**Function Execution-Trace Library**

This project started when I decided to write a program to report satellite passes. I have always wanted to be able to see a call graph of what a program is doing, and while a number of packages exist (callgrind and graphviz), however, these are large applications and somewhat overkill (callgrind is a virtual machine emulator and consequently and can run 10 to 50 times slower). As a result, while developing the tracking program, I decided to investigate whether I could come up with a simple solution.

The first approach resulted in a number of simple macros that did achieve what I wanted, which was to be able to draw a simple function call-tree giving the line number in the program where each function was called (a necessary feature since a function may be called from several different locations within its parent function depending on the program conditions). While this solution did work very well, however, it did have a number of limitations, not the least of which was that it required the programmer to adopt a very disciplined approach to the coding rules used when developing the code. For example, while the following common code structure could be used:

If (condition)  
{  
 func ();  
}

Two more frequent alternatives to this are:

If (condition)  
 func ();

and:

if (condition) func ();

Of these, only the second alternative could be adopted. Furthermore, this solution required that every function have a return statement, including void functions. This is particularly onerous on the programmer since missing a return on a void function would never be reported by the compiler and would result in the results being completely meaningless.

Another significant disadvantage is that these macros needed to be manually inserted throughout the entire program code, distracting from the actual program. Clearly this solution was, at best, garbage.

**Compiler Based Solution**

A little more investigation uncovered a little-known feature of the GCC compiler that automatically inserted hooks into each function immediately after the function call and prior to the return. This is the solution adopted by this Execution Trace library. The major advantage of this approach is that it places absolutely no restrictions on the developer, requiring absolutely no modification to the existing projects code base. The interaction with the user’s program is absolutely minimal and handled entirely by the compiler (there is no necessity to even include any header files in the project). To use the Execution Trace library simply requires that the program be linked to the project, and even this process can be automated through the use of makefile flags.

Although this alternative greatly simplified their use with any program and avoided placing any restrictions on the developer, it also has one major disadvantage. The hook functions provide no means of identifying the line number from which a function was called. Furthermore, the compiler does not seem to provide any assistance to solve this problem.

Called-From Line

Although the compiler provides the \_\_LINE\_\_ macro that gives the current line number, that cannot be (simply) used to identify where the function call originated. A trivial compiler-based solution would be for GCC to support an additional macro (e.g. \_\_FLINE\_\_) that was only updated to the current line number immediately prior to entering a function call that was not protected by the no-instrument-function attribute, however, this requires a modification to the compiler code (unlikely given the number of readily available profiling tools available).

Given the importance of the calling line feature, the Call Trace library does provide rudimentary support for this feature, should it be available, by means of the xt\_lineNo variable. To use this, the variable just needs to be somehow set to the line number of the parent (calling) function. If it is not provided, nothing happens.

Note: Called-From line feature is only available in non-real-time mode.

Possible Called-From Implementation

Given there is no clean way to obtain the line where a function is called (that I am aware of), the following technique provides a dirty way to implement the feature. This solution is not very elegant but may be seen as useful in some circumstances. Given the following program:

|  |  |
| --- | --- |
| **Original** | **Modified** |
| #include <stdio.h>   void main (); int square (int v);  void main () {  int res, val;   val = 3;  res = square (val);  printf (“%d squared is %d\n”,  val, res); }  int square (int x) {  return (x \* x); } | #include <stdio.h>  **#define M\_square(x) square(\_\_LINE\_\_,\_\_VA\_ARGS\_\_)** void main (); int square (**int cygln\_\_,** int v);  void main () {**\_**  int res, val;   val = 3;  res = **M\_**square (val);  printf (“%d squared is %d\n”,  val, res); }  int square (**int cygln\_\_,** int x) {\_  return (x \* x); } |

There are three important things to note about this code. Firstly, the parameter used to pass the line number into the function must be called cygln\_\_. Secondly, the underscore character following the opening bracket which copies the line number into the internal global variable cyg\_\_lineNo. Finally, the thing that makes it almost unusable is that it again required the programmer to be very disciplined.

If a function is defined in the normal way as shown in the original code, the line number on which it is stored will not be shown.

Finally, when compiling the program without the Execution Trace library, either still include the xt.h file, or add the following line to the start of every file where functions are defined as shown in the modified code. This prevents the compiler complaining about an unused variable (cygln\_\_) as well as the underscore at the beginning of each function.

#define \_ (void)(cygln\_\_);

**Compiling**

To build the Execution Trace library, the -rdynamic and -finstrument-functions options must be added to the GCC command line in addition to the -ldl option to link to the libdl library. Typically, the compile command would be:

**$** gcc -g -O2 -rdynamic -W -Wall -std=c11 -pedantic -Wshadow -Wcast-qual -Wconversion -Wwrite-strings -fno-builtin -finstrument-functions -o test -ldl -lm test.c

and the link command being:

**$** gcc -rdynamic test.o test2.o xt.o -ldl -lm -o test

**Use of the Execution Trace Library**

The following simple program is used to demonstrate some features of the Execution Trace library. The program is made of two simple C files containing three functions plus main().

|  |  |
| --- | --- |
| #include <stdio.h> #include "test.h"   int main (int argc, char \*argv[]) {  int k, n, r;   (void)(argv);   k = (argc > 1) ? 1 : 3;  n = 5;  r = func1 (n, 0);  r /= 2;  n = func1 (r, k);  r += 6;  n = func1 (r, 2);   return 0; }   int func1 (int x, int f) {  int y;  int \*p = NULL;   y = func2 (x, 2);  y = func2 (y, 3);   if (f != 1) /\* If f == 1, p remains a NULL so next \*/  p = &y; /\* statement causes a seg fault. \*/  \*p = 5;   return 2 \* y; } | #include <stdio.h> #include <unistd.h> #include "test.h"  int func2 (int a, int b) {  func3 ();  return (a + b); }   int func3 (void) {  sleep (1);  return (2); } |
| **File test.c** | **File test2.c** |

A header file (test.h) also exists that contains nothing but the function prototypes for the above files.

To use the Execution Trace library with the above files, use the following commands:

**$** gcc -g -O2 -W -Wall -std=c11 -pedantic -Wshadow -Wcast-qual -Wconversion -Wwrite-strings -fno-builtin -finstrument-functions -c test.c -o test.o

**$** gcc -g -O2 -W -Wall -std=c11 -pedantic -Wshadow -Wcast-qual -Wconversion -Wwrite-strings -fno-builtin -finstrument-functions -c test2.c -o test2.o

**$** gcc -g -O2 -W -Wall -std=c11 -pedantic -Wshadow -Wcast-qual -Wconversion -Wwrite-strings -fno-builtin -finstrument-functions -D XT\_X\_REAL\_TIME -c xt.c -o xt.o

**$** gcc -rdynamic test.o test2.o xt.o -ldl -lm -o test

Note that while the Execution Trace library only consists of two files, neither file is referenced in any way in the main project. The only connection between the library and the project it through the linker with the rdynamic and ldl arguments.

**Usage**

The library may be configured in a number of ways through several global variables in the xt.c file. While some of these could be manipulated under program control from the project it is not recommended. Firstly, doing so makes little sense, and secondly it would compromise the independence of the project file.

Since it makes more sense to set these parameters at compile time, it makes more sense to set most of the more frequently changed parameters through the make file.

The example make file shows how this is done. Generally, the project would be built by simply entering the make command as follows (assuming the make file used is based on the one provided here):

**$** make

To build a project using the XT library simply requires two steps as follows:

1. Copy the two files xt.h and xt.c into the project’s base directory.
2. Enter the make command using the USE\_XT command line option as follows:  
    **$** make USE\_XT=1

When the library is no longer needed, simply rebuild again with the normal command as follows:

**$** make

This will simply re-compile the project without the hook functions and no longer link to the library. It is the USE\_XT flag that compiles the XT library, adds the appropriate flags to the projects compile command and links the project with both the XT and dladdr libraries. (While the flag is set to 1 here, any value will do - see below.)

The following output shows the function call hierarchy of the simple program above linked against the Execution Trace library:

Main  
 func1  
 func2  
 func3  
 func2  
 func3  
 func1  
 func2  
 func3  
 func2  
 func3  
 func1  
 func2  
 func3  
 func2  
 func3

**Fig 1**

A more conventional tree structure, and in fact the one I originally visualised when I started this project, can also be produced. This clearly shows the function call hierarchy as follows:

Main  
├─── func1  
│ ├─── func2  
│ │ └─── func3  
│ └─── func2  
│ └─── func3  
├─── func1  
│ ├─── func2  
│ │ └─── func3  
│ └─── func2  
│ └─── func3  
└─── func1  
 ├─── func2  
 │ └─── func3  
 └─── func2  
 └─── func3

**Fig 2**

To identify the correct relationship and be able to draw the structural hierarchy as shown requires the entire structure be built in memory before being printed. This implies the program needs to run through to the end before the structure is rendered (i.e. the program runs through to the end and then the structure is printed immediately before exiting main).

While it may be pleasing to look at as it clearly emphasises the call relationship between the various functions, it has two significant drawbacks. Suppose a segmentation fault occur during, say, at some point after the fourth call to func3(). In this case the entire program would terminate and be dumped from memory before the execution tree could be generated. Consequently, the diagram could not be used to shed any light on what may have gone wrong. The second issue is that even if the tree structure was available, it does not help isolate the problem to a single function.

A far better approach would be to be able to create an execution tree in real time so that the output would be available when a problem occurred. This can be done by setting a flag at compile time and results in the following diagram in a fault situation.

Main  
 func1  
 func2  
 func3  
  
 func2  
 func3  
  
 func1  
 func2  
 func3  
  
 func2  
 func3  
  
Segmentation fault

**Fig 3**

This is an improvement, but it still does not localise any problems. While the nice tree structure shown in Fig 2 cannot be generated, a more useful call relationship diagram can be generated that will localise any problems to a single function, and possibly even a section of a function.

The alternative call relationship diagram is shown below for the complete program.

Main  
└─── func1  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
┌───┘  
└─── func1  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
┌───┘  
└─── func1  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
┌───┘  
┌───┘

**Fig 4**

The advantage of this diagram is that not only can it can be created as the program is executing in real time, but it also shows where the program execution is at any instant. While perhaps not as attractive as the previous tree diagram, it has the distinct advantage of being able to identify exactly where a problem such as a segmentation fault occurs. Although this type of fault immediately terminates the program the call hierarchy up to the point of the fault would already have been generated.

Using the example in Fig 3, it is clear that the segmentation fault occurs at some point after the fourth call to func3(), however, what is not obvious is whether the fault actually occurred in func3(), func2(), func1() or main().

Examining the execution trace, however, it is clear that the fault occurred, *not* within func3(), as suggested in Fig 3, but within func1(), and after the second call to func2() as shown below.

Main  
└─── func1  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
┌───┘  
└─── func1  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘  
 └─── func2  
 └─── func3  
 ┌───┘  
 ┌───┘

Segmentation fault

**Fig 5**

While this does not identify the exact location of the problem, on a large program it may help localise it to a specific function and section.

**Options**

The Execution Trace library is configured by a number of global variables at the beginning of the file xt.c.

Below is a list of the available global configuration variables defined in the xt.c file along with their description. While these can be set or cleared by a user’s program, in most cases it makes little sense to do that as they are more logically set at compile time. Variables identified with an asterisk can also be set directly from a make file (see below).

xt\_enabled

Set this value to 1 to enable the library, and 0 to disable it. The library can also be disabled un-setting the \_X\_TRACE\_\_ macro (this macro is defined immediately before including the xt.h file). This variable will also be used to disable tracing if outputting the tree to a user defined file but an error occurs.

xt\_realTime \*

Set this value to 1 to enable the real time mode, or 0 (default) to create a traditional call tree.

xt\_traceLines \*

If set to 0, only function calls are printed. Setting this to 1 will also draw the execution trace lines. This includes both calls to functions as well as function returns. This is useful for more precisely locating the exact location of unexpected program terminations.

This option applies real time trace mode only.

xt\_showTree \*

Set this value to 1 will display call tree lines, and 0 to hide them. This is only meaningful in non-real time mode.

xt\_addGaps \*

This variable is also used in non-real time mode only. By setting this value to 1 a blank line will be generated whenever a "block" of functions exit. This is defined as two or more function returns without an intervening function call. This may make the tree more appealing to view.

xt\_timer \*

This feature reports the execution time and is set to 0 (disabled) by default. The time value is shown at the end of the function name.

In non-real time mode, the value is the total time spent inside a function (including the time taken by each sub function it calls). In real time mode the time reported is the total program execution time from the start of the program up to when the function is called (entered).

Setting this variable to 1 reports the amount of CPU time used by each function as reported by the clock() function (this will usually be a lot less than you think). A value of 2 causes the actual elapsed time (as measured by a clock) to be reported.

xt\_treeType

This variable specifies the type of lines used to draw the call tree. The current options are for light lines (normal), heavy lines, or double lines. These line types are defined in the XTType enumeration (xt.h) as 0, 1 or 2 respectively. The default is 0.

xt\_pTreeCol

This variable is a pointer to a const char string specifying the ANSI codes that will be used to render the tree line color of the call tree. Standard color definition strings are specified in the header file (xt.h), however, any ANSI string can be used.

This only applies on non-real time mode.

xt\_pNameCol

This variable points to the ANSI codes that specify the color that will be used to print the function names. As before, any of the standard ANSI color control strings may be specified.

xt\_pOutputFile

This is a pointer to a character string specifying the name of the output file where execution trace is to be sent. If this variable is set to NULL, the call trace output is sent to the standard error path which may be re-directed if required (this will generally be fine for non-interactive programs). It is useful for interactive programs or where other output is being generated on the standard error path.

By default, this variable is set to NULL.

**Options Available via Make File**

At the top of xt.c file there are also a number of macros that extend the control of these parameters to the make file to allow the library to be configured immediately from the make command line rather than requiring the user to edit the xt.c file whenever the configuration needs to be altered.

|  |
| --- |
| **Note on Using Make Files to Define Parameters**  Using the C compiler, it is possible to pass macros on the command line using the -D option. This is the equivalent to the C pre-processor #define directive in a .c or .h file. Consequently, macros defined on the command line can also support optional arguments. (i.e. the command line string -D TEST is equivalent to the line #define TEST in C source code and by default it is assigned a value of 1 by the compiler, while -D TEST=5 is the same as #define TEST 5.)  Macros can also be passed into the make file from the command line in a similar way, however, in contrast with the C compiler which allows macros to be defined but not set, make always requires macros to be entered as name/value pairs, such as TEST=5. Entering a macro on the make command line without a value will simply leave the macro undefined.  As a result, to define a macro on the make command line that is to be passed into the compiler as simply -D TEST, the macro must be entered with an associated value, even though the value will not be used. |

To use the options listed below when building a project with the XT library using the make file provided, use the following format of make command:

**$** make USE\_XT=1 REAL\_TIME=1

The following table describes the various parameters that are available from the make file supplied. While many do not take an argument, a value will always need to be passed in to the make file. The final column identifies whether the value is used or not. When a value is used, it must be as described in the

|  |  |  |
| --- | --- | --- |
| **Macro** | **Description** | **Arg used** |
| REAL\_TIME | Enable realtime mode. | No |
| TRACE\_LINES | Include to draw execution trace lines in real time mode,h | No |
| SHOW\_TREE | Display the execution tree lines in non-real time mode. | No |
| ADD\_GAPS | Add additional gap line after a block. | No |
| TIMER | Show timing values at end of function. Argument values are:  0 – Disable timer (default)  1 –CPU process execution timing   2 – Elapsed time (clock time) | Yes |