21:13 Solving the Load Address; or, Fixing Useless Firmware Disassembly

by EVM

Our Objective

In this article, I present a little trick for determining the load address in a processor's memory space for a piece of embedded code when the load address is not on a nice clean boundary. Oftentimes, the load address is easy to figure out, but in certain cases it might not be, such as when Flash code is copied into RAM or when a firmware update file contains code for multiple processors.

The concept is to load the code at 0, locate all the absolute function calls, and sort them starting from the lowest address. Then choose the two lowest function addresses, f_1 and f_2. Calculate $d = f_2 - f_1$. Scan through the functions starting from 0, look for the first pair of functions that are offset by d, and call these s_1 and s_2. The load address is then determined by calculating $f_1 - s_1$.

The reader convinced of the need for such a technique will desire to know *why* this works, we'll cover that in a moment. First, we'll remind other readers how much of a pain finding a firmware load address can be.

Motivation; or, Why Bother?

The problem we're dealing with is determining the load address for a piece of firmware that doesn't load on a round number boundary. This problem seems to primarily arise in two situations.

PRICE LIST SENT FREE UPON APPLICATION

RIEGER'S

Monogram Whiskey

Purity and age guarantee **Good Whiskey.** Rieger's Monogram is absolutely pure and wholesome. **Guaranteed under the Pure Food Laws.** Its exquisite, smooth, mellow flavor has made it a lasting favorite with over 100,000 satisfied customers. We are U. S. Registered Distillers (Distillery No. 360, 5th Dist. of Ky.) Why pay exhorbitant prices, when you can buy Rieger's Monogram Whiskey at the regular wholesale dealer's price and save money by ordering your goods shipped direct?

WE PREPAY ALL EXPRESS CHARGES

8 Qts. RIEGER'S MONOGRAM PRIVATE STOCK $5.00

4 Qts. RIEGER'S MONOGRAM EXTRA FINE $3.00

FREE WITH EACH ORDER Two sample bottles of Rieger's Fine Monogram Whiskey. Gold-tipped Whiskey Glass and Patent Corkscrew.

No Marks on Packages to Indicate Contents

Send us an order, and when you get the Whiskey, test it for flavor, smoothness, and all the essentials of GOOD Whiskey. Compare it with other Whiskies (no matter what the price); test it for medicinal purposes; let your friends try it; use half of it if necessary to satisfy yourself on these points—then if you are not thoroughly convinced that "Rieger's Monogram" is as good as any Whiskey you ever drank, return the balance to us, and we will pay return charges and at once send you every cent of your money.

J. RIEGER & CO. *1512 Genesee Street KANSAS CITY, MO.*

The first situation is when a piece of firmware is written to run mostly from RAM. This firmware will generally have a small stub at the beginning that copies the code from Flash into RAM and then jumps to it.

The second situation is when you have a firmware update file that is used to update multiple processors on a system. (For example, a drone that's got a handful of processors but only one update process.) So we might be able to clearly identify a blob within the file as belonging to a particular CPU, but we don't know how that blob ends up loaded in the processor's address space.

Many times it's pretty easy to solve the load address by just doing a disassembly and guessing a round number as the base address for the code. For instance, you may try to load the image at address 0, which IDA does by default, and then realize that all of the absolute addresses of functions begin with `0xC0`. This works most of the time because sections of memory often start on a nice round number in their address space. If we see `sub_10`, `sub_24`, `sub_3A` and then absolute references to `0xC00010`, `0xC00024`, and `0xC0003A`, it's probably safe to assume that `0xC00000` is the load address.

However, it's not always that straightforward. (Figure 13.) What sometimes happens in Situation 1 is that the image might get loaded at some offset that's not a round number. In Situation 2, what can happen is you may not know exactly where the code begins, so it may load on a round number but you may not know exactly which part of the code gets loaded on the round number.

The main symptom of loading at the wrong load address is broken absolute calls, which results in some form of bad disassembly. Depending on the architecture or the size of the code, there may not be that many absolute calls, so the failure may be subtle. A secondary symptom of loading at the wrong address is that a disassembler will fail to recognize large sections of code as subroutines because it does not look like any code is calling functions at those addresses.

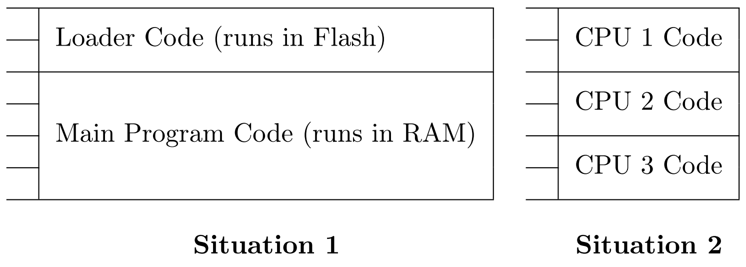


Figure 13: (Left) Situation 1: a program with an initial stub that copies the rest of the code into RAM and executes it. (Right) Situation 2: multiple pieces of code within a single firmware update file.

Our Clinical Trial

In an era where science is sometimes forgotten, it's helpful to look at a specific example as a study. For our clinical trial to determine technique efficacy, we'll look at a SH-2 microprocessor image that is an example of Situation 1. The code is initially run from Flash, but then most of the code was copied to RAM and run from there. The RAM code was loaded at a round number, but the RAM code began several functions into the binary image, which caused the disparity. I use IDA Pro for this example, but any other disassembler could just as easily be used.

Relative vs. Absolute Calls

Most processor architectures contain both absolute and relative jump and call instructions. In most programs, most of the jump and call instructions are relative. Relative means that the address of the target function is encoded within the instruction as an offset (usually from the start of the next instruction). Relative jumps and calls are position independent, meaning they can be moved around and will work correctly. They will disassemble to correctly target the right offset wherever they might be loaded.

Figure 14 shows an example of a relative call instruction (branch to subroutine) in SH-2. The processor makes the calculation: `0x401C32 + 4 + 2*(0x209) = 0x402048`, since `0x209` is the offset in 2-byte words, and the program counter value is actually four bytes ahead of the beginning of the instruction. (This is because it is just past the branch delay slot, which I'll just skip over explaining for now!)

However, you might notice that for SH-2, the maximum offset for the branch-to-subroutine instruction is ±`0x7FF`. This means you cannot jump more than 4K away from your current position, which is a problem. As such, we need to be able to specify the absolute address for a call. An absolute call (also known as a far call or far jump) specifies the full (in this case, 32-bit) address of the target function. Figure 15 shows an example of an absolute call instruction (`jsr`) in SH-2.

The processor makes the calculation `0x40083A + 4 + 2*(0xD) = 0x400870`, and this address is then dereferenced to get the full 32-bit address of the function, in this case `0x409FD8`.

You'll notice that we need to look at absolute calls for clues to where the program is loaded. From this one absolute call example above, we can see that the code is likely loaded somewhere in the area of `0x400000`.

Epson QX-10™? Graphics?

Q PLOTTER

GRAPHIC POWER FOR THE EPSON QX-10™

with one key . . .

■ Draw Circles ■ Squares
■ Lines ■ Axis, etc.

also zooming and panning, multiple screens, animation and much more . . .

Q Plotter also turns your Mbasic into a high powered graphics basic.

introductory price **$175.00**

To order call toll-free 1-800-824-7888 Operator 409
For technical information or dealer nearest you
Call (602) 747-0005 or Telex 709234

Metro Software Inc.
5648 E. Broadway Blvd., Dept. B, Tucson, Arizona 85711

QX-10 is a TM of Epson America • Mbasic is a TM of Microsoft

```
ROM:0401C32   B2 09           bsr sub_402048
```

Figure 14: An example of a relative call instruction (branch to subroutine) in SH-2.

```
ROM:0040083A   D3 0D          mov.l    #sub_409FD8, r3
ROM:0040083C   43 0B          jsr      @r3 ;sub_409FD8
...
ROM:00400870   00 40 9F D8    dword_400870:    .data.l h'409FD8
```

Figure 15: An example of an absolute call instruction in SH-2. Note that the full 32-bit target address is embedded as data just after the instruction.

Walkthrough: Measuring the (Social) Distance

In this example, IDA did an initial disassembly with the code loaded at 0. We see the following in the Functions window:

```
sub_0
sub_2A
sub_34
sub_66
sub_8A
sub_9C
```

Most of the absolute calls are in the `0x400000` range and are simply broken. Rebasing the image to `0x400000` (the obvious guess) makes those calls point to valid addresses; however, the calls land at seemingly random code addresses, in the middle of functions.

So here's the trick: we ask IDA to give us all of the absolute calls by doing a text search for the instruction, such as `jsr` for SH-2. We then sort these by the instruction. IDA helpfully labels the `jsr` instructions with their target functions, so this sorts these instructions by target function, with the smallest target function at the top:

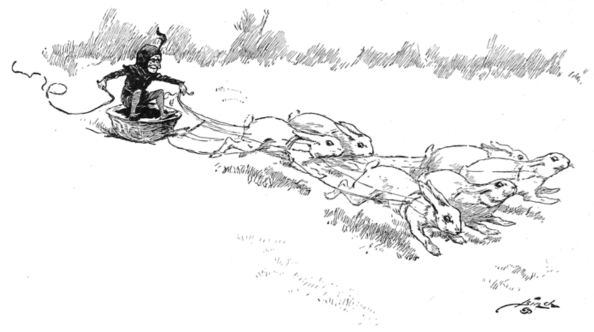
```
jsr @r2; sub_400000
jsr @r2; sub_400000
jsr @r2; sub_400036
jsr @r2; sub_400036
...
```

I'm oversimplifying here because the compiler will use different registers, so you may need to sort and then look for the low addresses used for each register. Regardless, the objective remains as locating the lowest two functions referenced by absolute calls.

Next, we choose the two smallest target functions. We calculate the difference (d) between the two functions. We then scan starting from the smallest function which IDA lists and look to see whether there is a function defined at distance d. For this example, $d =$ `0x36`. Scanning through the beginning of the functions list, we see `sub_66` and `sub_9C` are `0x36` apart. This means that `sub_66` gets loaded at `0x400000`, and consequently `sub_9C` gets loaded at `0x400036`.

See Figure 16 for some code to help with this if you use IDA. In this example, `sub_0` was an entry vector in the Flash code that eventually called `sub_34`, an initialization function that copied the Flash program beginning at offset `0x66` to `0x400000` in RAM.

Using our newfound knowledge, we can do two things. Either we can rebase the entire image to `0x3FFF9A`, or instead we can reload the input file, creating a new segment for RAM, telling IDA to start at offset `0x66` in the file. (Rebasing to an address beneath the section isn't technically correct, but it's quick and easy.)



```
define find_offset_functions(delta):
  f = idc.get_next_func(0)
  while (f != idc.BADADDR):
    f2 = idc.get_next_func(f)
    while ((f2 != idc.BADADDR) and (f2 - f <= delta)):
      if (f2 - f == delta):
        print("Functions %s and %s are 0x%x offset" %
              (idc.get_func_name(f), idc.get_func_name(f2), delta))
      f2 = idc.get_next_func(f2)
  f = idc.get_next_func(f)
```

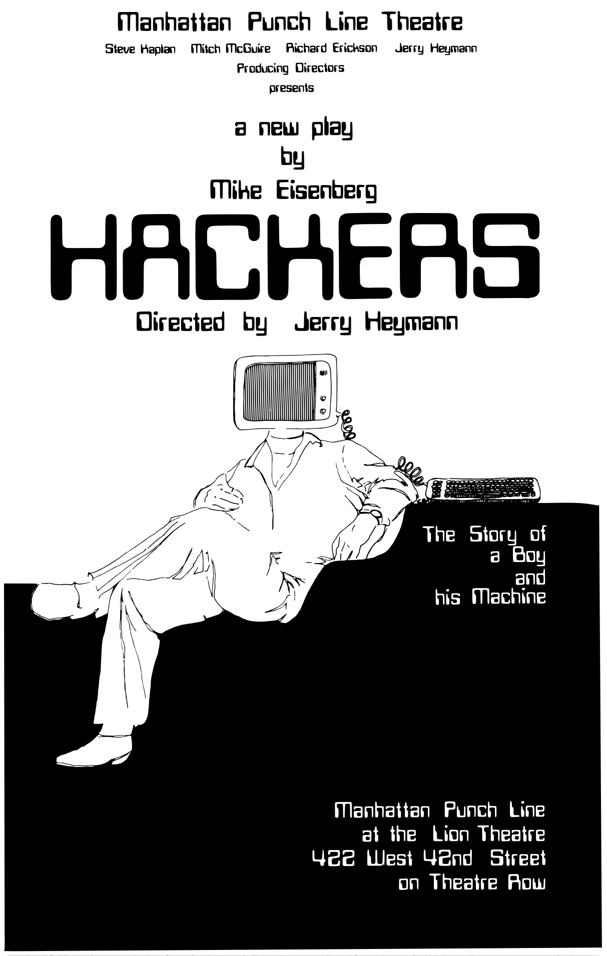
Figure 16: IDA Python code to print a list of functions offset by a given amount.

With whom might we share this?

In addition to matching deltas between absolute function calls, this process can be similarly applied in other situations where differences between objects in the binary can be matched to differences between absolute addresses in the code.

This approach has been successfully employed on a processor with separate code and data segments to determine the load address of the data segment. In this case, a function had been identified which was presumed to be a debug print function, and it took the address of a string as the first parameter. The load address was determined by dumping all of the first parameters in calls to the debug function, and then dumping all string offsets in the file, and matching the differences in those lists.

It seems possible to automate this approach by treating the two lists of offsets as signals and running a correlation function on both of them to determine the best match. The difficulty to automation would most likely be ensuring that disassembly is clean and accurate, and that a majority of subroutines are properly identified, even when the code is loaded at the wrong address.

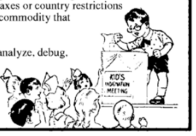

Manhattan Punch Line Theatre
Steve Kaplan Mitch McGuire Richard Erickson Jerry Heymann
Producing Directors
presents
a new play
by
Mike Eisenberg
HACKERS
Directed by Jerry Heymann
The Story of a Boy and his Machine
Manhattan Punch Line
at the Lion Theatre
422 West 42nd Street
on Theatre Row

The Age Of Personal Reverse Engineering has arrived!

Solved: That when tongues turn white, breath feverish, stomach sour and bowels constipated, that our mothers give us tiny portions of love and sugar, we claim pills and shells in exotic architectures in order to port the thing everywhere.

No need to wait more for this to happen! The era of personal reverse engineering has finally arrived. No taxes or country restrictions involved! Free radare2 licenses is a commodity that everybody can enjoy

With radare2 you can disassemble, analyze, debug, patch any binary for a wide range of CPUs and OSs even for your shiny 4004 running PC/M!



AT LAST...

SOFTWARE THAT TEACHES READING

PAL is the only diagnostic/remediation program ever written for reading education. PAL actually diagnoses the cause of reading problems, and provides remediation directly targeted at those problems.

PAL covers the entire scope and sequence of reading education for each grade 2 through 6, and evaluates up to 40 major skills and 160 subskills per grade level.

The **PAL MASTER DISK PACKAGE** (required for use with the Curriculum Packages) operates the PAL system. It includes an upper/lower case chip for the Apple II, so that lessons are presented in a 'real world' format. $99.95.

The **PAL READING CURRICULUM PACKAGES** provide the diagnosis and remediation. $99.95 per grade level. A two-disk demonstration package is available for only $9.95.

If you are uncertain about which grade level to purchase for your child, order the **PAL PLACEMENT TEST** (includes a $10.00 coupon good on your next PAL purchase). $29.95

THE WAY TEACHERS WANT READING TAUGHT.

System Requirements: Apple II with Applesoft, 48K RAM, one or two disk drives.
VISA, Mastercard, checks, COD accepted. Colorado residents add 3% sales tax.
Universal Systems for Education, Inc.
2120 Academy Circle, Suite E
Colorado Springs, Colorado 80909
(303) 574-4575