This document is an early draft.
Please do not redistribute it without prior permission from the authors.

The FIDDLE sources are in alpha stage.
If you had access to FIDDLE sources, please do not redistribute them without prior permission from the authors.

FIDDLE IS PROVIDED "AS IS" AND WITH NO WARRANTIES, EITHER IMPLIED OR EXPLICIT. USE IT AT YOUR OWN RISK.

Copyright © 1999 by Departamento de Informática of the FCT/UNL

João Lourenço <Joao.Lourenco@di.fct.unl.pt>
José C. Cunha <Jose.Cunha@di.fct.unl.pt>

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática
Quinta da Torre
P-2825-114 Monte de Caparica
Portugal
1 Introduction

FIDDLÉ
1: to move the hands or fingers restless
2: to spend time in aimless or fruitless activity

in Webster Dictionary

1.1 What is FIDDLÉ

FIDDLÉ is a distributed debugging engine, which allows a quick and easy interface with other tools acting as debugging interfaces.

FIDDLÉ is structured in 5 layers, and in Edition 0.2.8 just layers 0s, 0m, and 1m are finished and documented. This means that Chapter 8 [FIDDLÉ-2m], page 53 and Chapter 9 [FIDDLÉ-3m], page 55, are intentionally empty for now.

FIDDLÉ was developed as part of João Lourenço’s PhD work. If you are interested in distributed debugging in general, and in this work in particular, may be you would like to consult some of the publications listed in Appendix A [FIDDLÉ Related Publications], page 61, whose majority is available online at http://www-asc.di.fc.tul.pt/~jml/papers/.

1.2 FIDDLÉ vs. Traditional Debuggers

The generality of the traditional debuggers, such as DBX and GDB, have a monolithic architecture. This means it is impossible to dissociate the debugging functionalities from the debugging (user) interface, being the provided debugging interface the only way to access the debugging functionalities.

Many nice-looking and user-friendly sequencial debugging tools (e.g., XXGDB, DDD) are actually graphical front-ends which use a lower-level text-oriented debugging interface (such as GDB or DBX) as the back-end debugger.

In what concerns to the interface with the back-end debugger, in general all these tools act the same way: start the back-end debugger in the background connected to the graphical front-end by way of a bi-directional pipe-like channel. The debugging commands issued at the graphical level are converted into its textual form, and are sent to the back-end debugger. Then the graphical front-end waits for the resulting output text from the back-end debugger, parse it, extract the relevant data and change its status accordingly.

Parallel and distributed debuggers have mainly graphical interface has their primary debugging interface, which reduces drastically, or even eliminates, any possibility of integration with other tools.

FIDDLÉ aims to eliminate this restrictions, providing a distributed debugging engine accessible through a function-based interface. Due to its organization in layers with incremental functionalities, the user (programmer) can select which layer is more adequate to its needs (see Section 1.3 [FIDDLÉ Layers], page 2, for more details on FIDDLÉ layers and functionalities).

By using FIDDLÉ, it is possible to develop multiple debugging front-ends (FIDDLÉ clients), adapted to the user needs and providing different looks and capabilities. This debugging front-ends may operate simultaneously over the same set of target processes, as FIDDLÉ provides the basic mechanisms to assure that they have a coherent view of the target processes.
1.3 FIDDLE Layers

FIDDLE is structured in 5 layers. As of Edition 0.2.8, layers 0s, 0m, and 1m are finished and documented, and layers 2m and 3m are under development. Below you can find a brief description of the functionalities available in each layer.

**Fiddle-0s (Layer 0s)**

FIDDLE-0s aims to provide a function-based interface to access a back-end debugger. Currently the only back-end debugger supported is the GNU GDB (see Chapter 2 [Compiling and Installing FIDDLE], page 7 for more information on the versions of GDB supported).

The general architecture of FIDDLE-0s is presented in the figure below (the dashed line(s) delimit the physical machine(s) and the dotted line FIDDLE components).

![FIDDLE-0s Diagram](image)

FIDDLE-0s accepts only one single-threaded client, but is able to control multiple multi-threaded target processes\(^1\) running in the same machine.

This set of debugging services, accessible by function calls, transfer to FIDDLE-0s the responsibility to start new instances of the back-end debugger as needed, generate the appropriate debugging commands and send them to the appropriate back-end debugger; and to collect, parse and extract the relevant data from the response (text) of the back-end debugger.

FIDDLE-0s is explained in detail in Chapter 5 [FIDDLE-0s], page 23.

**Fiddle-0m (Layer 0m)**

The general architecture of FIDDLE-0m is presented in the figure below (the dashed line(s) delimit the physical machine(s) and the dotted line FIDDLE components).

---

\(^1\) As long as the back-end debugger supports threaded processes, such as the GNU GDB 4.17 running on the Linux/RedHat-5.2 system.
**FIDDLE 0m**

FIDDLE-0m extends FIDDLE-0s by supporting multi-threaded clients. Concurrent requests to different target processes are processed concurrently. Concurrent requests to the same target processes are serialized and executed one after another. FIDDLE-0m is explained in detail in Chapter 6 [FIDDLE-0m], page 45.

**Fiddle-1m (Layer 1m)**

The general architecture of FIDDLE-1m is presented in the figure below (the dashed line(s) delimit the physical machine(s) and the dotted line FIDDLE components).

FIDDLE-1m provides the same functionalities as FIDDLE-0m, but now the target-processes can reside in multiple (remote) machines, and not only in the local machine.

The set of arguments if some of the FIDDLE-1m services had to be changed to incorporate this new concept of remote processes.
FIDDLE-1m is explained in detail in Chapter 7 [FIDDLE-1m], page 49.

**FIDDLE-2m** *(Layer 2m)*

This layer is not implemented yet!

The general architecture of FIDDLE-2m is presented in the figure below (the dashed line(s) delimit the physical machine(s) and the dotted line FIDDLE components).

FIDDLE-2m extends FIDDLE-1m, in the sense that multiple client processes are now allowed. These multiple clients may be concurrently issuing debugging commands to the same processes.

It is assumed that the multiple clients use external means (to FIDDLE) to implement the coordination between themselves.

FIDDLE-2m is explained in detail in Chapter 8 [FIDDLE-2m], page 53.

**FIDDLE-3m** *(Layer 3m)*

This layer is not implemented yet!

The general architecture of FIDDLE-2m is presented in the figure below (the dashed line(s) delimit the physical machine(s) and the dotted line FIDDLE components).

(Figure to be done)

All previous layers were based in blocking services: a function was called that would perform a service in a local or remote process, blocking the (thread) caller until the service was completed.

FIDDLE-3m introduces the concept of *events* and *event handlers*. In this layer, when a service is requested, a *Request_ID* is returned immediately. When the requested
service is completed, an event will be generated by FIDDLE-3m and sent to the caller, which will activate an event handler function with the Request_ID as argument.

In FIDDLE-3m any client may specify its interest in being informed of the services requested by the other clients, and also on their results.

FIDDLE-3m is explained in detail in Chapter 9 [FIDDLE-3m], page 55.

**FIDDLE Layers Summarized**

The characteristics and properties of the FIDDLE layers can be summarized as follows:

<table>
<thead>
<tr>
<th>Client process</th>
<th>F0s</th>
<th>F0m</th>
<th>F1m</th>
<th>F2m</th>
<th>F3m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single client</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple clients</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Single-threaded</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-threaded</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

| Target process         |     |     |     |     |     |
| Multiple target processes |   | X   | X   | X   |

| Distribution           |     |     |     |     |     |
| One machine            | X   | X   |
| Multiple machine       |     | X   | X   | X   |

| Type of service        |     |     |     |     |     |
| Function call based    | X   | X   | X   | X   |
| Event based            |     |     |     | X   |

### 1.4 How FIDDLE Operates

FIDDLE was designed for building debuggers using a client-server model. You program the debugging user interface—so called FIDDLE client in this text—, and use FIDDLE services to access the debugging functionalities.

FIDDLE is thread-safe (except in layer 0s), so your client may be a single- or multi-threaded program.

Any thread requesting a FIDDLE service remains blocked until the service is completed (or fails). If you don’t want you client to be blocked, e.g., waiting for a process being debugged to hit a breakpoint, start a new thread and let it make the service request and wait for the reply, leaving the main thread for the general control of the user interface.

All FIDDLE services accept a reduced number (from zero to five) of simple basic-type parameters, such as integers and char arrays (strings).

All FIDDLE services, except initialization and termination, return a quite complex tkout_t structure. This structure encloses all the possible successful and erroneous reply values from FIDDLE. See Section 5.1.1 [The tkout_t structure], page 23. Mastering FIDDLE depends a lot on mastering the management of this structure and the interpretation of its contents. In Chapter 4 [A Tiny Example], page 13 we present the main ideas behind the management of the tkout_t structure.

### 1.5 Some Notes to the Reader
Thanks for trying/using FIDDLE.

We hope you’ll feel this reference manual clear and complete enough. Comments and suggestions to its improvement are always very welcome, and can be sent to the same addresses used to report bugs. See Chapter 3 [Reporting Bugs], page 11.

C source code presented in this manual

All the C source code presented in this manual, from simple data type definitions to full C programs, is extracted from the original source files at document-generation time and should be exactly equal (except for the line numbers), to the corresponding (types in the) source files.

For example, the piece of C code with the declaration of the ‘tkout_t’ data type, introduced in Section 5.1.1 [The tkout_t structure], page 23, was extracted from the file ‘lib/f0s/f0s_tkout.h’ just before generating this documentation file, so it should be up to date with the original.

Dependencies

FIDDLE uses an auxiliary library named libutil.a, where some general data structures and services are implemented, e.g., a reentrant double linked list and the pretty printing error and aborting messages, and is described in detail in Appendix B [FIDDLE Utilities Library], page 63. To master FIDDLE you must know how to use these library functions.

If you develop a FIDDLE client, you must link your program with this library too, and also with the pthreads library (even if your program is not multithreaded!), as explained in Chapter 2 [Compiling and Installing FIDDLE], page 7.

Availability

FIDDLE sources are in alpha stage, and (most probably) you have obtained your copy directly from the authors or with their permission.

Please be patient with the bugs you may found and report them (see Chapter 3 [Reporting Bugs], page 11) as soon as possible.

Also, do not redistribute FIDDLE without prior permission from the authors, as we plan to make it available to the community by using the GNU GPL Licence, or something similar, but we haven’t decided yet!
2 Compiling and Installing FIDDLE

2.1 Development and Testing Environment

Until now, the development and testing of FIDDLE has been restricted to the following environment:

Operating systems
Linux RedHat 6.2 (includes linuxthreads)

C compilers
egcs-2.91.66

Low-level debuggers
gdb 4.18

We would appreciate if you could send us some reports on it’s behavior in the same and other environments. See Chapter 3 [Reporting Bugs], page 11.

2.2 Getting FIDDLE

The last version of FIDDLE’s source code is available from
In the same address you may also find binaries for different architectures, as well as an on-line version of this and other FIDDLE related documents.

2.3 Compiling FIDDLE

FIDDLE is distributed with a ‘configure’ script, which should make your compile and install process much easier.

To compile FIDDLE, unpack the sources and call ‘configure’, a shell script in the source root directory, e.g.,

$ gzip -dc fiddle-0.2.8.tar.gz | tar xf -
$ cd fiddle-0.2.8
$ ./configure --prefix=$HOME/fiddle-0.2.8
$ make

This will prepare to install the FIDDLE libraries and binaries in the base directory "$HOME/fiddle-0.2.8". If you omit the ‘--prefix=...’ option, the directory ‘/usr/local’ will be used by default as the base directory.

Besides ‘--prefix’, other command-line options are accepted by configure, with the ones below deserving our special attention:

--enable-shared=[PKGS]
Build shared libraries [default=yes]

--enable-static=[PKGS]
Build static libraries [default=yes]

--with-libs-from=DIR
Pass compiler flags to look for libraries in DIR

--with-include-from=DIR
Pass compiler flags to look for header files in DIR
--with-servers-dir=DIR
   Where to find/install the servers [default=EPREFIX/bin]

--with-rsh=RSH_COMMAND
   Alternative remote shell command (e.g., ssh) [default=rsh]

If `configure --help` is executed, a complete list of the command-line options accepted by
configure are presented, each accompanied with a brief explanation.

If your system is one of the systems supported by FIDDLE, the compilation process will progress
smoothly and will terminate successfully. At this point you will have the FIDDLE libraries and
binaries ready to install.

To install FIDDLE, execute the following command:

   $ make install

Besides the default and the install make targets, there are some other useful targets you may consider:

info  Create a set of info files, in 'fiddle.info', 'fiddle.info-1', 'fiddle.info-2', etc.
      This target is called implicitly by the 'default' target.

dvi   Create a DVI version of the manual, in 'doc/fiddle.dvi'.

dv   Create a Postscript version of the manual, in 'doc/fiddle.ps'.

html  Create a HTML version of the manual, in 'doc/fiddle.html'.

clean Delete all object and archive files, but not the configuration files.

distclean Delete all files not included in the distribution.

uninstall Delete all files copied by 'make install'.

2.4 FIDDLE Target Processes

The target process(es), i.e., the processes that will be debugged using FIDDLE, have to be
compiled with the -g option, to direct the compiler to incorporate debugging information into
the generated executable file, e.g.,

   $ cc -g -o target target.c

If you use the GNU ‘gcc’ or ‘egcs’ compilers, use the option -gdbg instead. This will instruct
‘GDB’ to incorporate extra debugging information in the generated executable file, that will be
recognized by the ‘GNU GDB’ debugger. This will solve some unpleasant problems, such as when
the ‘GDB’ doesn’t find the source file.

2.5 FIDDLE Client Processes

To access FIDDLE services, you need to include the file ‘fiddle/ffiddleXX.h’ in your C files,
where the XX in ‘fiddleXX.h’ should be replaced with the desired layer number (0s, 0m, 1m,
2m or 3m), e.g.,

   #include <fiddle/ffiddle1m.h>

This file (‘fiddle/ffiddle1m.h’) will include all the remaining necessary files.
Also don’t forget to add the path of FIDDLE include files to the ‘-I’ flags of your C compiler,
e.g.,
$ cc -c -I$HOME/fiddle-0.2.8/include client.c

The above examples assumes `$HOME/fiddle-0.2.8/include` as the location of FIDDLE include files.

To generate the executable, you need to link your object files with the FIDDLE libraries for the desired layer and below, and also with the FIDDLE utilities library. You can use the `-L` flag to specify the path to the FIDDLE libraries (again, replace `XX` with the correct layer number). Remember that FIDDLE uses `linuxthreads`, and so do not forget to link with the `pthreads` library too, e.g.,

```
$ cc -o client client.o -L$HOME/fiddle-0.2.8/lib \
   -lfiddle1m -lfiddle0m -lfiddle0s -lfiddleutil -lpthread
```

The example above assumes FIDDLE libraries were installed in `$HOME/fiddle-0.2.8/lib` and that the `pthreads` library was installed in a standard location.
3 Reporting Bugs

If you think you have found a bug in FIDDLE, please investigate it and report it. We have made FIDDLE available to you, and if is not to ask too much from you, report the bugs that you find. There are a few things you should think about when you put your bug report together:

1. You have to send us a test case that makes it possible for us to reproduce the bug. Include instructions on how to run the test case.
2. You also have to explain what is wrong; if you get a crash, or if the results printed are incorrect and in that case, in what way.

If your bug report is good, we will do our best to help you to get a corrected version of FIDDLE; if the bug report is poor, we won’t do anything about it (aside of chiding you to send better bug reports).

Please send your bug reports to:
Joao.Lourenco@di.fct.unl.pt
Jose.Cunha@di.fct.unl.pt

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please send a note to the same addresses as above.
4 A Tiny Example

Just for you to get the flavor of what you can do with FIDDLE, this chapter will present a complete (but very limited) example. Don’t expect to understand completely all the details of this example, but come back to it at any time in the future, and compare what is exemplified with what you are reading/studying.

The same source files presented here (`client.c` and `target.c`) are available in the `examples/` directory. They should be exactly equal (except for the line numbers), as the printed versions are generated automatically from the corresponding C source files, at document generation time.

For a more complete example, please refer to the debugging consoles, available together with FIDDLE’s source code, in the directories `f0s/console/`, `f0m/console/`, `f1m/console/`, `f2m/console/` and `f3m/console/`. The remote debugging servers, available in `f0m/serv` and `f1m/serv`, are also good examples of simple FIDDLE clients.

4.1 Compiling the example source files

The example sources will be compiled at the same time as FIDDLE. If you want to have a better knowledge of what has been done, go to the `examples/` directory and type:

```
$ make clean
$ make
```

On the second make, you should get an output similar to the one below. The longer lines (i.e. lines 2 and 4) have been broken using `\`.

```
1 cc -DHAVE_CONFIG_H -I. -I. -I. -I./util -I./f0s/lib -g -c target.c
2 cc -g -o target target.o ../f0s/lib/.libs/libfiddle0s.a \
    ../util/.libs/libfiddleutil.a
3 cc -DHAVE_CONFIG_H -I. -I. -I. -I./util -I./f0s/lib -g -c client.c
4 cc -g -o client client.o ../f0s/lib/.libs/libfiddle0s.a \
    ../util/.libs/libfiddleutil.a
```

4.2 The Target Program

Let’s take a look at the `target.c` source file:

```
1 /* ***********************************************************/
2 * target.c
3 * $Id$
4 *
5 * Copyright (c) 1999 by Dep.Informatica of the FCT/UNL
6 * Time-stamp: <00/05/23 18:48:15 jml>
7 *
8 * Joao Lourenco <Joao.Lourenco@di.fct.unl.pt>
9 *
10 * Parallel and Distributed Processing Group
11 * Departamento de Informatica
12 * Faculdade de Ciencias e Tecnologia
13 * Universidade Nova de Lisboa
14 *
15 * -----------------------------------------------
16 *
```
17  * Description: Example of a 'target program'
18  *
19  *-----------------------------------------------------------------------------
20  */
21 
22  #include <stdio.h>
23  #include <string.h>
24 
25  /* A simple structure */
26  struct ss {
27     int x;
28     float y;
29     char z[40];
30  };
31 
32  /* This function returns a structure */
33  struct ss sfake (char *str)
34  {
35     struct ss s;
36     s.x = strlen (str);
37     s.y = 56.789;
38     strcpy (s.z, str);
39     return s;
40  }
41 
42  /* The main function */
43  int main (int argc, char *argv[])
44  {
45     int i, x, z;
46     struct ss s1;
47     /* Print the command line arguments */
48     for (i = 0; i < argc; i++)
49         printf ("argv[%d] = \%s\n", i, argv[i]);
50     /* Call a function that return a structure */
51     s1 = sfake("teste");
52     /* Return */
53     return 0;
54  }
55
4.3 The Client Program

Now, let's define a FIDDLE's client program (also available in FIDDLE's distribution in the 'examples/' directory) that will:

1. Initialize FIDDLE (line 244);
2. Load an instance of the target program into memory (see Section 4.2 [The Target Program], page 13) (line 247);
3. If the command line option `-tty device` was given, set this device as the default tty for I/O operations (line 257). Note that if this option is omitted, the output generated by the target program will be lost!
4. List the first 10 source lines of the `main()` function (line 269);
5. Put a break point in line 47 (in the first `for` loop) (line 279);
6. Put another breakpoint in the `sfake()` function (line 289);
7. Run the target program with the given arguments (line 299). It will stop when the first breakpoint (in the `for` loop) is hited!
8. Continue the execution of the target process (line 309). It will stop when the second breakpoint (in the `sfake()` function) is hited!
9. Print the current stack frames (line 319);
10. Kill the target process (line 329).

```c
/* ***************************************************************
 * client.c
 * $Id$
 * Copyright (c) 1999 by Dep.Informatica of the FCT/UNL
 * Time-stamp: <00/01/27 09:56:23 jm1>
 * <Joao.Lourenco@di.fct.unl.pt>
 * Parallel and Distributed Processing Group
 * Departamento de Informatica
 * Faculdade de Ciencias e Tecnologia
 * Universidade Nova de Lisboa
 * ---------------------------------------------------------------
 * Description: Example of a FIDDLE's 'client program'
 * ---------------------------------------------------------------
 */

#include <stdio.h>
#include "fiddle0s.h"

/* ***************************************************************
 * Print the usage string
 */
static void usage (char **argv)
{
    printf ("
    usage: "$ [-tty tty_device] <path_to_target> <arg_list>
    
    Complete pathway to the 'target' program.
    Command line arguments to the 'target' program.
    
    basename (argv[0]));
    exit (1);
}
```
static void check_for_error (tkout_t *tk)
{
    if (tk == NULL) {
        /* Memory error */
        puts ("Service returned NULL (memory error)!");
        exit (1);
    }
    if (tk->err != NULL) {
        /* Error! Print error message and abort */
        printf ("ERROR: %d = %s\n", tk->err->code, tk->err->str);
        exit (1);
    }
    if (tk->rep == NULL && tk->alw == NULL) {
        /* The 'rep' and 'alw' fields are NULL... Abort! */
        puts ("REP and ALW are NULL!");
        exit (1);
    }
    if (tk->status > 0) {
        printf ("ERROR: %s\n", f0s_err_msg (tk->status));
        exit (1);
    }
    /* OK! */
    return;
}

static void process_list (tkout_t *tk)
{
    listdata_t *line;
    chain_node_t *node;
    chain_t *lines = tk->rep->data.r_list.data;
for (node = chain_first (lines);
    node != NULL;
    node = chain_next (lines, node)) {
    line = (listdata_t *) chain_node_data (node);
    printf ("%4d: ", line->line_number);
    if (line->line_text != NULL)
        printf ("%s", line->line_text);
    puts ("\n");
}
f0s_tkout_delete (tk);
}

/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Process the result from 'f0s_break'
 */
static void process_break (tkout_t *tk)
{
    printf ("Breakpoint \%d at address '0x%x', \nline \%d', file \%s\n",
        tk->rep->data.r_break.bp_id,
        (unsigned)tk->rep->data.r_break.address,
        tk->rep->data.r_break.line,
        tk->rep->data.r_break.file);
    f0s_tkout_delete (tk);
}

/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Process the result from 'f0s_run'
 */
static void process_run (tkout_t *tk)
{
    /* Nothing to do! */
    f0s_tkout_delete (tk);
}

/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Process the result from 'f0s_continue'
 */
static void process_continue (tkout_t *tk)
{
    /* Nothing to do! */
    f0s_tkout_delete (tk);
}

/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Process the result from 'f0s_info_stack'
 * (use 'chain_walk()' to iterate on the resulting list)
 */
static int print_frame_line (void *obj, void *args)
framedata_t *f = (framedata_t *)obj;
if (f != NULL) {
    printf ("%d ", f->frame);
    if (f->address != NULL)
        printf ("0x%lx in ", (unsigned) f->address);
    if (f->function != NULL) {
        printf ("%s ", f->function);
        if (f->arguments != NULL)
            printf ("%s", f->arguments);
        printf ("");
    }
    if (f->file != NULL)
        printf (" at %s:%d", f->file, f->line);
    puts ("\n");
}
return 0;

static void process_info_stack (const tkout_t *tk)
{
    if (tk->rep->data.r_infostack.data != NULL)
        chain_walk (tk->rep->data.r_infostack.data,
                    print_frame_line, NULL);
}

/* XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Process the result from 'f0s_kill()
 */
static void process_kill (tkout_t *tk)
{
    /* Nothing to do! */
    f0s_tkout_delete (tk);
}

/* XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Process the result from 'f0s_tty()
 */
static void process_tty (tkout_t *tk)
{
    /* Nothing to do! */
    f0s_tkout_delete (tk);
}

/* XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Process the 'always' field of 'tkout_t'
 * (use 'chain_walk()' to iterate on the 'tk->alw' list)
 */
static int process_always_structure (void *obj, void *args)
{
    alwdata_t *alw = (alwdata_t *)obj;
192 switch (alw->code) {
193 case A0_BPHIT:
194     printf ("Breakpoint hit:
195         Number: %d
196         Function: %s (%s)
197         File: %s
198         Line: %d\n",
199         alw->data.a_bphit.bp_id,
200         alw->data.a_bphit.function,
201         alw->data.a_bphit.args,
202         alw->data.a_bphit.file,
203         alw->data.a_bphit.line);
204     break;
205 default:
206     /* Error! */
207     printf ("ERROR 'process_always_structure()': Invalid \n208         'alw->code'=%%d\n", alw->code);
209     exit (1);
210 }
211 return 0;
212 }

213 static void process_always (tkout_t *tk)
214 {
215     chain_walk (tk->alw, process_always_structure, NULL);
216 }
217
218 /* ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^_^}
puts ("Initializing FIDLE");
puts ("==================================");
f0s_initialize ();

/* Load a 'target' into memory */
puts ("==================================");
printf ("Loading %s' into memory\n", targets[0]);
puts ("==================================");
tk = f0s_file (-1, targets[0]);
check_for_error (tk);
process_always (tk);
tid = process_file (tk);
puts ("\n");

/* If '-tty' option was given, execute 'f0s_tty()' */
if (strcmp (argv[1], "-tty") == 0) {
    puts ("==================================");
    printf ("Setting %s as IO tty\n", argv[2]);
    puts ("==================================");
    tk = f0s_tty (tid, argv[2]);
    check_for_error (tk);
    process_always (tk);
    process_tty (tk);
    puts ("\n");
}

/* List the first 10 lines of 'main()' function */
puts ("==================================");
puts ("Listing the first 10 lines of 'main()' function");
puts ("==================================");
tk = f0s_list (tid, NULL, -1, "main", 10);
check_for_error (tk);
process_always (tk);
process_list (tk);
puts ("\n");

/* Put a break point in line 47 (in the first 'for' loop) */
puts ("==================================");
puts ("Put a break point in line 47");
puts ("==================================");
tk = f0s_break (tid, NULL, 47, NULL);
check_for_error (tk);
process_always (tk);
process_break (tk);
puts ("\n");

/* Put another breakpoint in the 'sfake()' function */
puts ("==================================");
puts ("Put another break point in 'sfake()' function");
puts ("==================================");
tk = f0s_break (tid, NULL, -1, "sfake");
check_for_error (tk);
process_always (tk);
process_break (tk);
puts ("\n");

/* Run the target program with the given arguments */
puts ("=================================================================");
puts ("Running target with the given arguments");
puts ("=================================================================");
tk = f0s_run_v (tid, &targs[1]);
check_for_error (tk);
process_always (tk);
process_run (tk);
puts ("\n");

/* Continue the execution of the target process */
puts ("=================================================================");
puts ("Continuing the execution");
puts ("=================================================================");
tk = f0s_continue (tid);
check_for_error (tk);
process_always (tk);
process_continue (tk);
puts ("\n");

/* Print the stack */
puts ("=================================================================");
puts ("Printing the stack");
puts ("=================================================================");
tk = f0s_info_stack (tid);
check_for_error (tk);
process_always (tk);
process_info_stack (tk);
puts ("\n");

/* Kill the target process */
puts ("=================================================================");
puts ("Killing the target process");
puts ("=================================================================");
tk = f0s_kill (tid);
check_for_error (tk);
process_always (tk);
process_kill (tk);
puts ("\n");

/* Finish */
return (0);
}
4.4 Debugging the Target Program with the Client Program

(must be done...)

5 FIDDLE-0s

In Section 1.3 [FIDDLE Layers], page 2, we have explained briefly the main ideas behind FIDDLE-0s. In the following sections of this Chapter, we’ll look in detail to FIDDLE-0s usage, by presenting and explaining the used data types and supported services functions.

5.1 F0s Data Types

In this and the following sections/chapters, it is assumed that the reader has good knowledge of the auxiliary data type(s) (specially the chain_t type) and associated management functions, defined in the utils library and described in detail in Appendix B [FIDDLE Utilities Library], page 63.

The reading of Appendix B [FIDDLE Utilities Library], page 63, is strongly advised before continuing with this chapter.

There are five main data types necessary to know in detail, in order to use FIDDLE-0s, which are: tkout_t, repdata_t, alwdata_t, posdata_t and errdata_t. They are presented below with the necessary detail.

5.1.1 The tkout_t structure

```
tkout_t
struct tkout_t {
  Data Type
  int tid;
  Data Type
  const char *output;
  errcode0_t status;
  repdata_t *rep;
  chain_t *alw; /* of alwdata_t */
  posdata_t *pos;
  errdata_t *err;
} tkout_t;
```

(Please ignore the line numbers, they are included for further reference to the structure contents).

**Definition**

- The base return type for the FIDDLE-0s service functions.

**Description**

- **int tid** *(in line 2)*
  Contains the **Symbolic Task ID** of the target process.

- **const char *output** *(in line 3)*
  Contains a full copy of the back-end debugger’s output (except the prompt).

- **errcode0_t status** *(in line 4)*
  Indicates the success or failure in the processing of the request according to the following rules:

- **status > FIDDLE_OK**
  The request was not accepted, e.g., due to an invalid set of arguments. In this case, `status` contains the error code, and the corresponding error message can
be obtained by calling f0s_err_msg() with status as the parameter (see Section 5.3.2 [F0s Management Functions], page 32).

status == FIDDLE_OK
The request was accepted and sent to the back-end debugger. The caller will be blocked (in a condition variable) waiting for the reply string. Once this reply string is received, it will be parsed to extract the relevant data to fill the rep, alu, pos and err fields.

repdata_t *rep (in line 5)
Contains the data relative to a successful processing of the service requested. This structure is explained in detail in Section 5.1.3 [The repdata_t structure], page 25.
In case of failure in the processing of the service, tkout_t->rep will point to NULL.

chain_t *alu (in line 6)
Contains the data relative to blocks of text in the reply string that have with a clear and well defined meaning but that are not always present in the output, and one cannot predict when they will be, e.g., the information that a breakpoint was hit.
If no unpredictable output was detected, tkout_t->rep will point to an empty list, i.e., a list with zero nodes.

posdata_t *pos (in line 7)
Contains the data relative to the current location in the source files, e.g., source file name and line number.
If this information is not present in the output, tkout_t->pos will point to NULL.

errdata_t *err (in line 8)
Contains the data relative to the error messages generated by the back-end debugger.
If the service processing was successful, tkout_t->err will point to NULL.

All FIDDLE-0s service functions (except initialization and termination) return a pointer to such a tkout_t structure, whose contents will be filled according to the success/failure of the service requested and to the output received from the back-end debugger.
To fill the contents of the tkout_t structure, memory is allocated dynamically as needed by FIDDLE. A service functions is provided for the user to release the memory used by a tkout_t structure and its fields when no longer needed (see Section 5.3.2 [F0s Management Functions], page 32).

5.1.2 The repcode_t enumerated
All service requests should originate a reply from FIDDLE. This reply is "typified" using the enumerated presented below.
Basically, it contains one entry for each possible reply to a service request. Some entries have "internal" in its comment, meaning that the entry is used internally, and this value should never be returned to the user.

repcode_t
enum repcode_e

Data Type
Data Type
Definition

```c
typedef enum repcode_e {
  R0_UNKNOWN = 00_UNKNOWN, /* For internal use */
  R0_NO_REGEX = 00_NO_REGEX, /* For internal use */
  R0_TIDS,
  R0_KILL,
  R0_ATTACH,
  R0_DETACH,
  R0_SYMBOL_FILE,
  R0_FILE,
  R0_RUN,
  R0_STEP,
  R0NEXT,
  R0_CONTINUE,
  R0_FINISH,
  R0_CALL_VOID,
  R0_BREAK,
  R0_BREAK_NO_SOURCE,
  R0_DELETE,
  R0_EVALUATE,
  R0_EVALUATE_ONE_LINE, /* For internal use */
  R0_EVALUATE_MANY_START, /* For internal use */
  R0_EVALUATE_MANY_MIDDLE, /* For internal use */
  R0_EVALUATE_MANY_END, /* For internal use */
  R0_DISPLAY_EXPRESSION, /* For internal use */
  R0_DISPLAY_ONE_LINE, /* For internal use */
  R0_DISPLAY_MANY_START, /* For internal use */
  R0_DISPLAY_MANY_MIDDLE, /* For internal use */
  R0_DISPLAY_MANY_END, /* For internal use */
  R0_UNDISPLAY,
  R0_SET_VARIABLE,
  R0_CALL, /* Not used! */
  R0_LIST,
  R0TTY,
  R0_INFO_LINE,
  R0_INFO_LINE_NO_CODE,
  R0_INFO_PROGRAM,
  R0_INFO_STACK,
  R0_INFO_BREAK,
  R0_INFO_BREAKPOINT,
  R0_INFO_WATCHPOINT,
  R0_INFO_BREAK_WATCH_DATA,
  R0_INFO_DISPLAY,
  R0_THREAD,
  R0_INFO_THREADS,
  R0_NEW_THREAD, /* For line processing */
  R0_SENTDO,
} repcode_t;
```

5.1.3 The repdata_t structure
repdata_t
struct repdata_s

Definition

1 typedef struct repdata_s {
2   const char *str;  /* The relevant part of the output */
3   repcode_t code;   /* The REPLY code */
4   union {
5      rtids_t r_tids;
6      rkill_t r_kill;
7      rattach_t r_attach;
8      rdetach_t r_detach;
9      rsymbolfile_t r_symbolfile;
10     rfile_t r_file;
11     rrun_t r_run;
12     rstep_t r_step;
13     rnext_t r_next;
14     rcontinue_t r_continue;
15     rfinish_t r_finish;
16     rbreak_t r_break;
17     rdelete_t r_delete;
18     rsetvariable_t r_setvariable;
19     revalue_t r_evaluate;
20     rdisplay_t r_display;
21     rundisplay_t r_undisplay;
22     rtty_t r_tty;
23     rlist_t r_list;
24     rinfoline_t r_infoline;
25     rinfoprogram_t r_infoprogram;
26     rinfostack_t r_infostack;
27     rinfobreak_t r_infobreak;
28     rinfodisplay_t r_infodisplay;
29     rinfothreads_t r_infothreads;
30     rsendto_t r_sendto;
31  } data;
32  } repdata_t;

(Please ignore the line numbers, they are included for further reference to the structure contents).

Description

Contains the data relative to a successful processing of the service requested.

const char *str (in line 2)
Contains a partial copy of tkout_t->output. This is the string that matched the reply pattern and from where the reply data was extracted.

repcode_t code (in line 3)
Contains the code for the service requested. See below the definition of repcode_t.

union {} data (in line 4 to 31)
Contains one structure for each service request supported, where the data extracted from the str field is kept. Each structure is explained in detail close to the associated service request, e.g., the
rbreak_t structure is explained together with the f0s_break() service request.

### 5.1.4 The alwdata_t structure

**Definition**

#### alwdata_t

```c
typedef struct alwdata_s {
    const char *str; /* The relevant part of the output */
    alwcode_t code; /* The ALWAYS code */
    union {
        adisplay_t a_display;
        abphit_t a_bphit;
        asignal_t a_signal;
        anewthread_t a_newthread;
        athread_t a_thread;
    } data;
} alwdata_t;
```

(Please ignore the line numbers, they are included for further reference to the structure contents).

**Description**

Contains the data extracted from tkout_t->output that matched an always pattern. Because more than one always pattern may be detected in tkout_t->output, the field is not of type alwdata_t * but a chain_t *, where each node in the list contain an alwdata_t * pointer.

**const char *str (in line 2)**

Contains a partial copy of tkout_t->output. This is the string that matched the always pattern and from where the always data was extracted.

**alwcode_t code (in line 3)**

Contains the code for the pattern detected. See below the definition of alwcode_t.

**union {} data (in line 4 to 10)**

Contains one structure for each possible always pattern, where the data extracted from tkout_t->output is kept. Each structure is explained in detail close to an associated service request, namely:

- **adisplay_t** Explained together with the f0s_display().
- **abphit_t** Explained together with the f0s_break().
- **asignal_t** Explained together with the f0s_signal().
- **anewthread_t & athread_t** Explained together with the f0s_thread().

### alwcode_t

#### enum alwcode_e

**Definition**
typedef enum alwcode_e {
    A0_UNKNOWN = 00_UNKNOWN,
    A0_NO_REGEX = 00_NO_REGEX,
    A0_DISPLAY,
    A0_BPHIT,
    A0_SIGNAL,
    A0_NEW_THREAD,
    A0_SWITCH_TO_THREAD,
} alwcode_t;

5.1.5 The posdata_t structure

typedef struct pposition_s {
    const char   *file;
    int           line;
    int           pos;
    const char   *where;
    const void   *frame;
} pposition_t;

(Data Type Data Type)

Definition

(Please ignore the line numbers, they are included for further reference to the structure contents).

Description

This structure contains information about the current location in the source code. This information is extracted from tkout_t->output when present. When some of the data is unavailable, the corresponding field will contain -1 (for int typed fields) or NULL (for pointer typed fields).

const char *file (in line 2)
The name of the current source file.

int line (in line 3)
The number of the current line number in file.

int pos (in line 4)
The number of the current character position in file.

const char *where (in line 5)
The current stack frame constants.

const void *frame (in line 6)
The current stack frame address.

typedef struct posdata_s {
    const char   *str;               /* The relevant part of the output
    poscode_t    code;              /* The POSITION code */
} union {

(Data Type Data Type)
5   pposition_t p_position;
6   } data;
7 } posdata_t;

Description
Contains the data extracted from tkout_t->output that matched a position pattern.

const char *str (in line 2)
Contains a partial copy of tkout_t->output. This is the string that matched the position pattern and from where the position data was extracted.

poscode_t code (in line 3)
Contains the code for the pattern detected (always P0_POSITION). See below for definition of poscode_t.

union {} data (in line 4 to 10)
Contains one and only one structure of type pposition_t (described just above).

The union {} could be avoided, resulting in a simplified posdata_t. This solution was adopted to keep all XXXdata_t types with the same basic structure.

poscode_t Data Type
enum poscode_e Data Type

Definition
1 typedef enum poscode_e {
2   P0_UNKNOWN = 00_UNKNOWN,
3   P0_NO_REGEX = 00_NO_REGEX,
4   P0_POSITION,
5 } poscode_t;

5.1.6 The errdata_t structure

errdata_t Data Type
struct errdata_s Data Type

Definition
1 typedef struct errdata_s {
2   const char *str; /* The relevant part of the output */
3   errcode0_t code; /* The ERROR code */
4   union {
5     } data;
6 } errdata_t;

(Please ignore the line numbers, they are included for further reference to the structure contents).

Description
Contains the data extracted from tkout_t->output that matched an error pattern.

const char *str (in line 2)
Contains a partial copy of tkout_t->output. This is the string that matched the error pattern and from where the error data was extracted.
**errcode_t code (in line 3)**
Contains the code for the pattern detected (the error code). See below the definition of errcode_t.

**union {} data (in line 4 to 10)**
Contains one structure for each possible error message, where the data extracted from the str field is kept.

**NOTE:** This extraction of the error data from the error message was not implemented yet. It can be done at any time if needed.

**5.2 F0s Returning Data**
The result of a service call is returned in a tkout_t structure, whose contents were explained in Section 5.1 [F0s Data Types], page 23.

Now, let's consider the following piece of code:

```c
... tkout_t *tk;
... tk = f0s_continue (tid);
...```

In this context, tk will point to a token (tkout_t structure) with the results of the service requested (f0s_continue()):

![Diagram of tkout_t structure](image)

One of the fields of the tkout_t structure—the alw field—is "special", as it contains a double linked list (chain) with each element containing a pointer to a alwdata_t structure, and not the structure itself as the other fields do. This augments the complexity in the processing of the reply, and justifies the following explanation on how to process its contents.

There are two main possibilities to process the contents of the alw list:
1. Iterate through the elements in the list, applying a function to each one.

```c
void process_data_element (alwdata_t *alw, int tid)
{
    switch (alw->code) {
    case A0_DISPLAY:
        /*
         * The 'alw->data.a_display' is a chain where each node
         * points to a 'displaydata_t'. Process 'alw->data' is a
         * similar way to 'tkout_t->alw'
         */
        break;
    case A0_BPHIT:
        /*
         * e.g. print with the contents of 'alw->data.a_bphit'
         */
        break;
    ... } }
}
main ()
{
    chain_node_t *node;
    alwdata_t *data;
    for (node = chain_first (tk->alw);
        node != NULL;
        node = chain_next (tk->alw, node)) {
        data = (alwdata_t *) chain_node_data (node);
        process_data_element (data, tk->tid);
    }
}
```

2. Traverse the list, applying a function to the data in each node.

```c
void process_data_element (alwdata_t *alw, int tid)
{
    switch (alw->code) {
    case A0_DISPLAY:
        /*
         * The 'alw->data.a_display' is a chain where each node
         * points to a 'displaydata_t'. Process 'alw->data' is a
         * similar way to 'tkout_t->alw'
         */
        break;
    case A0_BPHIT:
        /*
         * e.g. print with the contents of 'alw->data.a_bphit'
         */
        break;
    ... } }
```
int always_walker (void *data, void *args)
{
    int tid = *((int *) args);
    alwdat_t *alw = (alwdat_t *)data;
    process_data_element (alw, tid);
    return 0;
}

main ()
{
    ...
    chain_walk (tk->alw, always_walker, &tk->tid);
    ...
}

5.3 F0s Service Functions

The FIDDLE-0s service functions always return a dynamically allocated tkout_t structure, whose memory should be released by the programmer when its contents are no longer needed. If the NULL value is returned, then some kind of memory error occurred, and the programmer should not make any assumptions concerning the success or failure of this service call.

5.3.1 F0s Default Return Values

After a service request, two of the fields in the returned tkout_t structure are always initialized with meaningful values: the output and status fields (see Section 5.1.1 [The tkout_t structure], page 23). The remaining fields are by default initialized with NULL, except the alwdat_t field, which is initialized with an empty chain_t. This is what we call the "Default Return values".

5.3.2 F0s Management Functions

int f0s_initialize (void)    // Function
    Initializes the FIDDLE-0s (Layer-0s) engine.
    Returns 0 on success, and a negative number otherwise.

int f0s_terminate (void)     // Function
    Terminates the FIDDLE-0s (Layer-0s) engine.
    Returns 0 on success, and a negative number otherwise.

tkout_t * f0s_tids (const char *options)     // Function
    FIDDLE will assign a Symbolic Task Identifier to each process under its control. The options field should be set to NULL or the empty string "\".  
    Returns the following data (in the rep->data.r_tids field of the tkout_t structure):

1 The only option currently supported option is "-a" and is used internally for debugging.
typedef struct rtids_data_s { Data Type
    int tid;
    int tp_pid;
    int dbg_pid;
    int attached;
} tidsdata_t;

typedef struct rtids_s { Data Type
    chain_t *data; /* of tidsdata_t */
} rtids_t;

Definition

Each field of the above structures is presented below:

tid The Symbolic Task ID. This is a global identifier.

tp_pid The Target Process ID (PID). This identifier is local to the executing machine.

gdb_pid The Process ID (PID) of the back-end debugger associated with the Target Process. This identifier is local to the executing machine.

attached Flag whose possible values are:

-1 The back-end debugger referred in gdb_pid is free to be reused.

0 The target process was first loaded into the back-end debugger, and just later it was executed with fos_run(). On FIDDLE-0s termination this process will be terminated.

1 The control over the target process was acquired through the attachment of a the back-end debugger, using fos_attach(). On FIDDLE-0s termination the back-end debugger will be detached and the process will be left to run free.

data A linked list, where each node contains one tidsdata_t element.

int fos_fkout_delete (tkout_t *tk) Function

Releases the memory associated with tk. The fields of tk that point to dynamically allocated memory are released before tk.

Returns 0 on success, and a negative number otherwise.

char *fos_err_msg (errcode0_t errnum) Function

Gets the error message associated with errnum.

Returns a pointer to the error message on success, or NULL on failure.
5.3.3 F0s Process Control Functions

tkout_t * f0sAttach (int tp_pid)  
Function  
Finds a free back-end debugger (by reusing one in the holding state or starting a new one if needed) and attach it to the process with PID tp_pid.

ATTENTION: The process I/O will use the same terminal it was using before the attachment.

Returns the following data (in the rep->data.r_attach field of the tkout_t structure):

rattach_t  
struct rattach_s  

Definition

1 typedef struct rattach_s {  
2 int tid;
3 int tp_pid;
4 } rattach_t;

Description
Each field of the above structure is presented below:

tid  Contains the symbolic task ID of the newly acquired process. This is a global identifier.

tp_pid  Contains the Target Process ID (PID). This identifier is local to the executing machine and should be the same as the one given as argument to f0sAttach().

tkout_t * f0sDetach (int tid)  
Function  
Detaches the back-end debugger from the process. The process is left to run uncontrolled as it was before the attachment, and the back-end debugger is released.

Returns the following data (in the rep->data.r_detach field of the tkout_t structure):

rdetach_t  
struct rdetach_s  

Definition

1 typedef struct rdetach_s {  
2 int tp_pid;
3 } rdetach_t;

Description
Each field of the above structure is presented below:

tp_pid  Contains the Target Process ID (PID). This identifier is local to the executing machine.

tkout_t * f0sKill (int tid)  
Function  
Kills the target process associated with the symbolic task ID tid.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

---

2 As a performance optimization, the implementation keeps a pool of free debuggers which will be used in the next f0sAttach() or f0s_file() service requests.
tkout_t * f0s_symbol_file (int tid, const char *file) Function
Loads the symbols table for the task tid from file.
Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

tkout_t * f0s_file (const char *file) Function
Finds a free the back-end debugger (one in the holding state or starting a new one if needed) and load the executable file into memory. The symbols table is also loaded from file.
Returns the following data (in the rep->data.r_file field of the tkout_t structure):

rfile_t Data Type
struct rfile_s Data Type

Definition
1 typedef struct rfile_s {
2 int tid;
3 const char *file;
4 } rfile_t;

Description
Each field of the above structure is presented below:
tid Contains the symbolic task ID of the program that was loaded into memory. This is a global identifier. Note that the program is not running yet. To start running the program call f0s_run() (see below).

tkout_t * f0s_run (int tid) Function
Starts the execution of a program previously loaded with f0s_file() (see above). In general, it is a good idea to set a breakpoint somewhere in the program, e.g., in the beginning of the main() function, before running it.
The target process will stop when a breakpoint is hit (if any), if a signal is received or if the program terminates.
Returns the following data (in the rep->data.r_run field of the tkout_t structure):

rrun_t Data Type
struct rrun_s Data Type

Definition
1 typedef struct rrun_s {
2 const char *file;
3 } rrun_t;

Description
Each field of the above structure is presented below:
file This string contains the name of the file from where symbols were loaded.

tkout_t * f0s_step (int tid) Function
Continues the execution of tid until next instruction is reached. Steps into the called functions.
The target process will stop when the next instruction is reached, or before if a breakpoint is hit (if any), if a signal is received or if the program terminates.
Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).
tkout_t * f0s_next (int tid)  
Function  
Continues the execution of tid until next instruction is reached. Steps over the called functions.  
The target process will stop when the next instruction is reached, or before if a breakpoint is hit, if a signal is received or if the program terminates.  
Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

tkout_t * f0s_continue (int tid)  
Function  
Continues the execution of tid until it stops or terminates. The process can stop due to many reasons, such as by reaching a breakpoint or receiving a signal, and terminates when reaches the end of the main function.  
The target process will stop if a breakpoint is hit, if a signal is received or if the program terminates.  
Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

tkout_t * f0s_finish (int tid)  
Function  
Continues the execution of tid until the function associated with the current stack frame returns. The value returned by this function is also extracted and sent to the user.  
The target process will stop just before returning the the upper stack frame, or before if a breakpoint is hit, if a signal is received or if the program terminates.  
Returns the following data (in the rep->data.r_finish field of the tkout_rfinish structure):

rfinish_t  
struct rfinish_s  
Definition  
1  typedef struct rfinish_s {  
2    const void *s_address;  
3    const char *s_function;  
4    const char *s_args;  
5    const char *s_file;  
6    int s_line;  
7    int gdb_var;  
8    const char *value;  
9  } rfinish_t;  

Description  
Each field of the above structure is presented below:  
s_address  
This is the stack frame address.  
s_function  
The name of the function just executed.  
s_args  
The arguments passed when the function was called.  
s_file  
This string contains the name of the source file containing the function.  
s_line  
The line number.
gdb_var The back-end debugger variable number where the return value of the function called was stored. All values output by the back-end debugger are stored in variables with names 'gdb$N' where 'N' is an integer starting at 1 and incremented once for each new value.

value The return value of the function called.

tkout_t * f0s_signal (int tid, int signum) Function
Sends signal signum to process tid. If the process is stopped, it will receive the signal when the execution is resumed, otherwise the process execution is stopped.

All the signals except SIGINT will be passed to the program. The SIGINT signal will just interrupt the program execution without further consequences.

The information about the signal received will be returned in the alw field of the tkout_t structure. See Section 5.2 [F0s Returning Data], page 30.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

Every time the process execution is interrupted by the reception of a signal, the following data is returned in the alw->data.a_signal field of the tkout_t structure:

asignal_t Data Type
struct asignal_s Data Type

Definition

1 typedef struct asignal_s {
2 const char *name;
3 } asignal_t;

Description
Each field of the above structures is presented below:

name The string that identifies the signal received, e.g., "SIGINT".

tkout_t * f0s_break (int tid, const char *file, int line, const char *function) Function
Sets a breakpoint in line or in function of process tid. These two arguments are exclusive, so when calling this function, or line == -1 or function == NULL. If file is omitted (using file == NULL) the current source file is considered as the default. If function != NULL and there is only one function with such name in the program, the file which contains the source of the function is selected automatically.

Returns the following data (in the rep->data.r_break field of the tkout_t structure):

rbreak_t Data Type
struct rbreak_s Data Type

Definition

1 typedef struct rbreak_s {
2 int bpid;
3 const void *address;
4 const char *file;
5 int line;
6 } rbreak_t;

Description
Each field of the above structure is presented below:
bpid  The Breakpoint Identifier. This identifier is local to the process.
address The memory address where the breakpoint was set.
file The name of the source file which contains the line where the breakpoint was set.
line The source line where the breakpoint was set. This line number is relative to the source file above.

The information about the breakpoint that was hit will be returned in the alw field of the tkout_t structure. See Section 5.2 [F0s Returning Data], page 30.

Whenever a breakpoint is hit by the target process, one node in the list alw of the tkout_t structure will contain information about the breakpoint hit in the data.a_bphit field.

The a_bphit field has the type abphit_t, which is described below:

```
struct abphit_s
{
    int    bpid;
    const char  *function;
    const char  *args;
    const char  *file;
    int    line;
} abphit_t;
```

Each field of the above structures is presented below:

bpid  The Breakpoint Identifier. This identifier is local to the process.
function The name of the function where the program execution stopped, i.e., which contains the breakpoint.
file The name of the source file associated with the code location where the execution stopped.
line The line number associated with the code location where the execution stopped.

tkout_t * f0s_delete (int tid, int bpid)  Function

Deletes breakpoint bpid of process tid.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

tkout_t * f0s_set_variable (int tid, const char *variable, const char *expression)  Function

Sets the value of variable to the result of the evaluation of expression in the current context of tid. The variable must be accessible in the current context of tid.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

tkout_t * f0s_call (int tid, const char *expression)  Function

Evaluates expression, where expression is the name of a function and its arguments.

Returns the same as the f0s_evaluate() service request (see below).
5.3.4 F0s Process Inspection Functions

`tkout_t * f0s_evaluate (int tid, const char *expression)` Function

Evaluates `expression` in the current context of `tid`.

Returns the following data (in the `rep->data.r_evaluate` field of the `tkout_t` structure):

1. `typedef struct reevaluate_s {
   2   int gdb_var;
   3   const char *value;
   4 } reevaluate_t;

Each field of the above structures is presented below:

`gdb_var` The back-end debugger variable number where the return value of the function called was stored. All values output by the back-end debugger are stored in variables with names ‘$N’ where ‘N’ is an integer starting at 1 and incremented once for each new value.

`value` This string contains the result of the evaluation of `expression`. It is up to the user to parse the string and identify its components.

`tkout_t * f0s_display (int tid, const char *expression)` Function

Evaluates `expression` in the current context of `tid` every time the process stops running.

The expression to be evaluated and the result of its evaluation will be returned in the `alw` field of the `tkout_t` structure. See Section 5.2 [F0s Returning Data], page 30.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

`tkout_t * f0s_undisplay (int tid, int dispid)` Function

Removes the expression associated with `dispid` from the list of expressions to be evaluated every time the process stops running.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

`tkout_t * f0s_list (int tid, const char *file, int start_line, const char *function, int n_lines)` Function

Lists `n_lines` of a source file of `tid`, starting in `line` or in `function`. These two arguments are exclusive, so when calling this function, or `line == -1` or `function == NULL`. If `file` is omitted (using `file == NULL`) the current source file is considered as the default. If `function != NULL` and there is only one function with such name in the program, the file which contains the source of the function is selected automatically.

Returns the following data (in the `rep->data.r_list` field of the `tkout_t` structure):

1. `typedef struct rlist_data_s {
   2   int line_number;
   3   const char *line_text;
   4 } listdata_t;
2. `typedef struct rlist_s {
   5   chain_t *data;      /* of listdata_t */
   6   rlist_t;
   7 } rlist_t;

Each field of the above structures is presented below:

`line_number` The line number.
line_text

The text for the line.

data

A linked list where each node contains one line (listdata_t element) of a source file.

tkout_t * f0s_info_line (int tid, const char *file, int line, const char *function)

Function

Gets information about a source line of tid. The line can be specified by its number line or by a function name. These two arguments are exclusive, so when calling this function, or line == -1 or function == NULL. If file is omitted (using file == NULL) the current source file is considered as the default. If function != NULL and there is only one function with such name in the program, the file which contains the source of the function is selected automatically.

Returns the following data (in the rep->data.r_infoline field of the tkout_t structure):

1  typedef struct rinfo_line_s {
2   int     line;
3   const char  *file;
4   const char  *start_function;
5   int       start_function_offset;
6   const char  *end_function;
7   int       end_function_offset;
8   const void *start_address;
9   const void *end_address;
10  } rinfo_line_t;

Each field of the above structure is presented below:

line

The line number.

file

The name of the file containing line.

start_function

The name of the function which includes line.

start_function_offset

The memory offset of the beginning of line’s code relative to the beginning of the function’s code.

dead_function

The name of the next function following line.

dead_function_offset

The memory offset of the beginning of line’s code relative to the beginning of the next function’s code.

start_address

The memory address of the beginning of line’s code.

dead_address

The memory address of the end of line’s code.

tkout_t * f0s_info_program (int tid)

Function

Gets information about the program tid.

Returns the following data (in the rep->data.r_infoprogram field of the tkout_t structure):
typedef struct rinfo_program_s {
int       tp_pid;
const void *address;
const char *why;
int        bpid;
const char *signal;
} rinfo_program_t;

Each field of the above structure is presented below:

**tp_pid**  
Contains the Target Process ID (PID). This identifier is local to the executing machine.

**address**  
The current stack frame address.

**why**  
This string describes why the process did stop.

**bpid**  
If the program stopped because a breakpoint was reached, then this field contains the breakpoint ID, otherwise it contains -1.

**signal**  
If the program stopped because it received a signal, then this field contains the name of that signal, otherwise it contains NULL.

Function

```
ltkout_t * f0s_info_stack (int tid)
```

Gets information about the stack of *tid* (*backtrace*).

Returns the following data (in the rep->data.r_infostack field of the tkout_t structure):

```
typedef struct rframedata_s {
int       frame;
const void *address;
const char *function;
const char *arguments;
const char *file;
int        line;
} framedata_t;
typedef struct rinfostack_s {
chain_t   *data;          /* of framedata_t */
} rinfostack_t;
```

Each field of the above structures is presented below:

**frame**  
The frame number.

**address**  
The stack frame address.

**function**  
The function associated with *frame*.

**arguments**  
The arguments of the *function* associated with *frame*.

**file**  
The source file for *function*.

**line**  
The line number in *file* for *function*.

**data**  
This is chain (linked list) where each node contains one framedata_t element.

Function

```
ltkout_t * f0s_up (int tid)
```

Changes the current stack frame of *tid* one layer up (closer to the *main()* function.

Returns exactly the same as the f0s_info_stack() command (see above).
Note that in this command, the linked list will contain just one element with the data for the new selected stack frame.

**tkout_t * f0s_down (int tid)**

Function

Changes the current stack frame of *tid* one layer down (closer to the inner frame).

Returns exactly the same as the f0s_info_stack() command (see above).

Note that in this command, the linked list will contain just one element with the data for the new selected stack frame.

**tkout_t * f0s_frame (int tid, int frame)**

Function

Changes the current stack frame of *tid* to frame number *frame*.

Returns exactly the same as the f0s_info_stack() command (see above).

Note that in this command, the linked list will contain just one element with the data for the new selected stack frame.

**tkout_t * f0s_info_break (int tid)**

Function

Gets information about the current breakpoints in *tid*.

Returns the following data (in the rep->data.r_infobreak field of the tkout_t structure):

```c
1  typedef struct rbreak_data_s {
2    int        bpid;
3    const char *type;
4    const char *disp;
5    int        enabled;
6    const void *address;
7    const char *function;
8    const char *file;
9    int        line;
10   int        hit_counter;
11   int        ign_counter;
12   const char *condition;
13  } breakdata_t;
14  typedef struct rinfobreak_s {
15    chain_t     *data;    /* of breakdata_t */
16  } rinfobreak_t;
```

Each field of the above structures is presented below:

- **bpid** The breakpoint ID.
- **type** Ignore this field. It will be used in future versions of FIDDLE.
- **disp** Ignore this field. It will be used in future versions of FIDDLE.
- **enabled** Ignore this field. It will be used in future versions of FIDDLE.
- **address** The memory address where the breakpoint is placed.
- **function** The function where the breakpoint is placed.
- **file** The source file where the breakpoint is placed.
- **line** The line number in file where the breakpoint is placed.
hit_counter
The number of hits in this breakpoint, i.e., how many times did the program execution reached the breakpoint.

ing_counter
Ignore this field. It will be used in future versions of FIDDLE.

condition
Ignore this field. It will be used in future versions of FIDDLE.

data
This is chain (linked list) where each node contains one breakdata_t element.

\texttt{tkout_t * f0s_info_display (int tid)} 
Function
Gets information about the current display expressions in \textit{tid}.

Returns the following data (in the \texttt{rep->data.r_infodisplay} field of the \texttt{tkout_t} structure):

\begin{verbatim}
1 typedef struct rinfodisplay_s {
2     chain_t   *data;    /* of displaydata_t */
3 } rinfodisplay_t;
\end{verbatim}

Each field of the above structures is presented below:

dispid
The \textit{Display Identifier}. This identifier is local to the process.

enabled
Ignore this field. It will be used in future versions of FIDDLE.

expression
This string contains the expression to be evaluated.

value
This fields is not used in this command (should be NULL).

data
A linked list where each node contains one displaydata_t element.

5.3.5 F0s Thread Functions

\texttt{tkout_t * f0s_thread (int tid, int threadID)} 
Function
Selects thread \textit{threadID} of \textit{tid} as the target for the next debugging commands.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

\texttt{tkout_t * f0s_info_threads (int tid)} 
Function
Gets information about all the threads in \textit{tid}.

Returns the following data (in the \texttt{rep->data.r_infothreads} field of the \texttt{tkout_t} structure):

\begin{verbatim}
1 typedef struct rinfothreads_data_s {
2     int          active;
3     int          th_id;
4     int          tp_pid;
5     const void   *frame;
6     const char   *function;
7     const char   *arguments;
8     const char   *file;
9     int          line;
10 } infothreadsdata_t;
11 typedef struct rinfothreads_s {
\end{verbatim}
12    chain_t  *data;    /* of infothreadsdata_t */
13 } rinfothreads_t;

Each field of the above structures is presented below:

active    Flag indicating if the thread is active.
th_id    The thread ID. This identifier is local to the process.
tp_pid    The process ID (PID). This identifier is local to the executing machine.
frame    The stack frame address.
function    The function associated with frame.
arguments    The arguments of the function associated with frame.
file    The source file for function.
line    The line number in file for function.
data    This is chain (linked list) where each node contains one infothreadsdata_t element.

5.3.6 F0s Miscellaneous Functions

tkout_t * f0s_tty (int tid, const char *tty_dev)    Function
Sets the standard I/O tty to tty_dev for the future runs of the program being debugged.
A process running under FIDDLE control may have normal standard I/O operations if it was already running before, and control was aquired by attaching FIDDLE to it.

If the process was loaded into memory by FIDDLE, e.g., using the f0m_file() service, it shouldn’t do any standard I/O operations, as its output will be lost and its input will conflict with the back-end debugger commands. To avoid this problem, before running the program call this service, specifying a new tty for the standard I/O operations.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

tkout_t * f0s_sendto (int tid, const char *command)    Function
Sends a command directly to the back-end debugger associated with tid and get its plain reply (not parsed).

This command is useful if one needs to send a command to the back-end debugger that is not supported (yet) by the FIDDLE-0s library.

Returns the default values (see Section 5.3.1 [F0s Default Return Values], page 32).

Remember that the back-end debugger output is always available in the tkout_t structure.
6 FIDDLE-0m

In Section 1.3 [FIDDLE Layers], page 2, we have explained briefly the main ideas behind FIDDLE-0m. In the following sections of this Chapter, we'll look in detail to FIDDLE-0m usage, by presenting and explaining the used data types and supported services functions.

In many aspects, FIDDLE-0m and FIDDLE-0s services are very similar, and you are asked to report to the equivalent services or data types in FIDDLE-0s for further information.

6.1 F0m Data Types

FIDDLE-0m uses the same data structures as FIDDLE-0s, such as tkout_t and repdata_t. Please read Section 5.1 [F0s Data Types], page 23 to know more about the data types used in this layer.

6.2 F0m Service Functions

In what concerns to the syntax, FIDDLE-0m services have exactly the same names and arguments as FIDDLE-0s services, but the new service prefix is 'f0m_' instead of 'f0s_'.

In what concerns to the semantics, FIDDLE-0m and FIDDLE-0s services are also similar: the caller (and a thread in FIDDLE-0m and a process in FIDDLE-0s) remains blocked until the service is completed.

Concurrent FIDDLE-0m requests (issued by different threads) to different target processes are processed concurrently. Concurrent requests to the same target processes are serialized and executed one after another.

Please refer to the equivalent service in FIDDLE-0s for details about the syntax and semantics of FIDDLE-0m service requests.

6.2.1 F0m Management Functions

```c
int f0m_initialize (void)  // Function
int f0m_terminate (void)   // Function
tkout_t * f0m_tids (const char *options)  // Function
```

These services are used in the same way as the equivalent ones from FIDDLE-0s. See Section 5.3.2 [F0s Management Functions], page 32.

6.2.2 F0m Process Control Functions
tkout_t * f0m_kill (int tid)  
Function

Function

Function

Function

Function

Function

Function

Function

Function

Function

Function

Function

Function

These services are used in the same way as the equivalent ones from FIDDLE-0s. See Section 5.3.3 [F0s Process Control Functions], page 34.

6.2.3 F0m Process Inspection Functions

Function

Function

Function

Function

Function

Function

Function

Function

Function

Function

These services are used in the same way as the equivalent ones from FIDDLE-0s. See Section 5.3.4 [F0s Process Inspection Functions], page 39.

6.2.4 F0m Thread Functions

Function

Function

These services are used in the same way as the equivalent ones from FIDDLE-0s. See Section 5.3.5 [F0s Thread Functions], page 43.

6.2.5 F0m Miscellaneous Functions

Function

Function

These services are used in the same way as the equivalent ones from FIDDLE-0s. See Section 5.3.6 [F0s Miscellaneous Functions], page 44.
tkout_t * f0m_tkin (int tid, tkin_t *tk)
Send the request already packed into a tkin_t structure to process tid\(^1\).
It is up to the user to release the memory used by tk if needed.

\(^1\) This service is mainly for internal use, to support the FIDDLE-1m layer.
7 FIDDLE-1m

In Section 1.3 [FIDDLE Layers], page 2, we have explained briefly the main ideas behind FIDDLE-1m. In the following sections of this Chapter, we'll look in detail to FIDDLE-1m usage, by presenting and explaining the used data types and supported services functions.

In many aspects, FIDDLE-1m and FIDDLE-0m services are very similar, and you are asked to report to the equivalent services or data types in FIDDLE-0m for further information.

7.1 F1m Data Types

FIDDLE-1m uses the same data structures as FIDDLE-0m, such as tkout_t and repdata_t. Please read Section 6.1 [F0m Data Types], page 45 to know more about the data types used in this layer.

7.2 F1m Service Functions

In what concerns to the syntax, the majority of the FIDDLE-1m services have exactly the same names and arguments as FIDDLE-0m services, but the new service prefix is ‘f1m_’ instead of ‘f0m_’.

In what concerns to the semantics, FIDDLE-1m and FIDDLE-0m services are also similar. Whenever needed, please refer to the equivalent services in FIDDLE-0m for details about the syntax and semantics of FIDDLE-1m service requests.

7.2.1 F1m Management Functions

int f0m_initialize (void)
int f0m_terminate (void)
tkout_t * f0m_tids (const char *options)

These services are used in the same way as the equivalent ones from FIDDLE-1m. See Section 6.2.1 [F0m Management Functions], page 45.

7.2.2 F1m Process Control Functions

tkout_t * f1m_attach (const char *machine, int tp_pid)
tkout_t * f1m_detach (int tid)
tkout_t * f1m_kill (int tid)
tkout_t * f1m_symbol_file (int tid, const char *file)
tkout_t * f1m_file (const char *machine, const char *file)

Attach FIDDLE to the process tp_pid in machine
These services are used in the same way as the equivalent ones from FIDDLE-1m. See Section 6.2.2 [F0m Process Control Functions], page 45.

These services are used in the same way as the equivalent ones from FIDDLE-1m. See Section 6.2.2 [F0m Process Control Functions], page 45.

Load the executable file into memory of machine. file is local to machine.
Except for the extra argument (machine), the syntax and semantics of this service is equal to the f0m_file() service. More information can be found in section [No value for “F0m Process Control Functions”].
tkout_t * flm_run (int tid)                     Function
tkout_t * flm_step (int tid)                   Function
tkout_t * flm_next (int tid)                   Function
tkout_t * flm_continue (int tid)               Function
tkout_t * flm_finish (int tid)                 Function
tkout_t * flm_signal (int tid, int signum)     Function
tkout_t * flm_break (int tid, const char *file, int line, const char *function)
 tkout_t * flm_delete (int tid, int bpid)      Function
 tkout_t * flm_set_variable (int tid, const char *variable, const char *expression)
 tkout_t * flm_call (int tid, const char *expression)  Function

These services are used in the same way as the equivalent ones from FIDDLE-0m. See Section 6.2.2 [F0m Process Control Functions], page 45.

7.2.3 F1m Process Inspection Functions

tkout_t * flm_evaluate (int tid, const char *expression)  Function
tkout_t * flm_display (int tid, const char *expression)   Function
tkout_t * flm_undisplay (int tid, int dispid)             Function
tkout_t * flm_list (int tid, const char *file, int start_line, const char *function, int nlines)
 tkout_t * flm_info_line (int tid, const char *file, int line, const char *function)  Function
 tkout_t * flm_info_program (int tid)                    Function
 tkout_t * flm_info_stack (int tid)                      Function
 tkout_t * flm_up (int tid)                             Function
 tkout_t * flm_down (int tid)                            Function
 tkout_t * flm_frame (int tid, int frame)                Function
 tkout_t * flm_info_break (int tid)                      Function
 tkout_t * flm_info_display (int tid)                    Function

These services are used in the same way as the equivalent ones from FIDDLE-1m. See Section 6.2.3 [F0m Process Inspection Functions], page 46.

7.2.4 F1m Thread Functions

tkout_t * flm_thread (int tid, int threadID)             Function
 tkout_t * flm_info_threads (int tid)                    Function

These services are used in the same way as the equivalent ones from FIDDLE-1m. See Section 6.2.4 [F0m Thread Functions], page 46.

7.2.5 F1m Miscellaneous Functions

tkout_t * flm_tty (int tid, const char *tty_dev)         Function
 tkout_t * flm_sendto (int tid, const char *command)     Function

These services are used in the same way as the equivalent ones from FIDDLE-1m. See Section 6.2.5 [F0m Miscellaneous Functions], page 46.

tkout_t * flm_tkin (int tid, tkin_t *tk)                 Function

Send the request already packed into a tkin_t structure to process tid

---

1 This service is mainly for internal use, to support the FIDDLE-1m layer.
It is up to the user to release the memory used by \texttt{tk} if needed.
8 FIDDLE-2m

This is not done yet!!!
9 FIDDLE-3m

This is not done yet!!!
10 Unsolved Problems

In this Chapter we will list the (still) unsolved problems. If you have suggestions on how to solve any of them, please let us know. You can use the bug report address(es) for this (see Chapter 3 [Reporting Bugs], page 11).

10.1 Intercepting fork()

How to intercept the creation of a new Unix process during fork(), in order to have the child process executed under control of a back-end debugger too?

If this is achieved, we could automatically add the new processes to the FIDDLER environment and warn the user of these new processes, in a way similar to what is already being done with the new threads (in the case of new threads, the back-end debugger generates this warning).

10.2 I/O of remote processes

A simple solution to redirect the standard I/O of remote processes to a tty in the local machine.
11 Acknowledgements

We would like to present out thanks to:

João Seco (DI - FCT/UNL)
Nuno Frequiça (DI - FCT/UNL)

They accepted to read and comment an early draft of this document.

Alexandre Carissimi (Universidade Federal do Rio Grande do Sul, Brazil)

He read very carefully the version 0.2.7 of this document, and suggested many and valuable improvements to its contents.

Denise Stringhini (Universidade Federal do Rio Grande do Sul, Brazil)

She is using FIDDLE in her research work, and is a constant source of questions, comments and suggestions to the improvement of this manual and FIDDLE itself.
Appendix A  FIDDLE Related Publications

[CLA98]  
José C. Cunha, João Lourenço, and Vitor Duarte.  
*Using DDBG to support testing and high-level debugging Interfaces.*  

[CLA98]  
José C. Cunha, João Lourenço, and Tiago Antão.  
*An experiment in tool integration: the DDBG parallel and distributed debugger.*  
Elsevier Science Press.

[KCD+97]  
*A graphical development and debugging environment for parallel programs.*  
Elsevier Science Press.

[LC98b]  
João Lourenço, and José C. Cunha.  
*Replaying Distributed Applications with RPVM.*  
Report Series of the Institute of Applied Computer Science and Information Systems, University of Vienna.

[LC98a]  
João Lourenço, and José C. Cunha.  
*The PDBG process-level debugger for parallel and distributed programs [Poster].*  
In Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT’98), Welches, Oregon, USA, August 1998.  
ACM Press.

[LC98]  
J. Lourenço, and J. Cunha.  
*A thread-level distributed debugger.*  

[CLV+98]  
*A framework to support parallel and distributed debugging.*  
Lecture Notes on Computer Science, Elsevier Science Press.

[CML+98]  
*The DOTPAR project: Towards a framework supporting domain oriented tools for parallel and distributed processing.*

[CLD98]
José C. Cunha, João Lourenço, and Vitor Duarte.
An experiment in tool integration: the DDBG parallel and distributed debugger.

[LCK+97]
An integrated testing and debugging environment for parallel and distributed programs.
IEEE Computer Society Press.

[CLA96b]
A distributed debugging tool for a parallel software engineering environment.
In Proceedings of the 1st European Parallel Tools Meeting (EPTM’96), Châtillon, France, October 1996.
Edited by ONERA (French National Establishment for Aerospace Research).

[CLA96a]
A debugging engine for a parallel and distributed environment.
Edited by Hungarian Academy of Sciences, KFKI.
Appendix B FIDDLE Utilities Library

The fiddleutil library implements a set of functions to manage *double linked lists*. These functions are reentrant and protected by *mutexes* and *condition variables* when needed, to allow concurrent access to the same list by different threads.

As this library uses services and data typed defined in the *pthreads* library, if you use this library you have to include always the *pthreads* library, even if your program is single-threaded.

B.1 Chain types

**chain_t**

Lirabry Data Type (opaque)

*Definition*

This is an opaque type, and so its internal structure is not published.

*Description*

This is the base type for the double linked list.

**chain_node_t**

Lirabry Data Type (opaque)

*Definition*

This is an opaque type, and so its internal structure is not published.

*Description*

This is the base type for the nodes kept in the list.

**chain_collect_t**

Lirabry Data Type (opaque)

*Definition*

This is an opaque type, and so its internal structure is not published.

*Description*

This is the base type for the nodes kept in the list created by *chain_walk_collect()* (see Section B.2 [Chain functions], page 64).

**chain_walk_f**

Lirabry Data Type

*Definition*

```c
typedef int (*chain_walk_f)(void *data, void *args);
```

*Description*

Some functions, such as *chain_walk()* (see Section B.2 [Chain functions], page 64), traverse the chain, applying a function (the *walker*) to each node.

The function that is applied should be of this type.

If *walker* returns a negative number the traversing stops.

**chain_walk_collect_f**

Lirabry Data Type

*Definition*

```c
typedef void (*chain_walk_collect_f)(void *data, void *args);
```

*Description*

The function *chain_walk_collect()* (see Section B.2 [Chain functions], page 64), traverse the chain, applying a function (the *walker*) to each node.

The function that is applied should be of this type.

If *walker* returns NULL the traversing stops.
chain_cmp_f

Definition

typedef int (*chain_cmp_f) (const void *data,
   const void *what, void *args);

Description
Some functions, such as chain_find() (see Section B.2 [Chain functions], page 64), traverse the chain looking for a node whose data compares successfully with something (what), returning the comparing functions 0 (zero) in this situation. The function that is used for this test should be of this type.

If walker returns a negative number the traversing stops.

B.2 Chain functions

chain_t * chain_create (void)
   Creates a new empty chain.
   Returns a pointer to an opaque typed (chain_t *) chain, or NULL on failure.

void chain_destroy (chain_t *chain)
   Deletes chain. Assumes the data in the chain nodes was already cleaned or is accessible through some other data structure.
   Returns nothing (void).

int chain_length (const chain_t *chain);
   Returns the number of nodes in chain.

chain_node_t * chain_first (const chain_t *chain)
   Returns a pointer to the first node in chain, or NULL if chain is empty.

chain_node_t * chain_last (const chain_t *chain)
   Returns a pointer to the last node in chain, or NULL if chain is empty.

chain_node_t * chain_next (const chain_t *chain, const
   chain_node_t *node)
   Returns a pointer to the next node in chain, or NULL if node is the last one.

chain_node_t * chain_prev (const chain_t *chain, const
   chain_node_t *node)
   Returns a pointer to the previous node in chain, or NULL if node is the first one.

chain_node_t * chain_enqueue (chain_t *chain, void *data)
   Creates a new node to keep data and insert it at the tail of chain.
   Returns a pointer to the newly created node, or NULL on failure.

void * chain_dequeue (chain_t *chain)
   Removes (deletes) a node from chain. The data kept in the node is not deleted.
   Returns a pointer to the data in the (deleted) node, or NULL if chain is empty.
void * chain_dequeue_cond (chain_t *chain, chain_walk_f cond, void *args)  
Library Function
Blocks the calling thread until there is a node in chain for which the evaluation of
cond(args) returns negative, then remove this node. The data kept in the node is not
deleted.
Returns a pointer to the data in the (deleted) node.

void * chain_remove (chain_t *chain, chain_node_t *node)  
Library Function
Removes node from chain. The data kept in the node is not deleted.
Returns a pointer to the data in node.

chain_node_t * chain_prune (chain_t *chain, chain_walker_f pruner, void *args)
Remove all nodes from chain for which pruner(args) returns positive, leaving the node
untouched if pruner(args) returns 0 (zero). Stops the pruning if pruner(args) returns
negative. If the data in the node was dynamically allocated, the pruner() may release
that memory before returning.
Returns a pointer to the node where the pruning stopped, or NULL if all the chain was
traversed.

chain_t * chain_merge (chain_t *chain1, chain_t *chain2)  
Library Function
Append chain2 to chain1.
Returns a pointer to chain1.

chain_node_t * chain_find (chain_t *chain, chain_cmp_f compare, void *args)
Library Function
Find the first node in chain for which compare(args) returns 0 (zero).
Returns a pointer to the node or NULL if none was found.

chain_node_t * chain_walk (chain_t *chain, chain_walker_f walker, void *args)
Library Function
Traverses chain from head to tail (chain_walk()) or from tail to head (chain_rwalk()),
applying walker(args) to each node. Stops the traversing if walker(args) returns a
negative value.
Returns a pointer to the node where the traversing stopped, i.e., to the last node to which
walker was applied, or NULL if the all list was traversed.

chain_t * chain_walk_collect (chain_t *chain,
    chain_walk_collect_f collector, void *args)
Library Function
Traverses chain from head to tail applying walker(args) to each node and collecting the
return values of walker(args) into a new chain. Stops the traversing if walker(args)
returns NULL.
Returns a pointer to the newly created chain with the returning values from walker(args)
(this new chain has the same number of nodes as chain), or NULL on failure.

void * chain_collect_origin (const chain_collect_t *node)
Library Function
Returns a pointer to which walker(args) (see chain_walk_collect() above) was ap-
plied.
void * chain_collect_data (const chain_collect_t *node) Library Function
Returns the value returned by walker(args) (see chain_walk_collect() above) when
it was applied to the origin (see chain_collect_origin() above).
Appendix C Glossary

**Back-end application**
An application (or library) that provides a class of services, whose functionalities can be accessed by other applications (the *front-end tools*).

**Back-end debugger**
When a debugger is used as a *back-end application*.
FIDDL uses traditional command-line debuggers (such as GDB) as its back-end debuggers. If this *back-end debugger* supports multi-threaded processes, then FIDDL will also control/debug the multiple threads in the same *target process*.

**Client**
An application which is relying on FIDDL to provide debugging services. This will be a debugging *front-end application*.

**Component-level debugger**
The same as *back-end debugger*.

**Debugee**
The same as *target process*.

**Front-end application**
An application that masks another (*back-end*) application, e.g., XXGDB is a *front-end application* to the (*back-end debugger*) GNU GDB.
FIDDL clients will also act be debugging *front-ends application*, as they will rely in FIDDL debugging engine as its *back-end application*.

**Low-level debugger**
The same as *back-end debugger*.

**Target application**
The application being debugged, where FIDDL is controlling at least one of its processes (see *target process*).

**Target process**
A process being debugged under FIDDL control. This process is part of the *target application*.

**Token**
Data extracted from the back-end debugger output and packed into a *tkout_t* structure.
Appendix D  f0s_tkout.h

1 /*******************************
2 *
3 * $Id$
4 *
5 * Copyright (c) 1999 by Dep.Informatica of the FCT/UNL
6 * Time-stamp: <99/12/13 05:15:59 jml>
7 *
8 * Joao Lourenco <Joao.Lourenco@di.fct.unl.pt>
9 *
10 * Parallel and Distributed Processing Group
11 * Departamento de Informatica
12 * Faculdade de Ciencias e Tecnologia
13 * Universidade Nova de Lisboa
14 *
15 * ----------------------------------------------
16 *
17 * Description:
18 *
19 * ----------------------------------------------
20 */
21 #ifndef __F0_TKOUT_H
22 #define __F0_TKOUT_H
23
24 #include "chain.h"
25 #include "f0s_out_codes.h"
26
27 /*******************************
28 * One structure for each possible 'rep_outcode'
29 * *******************************/
30
31 /* start_RTIDS */
32 typedef struct rtids_data_s {
33       int     tid;
34       int     tp_pid;
35       int     dbg_pid;
36       int     attached;
37 } tidsdata_t;
38
typedef struct rtids_s {
39       chain_t   *data;   /* of tidsdata_t */
40 } rtids_t;
41 /* end_RTIDS */
42
43 /* start_RKILL */
44 typedef struct rkill_s {
45 } rkill_t;
46 /* end_RKILL */
47
48 /* start_RATTACH */
typedef struct rattach_s {
  int       tid;
  int       tp_pid;
} rattach_t;
/* end_RATTACH */

/* start_RDETACH */
typedef struct rdetach_s {
  int       tp_pid;
} rdetach_t;
/* end_RDETACH */

/* start_RSsymbolFILE */
typedef struct rsymbolfile_s {
  const char *file;
} rsymbolfile_t;
/* end_RSsymbolFILE */

/* start_RFILE */
typedef struct rfile_s {
  int tid;
  const char *file;
} rfile_t;
/* end_RFILE */

/* start_RRUN */
typedef struct rrun_s {
  const char *file;
} rrun_t;
/* end_RRUN */

/* start_RSTEP */
typedef struct rstep_s {
} rstep_t;
/* end_RSTEP */

/* start_RNEXT */
typedef struct rnext_s {
} rnext_t;
/* end_RNEXT */

/* start_RCONTINUE */
typedef struct rcontinue_s {
} rcontinue_t;
/* end_RCONTINUE */

/* start_RFINISH */
typedef struct rfinish_s {
  const void *s_address;
  const char *s_function;
  const char *s_args;
101  const char *s_file;
102  int s_line;
103  int gdb_var;
104  const char *value;
105 } rfinish_t;
106 /**< end_RFINISH */
107
108 /**< start_RBREAK */
109 typedef struct rbreak_s {
110 int bpid;
111 const void *address;
112 const char *file;
113 int line;
114 } rbreak_t;
115 /**< end_RBREAK */
116
117 /**< start_RDELETE */
118 typedef struct rdelete_s {
119 } rdelete_t;
120 /**< end_RDELETE */
121
122 /**< start_RSETVARIABLE */
123 typedef struct rsetvariable_s {
124 } rsetvariable_t;
125 /**< end_RSETVARIABLE */
126
127 /**< start_REVALUATE */
128 typedef struct revaluate_s {
129 int gdb_var;
130 const char *value;
131 } revaluate_t;
132 /**< end_REVALUATE */
133
134 /**< start_RDISPLAY */
135 typedef struct rdisplay_data_s {
136 int dispid;
137 int enabled;
138 const char *expression;
139 const char *value;
140 } displaydata_t;
141 typedef struct rdisplay_s {
142 char_t *data; /**< of displaydata_t */
143 } rdisplay_t;
144 /**< end_RDISPLAY */
145
146 /**< start_RUNDISPLAY */
147 typedef struct rundisplay_s {
148 } rundisplay_t;
149 /**< end_RUNDISPLAY */
150
151 /**< start_RLIST */
typedef struct rlist_data_s {
    int     line_number;
    const char *line_text;
} listdata_t;
typedef struct rlist_s {
    chain_t  *data;    /* of listdata_t */
} rlist_t;
/* end_RLIST */

/* start_RINFOLINE */
typedef struct rinfoline_s {
    int     line;
    const char *file;
    const char *start_function;
    int     start_function_offset;
    const char *end_function;
    int     end_function_offset;
    const void *start_address;
    const void *end_address;
} rinfoline_t;
/* end_RINFOLINE */

/* start_RINFOPROGRAM */
typedef struct rinfoprogram_s {
    int         tp_pid;
    const void *address;
    const char *why;
    int         bpid;
    const char *signal;
} rinfoprogram_t;
/* end_RINFOPROGRAM */

/* start_RINFOSTACK */
typedef struct rframedata_s {
    int     frame;
    const void *address;
    const char *function;
    const char *arguments;
    const char *file;
    int     line;
} framedata_t;
typedef struct rinfostack_s {
    chain_t  *data;    /* of framedata_t */
} rinfostack_t;
/* end_RINFOSTACK */

/* start_RINFOBREAK */
typedef struct rbreak_data_s {
    int        bpid;

203    const char  *type;
204    const char  *disp;
205    int        enabled;
206    const void  *address;
207    const char  *function;
208    const char  *file;
209    int        line;
210    int        hit_counter;
211    int        ign_counter;
212    const char  *condition;
213  } breakdata_t;
214 typedef struct rlinfobreak_s {
215    chain_t  *data;  /* of breakdata_t */
216  } rlinfobreak_t;
217  /* end_RINFOBREAK */
218
219  /* start_RINFODISPLAY */
220 typedef struct rlinfodisplay_s {
221    chain_t  *data;  /* of displaydata_t */
222  } rlinfodisplay_t;
223  /* end_RINFODISPLAY */
224
225  /* start_RINFOTHREADS */
226 typedef struct rlinfothreads_data_s {
227    int        active;
228    int        th_id;
229    int        tp_pid;
230    const void  *frame;
231    const char  *function;
232    const char  *arguments;
233    const char  *file;
234    int        line;
235  } infothreadsdata_t;
236 typedef struct rlinfothreads_s {
237    chain_t  *data;  /* of infothreadsdata_t */
238  } infothreads_t;
239  /* end_RINFOTHREADS */
240
241  /* start_RTTY */
242 typedef struct rtty_s {
243  } rtty_t;
244  /* end_RTTY */
245
246  /* start_RSENDTO */
247 typedef struct rsendto_s {
248    const char  *output;
249  } rsendto_t;
250  /* end_RSENDTO */
251
252  /* start_REPDATA */
253 typedef struct repdata_s {
const char *str;  /* The relevant part of the output */
repcode_t code;   /* The REPLY code */
union {
    rtid_t    r_tids;
    rkill_t   r_kill;
    rattach_t r_attach;
    rdetach_t r_detach;
    rsymbolfile_t r_symbolfile;
    rfile_t   r_file;
    rrun_t    r_run;
    rstep_t   r_step;
    rnext_t   r_next;
    rcontinue_t r_continue;
    rfinish_t r_finish;
    rbreak_t  r_break;
    rdelete_t r_delete;
    rsetvariable_t r_setvariable;
    reevaluate_t r_evaluate;
    rdisplay_t r_display;
    rundisplay_t r_undisplay;
    tty_t     r_tty;
    rlist_t   r_list;
    rinfo_t   r_info;
    rinfof_t   r_infoline;
    rinfofprogram_t r_infoprogram;
    rinfofstack_t r_infolstack;
    rinfofbreak_t r_infobreak;
    rinfofdisplay_t r_infodisplay;
    rinfofthreads_t r_infolthreads;
    rsendto_t r_sendto;
} data;
}

typedef struct abphit_s {
    int bpid;
    const char *function;
    const char *args;
    const char *file;
    int line;
} abphit_t;

/* start_ABPHIT */

typedef struct abphit_s {
    int bpid;
    const char *function;
    const char *args;
    const char *file;
    int line;
} abphit_t;

/* start ASIGNAL */
typedef struct asignal_s {
    char *name;
} asignal_t;

/* end_ASIGNAL */

/* start_ANEWTHREAD */
typedef struct anewthread_data_s {
    int id;
} anewthreaddata_t;

typedef struct anewthread_s {
    char *data;    /* of displaydata_t (ignore 'enabled' field) */
} anewthread_t;

/* end_ANEWTHREAD */

/* start_ATHREAD */
typedef struct athread_s {
    int pid;
} athread_t;

/* end_ATHREAD */

/* start_ALWDATA */
typedef struct alwdata_s {
    const char *str;    /* The relevant part of the output */
    code_t code;    /* The ALWAYS code */
    struct {
        display_t a_display;
        bphit_t a_bphit;
        asignal_t a_signal;
        anewthread_t a_newthread;
        athread_t a_thread;
    } data;
} alwdata_t;

/* end_ALWDATA */

/* #induction*/

/* start_PPOSITION */
typedef struct pposition_s {
    char *file;
    int line;
    int pos;
    const char *where;
    void *frame;
} pposition_t;

/* end_PPOSITION */
typedef struct posdata_s {
    const char *str; /* The relevant part of the output */
    poscode_t code; /* The POSITION code */
    union {
        pposition_t p_position;
    } data;
} posdata_t;

/* start_POSDATA */

/* One structure for all 'err_outcode'

/* start_ERRDATA */

typedef struct errdata_s {
    const char *str; /* The relevant part of the output */
    errcode0_t code; /* The ERROR code */
    union {
    } data;
} errdata_t;

/* end_ERRDATA */

/* The Output Token - level 0

/* start_TKOUT */

typedef struct tkout_s {
    int tid;
    const char *output;
    errcode0_t status;
    repdata_t *rep;
    chain_t *alw; /* of alwdata_t */
    posdata_t *pos;
    errdata_t *err;
} tkout_t;

/* end_TKOUT */

/* Create and Destroy Output Tokens

/* */
Function: tkout_t *tkout = f0s=tkout_new (errcode0_t code)
* Description: Create a new empty Output Token.
* Function: repdata_t *repdata_s = f0s=tkout_rep_new (void)
* Description: Create a new empty Reply for the Output Token.
* Function: posdata_t *posdata_s = f0s=tkout_pos_new (void)
* Description: Create a new empty Position for the Output Token.
* Function: alwdata_t *alwdata_s = f0s=tkout_alw_new (void)
* Description: Create a new empty Always for the Output Token.
* Function: errdata_t *errdata_s = f0s=tkout_err_new (void)
* Description: Create a new empty Error for the Output Token.
* Returns: Pointer to the allocated structure.
*/
tkout_t *f0s=tkout_new (errcode0_t code);
repdata_t *f0s=tkout_rep_new (void);
posdata_t *f0s=tkout_pos_new (void);
alwdata_t *f0s=tkout_alw_new (void);
errdata_t *f0s=tkout_err_new (void);

*/ --- EOF --- */
int f0s=tkout_delete (tkout_t *t);
int f0s=tkout_rep_delete (repdata_t *r);
int f0s=tkout_alw_delete (alwdata_t *r);
int f0s=tkout_pos_delete (posdata_t *r);
int f0s=tkout_err_delete (errdata_t *r);
458 #endif /* __F0_TKOUT_H */
Appendix E  How to use FIDDLE-0s (Example)

In directory FIDDLE_SOURCE_ROOT/f0/console you can find the sources of a complete application that uses FIDDLE-0s. This is probably the best place for you to look at examples on how to request FIDDLE-0s services and how to process its replies.

Below you can find a simplified example of the application above. This example was designed with tutoring purposes and will not compile.

```
1    /* **************************************************************
2    * */
3    * $Id$
4    *
5    */ Copyright (c) 1999 by Dep.Informatica of the FCT/UNL
6    */ Time-stamp: <99/06/30 22:41:50 jml>
7    */
8    */ Joao Lourenco <Joao.Lourenco@di.fct.unl.pt>
9    */
10   */ Parallel and Distributed Processing Group
11   */ Departamento de Informatica
12   */ Faculdade de Ciencias e Tecnologia
13   */ Universidade Nova de Lisboa
14   */
15   */ ******************************************************************
16    *//#include <stdio.h>
17    */
18   */ Don’t forget to add the CC compiler option "-I<path>", where <path>
19    */ is the path to FIDDLE include files */
20   */ #include "fidle/f0s_lib.h"
21    */
22   */#undefines, TYPES and GLOBAL (PRIVATE) VARIABLES
23   */#undefines, TYPES and GLOBAL (PRIVATE) VARIABLES */
24    */
25   */ PRIVATE functions
26   */ PRIVATE functions */
27    */
28   */ static int print_tids_line (void *obj, void *args)
29    {
30      char att;
31      TidsData *d = (TidsData *)obj;
32      switch (d->attached) {
33        case -1:
34          att = '\-';
35          break;
36        case 0:
37            
38```
att = 'n';
break;
case i:
  att = 'y';
break;
}
printf (FORMAT_INT, d->tid, att, d->tp_pid, d->dbg_pid);
return 0;
}

/* Print the contents of a Chain whose elements are "TidsData" structures */
static void print_tids (const RepData *rep)
{
  printf (FORMAT_STRING, "TID", "ATT", "T0_PID", "D0_PID");
  if (rep->data.r_tids.data != NULL)
    chain_walk (rep->data.r_tids.data, print_tids_line, NULL);
}

/* ============================================================== */

static void test_error (ErrData *err)
{
  switch (err->code) {
  case E0_RTS_SEND_RECEIVE:
    [...];
    break;
  case E0_RTS_INVALID_TP_PID:
    [...];
    break;
  case E0_RTS_NO_CLD_AVAILABLE:
    [...];
    break;
  [...];    /* For the remaining error conditions */
  }
}

static void test_reply (RepData *rep)
{
  switch (rep->code) {
  case R0_TIDS:
    print_tids (rep->data.r_tids.data);
    break;
  case R0_KILL:
    /* Do nothing */
    break;
  case R0_ATTACH:
printf ("Attached to process %d\n", rep->data.r_attach.tp_pid);
printf ("FIDDLE Task ID is %d\n", rep->data.r_attach.tid);
break;
case R0_DETACH:
    printf ("Detached from process %d\n", rep->data.r_detach.tp_pid);
    break;
case R0_SYMBOL_FILE:
    printf ("Using symbols from file %s\n",
        astr_chars (rep->data.r_symbolfile.file));
    break;
[...];  /* For the remaining possible replies */
}

static void test_always (AlwData *alw)
{
    switch (alw->code)
    [...];  /* Similar to the 'test_reply()' function */
}

static void test_position (AlwData *alw)
{
    switch (alw->code)
    [...];  /* Similar to the 'test_reply()' function */
}

/* -*- --------------- */

/* Read one line from keyboard, possibly using 'gets()' */
/* or GNU 'readline()' */
static char *read_line (void)
{
    [...];
}

/* Break a string into an 'argv[]' type vector. */
/* The array should be NULL terminated */
static char *parse_line (char *line)
{
    [...];
}
/* Execute the command identified in 'av[0]' with the arguments * av[1]', 'av[2]', ... */
static TkOut0 *execute (char *av[])
{
    TkOut0 *tk;
    int cmd_code;
    cmd_code = get_command_code (av[0]);
    if (cmd_code == -1)
        return make_error_tkout ("Invalid command");
    switch (cmd_code) {
    case COS_TIDS:
        tk = f0s_tids (av[1]);
        break;
    case COS_KILL:
        tk = f0s_kill (atoi (av[1]));
        break;
    case COS_ATTACH:
        tk = f0s_attach (atoi (av[1]));
        break;
    case COS_DETACH:
        tk = f0s_detach (atoi (av[1]));
        break;
    case COS_SYMBOL_FILE:
        tk = f0s_symbol_file (atoi (av[1]), av[2]);
        break;
    case COS_FILE:
        tk = f0s_symbol_file (av[1]);
        break;
    [...] /* Continue with the remaining commands */
    }
    return tk;
}

/* Process the output token. Identify the fields that were filled * and act accordingly */
static void process_output_token (TkOut0 *tk)
{
    if (tkout->status < E0-RTS_OK)
        non_fiddle_error (tkout->status);
    else if (tkout->status > E0-RTS_OK)
        fiddle_error (tkout->status, tkout->output);
    else
        fiddle_ok (tkout);
}
198 /* Process the reply from GDB */
199 static void fiddle_ok (TkOut0 *tk)
200 {
201   if (tk->error != NULL)
202     /* Examine the error message from GDB and act accordingly */
203     test_error (rk->error);
204   else {
205     /* Examine the reply from GDB to the issued command and
206      * act accordingly */
207     test_reply (tkout->rep);
208     test_always (tkout->alw);
209     test_position (tkout->pos);
210   }
211 }
212 }
213
214
215
216
217 /* ########################################################################
218   * PUBLIC functions
219   * ######################################################################## */
220
221 main ()
222 {
223   char **av;
224   char *command;
225   TkOut0 *tk;
226
227   /* 'read_line()' returns NULL to finish */
228   while ((command = read_line()) != NULL) {
229
230     /* Break 'command' into an 'argv[]' type vector */
231     av = parse_line(command);
232
233     /* Execute the command */
234     tk = execute (av);
235
236     /* Test for memory error */
237     if (tk == NULL)
238       memory_error(); /* Probably the easiest solution is to
239                       * print a memory error message and
240                       * abort the program.... */
241
242     /* Process the output token with the reply from GDB */
243     process_output_token (tk);
244
245     /* Release the memory no longer needed */
246     free (command);
247     free (av);
248     f0s_tkout_delete (tk);
249     }
250     }
Concept Index

A
Advices ................................................. 5
Attach (F0m) ......................................... 46
Attach (F0s) ......................................... 34
Attach (F1m) ......................................... 49

B
Back-end debugger .................................... 1
back-end debugger (def) ............................. 67
Back-end debugger (def) ............................. 67
Back-end tool (def) .................................... 67
Basic, FIDDLE ........................................... 2
Break (F0m) ........................................... 46
Break (F0s) ........................................... 37
Break (F1m) ........................................... 50
Breakpoint, Delete (F0m) ............................. 46
Breakpoint, Delete (F0s) ............................. 38
Breakpoint, Delete (F1m) ............................. 50
Breakpoint, Information (F0m) ...................... 46
Breakpoint, Information (F0s) ...................... 42
Breakpoint, Information (F1m) ...................... 50
Bugs, Reporting ....................................... 11
Bypass output parsing (F0m) ......................... 46
Bypass output parsing (F0s) ......................... 44
Bypass output parsing (F1m) ......................... 50

C
Call (F0m) ............................................... 46
Call (F0s) ............................................... 38
Call (F1m) ............................................... 50
Call a function (F0m) ................................. 46
Call a function (F0s) ................................. 38
Call a function (F1m) ................................. 50
chain, collect data ................................. 66
chain, collect origin ............................... 65
chain, collecting traversing results ............... 65
chain, collecting traversing results, data .......... 66
chain, collecting traversing results, origin ....... 65
chain, conditional dequeue .......................... 65
chain, create ......................................... 64
chain, dequeue ........................................ 64
chain, dequeue conditional .......................... 65
chain, destroy ........................................ 64
chain, enqueue ........................................ 64
chain, find ............................................. 65
chain, first ............................................ 64
chain, last ............................................. 64
chain, length ......................................... 64
chain, merge .......................................... 65
chain, next ............................................. 64
chain, previous ...................................... 64
chain, pruning ........................................ 65
chain, remove ........................................ 65
chain, traversing .................................... 65
chain, walker ......................................... 65
Change stack frame (F0m) ........................... 46
Change stack frame (F0s) ........................... 41, 42
Change stack frame (F1m) ........................... 50
Client (def) ........................................... 67
Compiling ............................................. 7
Compiling FIDDLE .................................... 7
Compiling with FIDDLE ............................... 8
Component-level debugger (def) .................... 67
Continue (F0m) ....................................... 46
Continue (F0s) ....................................... 36
Continue (F1m) ....................................... 50
Current line information (F0m) ...................... 46
Current line information (F0s) ...................... 40
Current line information (F1m) ...................... 50

D
Data types (F0m) ...................................... 45
Data types (F0s) ...................................... 23
Data types (F1m) ...................................... 49
Data, Return (F0s) .................................... 30
Debugee (def) ........................................ 67
Debugging (Example) ................................. 22
Debugging interfaces ................................ 1
Default return values (F0s) .......................... 32
Definition, back-end debugger ....................... 67
Definition, Back-end debugger ....................... 67
Definition, Back-end tool ............................ 67
Definition, Client .................................... 67
Definition, Component-level debugger ............. 67
Definition, Debugee ................................. 67
Definition, Front-end ................................ 67
Definition, Target application ........................ 67
Definition, Target process ........................... 67
Definition, Token ..................................... 67
Delete (F0m) ........................................... 46
Delete (F0s) ........................................... 38
Delete (F1m) ........................................... 50
Delete a breakpoint (F0m) ........................... 46
Delete a breakpoint (F0s) ........................... 38
Delete a breakpoint (F1m) ........................... 50
Detach (F0m) ........................................... 46
Detach (F0s) ........................................... 34
Detach (F1m) ........................................... 49
Detachment environment ............................. 7
Display (F0m) ........................................ 46
Display (F0s) ........................................ 39
Display (F1m) ........................................ 50
Display, Information (F0m) .......................... 46
Display, Information (F0s) .......................... 43
Display, Information (F1m) .......................... 50
Distributed debugging ........................................ 4
Double linked list ........................................... 63
Down (F0m) .................................................. 46
Down (F0s) .................................................. 42
Down (F1m) .................................................. 50

E
Error messages (F0s) ........................................ 33
Evaluate (F0m) ............................................... 46
Evaluate (F0s) ............................................. 39
Evaluate (F1m) ............................................... 50
Evaluate an expression (F0m) ................................ 46
Evaluate an expression (F0s) ................................ 39
Evaluate an expression (F1m) ................................ 50
Example using FIDDLE-0s .................................... 79
Example, Client program ..................................... 14
Example, Compiling the sources ............................ 13
Example, Target program .................................... 13
Example, Tiny ................................................ 13
Expression, Evaluate an (F0m) ............................. 46
Expression, Evaluate an (F0s) ............................. 39
Expression, Evaluate an (F1m) ............................. 50
Expression, Undisplay (F0m) ................................ 46
Expression, Undisplay (F0s) ................................ 39
Expression, Undisplay (F1m) ................................ 50

F
F0m Management functions ................................. 45
F0m Miscellaneous functions ............................... 46
F0m Process control functions ............................. 45
F0m Process inspection functions .......................... 46
F0m Service Functions ...................................... 45
F0m Thread functions ....................................... 46
F0s return values, Default ................................. 32
F0s Returning Data ........................................... 30
F0s Service Functions ....................................... 32
FI1m Management functions ............................... 49
FI1m Miscellaneous functions ............................. 50
FI1m Process control functions ........................... 49
FI1m Process inspection functions ........................ 50
FI1m Service Functions ...................................... 49
FI1m Thread functions ....................................... 50
FIDDE Client processes ..................................... 8
FIDDE clients ............................................... 1
FIDDE operation ............................................. 5
FIDDE Target processes ..................................... 8
FIDDE-0m ..................................................... 2, 45
FIDDE-0s ..................................................... 2, 23
FIDDE-0s example ........................................... 79
FIDDE-1m ..................................................... 3, 49
FIDDE-2m ..................................................... 4
FIDDE-3m ..................................................... 4

File (F0m) .................................................... 46
File (F0s) .................................................... 35
File (F1m) .................................................... 49
Finish (F0m) ................................................ 46
Finish (F0s) ................................................ 36
Finish (F1m) ................................................ 50
Frame (F0m) ................................................ 46
Frame (F0s) ................................................ 42
Frame (F1m) ................................................ 50
Front-end (def) ............................................. 67
Front-end debugger .......................................... 1
Functions, Management (F0m) ............................ 45
Functions, Management (F0s) ............................ 32
Functions, Management (F1m) ............................ 49
Functions, Miscellaneous (F0m) .......................... 46
Functions, Miscellaneous (F0s) .......................... 44
Functions, Miscellaneous (F1m) .......................... 50
Functions, Process control (F0m) ........................ 45
Functions, Process control (F0s) ........................ 34
Functions, Process control (F1m) ........................ 49
Functions, Process inspection (F0m) ...................... 46
Functions, Process inspection (F0s) ...................... 39
Functions, Process inspection (F1m) ...................... 50
Functions, Service (F0m) ................................... 45
Functions, Service (F0s) ................................... 32
Functions, Service (F1m) ................................... 49
Functions, Thread (F0m) ................................... 46
Functions, Thread (F0s) ................................... 43
Functions, Thread (F1m) ................................... 50

G
Getting FIDDLE ............................................ 7
Glossary ..................................................... 67

H
Header files ................................................... 8
How to use FIDDLE-0s ...................................... 79

I
Info break (F0m) ........................................... 46
Info break (F0s) ........................................... 42
Info break (F1m) ........................................... 50
Info display (F0m) .......................................... 46
Info display (F0s) .......................................... 43
Info display (F1m) .......................................... 50
Info line (F0m) ............................................. 46
Info line (F0s) ............................................. 40
Info line (F1m) ............................................. 50
Info program (F0m) ......................................... 46
Info program (F0s) ......................................... 40
Info program (F1m) ......................................... 50
Concept Index

Info stack (F0m) ........................................... 46
Info stack (F0s) ........................................... 41
Info stack (F1m) ........................................... 50
Info threads (F0m) ........................................... 46
Info threads (F0s) ........................................... 43
Info threads (F1m) ........................................... 50
Initialization (F0m) .................................... 45
Initialization (F0s) .................................... 32
Initialization (F1m) .................................... 49
Installing ............................................... 7
Introduction ............................................. 1
K
Kill (F0m) .................................................. 46
Kill (F0s) .................................................. 34
Kill (F1m) .................................................. 49
L
Layer-0m .............................................. 2, 45
Layer-0s .............................................. 2, 23
Layer-1m .............................................. 3, 49
Layer-2m ............................................... 4
Layer-3m ............................................... 4
Library, Utilities ..................................... 63
Linking with FIDDLE .................................. 9
List (F0m) .............................................. 46
List (F0s) .............................................. 39
List (F1m) .............................................. 50
Load a program ....................................... 35
M
Management functions (F0m) ...................... 45
Management Functions (F0s) ...................... 32
Management functions (F1m) ...................... 49
Miscellaneous functions (F0m) .................... 46
Miscellaneous functions (F0s) .................... 44
Miscellaneous functions (F1m) .................... 50
N
Next (F0m) .............................................. 46
Next (F0s) .............................................. 36
Next (F1m) .............................................. 50
O
Output redirection (F0m) ............................ 46
Output redirection (F0s) ............................ 44
Output redirection (F1m) ............................ 50
Output, Parsing bypass (F0m) ..................... 46
Output, Parsing bypass (F0s) ..................... 44
P
Process control functions (F0m) .................. 45
Process control functions (F0s) .................. 34
Process control functions (F1m) .................. 49
Process inspection functions (F0m) .......... 46
Process inspection functions (F0s) .......... 39
Process inspection functions (F1m) .......... 50
Program information (F0m) ..................... 46
Program information (F0s) ..................... 40
Program information (F1m) ..................... 50
R
Releasing token memory (F0s) ............... 33
Reply to service requests ....................... 23
Reporting bugs .................................. 11
Returning Data (F0s) ............................. 30
Run (F0m) ........................................... 46
Run (F0s) ........................................... 35
Run (F1m) ........................................... 50
S
Send a signal (F0s) ..................................... 37
Sendto (F0m) ........................................... 46
Sendto (F0s) ........................................... 44
Sendto (F1m) ........................................... 50
Service Functions (F0m) ....................... 45
Service Functions (F0s) ....................... 32
Service Functions (F1m) ....................... 49
Service reply ....................................... 23
Service reply, always field ..................... 27
Service reply, data field ......................... 25
Service reply, error field ......................... 29
Service reply, position field .................... 28
Set a breakpoint (F0m) ......................... 46
Set a breakpoint (F0s) ......................... 37
Set a breakpoint (F1m) ......................... 50
Set a variable (F0m) .............................. 46
Set a variable (F0s) .............................. 38
Set a variable (F1m) .............................. 50
Signal (F0m) ........................................... 46
Signal (F0s) ........................................... 37
Signal (F1m) ........................................... 50
Source listing (F0m) .............................. 46
Source listing (F0s) .............................. 39
Source listing (F1m) .............................. 50
Stack, Backtrace (F0m) ..................... 46
Stack, Backtrace (F1m) ..................... 46
<table>
<thead>
<tr>
<th>Stack, Backtrace (F0s)</th>
<th>Stack, Backtrace (F1m)</th>
<th>Stack, Change frame (F0m)</th>
<th>Stack, Change frame (F0s)</th>
<th>Stack, Change frame (F1m)</th>
<th>Stack, Change frame (F1s)</th>
<th>Stack, Information (F0m)</th>
<th>Stack, Information (F0s)</th>
<th>Stack, Information (F1m)</th>
<th>Stack, Information (F1s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>50</td>
<td>46</td>
<td>41, 42</td>
<td>46</td>
<td>46</td>
<td>41</td>
<td>50</td>
<td>41</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step (F0m)</th>
<th>Step (F0s)</th>
<th>Step (F1m)</th>
<th>Switch thread (F0m)</th>
<th>Switch thread (F0s)</th>
<th>Switch thread (F1m)</th>
<th>Symbol file (F0m)</th>
<th>Symbol file (F0s)</th>
<th>Symbol file (F1m)</th>
<th>Symbolic task ID (F0m)</th>
<th>Symbolic task ID (F0s)</th>
<th>Symbolic task ID (F1m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>35</td>
<td>50</td>
<td>46</td>
<td>43</td>
<td>50</td>
<td>46</td>
<td>35</td>
<td>49</td>
<td>45</td>
<td>32</td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread (F1m)</th>
<th>Thread functions (F0m)</th>
<th>Thread functions (F0s)</th>
<th>Thread functions (F1m)</th>
<th>Thread functions (F1s)</th>
<th>Thread, Switch (F0m)</th>
<th>Thread, Switch (F0s)</th>
<th>Thread, Switch (F1m)</th>
<th>Thread, Switch (F1s)</th>
<th>Threads, Information (F0m)</th>
<th>Threads, Information (F0s)</th>
<th>Threads, Information (F1m)</th>
<th>Threads, Information (F1s)</th>
<th>Tiny Example</th>
<th>Tkin (F0m)</th>
<th>Tkin (F1m)</th>
<th>Token (def)</th>
<th>Token, releasing memory (F0s)</th>
<th>Tty (F0m)</th>
<th>Tty (F0s)</th>
<th>Tty (F1m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>46</td>
<td>43</td>
<td>50</td>
<td>46</td>
<td>43</td>
<td>33</td>
<td>46</td>
<td>44</td>
<td>50</td>
<td>43</td>
<td>47</td>
<td>50</td>
<td>13</td>
<td>47</td>
<td>50</td>
<td>67</td>
<td>33</td>
<td>46</td>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U</th>
<th>Undisplay (F0m)</th>
<th>Undisplay (F0s)</th>
<th>Undisplay (F1m)</th>
<th>Up (F0m)</th>
<th>Up (F0s)</th>
<th>Up (F1m)</th>
<th>Using FIDDLE</th>
<th>Utilities Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>39</td>
<td>50</td>
<td>46</td>
<td>41</td>
<td>50</td>
<td>8</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
<th>What is FIDDLE</th>
<th>What is FIDDLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Type Index

A
abphit_t ........................................... 38
alvcode_t ........................................ 27
alvdata_t ........................................ 27
asignal_t ......................................... 37

B
breakdata_t ...................................... 42

C
Cabdata ............................................. 38
chain_cmp_f ...................................... 64
chain_collect_t ................................. 63
chain_node_t ..................................... 63
chain_t ........................................... 63
chain_walk_collect_f .......................... 63
chain_walk_f .................................... 63

D
displaydata_t ................................... 30, 43

E
eenum .............................................. 24, 27, 29, 30
errcode_t ......................................... 30
errdata_t ......................................... 29

F
framedata_t ....................................... 41

L
listdata_t ......................................... 39

P
poscode_t ......................................... 29
posdata_t ......................................... 28
pposition_t ....................................... 28

R
rattach_t .......................................... 34
rbreak_t .......................................... 37
Rcall ............................................. 38
rdetach_t ........................................ 34
rdisplay_t ........................................ 39
repcode_t ......................................... 24
repdata_t ......................................... 25
revaluate_t ...................................... 39
rfile_t ........................................... 35
rfinish_t .......................................... 36
rinfobreak_t ..................................... 42
rinfodisplay_t .................................. 43
rinfoline_t ....................................... 40
rinfoprogram_t .................................. 40
rinfostack_t ..................................... 41
rlist_t ............................................ 39
rrun_t ............................................ 35
rsymbolfile_t .................................... 35
rtids_t ............................................ 33

S
struct ............................................. 23, 25, 27, 28, 29, 33, 34, 35, 36, 37, 38
struct rattach_s .................................. 34
struct rbreak_sdata ............................. 42
struct rcachedata ............................... 38
struct rcall ...................................... 38
struct rdisplay_s ................................ 39
struct rdatal_s ................................... 39, 43
struct revaluate_s ................................ 39
struct rfile_s .................................... 35
struct rfinish_s .................................. 36
struct rframedata_s ............................. 41
struct rinfobreak_s ............................. 42
struct rinfodisplay_s ........................... 43
struct rinfoline_s ................................ 40
struct rinfoprogram_s ........................... 40
struct rinfostack_s .............................. 41
struct rlist_s .................................... 39
struct rlist_sdata_s ............................. 39
struct rrun_s ..................................... 35
struct rsymbolfile_s ............................ 35

T
tids_data_t ....................................... 33
tkout_t .......................................... 23
Function Index

C
chain_collect_data ........................................... 66
chain_collect_origin ......................................... 65
chain_create .................................................. 64
chain_dequeue ................................................ 64
chain_dequeue_cond .......................................... 65
chain_destroy ................................................ 64
chain_enque ................................................. 64
chain_find .................................................... 65
chain_first .................................................... 64
chain_last ..................................................... 64
chain_length .................................................. 64
chain_merge ................................................... 65
chain_next ..................................................... 64
chain_prev ..................................................... 64
chain_prune ................................................... 65
chain_remove .................................................. 65
chain_rwlock .................................................. 65
chain_walk ..................................................... 65
chain_walk_collect .......................................... 65

F
f0m_attach ..................................................... 46
f0m_break ...................................................... 46
f0m_call ....................................................... 46
f0m_continue ............................................... 46
f0m_delete .................................................... 46
f0m_detach .................................................... 46
f0m_display ................................................... 46
f0m_down ....................................................... 46
f0m_evaluate .................................................. 46
f0m_file ....................................................... 46
f0m_finish ..................................................... 46
f0m_frame ..................................................... 46
f0m_info_break .............................................. 46
f0m_info_display ............................................ 46
f0m_info_line ............................................... 46
f0m_info_program ........................................... 46
f0m_info_stack .............................................. 46
f0m_info_threads ............................................ 46
f0m_initialize ............................................... 45, 49
f0m_kill ....................................................... 45
f0m_list ...................................................... 46
f0m_next ...................................................... 46
f0m_run ....................................................... 46
f0m_sendto ................................................... 46
f0m_set_variable ............................................ 46
f0m_signal .................................................... 46
f0m_step ....................................................... 46
f0m_symbol_file ............................................. 46
f0m_terminate ............................................... 45, 49
f0m_thread ................................................... 46
f0m_tids ...................................................... 45, 49
f0m_tty ....................................................... 47
f0m_undisplay ............................................... 46
f0m_up ......................................................... 46
f0s_attach .................................................... 34
f0s_break ..................................................... 37
f0s_call ....................................................... 38
f0s_continue ................................................ 36
f0s_delete .................................................... 38
f0s_detach .................................................... 34
f0s_display .................................................. 39
f0s_down ....................................................... 42
f0s_ensure .................................................... 33
f0s_evaluate .................................................. 39
f0s_file ....................................................... 35
f0s_finish ..................................................... 36
f0s_frame ..................................................... 42
f0s_info_break .............................................. 42
f0s_info_display ............................................ 43
f0s_info_line ............................................... 40
f0s_info_program ............................................ 40
f0s_info_stack .............................................. 41
f0s_info_threads ............................................ 43
f0s_initialize ............................................... 32
f0s_kill ....................................................... 34
f0s_list ....................................................... 39
f0s_next ...................................................... 36
f0s_run ....................................................... 35
f0s_sendto ................................................... 44
f0s_set_variable ............................................ 38
f0s_signal .................................................... 37
f0s_step ....................................................... 35
f0s_symbol_file ............................................. 35
f0s_terminate ............................................... 32
f0s_thread ................................................... 43
f0s_tids ...................................................... 32
f0s_tkill ..................................................... 33
f0s_tkill_delete ............................................ 33
f0s_tty ....................................................... 44
f0s_undisplay ............................................... 39
f0s_up ........................................................ 41
fim attach ..................................................... 49
fim_break ..................................................... 50
fim_call ....................................................... 50
fim_continue ................................................ 50
fim_delete .................................................... 50
fim_detach ................................................... 49
fim_display ................................................... 50
fim_down ...................................................... 50
fim_evaluate .................................................. 50
fim_file ...................................................... 49
fim_finish .................................................... 50
fim_frame ..................................................... 50
fim_info_break .............................................. 50
fim_info_display ............................................ 50
fim_info_line ............................................... 50
fim_info_program ............................................ 50
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>fim_info_stack</td>
<td>50</td>
</tr>
<tr>
<td>fim_info_threads</td>
<td>50</td>
</tr>
<tr>
<td>fim_kill</td>
<td>49</td>
</tr>
<tr>
<td>fim_list</td>
<td>50</td>
</tr>
<tr>
<td>fim_next</td>
<td>50</td>
</tr>
<tr>
<td>fim_run</td>
<td>50</td>
</tr>
<tr>
<td>fim_sendto</td>
<td>50</td>
</tr>
<tr>
<td>fim_set_variable</td>
<td>50</td>
</tr>
<tr>
<td>fim_signal</td>
<td>50</td>
</tr>
<tr>
<td>fim_step</td>
<td>50</td>
</tr>
<tr>
<td>fim_symbol_file</td>
<td>49</td>
</tr>
<tr>
<td>fim_thread</td>
<td>50</td>
</tr>
<tr>
<td>fim_tkm</td>
<td>50</td>
</tr>
<tr>
<td>fim_tty</td>
<td>50</td>
</tr>
<tr>
<td>fim_undisplay</td>
<td>50</td>
</tr>
<tr>
<td>fim_up</td>
<td>50</td>
</tr>
</tbody>
</table>
# Table of Contents

1 Introduction ............................................. 1
   1.1 What is FIDDLE ......................................... 1
   1.2 FIDDLE vs. Traditional Debuggers ....................... 1
   1.3 FIDDLE Layers .......................................... 2
   1.4 How FIDDLE Operates ..................................... 5
   1.5 Some Notes to the Reader .................................. 5

2 Compiling and Installing FIDDLE ....................... 7
   2.1 Development and Testing Environment .................... 7
   2.2 Getting FIDDLE ......................................... 7
   2.3 Compiling FIDDLE ......................................... 7
   2.4 FIDDLE Target Processes .................................. 8
   2.5 FIDDLE Client Processes .................................. 8

3 Reporting Bugs ........................................... 11

4 A Tiny Example ........................................... 13
   4.1 Compiling the example source files ....................... 13
   4.2 The Target Program ....................................... 13
   4.3 The Client Program ....................................... 14
   4.4 Debugging the Target Program with the Client Program .... 22

5 FIDDLE-0s .................................................. 23
   5.1 F0s Data Types .......................................... 23
      5.1.1 The tkout_t structure ................................ 23
      5.1.2 The repcode_t enumerated ............................ 24
      5.1.3 The repdata_t structure ............................... 25
      5.1.4 The alwdata_t structure ............................... 27
      5.1.5 The posdata_t structure ............................... 28
      5.1.6 The erodata_t structure ............................... 29
   5.2 F0s Returning Data ....................................... 30
   5.3 F0s Service Functions ..................................... 32
      5.3.1 F0s Default Return Values ............................ 32
      5.3.2 F0s Management Functions ............................. 32
      5.3.3 F0s Process Control Functions ......................... 34
      5.3.4 F0s Process Inspection Functions ...................... 39
      5.3.5 F0s Thread Functions ................................ 43
      5.3.6 F0s Miscellaneous Functions ......................... 44

6 FIDDLE-0m .................................................. 45
   6.1 F0m Data Types .......................................... 45
   6.2 F0m Service Functions ..................................... 45
      6.2.1 F0m Management Functions ............................. 45
      6.2.2 F0m Process Control Functions ......................... 45
      6.2.3 F0m Process Inspection Functions ...................... 46
      6.2.4 F0m Thread Functions ................................ 46
      6.2.5 F0m Miscellaneous Functions ......................... 46
7   FIDDLE-1m ........................................ 49
    7.1  Flm Data Types ........................................ 49
    7.2  Flm Service Functions ................................. 49
      7.2.1  Flm Management Functions ....................... 49
      7.2.2  Flm Process Control Functions .................. 49
      7.2.3  Flm Process Inspection Functions ............... 50
      7.2.4  Flm Thread Functions ............................ 50
      7.2.5  Flm Miscellaneous Functions .................... 50

8   FIDDLE-2m ........................................ 53

9   FIDDLE-3m ........................................ 55

10  Unsolved Problems ................................. 57
    10.1  Intercepting fork() .................................. 57
    10.2  I/O of remote processes ............................ 57

11  Acknowledgements ................................. 59

Appendix A   FIDDLE Related Publications .......... 61

Appendix B   FIDDLE Utilities Library .......... 63
  B.1  Chain types .......................................... 63
  B.2  Chain functions .................................... 64

Appendix C   Glossary ................................. 67

Appendix D   f0s_tkout.h ............................... 69

Appendix E   How to use FIDDLE-0s (Example) ...... 79

Concept Index ........................................ 85

Type Index ............................................. 89

Function Index ...................................... 91