```
struct gdb_input_vector
{
  int (*query) (struct gdb_input_vector *,
                char * prompt,
                int quit_allowed);
  int * (*selection) (struct gdb_input_vector *,
                      char * prompt,
                      char ** choices);
  char * (*read_string) (struct gdb_input_vector *,
                         char * prompt);
  char ** (*read_strings) (struct gdb_input_vector *,
                           char * prompt);
}
```

Use the function `memset` or something equivalent to initialize an input vector to all bits zero. Then fill in the function pointers with your functions.

There are four kinds of input requests explicitly made by libgdb.

A *query* is a yes or no question. The user can respond to a query with an affirmative or negative answer, or by telling gdb to abort the command (in some cases an abort is not permitted). Query should return 'y' or 'n' or 0 to abort.

A *selection* is a list of options from which the user selects a subset. Selections should return a NULL terminated array of integers, which are indexes into the array of choices. It can return NULL instead to abort the command. The array returned by this function will be passed to `free` by libgdb.

A *read_string* asks the user to supply an arbitrary string. It may return NULL to abort the command. The string returned by `read_string` should be allocated by `malloc`; it will be freed by libgdb.

A *read_strings* asks the user to supply multiple lines of input (for example, the body of a command created using 'define'). It, too, may return NULL to abort. The array and the strings returned by this function will be freed by libgdb.

I/O Redirection from the Application Top-Level

`struct` **gdb_io_vecs** `gdb_set_io` (`struct gdb_io_vecs *`) Function

```
struct gdb_io_vecs
{
  struct gdb_input_vector * input;
```

```
        struct gdb_output_vector * error;
        struct gdb_output_vector * info;
        struct gdb_output_vector * value;
      }
```

This establishes a new set of i/o vectors, and returns the old setting. Any of the pointers in this structure may be NULL, indicating that the current value should be used.

This function is useful for setting up i/o vectors before any libgdb commands have been invoked (hence before any input or output has taken place).

It is explained in a later chapter how to redirect output temporarily. (See Chapter 5 [Invoking], page 7.)

I/O Redirection in Debugger Commands

A libgdb application creates input and output vectors and assigns them names. Which input and output vectors are used by libgdb is established by executing these debugger commands:

set input-vector *name* Function
set error-output-vector *name* Function
set info-output-vector *name* Function
set value-output-vector *name* Function

Choose an I/O vector by name.

A few debugger commands are for use only within commands defined using the debugger command 'define' (they have no effect at other times). These commands exist so that an application can maintain hooks which redirect output without affecting the global I/O vectors.

with-input-vector *name* Function
with-error-output-vector *name* Function
with-info-output-vector *name* Function
with-value-output-vector *name* Function

Set an I/O vector, but only temporarily. The setting has effect only within the command definition in which it occurs.

Initial Conditions

When libgdb is initialized, a set of default I/O vectors is put in place. The default vectors are called `default-input-vector`, `default-output-vector`, &c.

The default query function always returns 'y'. Other input functions always abort. The default output functions discard output silently.

5 Invoking the Interpreter, Executing Commands

This section introduces the libgdb functions which invoke the command interpreter.

`void` **gdb_execute_command** (*command*) Function

```
char * command;
```

Interpret the argument debugger command. An error handler must be set when this function is called. (See Chapter 3 [Top Level], page 2.)

It is possible to override the current I/O vectors for the duration of a single command:

`void` **gdb_execute_with_io** (*command*, *vecs*) Function

```
char * command;
struct gdb_io_vecs * vecs;

struct gdb_io_vecs
{
  struct gdb_input_vector * input;
  struct gdb_output_vector * error;
  struct gdb_output_vector * info;
  struct gdb_output_vector * value;
}
```

Execute *command*, temporarily using the i/o vectors in *vecs*.

Any of the vectors may be NULL, indicating that the current value should be used. An error handler must be in place when this function is used.

`struct gdb_str_output` **gdb_execute_for_strings** (*cmd*) Function

```
char * cmd;
```

`struct gdb_str_output` **gdb_execute_for_strings2** (*cmd*, *input*) Function

```
char * cmd;
struct gdb_input_vector * input;
```

```
struct gdb_str_output
{
  char * error;
  char * info;
  char * value;
};
```

Execute *cmd*, collecting its output as strings. If no error occurs, all three strings will be present in the structure, the empty-string rather than NULL standing for no output of a particular kind.

If the command aborts with an error, then the `value` field will be NULL, though the other two strings will be present.

In all cases, the strings returned are allocated by malloc and should be freed by the caller.

The first form listed uses the current input vector, but overrides the current output vector. The second form additionally allows the input vector to be overridden.

This function does not require that an error handler be installed.

`void` **execute_catching_errors** (*command*) Function

`char *` *command*;

Like `execute_command` except that no error handler is required.

`void` **execute_with_text** (*command*, *text*) Function

`char *` *command*;
`char **` *text*;

Like `execute_catching_errors`, except that the input vector is overridden. The new input vector handles only calls to `query` (by returning 'y') and calls to `read_strings` by returning a copy of *text* and the strings it points to.

This form of execute_command is useful for commands like `define`, `document`, and `commands`.

6 How New Commands are Created

Applications are, of course, free to take advantage of the existing GDB macro definition capability (the `define` and `document` functions).

In addition, an application can add new primitives to the GDB command language.

`void` **gdb_define_app_command** (*name*, *fn*, *doc*) Function

`char *` *name*;
`gdb_cmd_fn` *fn*;
`char *` *doc*;

```
typedef void (*gdb_cmd_fn) (char * args);
```

Create a new command call *name*. The new command is in the `application` help class. When invoked, the command-line arguments to the command are passed as a single string.

Calling this function twice with the same name replaces an earlier definition, but application commands can not replace builtin commands of the same name.

The documentation string of the command is set to a copy the string *doc*.

7 How Builtin Variables are Defined

Convenience variables provide a way for values maintained by libgdb to be referenced in expressions (e.g. `$bpnum`). Libgdb includes a means by which the application can define new, integer valued convenience variables:

void **gdb_define_int_var** (*name*, *fn*, *fn_arg*) Function

```
char * name;
int (*fn) (void *);
void * fn_arg;
```

This function defines (or undefines) a convenience variable called *name*. If *fn* is NULL, the variable becomes undefined. Otherwise, *fn* is a function which, when passed *fn_arg* returns the value of the newly defined variable.

No libgdb functions should be called by *fn*.

One use for this function is to create breakpoint conditions computed in novel ways. This is done by defining a convenience variable and referring to that variable in a breakpoint condition expression.

8 Scheduling Asynchronous Computations

A running libgdb function can take a long time. Libgdb includes a hook so that an application can run intermittently during long debugger operations.

void **gdb_set_poll_fn** (*fn*, *fn_arg*) Function

```
void (*fn)(void * fn_arg, int (*gdb_poll)());
void * fn_arg;
```

Arrange to call *fn* periodically during lengthy debugger operations. If *fn* is NULL, polling is turned off. *fn* should take two arguments: an opaque pointer passed as *fn_arg* to `gdb_set_poll_fn`, and a function pointer. The function pointer passed to *fn* is provided by libgdb and points to a function that returns 0 when the poll function should return. That is, when `(*gdb_poll)()` returns 0, libgdb is ready to continue *fn* should return quickly.

It is possible that `(*gdb_poll)()` will return 0 the first time it is called, so it is reasonable for an application to do minimal processing before checking whether to return.

No libgdb functions should be called from an application's poll function, with one exception: `gdb_request_quit`.

void **gdb_request_quit** (void) Function

This function, if called from a poll function, requests that the currently executing libgdb command be interrupted as soon as possible, and that control be returned to the top-level via an error.

The quit is not immediate. It will not occur until at least after the application's poll function returns.

9 Debugger Commands for Libgdb Applications

The debugger commands available to libgdb applications are the same commands available interactively via GDB. This section is an overview of the commands newly created as part of libgdb.

This section is not by any means a complete reference to the GDB command language. See the GDB manual for such a reference.

9.1 Setting Hooks to Execute With Debugger Commands.

Debugger commands support hooks. A command hook is executed just before the interpreter invokes the hooked command.

There are two hooks allowed for every command. By convention, one hook is for use by users, the other is for use by the application.

A user hook is created for a command XYZZY by using `define-command` to create a command called `hook-XYZZY`.

An application hook is created for a command XYZZY by using `define-command` to create a command called `apphook-XYZZY`.

Application hooks are useful for interfaces which wish to continuously monitor certain aspects of debugger state. The application can set a hook on all commands that might modify the watched state. When the hook is executed, it can use i/o redirection to notify parts of the application that previous data may be out of date. After the top-level loop resumes, the application can recompute any values that may have changed. (See Chapter 4 [I/O], page 4.)

9.2 View Commands Mirror Show Commands

The GDB command language contains many `set` and `show` commands. These commands are used to modify or examine parameters to the debugger.

It is difficult to get the current state of a parameter from the `show` command because `show` is very verbose.

```
(gdb) show check type
Type checking is "auto; currently off".
(gdb) show width
Number of characters gdb thinks are in a line is 80.
```

For every `show` command, libgdb includes a `view` command. `view` is like `show` without the verbose commentary:

```
(gdb) view check type
auto; currently off
(gdb) view width
80
```

(The precise format of the ouput from `view` is subject to change. In particular, `view` may one-day print values which can be used as arguments to the corresponding `set` command.)

9.3 The Application Can Have Its Own Breakpoints

The GDB breakpoint commands were written with a strong presumption that all breakpoints are managed by a human user. Therefore, the command language contains commands like 'delete' which affect all breakpoints without discrimination.

In libgdb, there is added support for breakpoints and watchpoints which are set by the application and which should not be affected by ordinary, indiscriminate commands. These are called *protected* breakpoints.

break-protected *...* Debugger Command
watch-protected *...* Debugger Command

> These work like `break` and `watch` except that the resulting breakpoint is given a negative number. Negative numbered breakpoints do not appear in the output of `info breakpoints` but do in that of `info all-breakpoints`. Negative numbered breakpoints are not affected by commands which ordinarily affect 'all' breakpoints (e.g. `delete` with no arguments).
>
> Note that libgdb itself creates protected breakpoints, so programs should not rely on being able to allocate particular protected breakpoint numbers for themselves.

More than one breakpoint may be set at a given location. Libgdb adds the concept of *priority* to breakpoints. A priority is an integer, assigned to each breakpoint. When a breakpoint is reached, the conditions of all breakpoints at the same location are evaluated in order of ascending priority. When breakpoint commands are executed, they are also executed in ascending priority (until all have been executed, an error occurs, or one set of commands continues the target).

priority *n bplist* Debugger Command

> Set the priority for breakpoints *bplist* to *n*. By default, breakpoints are assigned a priority of zero.

9.4 Structured Output, The `Explain` Command

(This section may be subject to considerable revision.)

When GDB prints a the value of an expression, the printed representation contains information that can be usefully fed back into future commands and expressions. For example,

```
(gdb) print foo
$16 = {v = 0x38ae0, v_length = 40}
```

On the basis of this output, a user knows, for example, that `$16.v` refers to a pointer valued `0x38ae0`

A new output command helps to make information like this available to the application.

explain *expression* Debugger Command
explain */format expression* Debugger Command

> Print the value of *expression* in the manner of the `print` command, but embed that output in a list syntax containing information about the structure of the output.

As an example, `explain argv` might produce this output:

```
(exp-attribute
   ((expression "$19")
    (type "char **")
    (address "48560")
    (deref-expression "*$19"))
   "$19 = 0x3800\n")
```

The syntax of output from `explain` is:

```
<explanation> :=    <quoted-string>
                  | (exp-concat <explanation> <explanation>*)
                  | (exp-attribute <property-list> <explanation>)

<property-list> := ( <property-pair>* )

<property-pair> := ( <property-name> <quoted-string> )
```

The string-concatenation of all of the `<quoted-string>` (except those in property lists) yields the output generated by the equivalent `print` command. Quoted strings may contain quotes and backslashes if they are escaped by backslash. `"\n"` in a quoted string stands for newline; unescaped newlines do not occur within the strings output by `explain`.

Property names are made up of alphabetic characters, dashes, and underscores.

The set of properties is open-ended. As GDB acquires support for new source languages and other new capabilities, new property types may be added to the output of this command. Future commands may offer applications some selectivity concerning which properties are reported.

The initial set of properties defined includes:

- `expression`

  This is an expression, such as `$42` or `$42.x`. The expression can be used to refer to the value printed in the attributed part of the string.

- `type`

  This is a user-readable name for the type of the attributed value.

- `address`

  If the value is stored in a target register, this is a register number. If the value is stored in a GDB convenience variable, this is an integer that is unique among all the convenience variables. Otherwise, this is the address in the target where the value is stored.

- `deref-expression`

  If the attributed value is a pointer type, this is an expression that refers to the dereferenced value.

Here is a larger example, using the same object passed to `print` in an earlier example of this section.

```
(gdb) explain foo
(exp-attribute
  ( (expression "$16")
```

```
    (type "struct bytecode_vector")
    (address 14336) )
  (exp-concat
    "$16 = {"
    (exp-attribute
      ( (expression "$16.v")
        (type "char *")
        (address 14336)
        (deref-expression "*$16.v") )
      "v = 0x38ae0")
    (exp-attribute
      ( (expression "$16.v_length")
        (type "int")
        (address 14340) )
      ", v_length = 40")
     "}\n"))
```

It is undefined how libgdb will indent these lines of output or where newlines will be included.