Trace Implementation Framework 1

The *Trace Implementation Framework* (TIF) lets you insert probe points into source code. With these probe points, you can:

- Trace C or C++ program execution
- Debug your C or C++ programs
- Gather performance data on C or C++ programs

Use TIF to insert probe points anywhere in C or C++ code, including `.init` sections, `.fini` sections, multithreaded code, shared objects, and shared objects opened by `dlopen(3X)`. A *probe point* or a *probe* is a place in the source code that generates a trace record to a trace file.

You can also connect functions called probe functions to these probe points. Currently, the only available probe function is a debug function that writes output to `stderr`.

TIF supplies

- the library that you link with to trace or debug execution (`libtnfprobe`)
- the program you use to manipulate the probe points and the kinds of data they gather (`prex`)
- the program you use to convert the binary TNF[1] trace file to an ASCII file (`tnfdump`)

The following list shows the steps in a TIF session.

1. Use `libtnfprobe` macros to insert probe points in your program or library

1

2. Compile your code
3. Run one or more `prex` sessions to manipulate the probe points and to create the binary trace file
4. Run `tnfdump(1)` to convert the binary trace file to ASCII
5. Read information from the ASCII file

Types of Users

Not everyone will use all the TIF pieces provided. If you use the probe points that are shipped with your program or library, follow the instructions for "Using Existing Probe Points" on page 22. If you are debugging a program, follow the instructions under "Debugging a Program" on page 23. If you are inserting probe points in a library, all the sections in this chapter will be helpful.

Using Existing Probe Points

If your program or library was shipped with probe points, these are the sections that you should read:

First, read the documentation provided with the program or library that gives the names of the probe points and explains the data gathered.

Then read the following sections in this chapter.

- "Compiling and Linking" on page 3
- "Manipulating Probe Points With prex" on page 4
- "Reading the Trace File" on page 39

1. The binary trace file that is gathered by TIF is written in a format called "Trace Normal Form." This is not something a TIF user needs to know anything about, but it explains why the letters "TNF" appear so often in this chapter.

Debugging a Program

Because you will both insert probe points and gather information from them, you will need to read all the sections listed under "Using Existing Probe Points" on page 22, as well as the section on "Inserting Probe Points" on page 41.

Inserting Probe Points in Libraries for Shipping

While including probe points in a library for shipping involves the same steps as debugging a program, you also will provide information about the probe points for those who use your library. For you, all sections in this chapter apply.

Compiling and Linking

Deciding to Link With libtnfprobe

Linking with the library increases the size of your program. If you are not using the probe points and you will never want to use them, don't link with `libtnfprobe`.

But, for example, if a program is not easy to restart and is a long-running program that you might someday want to gather information about, then *do* link with `libtnfprobe`. A window server is a good example of this — not the sort of program you want to stop and restart often, so you probably want to link with `libtnfprobe` before you start it running.

If you are attaching[1] from `prex`, `libtnfprobe` must be linked to your program before you attach. You can link with `libtnfprobe` in one of two ways:

- At compile time, include `-ltnfprobe` on the `cc` line (if using `-lthread`, always put `-ltnfprobe` before `-lthread`), or
- `$LD_PRELOAD=libtnfprobe.so.1 a.out` (see the Linker & Libraries book for an explanation of `LD_PRELOAD`)

1. Both attaching and loading are explained in "Starting prex" on page 24

then attach.

You can compile probe points out with -DNPROBE (similar to the NDEBUG for assert(3X)).

Note that libtnfprobe is automatically linked in (with LD_PRELOAD) when you load[1] your program with prex.

Manipulating Probe Points With prex

Starting prex

Start prex running on your program in one of two ways: load your program from prex, or attach prex to your already-running process. (Note that prex works with dynamic executables, and not static executables.)

Loading your program from prex [1]

When you load your program from prex, prex starts your program running and then stops the program before any user code is executed. Code is stopped even before the .init sections are executed, so you can place probe points in the .init sections to trace initialization code.

```
prex [-o outfilename] [-s kbytes-size] [-l sharedobjs] cmd [cmd-args...]
```

The -o, -s, and -l options are explained in "Options to prex" on page 6.

In the simplest case, and with an executable named a.out that takes no arguments and that contains the probe points, this looks like

```
$ prex a.out
```

1. When you load your program with prex, it preloads libtnfprobe, so the program does not have to be explicitly linked with libtnfprobe.

Attaching `prex` *to your already-running process*[1]

```
$ prex [-o outfilename] [-s kbytes_size] [-l sharedobjs] -p pid
```

Again, in the simplest case, if the program named `a.out` is running and the `ps(1)` command shows that `a.out` has a process ID (PID) of 2374, then the following command attaches `prex` to `a.out`.

```
$ prex -p 2374
```

1. Your running process should have `libtnfprobe` already linked in—see "Compiling and Linking" on page 3.

Options to `prex`

Table 1-1 prex Command-Line Options

Option	Definition
-o *trace_file*	The trace file contains information gathered by the `prex` command.The *trace_file* location is assumed to be relative to the directory from which you started `prex`. When no *trace_file* is specified, the default location is `/$`*TMPDIR*`/trace-$`*PID* where $*PID* is the process ID of your program. If $*TMPDIR* is not set, `/tmp` is used. When a program is traced, the trace file that is set at the start (the default or the `-o option` name) is used for the life of the program.
-l *libraries*	The *libraries* argument contains the names of libraries to `LD_PRELOAD` for your programs. It follows the `LD_PRELOAD` rules on how libraries should be specified and where they will be found. Enclose the space-separated list of library names in double quotes.
-s *size*	*size* is the size of the trace file in kilobytes. The default size of a trace file is four megabytes.

When you see the `prex` prompt, `prex` is running and your program is stopped.

```
prex>
```

Enter `prex` commands to control the probe points: e.g. listing probe points, turning on tracing, and/or debugging.

To continue running your program, enter the `continue` command.

```
prex> continue
```

Type control-c (^c) to stop your program and return to `prex`.

```
$ ^c
prex>
```

Running `prex`

You can attach and detach prex from your target repeatedly, creating multiple sessions. For example, in your first session you could start the target with prex, set up the probes, and type `quit resume`. In a following session, you could attach prex to the same target, reconfigure the probes, and again type `quit resume`. You can do this any number of times. Note that all traces are written to the trace file name specified in the first session - it cannot be changed.

The Parts of the `prex` *Command*

Attributes and Pseudo-Attributes

Attributes are characteristics by which you identify probe points in your code. Pseudo-attributes are not seen in source files, but can be accessed through the `prex` command. Also, a programmer can include user-defined attributes with the TNF_PROBE macros (see "Using the TNF_PROBE_* Macros" on page 42.

Table 1-2 Characteristics of Attributes

Attribute	Characteristics
name	The name of the probe point
keys	The groups to which the probe point belongs. If any key in a probe point is enabled, then that probe point is enabled.
file	The name of the file containing the probe point
line	The line number in the code on which the probe point occurs
slots	The names of the probe point argument. (*arg_name_n*, see page 24)

Table 1-3 Pseudo-Attributes

Pseudo-Attribute	Characteristics
enable	A probe point performs the action that it has been set up for only when it is enabled. For example, even when the tracing state is set, tracing occurs only if the probe point has been enabled.
trace	When tracing for a probe point is on, a line is written to the trace file each time the probe point is *hit* in your program.
object	The name of the shared object or executable that the probe is in. Useful for selecting all the probes in a particular module.
funcs	Shows the list of probe functions connected to this probe. Currently, only the debug function is available.

Grammar for Matching Probes

Select probes based on the attributes listed in your *selector_list*. Each attribute can be described by:

- Identifiers – attribute names using letters, numbers, and the symbols `_\.` and %; cannot begin with a number.
- Quoted strings – names of attributes in single quotes, taken literally; useful when the attribute names are reserved words
- Regular expressions – attribute names or parts of attribute names enclosed in slashes (//); expanded for any match following the rules for `egrep`; when slashes occur in the regular expression, as in a path name, escape the slashes with backslashes: `/\/tmp\/filename/`

The *selector_list* lets you select probe points based on the values you associate with the attribute. For example, if you choose to turn tracing on for all probe points that have the following values for the `name` and `file` attributes

```
name=/first/  file='sampleZ.c'
```

then all probe points with names containing the word `first` and all probe points with a `file` value of `sampleZ.c` will have tracing turned on.

Use $*set_name* as a shorthand alias for *selector_list*. The *set_name* follows the Identifier naming rules, above. The `$all` default set selects all probes in the program.

Reserved Words

The following words are reserved, so enclose them in single quotes if you use them to select attributes or values

alloc	buffer	clear
connect	continue	create
delete	disable	enable
fcns	file	filter
funcs	help	history
keys	kill	ktrace
line	list	name
off	on	pfilter
probes	quit	resume
sets	slots	source
suspend	trace	untrace
values		

Table 1-4 `prex` Commands

Command	Action
`create` $*set_name selector_list*	Create a set with probe points matching *selector_list*. Effectively creates an alias, $*set_name*, for the *selector_list*.
list sets	List the defined sets.
list fcns	List the defined functions (currently, only &debug is defined).
connect &debug $*set_name* connect &debug *selector_list*	Connect a function to the probe points (currently, only the debug function is available; does not enable the probe points); note that the debug function sends its output to `stderr` and not to the trace file.
clear $*set_name* clear *selector_list*	Disconnect connected probe functions.
trace $*set_name* trace *selector_list* untrace $*set_name* untrace *selector_list*	Toggle the tracing action of the probe points (determine whether or not a probe point generates a trace record when it is hit; does not enable the probe points). (default mode is tracing on) The `untrace` command is useful when you are only interested in getting debug output, in which case your probe should be enabled, with tracing off and debug on. Tracing disturbs your program less than the debug function does because `trace` writes to a file while `debug` writes to `stderr`.
enable $*set_name* enable *selector_list* disable $*set_name* disable *selector_list*	Control whether or not the probe points perform the action they are set up for (probe points are disabled by default; does not turn on tracing).The cheapest way to stop a probe from tracing is to use the `disable` command. The enable and disable commands are a master switch. If a probe point is not enabled, even if the probe point is connected to the `debug` function and `trace is` on, no information is sent to `stderr` or to the trace file.
list history	List the control command history (list the commands `connect`, `clear`, `trace`, `untrace`, `enable`, and `disable`—the commands that are executed whenever a new shared object is brought in to your program through `dlopen`)

Command	Action
list *attributes* probes $*set_name* list *attributes* probes *selector_list*	List whether specified probe points are enabled or disabled, whether they have tracing on or off, and what the connected probe function is. For example, `list name file probes name=first file='sampleZ.c'` lists only `name` and `file` values for matching probe points, while the command `list probes name=first file='sampleZ.c'` lists all default attributes and their values (`name`, `enable`, `trace`, `file`, `line`, and `funcs`).
list values *attributes*	List the values associated with attributes like name, keys, and file
help	List all available `prex` commands
source *filename*	Source a file of `prex` commands. *filename* is a quoted string.
continue	Resume execution of your program
quit kill	Quit `prex` and kill your program
quit resume	Quit `prex` and continue executing your program
quit suspend	Quit `prex` and leave your program suspended
quit	Quit `prex`; if your program is loaded with `prex`, it will be killed; if your program was attached to `prex`, your program will continue

Tracing, Enabling, and Connecting

To trace a probe point, the probe point must have trace on and it must be enabled. To debug a probe point, the probe point must be connected to the debug function and must be enabled.

The following table shows which combinations of tracing, enabling, and connecting result in which actions.

Table 1-5

Enabled or Disabled	Tracing State (On/Off)	Debug State (Connected/Cleared)	Results In
Enabled	On	Connected	Tracing and Debugging
Enabled	On	Cleared	Tracing, only
Enabled	Off	Connected	Debugging, only
Enabled	Off	Cleared	nothing

Table 1-5

Enabled or Disabled	Tracing State (On/Off)	Debug State (Connected/Cleared)	Results In
Disabled	On	Connected	nothing
Disabled	On	Cleared	nothing
Disabled	Off	Connected	nothing
Disabled	Off	Cleared	nothing

Controlling `prex` *From a Script*

Enter commands to `prex` either from the `prex` command line or from a file containing `prex` commands.

When you start `prex`, it searches for a file named `.prexrc` first in $*HOME*/ and next in the directory from which you started prex (./). Commands are read from all files that are found. Therefore it is possible to have a `.prexrc` file in the current directory that overrides defaults that are set up by the `.prexrc` file in the home directory.

After reading any `.prexrc` files that are found, input is expected from the `prex` command line. To set up an experiment entirely using `.prexrc` files, the last statement in it can be "`quit resume`" which will quit `prex` but let your program continue.

Also, when `prex` is running you can use the **`source`** *filename* command to specify a file from which `prex` reads commands. This file can have any name you like.

As with commands entered from the `prex` command line, all commands in the script should be in ASCII.

- Terminate each command with the newline character.
- To continue a command onto the next line, end the previous line with a backslash (\) character.
- Separate tokens by whitespace (one or more spaces or tabs).
- Start comments with a hashmark (#).

While the command language is the same for `prex` commands entered from the command line and for commands from a script, commands that return output (such as `list probes...`) make little sense in a script because output goes to `stdout`.

1

A Sample C Program

```
Line 1    #include <stdio.h>
          #include <string.h>

          #include <tnf/probe.h>

          #define MAX_RESPONSE_SIZE        256

          void find_factor(int n);

Line 10   int
          main(int argc, char **argv)
          {
              boolean_t    shouldexit = B_FALSE;
              int          sum = 0, max_loop = 5;
              int          i;

              TNF_PROBE_0(start, "cookie main", "sunw%debug starting main");

              while (!shouldexit) {
Line 20       char         response[MAX_RESPONSE_SIZE];

              printf("give me a COOKIE! ");
              scanf("%s", response);

              if (!strcmp(response, "COOKIE")) {
                  printf("thanks!\n");
                  shouldexit = B_TRUE;
              }

Line 30       else if (!strcmp(response, "loop")) {
                  for (i = 0; i < max_loop; i++) {
                      TNF_PROBE_2(inloop, "cookie main loop",
                          "sunw%debug in the loop",
                          tnf_long, loop_count,          i,
                          tnf_long, total_iterations,    sum);
                  sum++;
                  }
                  max_loop += 2 ;
              }
```

```
Line 40         else if (!strcmp(response, "factor")) {
                    find_factor(sum);
                }

                else {
                    printf("not a %s, ", response);
                }
                }

                return 0;
Line 50     }   /* end main */

            void
            find_factor(int n)
            {
                int i;

                TNF_PROBE_1(factor_start, "factor", "",
                    tnf_long, input_number, n);
                fprintf(stdout, "\tfactors of %d = ", n);
Line 60
                for (i=2; i <= n; i++) {
                while (n % i == 0) {
                    TNF_PROBE_2(found_a_factor, "cookie find_factor", "",
                        tnf_long, searching_for, n,
                        tnf_long, factor, i);
                    fprintf(stdout, "%ld ", i);
                    n /= i;
                }
                }
Line 70         fprintf(stdout, "\n");
                TNF_PROBE_0(factor_end, "factor", "");
            }
```

1

An Example `prex` *Session*

Blank lines inserted for legibility.

```
boulderc% prex
prex: need to specify cmd or pid
usage: prex [options] <cmd> [cmd-args...]
usage: prex [options] -p <pid>
options:
     -o <outfilename>   set trace output file name
     -s <kbytes-size>   set trace file size
     -l <sharedobjs>    shared objects to be preloaded (cmd only)
     -k  operate on the kernel (kernel mode)
boulderc% prex cookie
Target process stopped
Type "continue" to resume the target, "help" for help ...

prex> list sets
$all       'keys'=/.*/

prex> list fcns
&debug    tnf_probe_debug

prex> list probes $all
name=factor_end         enable=off trace=on file=cookie.c line=74 funcs=<no value>
name=found_a_factor     enable=off trace=on file=cookie.c line=68 funcs=<no value>
name=factor_start       enable=off trace=on file=cookie.c line=61 funcs=<no value>
name=inloop             enable=off trace=on file=cookie.c line=35 funcs=<no value>
name=start              enable=off trace=on file=cookie.c line=18 funcs=<no value>

prex> create $factor /factor/

prex> list sets
$all       'keys'=/.*/
$factor    'keys'=/factor/

prex> list probes $factor
name=factor_end         enable=off trace=on file=cookie.c line=74 funcs=<no value>
name=found_a_factor     enable=off trace=on file=cookie.c line=68 funcs=<no value>
name=factor_start       enable=off trace=on file=cookie.c line=61 funcs=<no value>

prex> enable $all

prex> list probes $all
name=factor_end         enable=on trace=on file=cookie.c line=74 funcs=<no value>
name=found_a_factor     enable=on trace=on file=cookie.c line=68 funcs=<no value>
name=factor_start       enable=on trace=on file=cookie.c line=61 funcs=<no value>
name=inloop             enable=on trace=on file=cookie.c line=35 funcs=<no value>
name=start              enable=on trace=on file=cookie.c line=18 funcs=<no value>

prex> list history
[0] enable  'keys'=/.*/
```

```
prex> list values /.*/
enable =
        on
file =
        cookie.c
funcs =

keys =
        cookie
        factor
        find_factor
        loop
        main
line =
        18
        35
        61
        68
        74
name =
        factor_end
        factor_start
        found_a_factor
        inloop
        start
object =
        cookie
slots =

        factor
        input_number
        loop_count
        searching_for
        total_iterations
sunw%debug =
        in
        loop
        main
        starting
        the
trace =
        on
prex> list values name
name =
        factor_end
        factor_start
        found_a_factor
        inloop
        start

prex> list values object
object =
        cookie
```

```
prex> list /.*/ probes $all
     enable=on
     trace=on
     object=cookie
     funcs=<no value>
     name=factor_end
     slots=<no value>
     keys=factor
     file=cookie.c
     line=74

     enable=on
     trace=on
     object=cookie
     funcs=<no value>
     name=found_a_factor
     slots=searching_for factor
     keys=cookie find_factor
     file=cookie.c
     line=68

     enable=on
     trace=on
     object=cookie
     funcs=<no value>
     name=factor_start
     slots=input_number
     keys=factor
     file=cookie.c
     line=61

     enable=on
     trace=on
     object=cookie
     funcs=<no value>
     name=inloop
     slots=loop_count total_iterations
     keys=cookie main loop
     file=cookie.c
     line=35
     sunw%debug=in the loop

enable=on
     trace=on
     object=cookie
     funcs=<no value>
     name=start
     slots=<no value>
     keys=cookie main
     file=cookie.c
     line=18
     sunw%debug=starting main
```

```
prex> connect &debug name=inloop

prex> continue
give me a COOKIE! loop

probe inloop; sunw%debug "in the loop"; loop_count=0; total_iterations=0;
probe inloop; sunw%debug "in the loop"; loop_count=1; total_iterations=1;
probe inloop; sunw%debug "in the loop"; loop_count=2; total_iterations=2;
probe inloop; sunw%debug "in the loop"; loop_count=3; total_iterations=3;
probe inloop; sunw%debug "in the loop"; loop_count=4; total_iterations=4;
give me a COOKIE! factor
     factors of 5 = 5
give me a COOKIE! >^CTarget process stopped
Type "continue" to resume the target, "help" for help ...

prex> clear name=inloop

prex> continue

help
not a help, give me a COOKIE! loop
give me a COOKIE! factor
     factors of 12 = 2 2 3
give me a COOKIE! biscuit
not a biscuit, give me a COOKIE! oreo
not a oreo, give me a COOKIE! cookie
not a cookie, give me a COOKIE! COOKIE
thanks!
prex: target process finished
```

Reading the Trace File

The binary trace file you create with `prex` contains information determined by the `prex` commands you chose (see "Running prex" on page 27).

By default, this file is created in /$*TMPDIR*/`trace`-<pid>, where <pid> is the process ID of the target program. If $*TMPDIR* is not set, the file is created in `/tmp/trace`-<pid>. You can override these default locations with the `-o` option to the `prex` command (see "Options to prex" on page 6 for a full explanation).

The binary trace file can be thought of as a least-recently-used circular buffer. When the file has been filled, newer events overwrite older events.

Once a program is tracing to a file, there is no way to clear the trace file or to give it a different name for the lifetime of that program. This means that if you disconnect from the target and attach later with a different trace file name, the second name is ignored.

Converting the Binary File to Readable Format

To convert the binary trace file to an ASCII file, use the `tnfdump(1)` command and the name of the binary trace file. Because `tnfdump` output goes to `stdout`, you probably want to redirect it into a file.

```
$ tnfdump filename > newfile
```

The `-r` option to tnfdump provides detailed (raw) TNF output. Reading this output requires an understanding of TNF that is beyond the scope of this chapter.

Reading the `tnfdump` *File*

The following tables show the output of part of "`prex cookie`" from the "An Example prex Session" on page 36. The output of the `tnfdump` file is very wide—open a very wide window to display it. Here, for legibility, the output is split into two displays.

```
probe tnf_name: "start" tnf_string: "keys cookie main;file cookie.c;line 18;sunw%debug starting main"
probe tnf_name: "inloop" tnf_string: "keys cookie main loop;file cookie.c;line 35;sunw%debug in the loop"
probe tnf_name: "factor_start" tnf_string: "keys factor;file cookie.c;line 61;"
probe tnf_name: "found_a_factor" tnf_string: "keys cookie find_factor;file cookie.c;line 68;"
probe tnf_name: "factor_end" tnf_string: "keys factor;file cookie.c;line 74;"
```

```
---------------- ----------------- ----- ----- ---------- --- -------------------------- ------------------------
    Elapsed (ms)       Delta (ms)   PID LWPID     TID     CPU Probe Name                Data / Description . . .
---------------- ----------------- ----- ----- ---------- --- -------------------------- ------------------------
        0.000000       0.000000  2269   1         0   - start
     3479.248000    3479.248000  2269    1          0   - inloop                    loop_count: 0 total_iterations: 0
     3487.791000       8.543000  2269    1          0   - inloop                    loop_count: 1 total_iterations: 1
     3489.156000       1.365000  2269    1          0   - inloop                    loop_count: 2 total_iterations: 2
     3490.962000       1.806000  2269    1          0   - inloop                    loop_count: 3 total_iterations: 3
     3492.382000       1.420000  2269    1          0   - inloop                    loop_count: 4 total_iterations: 4
     7559.819000   4067.437000  2269   1         0   - factor_start              input_number: 5
     7561.332000      1.513000  2269   1         0   - found_a_factor            searching_for: 5 factor: 5
     7562.782000      1.450000  2269   1         0   - factor_end
    94445.007000    86882.225000  2269    1          0   - inloop                    loop_count: 0 total_iterations: 5
    94445.549000       0.542000  2269    1          0   - inloop                    loop_count: 1 total_iterations: 6
    94445.572000       0.023000  2269    1          0   - inloop                    loop_count: 2 total_iterations: 7
    94445.593000       0.021000  2269    1          0   - inloop                    loop_count: 3 total_iterations: 8
    94445.615000       0.022000  2269    1          0   - inloop                    loop_count: 4 total_iterations: 9
    94445.637000       0.022000  2269    1          0   - inloop                    loop_count: 5 total_iterations: 10
    94445.659000       0.022000  2269    1          0   - inloop                    loop_count: 6 total_iterations: 11
   100870.335000   6424.676000  2269   1         0   - factor_start              input_number: 12
   100871.663000      1.328000  2269    1          0   - found_a_factor            searching_for: 12 factor: 2
   100871.773000      0.110000  2269    1          0   - found_a_factor            searching_for: 6 factor: 2
   100871.866000      0.093000  2269    1          0   - found_a_factor            searching_for: 3 factor: 3
   100872.751000      0.885000  2269    1          0   - factor_end
```

Looking at the `tnfdump` display, you can see how long it takes to find a factor by subtracting the `factor_start` time from the `factor_end` time. Factoring 5 took 7562.782000 – 7559.819000, or 2.963000 milliseconds. Factoring 12 took 100872.751000 – 100870.335000, or 2.416000 milliseconds.

Inserting Probe Points

Insert probe points in your code to

- Trace the values of variables in your program
- Get internal state information that is useful for debugging or for performance analysis that is not available through the exported interface. For example: Find the information hidden in C++ private classes or performance data such as the collision rate in a hash table. By placing a probe point in the hash table code, the probe point can write to the trace file each time a collision is detected.

The interface for inserting probes is defined by the TNF_PROBE macros: TNF_PROBE_0 through TNF_PROBE_5. The numbers zero through five are the number of variables being traced by the macro.

With these macros, you can insert probe points anywhere in your code to get the values of variables and to trace program execution. The `libtnfprobe` library defines the standard scalar types (ints, longs, floats, and so on), but you can define more complex structures with the TNF_DECLARE_RECORD and TNF_DEFINE_RECORD macros.

Using the TNF_PROBE_* Macros

In the simplest case, TNF_PROBE_0, you give no argument types. The syntax for this is

```
TNF_PROBE_0 (name, keys, detail);
```

name is the name of the probe, following all the syntax guidelines for identifiers in ANSI C. The use of name declares it, so no separate declaration is necessary. This is a block scope declaration, so it does not affect the name space of the program.

keys is a list of groups the probe belongs to. The list is a string containing space-separated keywords and cannot contain a semicolon (;). When any of the groups are enabled, the probe point is enabled. *keys* cannot be a variable–it has to be an inlined string.

detail provides an attribute that you supply to `prex` to select probes and to annotate a probe with a string that is written out to a trace file once.

The detail string is made up of attribute-value pairs separated by a semicolon, although the value is optional. Neither the attribute nor the value can contain a semicolon. `prex` uses spaces to tokenize the value when searching for a match. Spaces around the semicolon delimiter are allowed.

Table 1-6 Attributes Created by TNF_PROBE Use

Attribute	Contents
name	name of the probe, following all the syntax guidelines for identifiers in ANSI C. The use of name declares it, so no separate declaration is necessary. There are no name space issues because this is a block scope declaration.
keys	a list of groups the probe belongs to. The list is a string containing space-separated keywords and cannot contain a semicolon (;). When any of the groups are enabled, the probe point is enabled.
file	file name of the probe (`__FILE__`)
line	line number of the probe (`__LINE__`)
slots	slot names of the probe event (string versions of `arg_name_`*n*)

You can also define your own attributes by assigning an attribute name containing a % symbol (preferred usage is to prefix the attribute name with a vendor stock symbol followed by the % character). In the following example, four attributes are defined: `sunw%debug`, `comX%exception`, `comY%func_entry`, and `comY%color`.

```
sunw%debug entering A; comX%exception; comY%func_entry; comY%color red blue
```

Table 1-7 Examples of User-Defined Attributes

Attribute	Value	Values prex matches on
`sunw%debug`	`entering function A`	`entering` or `function` or `A`
`comX%exception`	`no file`	`no` or `file`
`comY%func_entry`		`/.*/` (regular expression)
`comY%color`	`red blue`	`red` or `blue`

`libtnfprobe` reserves all attribute names that are not prefixed by a vendor symbol (it reserves all attributes that do not have the % character in them) The code for cookie.c in "A Sample C Program" on page 14 contains the following use of TNF_PROBE_0

```
TNF_PROBE_0(start, "cookie main", "sunw%debug starting main");
```

TNF_PROBE_1 through TNF_PROBE_5

The numbers 1 through 5 in the `arg*` names are used here to illustrate the number of variables you give to the probe point. For example, the syntax for TNF_PROBE_1 is

```
TNF_PROBE_1(name, keys, detail,
            arg_type_1, arg_name_1, arg_value_1);
```

and the syntax for TNF_PROBE_5 is

```
TNF_PROBE_5(name, keys, detail,
            arg_type_1, arg_name_1, arg_value_1
            arg_type_2, arg_name_2, arg_value_2
            arg_type_3, arg_name_3, arg_value_3
            arg_type_4, arg_name_4, arg_value_4
            arg_type_5, arg_name_5, arg_value_5);
```

arg_type_*n*, where *n* is a number from 1 through 5, is the type of the *n*th argument. The predefined types are listed in the following table (see "Defining User Types for Probe Points" on page 46 for information about defining your own types).

Table 1-8 Types

Type	Associated C type and Semantics
tnf_long	int, long
tnf_ulong	unsigned int, unsigned long
tnf_longlong	long long (if implemented in compilation system)
tnf_ulonglong	unsigned long long (if implemented in compilation system)

Table 1-8 Types

Type	Associated C type and Semantics
tnf_float	float
tnf_double	double
tnf_string	char *
tnf_opaque	void *

arg_name_*n* is the name you give the *n*th argument. Follow the ANSI C rules for identifiers, and do not use quotation marks around the argument name. (Note that the `slots` attribute mentioned on page 7 contains a string version of this name.)

arg_value_*n* is the expression that is evaluated to a value that is included in the trace file. A read access is done on any variables that are mentioned in value_*n*. In a multithreaded program, place locks around the TNF_PROBE_*n* macro if value_*n* contains a variable that should be read protected.

For example, the cookie.c program on page 14 uses TNF_PROBE_2 as follows:

```
TNF_PROBE_2(inloop, "cookie main loop","sunw%debug in the loop",
              tnf_long, loop_count, i,
              tnf_long, total_iterations, sum);
```

Example

In this example, probe points are placed at the entry and exit of a function to see how much time is spent in the function. The probe at the function entry also logs the arguments to the function.

When `prex` encounters a probe point at run time that is enabled for tracing, it writes a record to the trace file. Each probe point logs the time when it was encountered and also references a tag record containing information like the file name, line number, name, keys, and detail of the probe point.

The first probe point, `work_args`, also logs the value of the two arguments of the probe point (`state` and `message`).

```
#include <tnf/probe.h>

int
work(int state, char *message)
{
    TNF_PROBE_2(work_start, "work_module work"
                "sunw%debug in function work",
                tnf_long, int_input, state,
                tnf_string, string_input, message);
   ...
    TNF_PROBE_0(work_end, "work_module work", "");
}
```

Defining User Types for Probe Points

To trace a structure in your program, define a new type with the TNF_DECLARE_RECORD and TNF_DEFINE_RECORD_*n* macros. These are parts of a compile time interface for extending the types sent in to probe points.

```
TNF_DECLARE_RECORD(c_type, tnf_type);
```

```
TNF_DEFINE_RECORD_1(c_type, tnf_type,
                    tnf_member_type_1, c_member_name_1)
```

Create only one `TNF_DECLARE_RECORD` and one `TNF_DEFINE_RECORD` for each new type you define. The `TNF_DECLARE_RECORD` should precede the `TNF_DEFINE_RECORD`. It can be in a header file that multiple source files share if those source files need to use the `tnf_type` being defined. The `TNF_DEFINE_RECORD` should appear in only one of the source files.

The `TNF_DEFINE_RECORD` macro interface defines a function as well as a couple of data structures. Therefore, use this interface in a source file (.c file or .cc file) at file scope and not inside a function.

Do not put a semicolon after the TNF_DEFINE_RECORD statement. Having one will generate a compiler warning. Compiling with the preprocessor option –DNPROBE (see cc(1)), or with the preprocessor control statement #define NPROBE ahead of the #include <tnf/probe.h> statement, stops probe points as well as TNF type extension code from being compiled into the program.

c_type must be a C struct type. It is the template from which the new tnf_type is created. Not all the elements of the C struct need be provided in the TNF type being defined.

tnf_type is the name given to the newly created type. This interface uses the name space prefixed by the tnf_type. So, if a new type called xxx_type is defined by a library, then the library should not use xxx_type as a prefix in any other symbols it defines.

The policy on managing the type name space is the same as that for managing any other name space in a library: prefix any new tnf types by the unique prefix that the rest of the symbols in the library use. This prevents name space collisions when you link multiple libraries that define new tnf types.

For example, if a library called libpalloc.so uses the prefix pal for all symbols it defines, then it should also use the prefix pal for all new tnf types being defined.

tnf_member_type_n is the TNF type of the nth provided member of the C structure.

tnf_member_name_n is the name of the nth provided member of the C structure.

Examples

This example shows how a new TNF type is defined and used in a probe. This code is assumed to be part of a fictitious library called `libpalloc.so` that uses the prefix `pal` for all its symbols.

```
#include <tnf/probe.h>

typedef struct pal_header {
        long    size;
        char *  descriptor;
        struct pal_header *next;
} pal_header_t;

TNF_DECLARE_RECORD(pal_header_t, pal_tnf_header);
TNF_DEFINE_RECORD_2(pal_header_t, pal_tnf_header,
                        tnf_long,   size,
                        tnf_string, descriptor)

/*
 * Note: name space prefixed by pal_tnf_header should not be used
 * by this client any more.
 */

void
pal_free(pal_header_t *header_p)
{
        int state;

        TNF_PROBE_2(pal_free_start, "palloc pal_free",
                "sunw%debug entering pal_free",
                tnf_long,       state_var, state,
                pal_tnf_header, header_var, header_p);
        . . .
}
```

It is possible to make a `tnf_type` definition be recursive or mutually recursive, such as in a structure that uses the `next` field to point to itself (a linked list). When such a structure is sent in to a `TNF_PROBE`, then the entire linked list is logged to the trace file (until the `next` field is `NULL`). But, when the list is circular, it results in an infinite loop. To break the recursion, either don't include the `next` field in the `tnf_type`, or define the type of the `next` member as `tnf_opaque`.

Advanced Topics

Performance Issues

Don't place probe points in sections of code that are traversed frequently, as in a mutex lock that is hit often.

Estimate about 200ns for each disabled probe on a SS10. You can control the performance degradation of the application by controlling the number and placement of probes.

If you are shipping a library with probe points, it is important to run benchmarks to ensure that the performance is still at an acceptable level. Reduce the number of probes or change their positions to increase performance.

/proc

`dbx`, `truss`, and `prex` all use `/proc` to control the target process. `/proc` allows only one client to control a target process safely. Because of this, you cannot run programs like `dbx` and `prex` simultaneously on the same target program. If you try to run `prex` on a target while `dbx` or `truss` is running on the same target, `prex` displays the message "Cannot attach to target."

You can, however, interleave `prex` and `dbx` execution by following these steps:

1. Start `prex`,
2. Set up the state of the probe points,
3. Give the command "`quit suspend`"
4. Start `dbx`
5. Attach to the suspended program.

The target will not execute any code between `prex` and `dbx`.

You can also suspend the target by sending it a `SIGSTOP` signal and then typing "`quit resume`" to `prex`. If you do this, you should also send a `SIGCONT` signal after invoking `dbx` on the stopped process (or else `dbx` will hang).

`dlopen()` *and* `dlclose()` *and History*

Probes in shared objects that are brought in by a `dlopen(3X)` are automatically set up according to the `prex` command history. When a shared object is removed by a `dlclose(3X)`, `prex` refreshes its understanding of the probes in the target program. This implies that there is more work to do for `dlopen` and `dlclose`, so they will take slightly longer.

If you are not interested in this feature and don't want `dlopen` and `dlclose` to be perturbed, detach `prex` from the target.

Signals

`prex` does not interfere with signals that are delivered directly to the target program. However, `prex` receives all terminal-generated signals (such as ^c (SIGINT), ^z (SIGSTOP)) and does not forward them to the target program.

Use the `kill(1)` command from a shell to signal the target program.

Failure of Event Writing Operations

A few failure points are possible when writing out events to a trace file — for example, system call failures. These failures result in a failure code being set in the target process. The target process continues normally (but no trace records are written).

Whenever a user types ^c to `prex` to get to a `prex` prompt, `prex` checks the failure code in the target and informs the user if there was a tracing failure.

Target Executing a `fork` *or* `exec`

If your program does a `fork`, any probes that the child encounters will be logged to the same trace file. Events are annotated with a process ID, so it is possible to determine which process a particular event came from.

There is a race condition with a thread in a multithreaded program doing a `fork` while the other threads are still running—for the trace file not to get corrupted, make sure that the other threads are quiescent when doing a `fork`, or else use `fork1(2)`.

If the target program itself (not any children it might fork) does an `exec`(2), `prex` detaches from the target and exits. The user can reconnect prex with

```
prex -p pid
```

Building a Library for Shipping

In addition to the usual documentation of the API, there needs to be documentation specifying the names, keys, and arguments of each probe. The semantics of each probe and its arguments also needs to be described.

Error Messages

What they are, what they mean, and how to recover.

1