# Scripted Debugging : Using Tcl and GDB to debug C code

Moses DeJong
mdejong@users.sourceforge.net

## Abstract

This document describes a Tcl scripting interface to the gdb debugger. Scripting interaction with gdb makes it possible to debug a number of problems that would be difficult or impossible to address using a command line or graphical debugger. Examples of bugs that are well-suited to scripted debugging will be presented, along with a short discussion about implementation and future directions.

## Keywords

Scripting, Debugging, Visualization, Introspection, Automation

## Introduction

Like any human endeavor, programming is a balancing act. On one side the formidable inertia of the status quo, and on the other, the fads embraced by eager participants. One could make a convincing case that new ideas and new approaches have led to more productive programming languages and environments. Unfortunately, similar advances in debugging technology are significantly more difficult to identify [1,2,3,4,5,6]. The reality is that debugging is not a particularly exciting area of study. Many programmers would rather do something else, and some even advocate avoiding debuggers entirely, suggesting that **`printf`** statements are sufficient. Personal experience has shown that programming methodologies like Extreme Programming [7] and the consistent use of regression testing [8,9] can dramatically reduce the number of bugs introduced into software in the first place. Despite these advances, bugs continue to sneak into the code and programmers continue to need effective debugging tools to help exterminate them.

One tool commonly held up as a great advance in debugging technology is the graphical debugger. A graphical debugger is useful for displaying source code with a call stack, and values in memory can be displayed with a few clicks of the mouse. While certainly an improvement over a command line debugger, there are a variety of bugs for which a graphical debugger is of very little help.

This document will explore a number of these difficult problems and will introduce a debugging tool written to solve them. The tool will be referred to as the *GDBMI package*. It is a Tcl library that automates interaction with the gdb debugger using a relatively new gdb interface called the *MI* (aka GDB Machine Interface). Each of the examples involves debugging C code, but in practice a similar approach can be used to debug any language supported by gdb. Implementation details will be discussed later, but the short explanation is that the GDBMI package works by opening a pipe to a gdb process and then sending commands while interpreting results similar to how a user would interact with the command line version of gdb. Because the interaction is automated, it is possible to interpret results and take actions based on Tcl code that is completely separate from the C code being debugged.

## Example 1

The following example demonstrates the basics of interacting with the debugger, querying a value in the program being debugged, setting a breakpoint, and stepping through code. Assume the following C code has been compiled with debug symbols, like so:

```
% cat stepping.c
int main() {
   int i = 0;
   i++;
   i++;
   i++;
   return 0;
}
% gcc -o stepping -g stepping.c
```

The following Tcl code will load the GDBMI package, create a pipe to gdb behind the scenes, and step through the **`stepping`** executable. The version of gdb on the path must be 5.1 or newer and the commands must be run in wish so that I/O events are processed at idle time

```
package require GDBMI

set s [GDBMI::MIPipeStream s]
set d [GDBMI::MIDispatcher d]
$d setStream $s

$d mi_setExecutable ./stepping

$d mi_setBreakpoint stepping.c:2
$d setBreakpointHandler stop 1

proc stop { d brknum tid info } {
    puts "stopped at breakpoint"
    $d setStepHandler step
    $d mi_next
}

proc step { d tid info } {
    array set info_array $info
    puts -nonewline "(step) in\
        $info_array(func) at\
        $info_array(file):$info_array(line)"
```

```
    if {$info_array(func) == "main"} {
        set i [$d mi_gdbEval i]
        puts -nonewline " i is $i"
    }
    puts ""
    $d mi_next
}

$d mi_run
```

When the **mi_run** command is executed, the stepping executable will be started and the following will be printed.

```
stopped at breakpoint
(step) in main at stepping.c:3 i is 0
(step) in main at stepping.c:4 i is 1
(step) in main at stepping.c:5 i is 2
(step) in main at stepping.c:6 i is 3
(step) in main at stepping.c:7 i is 3
(step) in __libc_start_main at
../sysdeps/generic/libc-start.c:92
```

Creating the stream and dispatcher objects, setting the executable, and setting a breakpoint should be self-explanatory. The interesting part of this example is the ability to create Tcl callbacks that are invoked when something happens in the debugger. The **mi_setBreakpoint** method sets the breakpoint in the debugger and **setBreakpointHandler** associates a Tcl callback with the breakpoint. When **mi_run** is executed, the program being debugged runs until it hits the first breakpoint at line 2 (the line where the variable **i** is declared). At that point, a callback is evaluated and the **stop** Tcl command is run. The **stop** command prints the text "stopped at breakpoint" and then sets up another callback using **setStepHandler**, so that the **step** Tcl command will be invoked when the next gdb step operation has finished. Finally, the **mi_next** method is invoked to tell gdb to step to the next line in the file.

The **step** command introduces some new concepts. First, the **info** argument, which contains information about the current stack frame in gdb, is converted from a list to an array. Second, the **mi_gdbEval** method is used to query the value of the local variable **i** inside the program being debugged. This example demonstrates how easily a simple interaction with gdb can be scripted and how easily the script can be customized to address a specific need.

## Example 2

Now that we have had some exposure to the basics of scripted interaction with a program running inside the debugger, let's move on to a more complex and vastly more interesting example. Resource management is an area that consistently provides programmers with bugs that can be very difficult to track down. The following example shows how the GDBMI package can be used to find a leaking resource. In this case, a memory region that was allocated but was not freed.

```
% cat resources.c
#include <malloc.h>

typedef struct Resource { int num; } Resource;

typedef struct Container {
    char*     ident;
    Resource* res;
} Container;

Container* allocContainer(char* ident) {
    Container* c = (Container *) malloc(
sizeof(Container));
    c->ident = ident;
    c->res = (Resource *) malloc( sizeof(
Resource));
    return c;
}

void deallocContainer(Container* c) {
    free(c->res);
    free(c);
}

int main(int argc, char ** argv) {
  Container *c1, *c2, *c3, *c4, *c5;

  c1 = allocContainer("c1");
  c2 = allocContainer("c2");
  c3 = allocContainer("c3");
  c4 = allocContainer("c4");
  c5 = allocContainer("c5");

  /* Do A Bunch Of Stuff */

  deallocContainer(c5);
  deallocContainer(c4);
  if (argc == 1)
      deallocContainer(c3);
  deallocContainer(c2);
  deallocContainer(c1);

  exit(0);
}
```

To track the allocation and deallocation of **Resource** structs in this program, two breakpoints will be set. The first breakpoint will be set at line 14, just after the second **malloc** statement in the **allocContainer** function. The second breakpoint will be set at line 18 in the function **deallocContainer**, just before the **Resource** struct pointer is passed to **free**. The following Tcl code creates the basic stream objects and sets the breakpoints.

```
package require GDBMI
set s [GDBMI::MIPipeStream s]
set d [GDBMI::MIDispatcher d]
$d setStream $s

$d mi_setExecutable ./resources
```

```
$d mi_setBreakpoint resources.c:14
$d setBreakpointHandler allocResource 1

$d mi_setBreakpoint resources.c:18
$d setBreakpointHandler deallocResource 2
```

Now the callbacks that will be invoked when these breakpoints are hit will need to be defined. Our goal here is to track the allocation and deallocation of **Resource** structs, so we need to match up the **malloc** call that returns an address with the **free** call for that same address. A Tcl array that uses a memory address as the hash key can be used to solve this problem. The following callbacks will query the memory address of the **Resource** struct inside **allocContainer** and use the result as an index into the global array **map**.

```
proc allocResource { d brknum tid info } {
    set ptr [$d mi_gdbEval {(void *) c->res}]
    lappend ::map($ptr) allocated
    $d mi_continue
}

proc deallocResource { d brknum tid info } {
    set ptr [$d mi_gdbEval {(void *) c->res}]
    lappend ::map($ptr) deallocated
    $d mi_continue
}
```

Now that everything is setup, the program can be run with the following commands:

```
if {[info exists map]} {unset map}
$d mi_run
```

When the program finishes, the collected information can be printed:

```
% parray map
map(0x80496b8) = allocated deallocated
map(0x80496d8) = allocated deallocated
map(0x80496f8) = allocated deallocated
map(0x8049718) = allocated deallocated
map(0x8049738) = allocated deallocated
```

The above output shows that everything is working properly. Each of the resources that got allocated was later deallocated. The attentive reader might have noticed that in the main function presented above, a **Container** struct is leaked only when a command line argument is passed to the program. The following commands will trigger the leaking bug in main by passing a single command line argument and then restarting the program.

```
GDBMI::setArgs $d bugon
if {[info exists map]} {unset map}
$d mi_run
```

Sure enough, one of the resources is now being leaked.

```
% parray map
map(0x80496b8) = allocated deallocated
map(0x80496d8) = allocated deallocated
map(0x80496f8) = allocated
map(0x8049718) = allocated deallocated
map(0x8049738) = allocated deallocated
```

At this point, the scope of the problem is much clearer. A **Resource** struct was allocated at memory address **0x80496f8** but that memory was never deallocated. Something similar could be accomplished using Tcl's **memory trace** command or other memory systems like Purify, but it would take additional work to limit the results to include only **Resource** structs.

Knowing that one **Resource** struct is being leaked could be useful. For example, if one found and fixed the problem, then this approach could be used to verify that the fix worked. Unfortunately, just knowing the memory address of a leaking resource can be of very little use in a complex system. Often, one resource is tied to another and knowing the memory address of one may not lead to the other. For example, the Tk library will allocate system resources and associate them with a window object that has a name like **.top**. Knowing that a leaking system resource is at memory address **0x8035723** would not be nearly as useful as knowing that the resource was associated with the **.top** window.

In the example C code, a **Container** struct holds a **Resource** struct, so the debugging code needs to find the container that contains the leaking resource. The following code implements this by making a small modification to the **allocResource** callback presented earlier.

```
proc allocResource { d brknum tid info } {
    set ptr [$d mi_gdbEval {(void *) c->res}]
    set cid [GDBMI::getString \
        [$d mi_gdbEval {c->ident}]]
    lappend ::map($ptr) $cid allocated
    $d mi_continue
}
```

Instead of just appending the string "allocated" to the array, the callback also appends the string value of the **ident** member of the **Container** struct. When the program is run again after defining this new callback, it outputs the following:

```
% parray map
map(0x80496b8) = c1 allocated deallocated
map(0x80496d8) = c2 allocated deallocated
map(0x80496f8) = c3 allocated
map(0x8049718) = c4 allocated deallocated
map(0x8049738) = c5 allocated deallocated
```

It is obvious from the output above that the **Container** with the "c3" identifier is the one that is leaking the **Resource** object. While the results of this small

example could be duplicated using a GUI debugger, a pencil, and some paper, it would not be possible in a real system that contained hundreds or thousands of **`Container`** and **`Resource`** structs.

The results shown are much more useful than those from a low-level memory tool that is able to output only a file name and line number where a leaked memory allocation occurred. Knowing where a leaked allocation occurred might get you in the neighborhood, but if a whole class of resources is allocated at the same line in a file, there is no way to differentiate specific instances of that resource. The approach shown in this example is not presented as a replacement for a low-level malloc debug library [10, 11, 12]. Instead, the example shows how specific information about a leak can be uncovered after it is known that there is a leak somewhere in a class of resources.

## Example 3

In this next example, the C stack will be inspected by Tcl code and interesting values will be displayed. The program being debugged will also include a Tcl interpreter, so don't confuse the two. Assume that there is a crashing bug in the Tcl **`string`** command of the program being debugged and that the developer needs to find the Tcl call stack before the source of the problem can be identified. The following Tcl code will initialize gdb, have it create the child process to be debugged, set a breakpoint in the C function that is crashing, and then display the call stack after the breakpoint is hit.

```
package require GDBMI
set s [GDBMI::MIPipeStream s]
set d [GDBMI::MIDispatcher d]
$d setStream $s

$d mi_setExecutable ./tclsh

set fd [open foo.tcl w]
puts $fd {
  proc p1 { arg1 } {
      eval "string compare f1 f2"
  }
  proc p2 { arg1 arg2 } {
      p1 0
  }
  p2 1 2
}
close $fd

GDBMI::setArgs $d foo.tcl

# Set breakpoint in Tcl's string cmd
$d mi_setBreakpoint tclCmdMZ.c:1214
$d mi_run
```

At this point the user would wait until the subprocess had started up, begun processing the **`foo.tcl`** script, and hit the breakpoint. A small helper function is used to print the C stack in a readable format.

```
proc print_stack { frames } {
  foreach frame $frames {
    if {[array exists map]} {unset map}
    array set map $frame

    puts -nonewline $map(func)
    puts -nonewline "("

    foreach {varname varvalue} $map(args) {
      puts -nonewline " $varname"
    }
    puts " )"

    # Ignore __libc_start_main entry point
    if {$map(func) == "main"} { break }
  }
}

% set stack [GDBMI::getStack $d]
% print_stack $stack
0 Tcl_StringObjCmd( dummy interp objc objv )
1 TclEvalObjvInternal( interp objc objv
command length flags )
2 Tcl_EvalEx( interp script numBytes flags )
3 Tcl_EvalObjEx( interp objPtr flags )
4 Tcl_EvalObjCmd( dummy interp objc objv )
5 TclEvalObjvInternal( interp objc objv
command length flags )
6 TclExecuteByteCode( interp codePtr )
7 TclCompEvalObj( interp objPtr )
8 Tcl_EvalObjEx( interp objPtr flags )
9 TclObjInterpProc( clientData interp objc
objv )
10 TclEvalObjvInternal( interp objc objv
command length flags )
11 TclExecuteByteCode( interp codePtr )
12 TclCompEvalObj( interp objPtr )
13 Tcl_EvalObjEx( interp objPtr flags )
14 TclObjInterpProc( clientData interp objc
objv )
15 TclEvalObjvInternal( interp objc objv
command length flags )
16 Tcl_EvalEx( interp script numBytes flags )
17 Tcl_FSEvalFile( interp pathPtr )
18 Tcl_Main( argc argv appInitProc )
main( argc argv )
```

Unfortunately, the C stack may be of little value to a Tcl developer. Often, one needs to know the state of the Tcl stack at the time a crash occurred. This state can be discovered by poking around in the C stack, but the process can take a long time and it requires detailed knowledge about how Tcl is implemented. The following helper function automates this process.

```
proc print_tcl_stack { frames d } {
  foreach frame $frames {
    if {[array exists map]} {unset map}
    array set map $frame
    if {[array exists args_map]} {
      unset args_map
    }
    array set args_map $map(args)

    if {$map(func) == "Tcl_FSEvalFile"} {
```

```
      set res [$d mi_gdbEval [format \
          {(%s)->bytes} \
          $args_map(pathPtr)]]
      set file [GDBMI::getString $res]

      puts "C stack $map(level) :\
          Tcl : source $file"
    } elseif {$map(func) == \
        "TclEvalObjvInternal"} {
      set tcl_args [list]
      for {set i 0} {$i < $args_map(objc)} \
          {incr i} {
        set res [$d mi_gdbEval \
            [format {(%s)[%d]->bytes} \
            $args_map(objv) $i]]
        set str [GDBMI::getString $res]
        lappend tcl_args $str
      }
      puts "C stack $map(level) : \
          Tcl : $tcl_args"
    }

    # Ignore __libc_start_main entry point
    if {$map(func) == "main"} { break }
  }
}

% print_tcl_stack $stack $d
C stack 1 : Tcl : string compare f1 f2
C stack 5 : Tcl : eval string
C stack 10 : Tcl : p1 0
C stack 15 : Tcl : p2 1 2
C stack 17 : Tcl : source "foo.tcl"
```

In the example above, arguments to the `TclEvalObjvInternal` function are printed along with any file name passed to `Tcl_FSEvalFile`. C stack levels that don't provide useful information about the Tcl stack are ignored. This example of filtering information from C stack frames is simple, but it shows how easily a complex data structure like the C stack can be inspected using a script.

## GDBMI Package Implementation

The GDBMI package is implemented entirely in Tcl code. A pipe to a gdb process is created and messages are read from and written to the pipe. Typically, gdb will accept an MI command and then return a result indicating that the command was accepted. The output below shows a Tcl command and the resulting data that is written to and read from the pipe.

```
% $d mi_setExecutable ./stepping
-file-exec-and-symbols ./stepping
^done
% $d mi_setBreakpoint stepping.c:2
-break-insert stepping.c:2
^done,bkpt={number="1",type="breakpoint",disp=
"keep",enabled="y",addr="0x080483a6",func="mai
n",file="stepping.c",line="2",times="0"}
% $d mi_run
-exec-run
^running
```

Asynchronous messages are also generated by gdb. These messages are not read from the pipe as the result of a command; instead, they are generated when some event of interest occurs inside the debugger. For example, when the breakpoint set in the commands above is hit, the following is read from the pipe.

```
*stopped,reason="breakpoint-hit",bkptno="1",
thread-id="0",frame={addr="0x080483a6",
func="main",args=[],file="stepping.c",
line="2"}
```

When an asynchronous message like the above is received, a Tcl callback is invoked to inform the script that something of interest happened and that additional actions can be taken.

Readers familiar with the command line version of gdb will have noticed a couple of key differences in the way that the MI and the regular command line interfaces work. MI commands return a result right away and then generate an asynchronous message when an event has occurred. The regular command line version of gdb will block until the requested operation has finished. MI commands also return information in a structured format that is easily parsed while the regular command line gdb interface returns information in a human readable format. The following output shows how the commands presented above would be entered at the regular gdb command prompt.

```
(gdb) file ./stepping
Reading symbols from ./stepping...done.
(gdb) break stepping.c:2
Breakpoint 1 at 0x80483a6: file stepping.c,
line 2.
(gdb) run
Starting program: ./stepping
Breakpoint 1, main () at stepping.c:2
2           int i = 0;
```

Writing a parser to interpret the results above would be significantly more difficult and error-prone when compared to parsing MI output. The fact that commands in the regular gdb interface block also causes problems for the program trying to interact with gdb. Users of the ddd GUI debugger have no doubt seen a generic "debugger is busy" error, which shows up when the command line version of gdb has blocked the pipe as the result of some command. Solving the blocking issue and presenting data in a machine-parseable format were some of the primary design goals of the MI.

Readers familiar with the Insight debugger [13], a graphical debugger written in Tcl/Tk, will no doubt wonder how the MI implementation differs from the Insight implementation. Many of the commands supported by the MI are modeled after the Tcl

commands in Insight. Insight actually predates the MI by a couple of years, and lessons learned implementing Insight were considered when creating the MI. Insight is certainly a useful tool, but the reality is that it can't really be called stable since it is highly crash-prone. Insight is not a GUI that talks to gdb via a pipe; both the gdb code and the Tcl/Tk code exist in the same process. When something goes wrong in Insight's gdb code, it crashes and takes the GUI down with it. That leaves the user wondering what happened and makes reproducing problems extremely difficult.

Parts of gdb were written with the assumption that commands would always block, so special code was needed to get the gdb code in Insight to cooperate with the Tcl/Tk event loop. These special code paths were used only by Insight and as a result were not tested as part of the normal gdb release process. That lead to breakage in the Insight code as a result of gdb maintenance. The end result is that Insight tends to be highly crash-prone and it is not clear if or how things are going to improve in the future.

On the other hand, the MI has a number of regression tests included in the normal gdb testing process, so it is less likely to be broken at any point. It should not be difficult to create a reproducible test case for a crashing bug in the MI, but that issue seems to be moot since the MI does not appear to suffer from the crashes that plague Insight. While the MI is not perfect, it appears that the MI is and will continue to be a more stable and easier to use method of interacting with gdb from Tcl.

## Future Directions

While the GDBMI package has proven useful in a number of situations, some areas could be improved. Writing to stdout or stderr and reading from stdin by the process being debugged is not implemented properly in the MI. I/O of this sort must to be redirected to or from files when using the GDBMI package. Providing a means to set a controlling terminal for the process being debugged would also be a useful improvement.

Improving regression testing integration is another area that deserves further exploration. Currently, breakpoints are set at a line number in a file, and the line numbers need to be updated when the source file changes. It is obvious that having to update line numbers in every regression test after a change to the source code would be tedious and error-prone. A simple solution to this problem is to create a new type of breakpoint that contains a function name and an offset in lines from the start of the function. In this way, a breakpoint specification like `{my_func + 5}` could be given, the line number of `my_func` could be queried, and the breakpoint could be set at the returned line plus five. This would insulate breakpoints from changes in the source file that shift a function up or down some number of lines.

Effective regression tests are also going to need to deal with timeouts, programs that get stuck in a loop, signaled termination, and other unexpected exit conditions. The Expect package could prove to be very useful in these situations; further exploration of the combination of GDBMI commands and Expect is warranted.

Integration into the Source-Navigator IDE [14] is another area that deserves additional exploration. Currently, there is some integration of the Source-Navigator IDE and the Insight debugger, but it is far from ideal. Source-Navigator is great for static source code analysis, but it is lacking any runtime analysis or visualization features. Combining runtime information gathered using the GDBMI package with existing static code analysis features in Source-Navigator could prove to be very useful.

## Source Code

The GDBMI package can be downloaded and put to use today. Source is available under a Tcl like license at the following URL:

`http://www.uncounted.org/tcl/gdbmi-0.1.tgz`

## References


[1] H. Lieberman. The Debugging Scandal and What to Do About It. http://web.media.mit.edu/~lieber/Lieberary/Softviz/CACM-Debugging/CACM-Debugging-Intro.html#Intro

[2] R. Baecker, C. DiGiano, A. Marcus. Software Visualization for Debugging http://web.media.mit.edu/~lieber/Lieberary/Softviz/CACM-Debugging/SoftViz/SoftViz.html

[3] P. Dibble. Visualize a Better Debugger. http://www.embedded.com/story/OEG20021217S0035

[4] http://www.trnmag.com/Stories/2002/080702/Programming_tool_makes_bugs_sing_080702.html

[5] A. Hunt, D. Thomas. The Pragmatic Programmer. http://c2.com/cgi/wiki?RubberDucking

[6] http://www.lambdacs.com/debugger/debugger.html

[7] http://www.extremeprogramming.org/

[8] D. Libes. Regression Testing and Conformance Testing Interactive Programs http://expect.nist.gov/doc/regress.ps

[9] http://www-106.ibm.com/developerworks/linux/library/l-jacks/

[10] http://www.rational.com/products/pqc/index.jsp?SMSESSION=NO

[11] http://www.gnu.org/software/gcc/projects/bp/main.html

[12] http://www.gnome.org/projects/memprof/

[13] J Ingham. GDBTk: Integrating Tcl/Tk into a Recalcitrant Command-Line Application. http://www.usenix.org/publications/library/proceedings/tcl2k/full_papers/ingham/ingham_html/index.html

[14] http://sourcenav.sourceforge.net/