console-dev.de

Home of VisualHAM, N3D and HEL Library

« sine approximation with fixed point math part 2

Private:

introduction

I was experimenting how I could retrieve a stack trace on the Nintendo DS, using the devkitARM compiler, recently. A callstack is a list of active subroutines of a computer program. This can be used to identify from which code-path a particular subroutine was called, which is enormously helpful when detecting logical errors at run-time.

I tend to flood my code with "ASSERT's". ASSERT is used to identify logic errors during program development by implementing the expression argument to evaluate to FALSE only when the program is operating incorrectly and then calling a function to report the error:

```
1 void Vector2Add(Vector2 *dest, const Vector2 *v0, const Vector2 *v1)
2 {
3   ASSERT(dest != NULL);
4   ASSERT(v0 != NULL);
5   ASSERT(v1 != NULL);
6
7   dest->x = v0->x + v1->x;
8   dest->y = v0->y + v1->y;
9 }
```

When any of the incoming arguments point to NULL, ASSERT will call a subroutine to report the error. Typically the report includes the filename, line number and functionname. This information is helpful, but lacks of information from where the function was called.

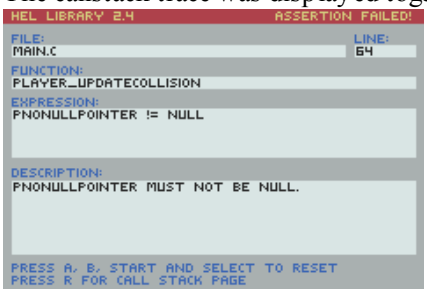
I use Vector2Add all over the place, how should I know what part in my program code does not work, when I only know at some point a NULL argument is passed to Vector2Add?

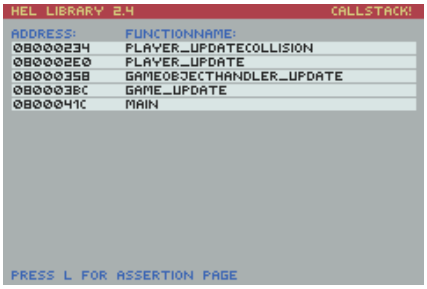
Right, I need to know from where Vector2Add was called, I need a callstack trace!

my earlier instrumentation approach

Around 2005-2006 I implemented a callstack trace feature in HEL Library. HEL Library is a middleware solution to ease development of Game Boy Advance titles.

The callstack trace was displayed together with an assertion-report, that looked like:





Back then I used function instrumentation (-finstrument-functions) to build a list of subroutine calls at run-time. It was basically a callback that was called for every function that is about to be entered (__cyg_profile_func_enter) and left (__cyg_profile_func_exit).

This was a huge overhead and slowed down program-execution dramatically, but the callstack trace was so precious, that the performance penalty didn't matter for the debug library.

fail does not mean to stop

When I started the Nintendo DS version, one goal was to use an approach which does not have any impact on run-time performance! I have tried several libraries, unfortunalety I didn't find anything that worked for me.

This includes libunwind, backtrace, several GCC builtin-keywords to get return addresses of different calldepths, just to name a few.

I gave all up. It was either a never ending story to integrate the library in to the project or headers / functions were missing in libraries that come with devkitARM.

hacking to success, then fail

Rather than giving up, I decided to accept it as challenge and come up with my own callstack trace code. It was a long and rocky journey, but at the end I had something that worked suprisingly well.

What I do is basically pretty simple, I interpret instructions of interest to simulate the beaviour of the Program Counter (PC) and Stack Pointer (SP). The Program Counter indicates where the program is in its intruction sequence. Stack Pointer indicates the current top of stack memory. The stack this is where local variables are located.

Armed with my own Program Counter and Stack Pointer variables, I can traverse program code and react when Program Counter is assigned a new value, which basically means to jump to a different address and continue to operate there.

Example:

```
1  void FirstFunc()
2  {
3    SecondFunc();
4  }
5
6  void SecondFunc()
7  {
8    // Program Counter is located here  <--
9    ThirdFunc();
10 }
```

Let's presume the Program Counter is located in SecondFunc, which was previously called by FirstFunc.

The thumb assembler version of this looks like:

```
1  FirstFunc:
2    push {lr}        ; Push Link Register (LR) on stack
3    sub sp, sp, #4   ; Update Stack Pointer by 4bytes because LR was pushed
4    bl  SecondFunc   ; Call to SecondFunc()
5    add sp, sp, #4   ; Update Stack Pointer to point to pushed LR again
6    pop {pc}         ; Pop LR from stack and store in Program Counter, this will return to caller
7
8  SecondFunc:
9    push {lr}        ; Push Link Register (LR) on stack
10   sub sp, sp, #4   ; Update Stack Pointer by 4bytes because LR was pushed
11   bl  ThirdFunc    ; Call to ThirdFunc()
12   add sp, sp, #4   ; Update Stack Pointer to point to pushed LR again
13   pop {pc}         ; Pop LR from stack and store in Program Counter, this will return to caller
```

I need to get my hands on the Link Register (LR) to return to SecondFunc's caller (*1). Since LR was being pushed on the stack, I know where to look! What is left is to traverse the program code in reversed order and interpret the "sub" and "push" instructions.

Moving the Program Counter in reversed order, which actually would be a backtrace imo, didn't work out for some reasons and I wasn't able to solve those problems. It did, however, worked in some cases, unfortunalety too unreliable.

when in no doubt, try it out

I stayed away from this problem for week and then had the idea why not trying to simulate the Program Counter the same way it would walk along when it continues program execution normally. It means rather then traversing program code backwards, I tried what happens, when I forward advance PC. Now I have to interpret "add" and "pop" instructions and this works suprisingly well!

From here on it took a few hours to come up with a version that is robust enough to work in all places that I tested with my code base, which consists of both, C and C++ source code.

Obviously it does not mean it always works in every sitiuation with all source code in the world. For example, optimization level -O0 generates so many unconditional branches and places return code all over the place, that the *Stacktrace* function fails. It supports thumb code only and has problens with hand written/optimized assembler routines (when LR is not push'ed and PC not pop'ed).

I added a couple of more instruction *Stacktrace* looks out for:

```
1 pop {...pc} ; adjust PC to the pop'ed value
2 add sp, nn  ; adjust SP by nn
3 sub sp, nn  ; adjust SP by nn
4 b   ofs     ; set PC to the addr PC+ofs
5 add sp, rn  ;  adjust SP by the value rn points to
```

The unconditional branch instruction "b" is from hell, because it could lead to an infinite loop in *Stacktrace*, such as *while(1) {}*. For this reason I added two special cases:

- Limit how many instructions are allowed to touch
- Branch only to the target address when it's different from PC

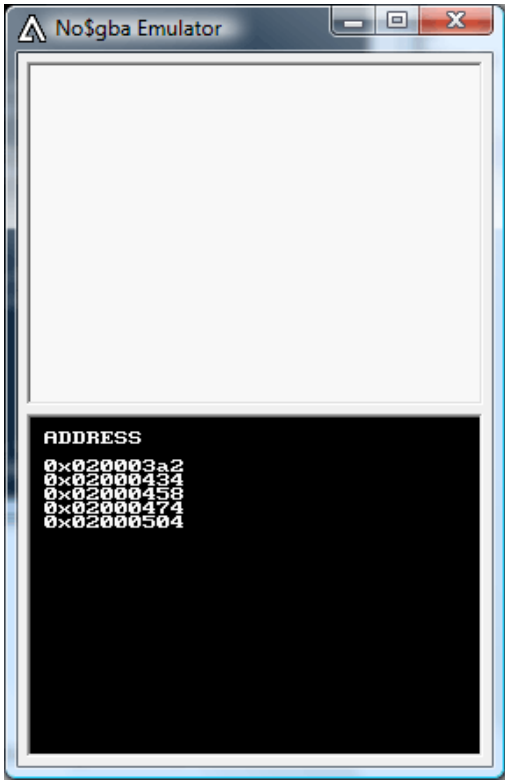
When any of the conditions evaluates to true, *Stacktrace* aborts and outputs the callstack only to this position. It will, however, report that an error occured.

Support for the unconditional branch was important, because the compiler generates code like this sometimes:

```
1  ExampleFunc:
2    push    {lr}
3    ...
4    tst     r0, #2
5    bne     .L92 ; if r0 is not 2 then jump to L92
6    ...
7  .L90:
8    pop     {pc} ; returns to caller
9
10 .L92:
11   bl      AnotherFunction
12   b       .L90 ; jump to L90 (above)
```

When we return from *AnotherFunction* we must jump to *L90* to pop the return address from the stack and return to the caller. This is the reason why I added support for unconditional branches. I haven't noticed other branches that are of importance for *Stacktrace* yet.

At this point of time I was able to return to each function-caller and have the return addresses, e.g:



resolving function names by hand

In order to map the memory address to an actual function, we have to look up the address in the so called Map file. Map files are typically plain text files that indicate the relative offsets of functions for a given version of a compiled binary. This file is generated by the linker when you pass *-Map* to it, here is a part of it:

```
1 .text    0x02000310    0xdf28
2 .text    0x02000380    0x1ac main.o
3          0x02000394      PrintStacktrace
4          0x0200046c      FirstFunc
5          0x0200040c      ThirdFunc
6          0x020004c8      main
7          0x02000450      SecondFunc
```

Line 1 specifies where the .text section (program code) starts and its size. Line 2 specifies the start address of the object file *main.o* followed by all functions it contains. While writing this article, I have learned *static* functions are not listed in the map file.

In order to find the function name of the memory address *0×02000458* (from screenshot above), we have to check in which range it is located. But it would be too simple when the .map file has everything sorted by addresses, so we have to do this instead.

Here is a sorted list by addresses:

```
1 .text   0x02000380    0x1ac main.o
2         0x02000394      PrintStacktrace
3         0x0200040c      ThirdFunc
4         0x02000450      SecondFunc
5         0x0200046c      FirstFunc
6         0x020004c8      main
```

The memory address *0×02000458* is between:

```
1         0x02000450      SecondFunc
2         0x0200046c      FirstFunc
```

So *0×02000458* belongs to *SecondFunc*. While this approach seems to work, it is complicated and really only works when you have the exact .map file available that was used to build the project. You can't use a .map file of a different version of your code, this is very unlikely to work.

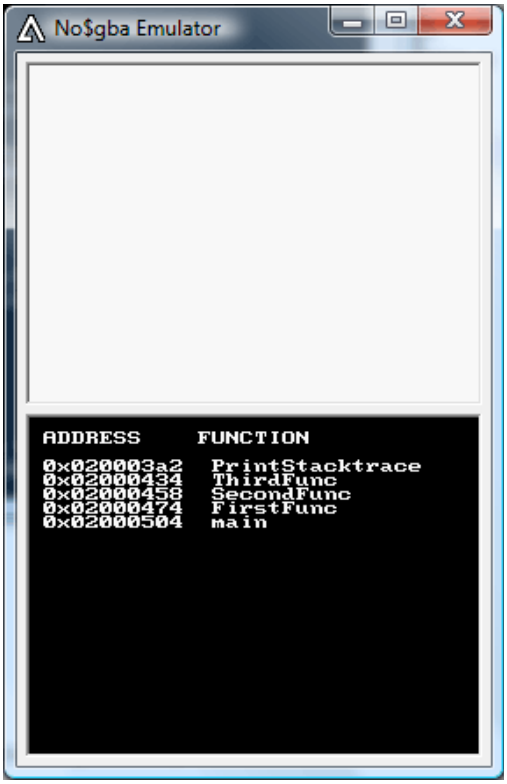
It would be so much better when the actual function name could be displayed instead!

what is your function name, dear memory address

Fortunalety, the compiler has an option to insert function names in plain-text infront of each function. This can be activated by adding *-mpoke-function-name* to the *CFLAGS* variable in your makefile.

But it not only inserts the name, it also inserts a bit-pattern to recognize that there is a function name as well as a relative offset to the name from this pattern.

What *Stacktrace* function does, it takes a memory address and then decreases this address until it detects the bit-pattern of the function name and we have it!



Well, there is of course a special case again. When *-mpoke-function-name* has not been specified, the function name bit-pattern is missing, thus *Stacktrace* would try to find it until it reaches the start of the text section. While this makes little sense, I added a limit how many 32bit words the "GetFunctionnameTag" function is allowed to touch before it returns "did not find".

how to integrate in your own project

The stacktrace.zip archive contains the entire source code, you only need those two files:

- stacktrace.h
- stacktrace.c

Copy them to your source code directory. Then open the makefile file which should be located in your project directory also. You have to adjust the following variables:

```
1 ARCH   := -mthumb -mthumb-interwork
2 CFLAGS := -O2 -mpoke-function-name
```

Now do a rebuild, rather than a regular build. You can do this by performing a "make clean" first, then a "make". Don't forget to include *stacktrace.h* in that file where you want to use *Stacktrace*.

Optimization level -O2 worked best with *Stacktrace* so far. When you use more aggressive optimization levels the compiler also starts to inline code heavily, which can be quite confusing when the *Stacktrace* output does not reflect what you see in your high level code.

stacktrace interface and usage

The usage of *Stacktrace* function is fairly simple.

All it expects is an array of *StacktraceEntry* elements and the count of it. Here is the interface from stacktrace.h:

```
typedef struct
{
  unsigned int addr;  //!< Return address
  const char *name;   //!< Name of function
} StacktraceEntry;

unsigned int Stacktrace(StacktraceEntry* dest, unsigned int count);
```

Use it like this:

```
void OutputStacktrace()
{
  unsigned int n;
  unsigned int count;
  StacktraceEntry entries[32]; // allocate memory for 32 stacktrace entries

  // get only the 10 deepest stacktrace entries
  count = Stacktrace(entries, 10);

  for (n=0; n<count; ++n)
    printf("0x%08x: %s", entries[n].addr, entries[n].name);
}
```

improving stacktrace

If you want to improve *Stacktrace*, I recommend to read through the comments in *stacktrace.c* and enable *STACKTRACE_VERBOSE*, by setting it to 1.

This will, as far as *OutputDebugString* (also in stacktrace.c) uses a debug message type your Nintendo DS software emulator understands, output what instructions *Stacktrace* comes across and interprets.

I've added an opcode structure for all instructions *Stacktrace* interprets as well as print-routines to dump the contents.

I have used no$gba during development, which was of great help because of its "debugger". Usually I yell about only see the assembler code in the shareware version of no$gba, but for this project it was exactly what I needed

conclusion

The *Stacktrace* function…

- works suprisingly well for being such a hack
- does not need debugging information/symbols
- it runs directly on the target device, not needed to be connected to GDB or a similar debugger
- does not come with any run-time performance penalty
- works best with optimization level -O2
- is most informative when adding -mpoke-function-name to CFLAGS in your makefile
- comes with full documented source code
- uses very little Nintendo DS specifics, except the text section start and length (portable to other ARM devices?)
- works with thumb code only
- does not detect when switching from thumb to arm mode
- depends on LR / PC being push'ed and pop'ed, hand-written assembler routine can cause trouble
- is tested in/with my code base only

frequently asked questions (faq)

Q: Why do I see so weird function names, like *_ZN13DEMOPARTQUEUE9CutToPartEj* and the like?

A: Because in C++, the function names go through name mangling. Name mangling is a technique used to solve various problems caused by the need to resolve unique names for programming entities.

download

Dear visiter, 100% OFF on console-dev.de, take your chance and download stacktrace.zip now!

what I didn't tell you yet

*1) LR does not always contains the address from where it was being called, but it holds the address where to return when the function completes. When using code optimization levels above -O2 the compiler e.g. generates code that does not return to functions where it was the last instruction involved anyway.

This entry was posted on Thursday, August 13th, 2009 at 8:13 pm and is filed under GBA, NDS, Programming. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site. Edit this entry.

Leave a Reply

Logged in as Peter Schraut. Log out »

Submit Comment

---

console-dev.de is proudly powered by WordPress
Entries (RSS) and Comments (RSS).