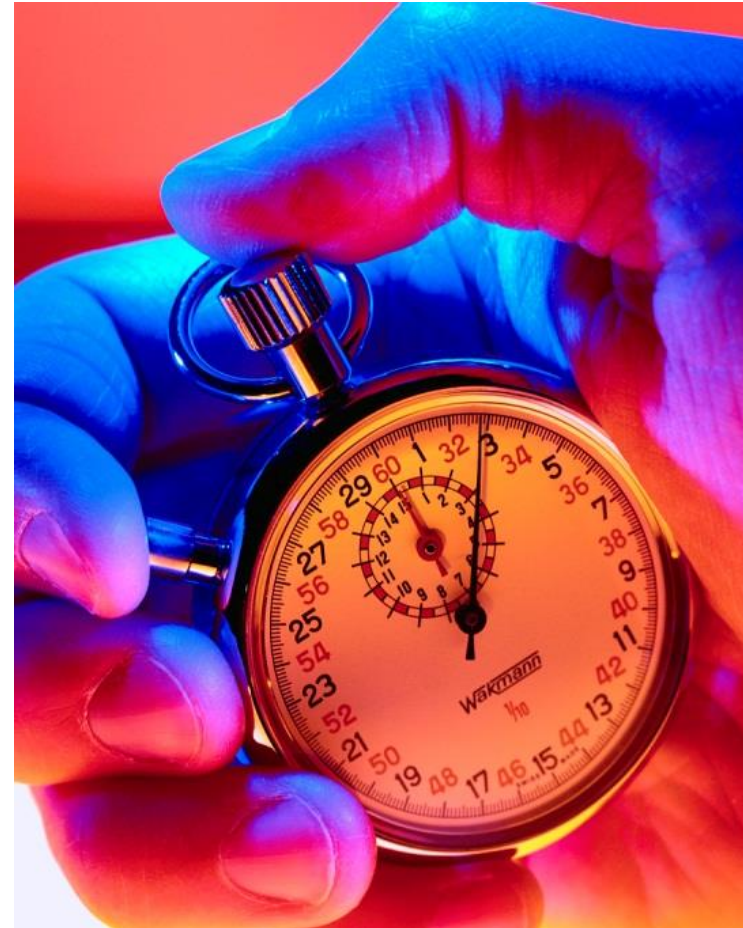


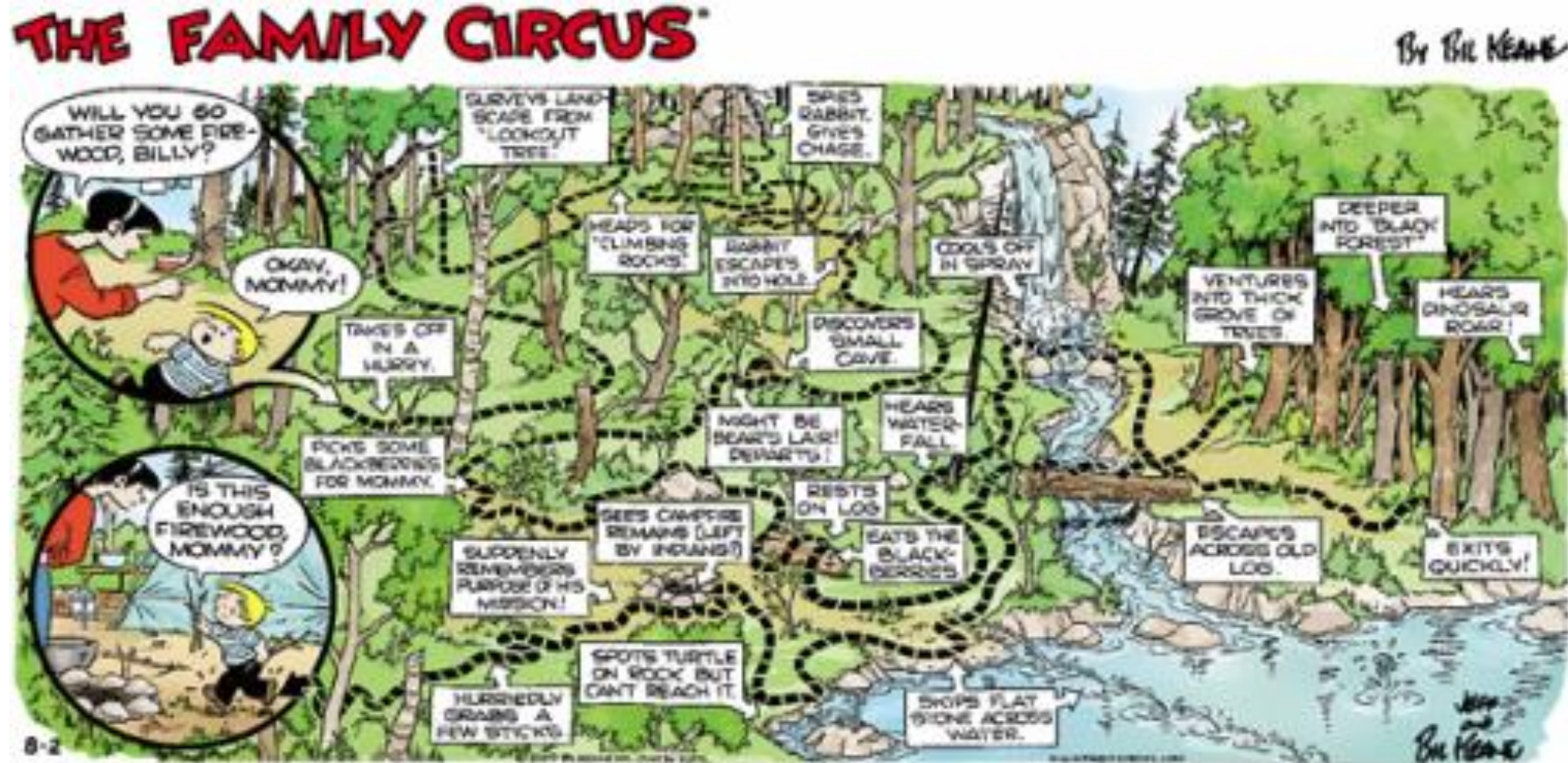
# Profiling Code for Execution Time

# Overview

- Which part of the code uses the most **time**?
- That's the best place to start optimization for speed, as it has the largest impact



# How Does The Program Spend Its Time?



- **What** is the program really doing? Anything unexpected or extra?
- **How** is the program doing it? Is it reasonably efficient?
- The vital few, the trivial many *J & B. Keane, King Features Syndicate*
- The 80/20 rule
- Pareto Principle, Juran's Principle



# Optimization Approaches: Which Book to Use?

Optimizing Your  
ECE 561  
Project 1 Code  
for **Speed** in  
*Just Two Hours*  
A.G. Dean

- Ideal: Service manual
  - Diagnostic manual customized to your program
  - Does not exist!

- Detective story
  - Filled with clues, but also many distractions (“red herrings”)
  - Want to see the clues and nothing else

A Day in the  
Life of Your  
ECE 561  
Project 1 Code  
N. Author

- Cookbook
  - Many different possible optimizations,
  - But which should be applied, and where?

- Resulting approach
  - Use **detective story** to find clues and identify performance problems
  - Use **cookbook** to solve performance problems

A Cookbook of All  
Possible Code  
**Optimizations**  
for **Speed**  
N. Author

ECE 561  
Project 1  
Report on  
Speed  
Optimization  
Stuart “Stu” Dent

## How is the Program Spending Its Time?

- Two basic approaches to find the right code to optimize
  - Estimation: Examine source code (and maybe object code too, for more accuracy) and build simple model of execution time relationships. We just saw this, and how risky it is.
  - Measurement: Measure a real system to get real\* numbers
- \* How real the numbers are depends on real the system and its inputs are

## Measurement: Sampling a Program

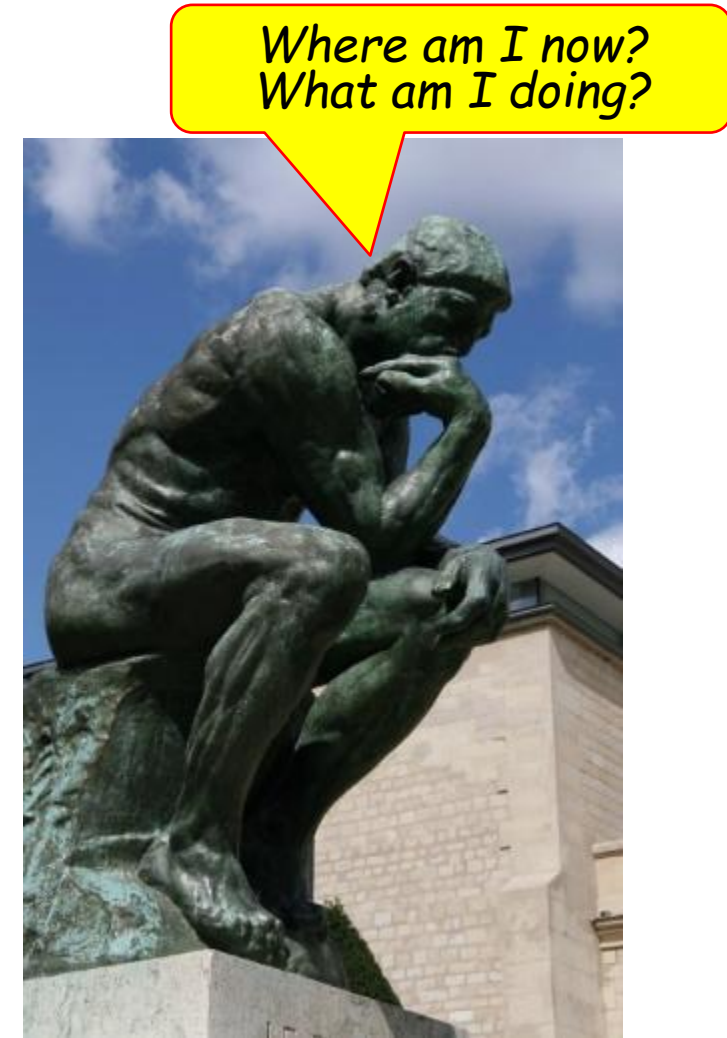
- Keep an array of counter variables, one per function
- Periodically sample the program as it runs
  - Find which function is running
  - Increment the counter variable for that function
- After run, examine the counter array to see which functions dominate execution time



Activity	Count
Studying calculus	37
Walking/biking/rollerblading	12
Soldering	7
Writing code	55
Eating	5
Study break	6
Sleeping	14

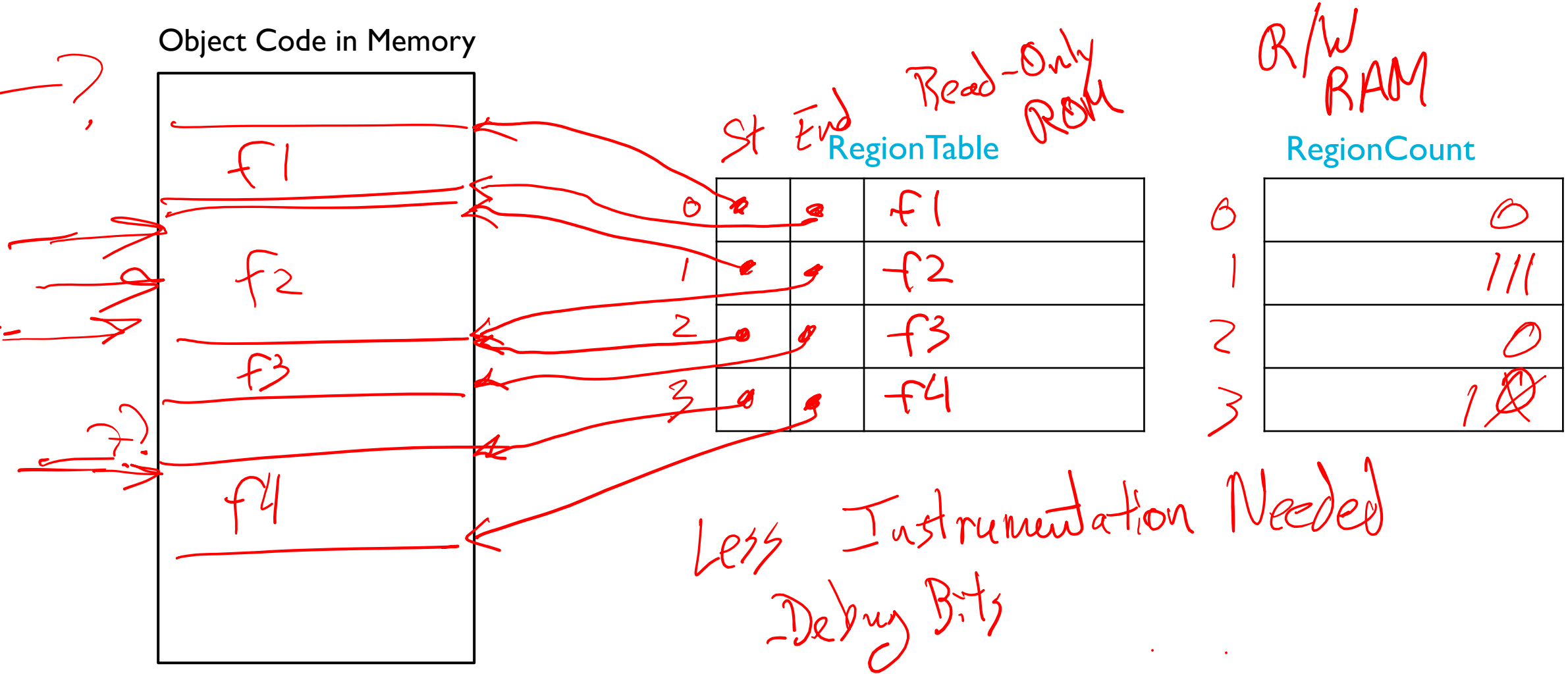
# Making the Program Self-Sampling

- Want target program to monitor itself
- Periodically samples itself as it runs
  - Trigger PC sampling with a timer interrupt
  - See which function was running by finding the ISR's return address
  - Increment the counter variable for that function
- Target program needs data structures:
  - Address-to-function translation table
  - Execution count for each region
- Helper program will generate address-to function translation table and integrate into application's source code



# Regions

- Target program needs an array to use while running to translate an address to a function name





# Region Information

- Getregions program
  - Analyzes .axf file to find function names, starting addresses and lengths
  - Generates C code to define and initialize an array of REGION\_T elements
- Each function is described by an entry in RegionTable
  - Start: first address used by this region
  - End: last address used by this region
  - Region name: simplifies analysis
- Place table in ROM (using **const**) to save RAM space

```
typedef struct {
    unsigned int Start;
    unsigned int End;
    char Name[24];
} REGION_T;
```

```
#include "region.h"
const REGION_T RegionTable[] = {
    {0x000000d5, 0x0000014c, "SystemInit"}, // 0
    {0x0000014d, 0x0000022c, "SystemCoreClockUpdate"}, // 1
    {0x00000261, 0x00000268, "Reset_Handler"}, // 2
    {0x00000269, 0x0000026a, "NMI_Handler"}, // 3
}
```

## Additional Profile Information

- **RegionCount** array
  - Placed in RAM
  - Updated by ISR during sampling, so declare as volatile
- **NumProfileRegions**
  - Total number of regions in table
- **profile\_samples**
  - Number of samples taken
- **profiling\_enabled**
  - Flag controlling profiling
- **num\_lost**
  - Number of addresses which couldn't be found in region table
- **adx\_lost**
  - Last address which wasn't found, helps for debugging

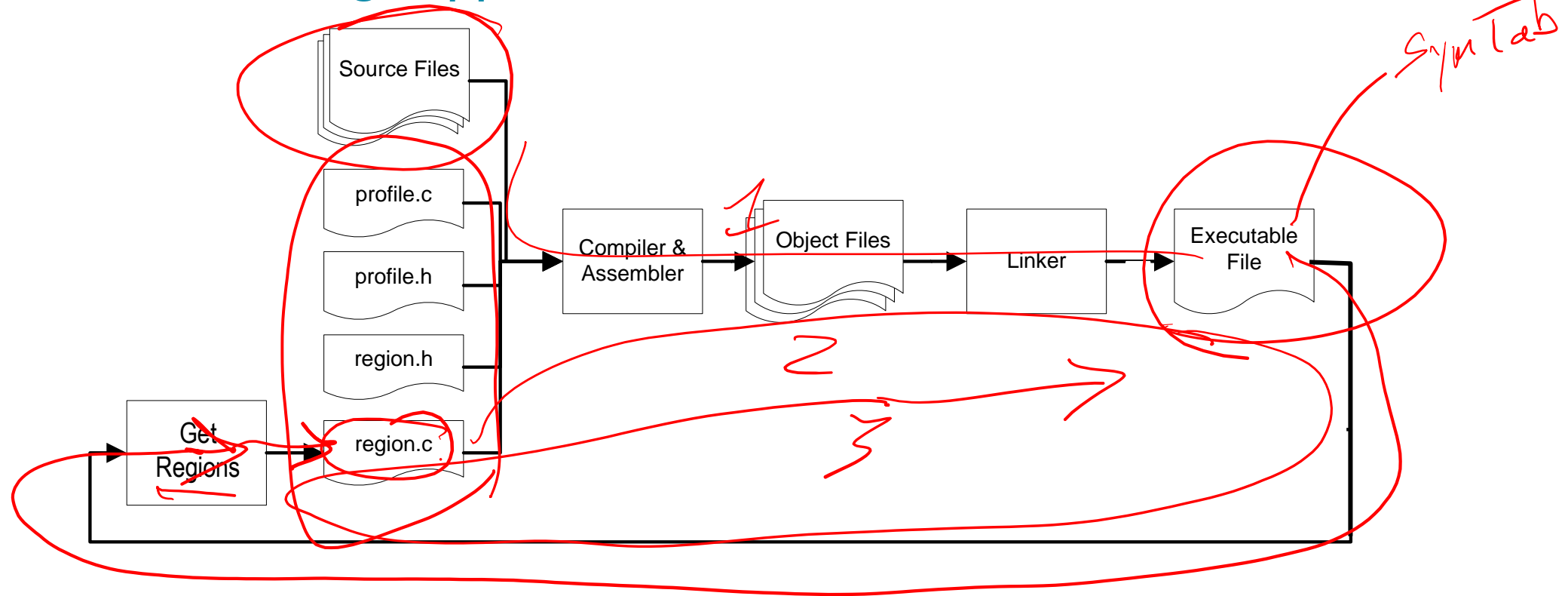
### Region.c

```
const unsigned NumProfileRegions=33;  
volatile unsigned RegionCount[33];
```

### Profile.c

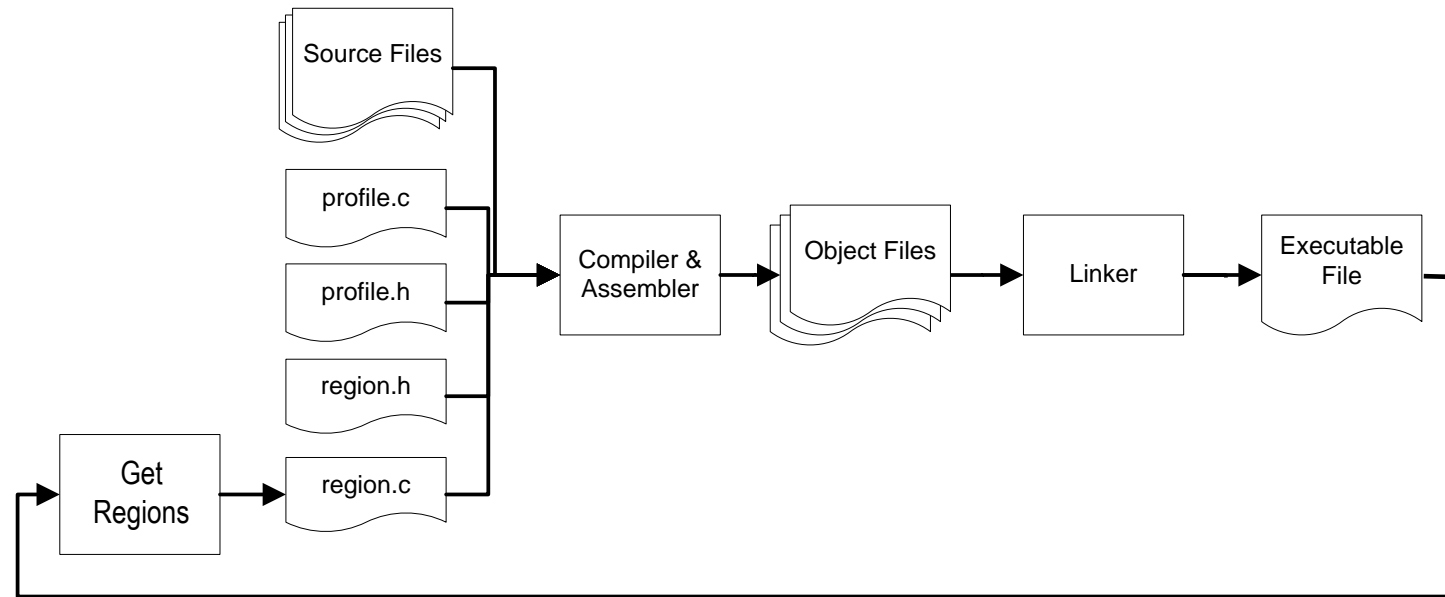
```
volatile unsigned int adx_lost=0, num_lost=0;  
volatile unsigned long profile_samples=0;  
unsigned char profiling_enabled = 0;
```

# Data Flow for Profiling Support



- Helper program (GetRegions) examines executable to make a C file which declares and initializes the region table.
- Run GetRegions
  - Examines executable file's symbol table (with addresses of all functions)
  - Generates C files which define address translation table (RegionTable), sample count table (RegionCount), other data
- Then rebuild program using updated region table

# Details of the Repeated Build Sequence



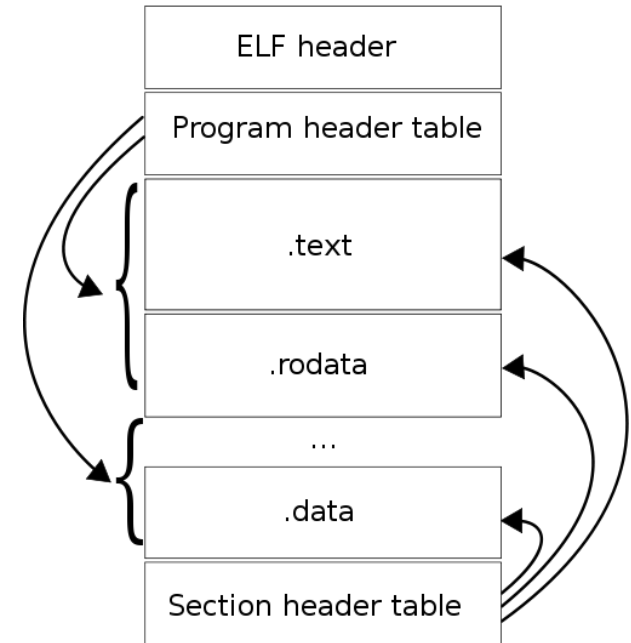
1. Build the program using a dummy region.c file. Region.c has **wrong number of regions, wrong addresses**.
2. Run tool to create the region table from the executable. Region.c has **right number of regions**, but **wrong addresses**.
3. Rebuild the program. The resulting executable symbol table has the correct function addresses.
4. Run tool to create the region table from the executable. **Region.c has right number of regions, right addresses**.
5. Rebuild the program. **Executable's region table has right number of regions, right addresses**

# EXTRACTING FUNCTION ADDRESSES FROM EXECUTABLE



# Finding Function Addresses

- MDK creates executable image with .axf extension
  - Format of .axf file is ELF (Executable and Linkable Format)
  - <http://man7.org/linux/man-pages/man5/elf.5.html>
- Example sections
  - .text: executable program instructions (machine code)
  - .data: initialized data
  - .bss: uninitialized data (actually cleared to zero)
  - .symtab: symbol table for debugging
  - .strtab: string table with names of sections and symbols
- Symbol Table holds information for each symbol (e.g. function, variable)
  - st\_value: beginning address of symbol
  - st\_size: size of symbol
  - st\_name: location of symbol's name in string table



```
typedef struct {
    uint32_t      st_name;
    Elf32_Addr    st_value;
    uint32_t      st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t      st_shndx;
} Elf32_Sym;
```

Image: Wikipedia

# Using GetRegions Manually

1. Build your program in  $\mu$ Vision.
2. Run `update_regions_manually.bat` (in the Scripts folder) to create a new `region.c` file (which is placed in the `src` folder). This file will have the correct number of entries but the addresses may be wrong (this is ok – a later pass will fix this).
3. Rebuild your program in  $\mu$ Vision to generate an executable with the correct size region table. Note that this table still uses the old function addresses, which will be wrong if the previous region table had a different number of entries.
4. Run `update_regions_manually.bat` to create a new `region.c` file with the correct function addresses and the correct number of entries.
5. Rebuild your program in  $\mu$ Vision to generate an executable with the correct region addresses.
6. Download and execute your program on the target hardware. This will populate the `RegionCount` table to indicate the execution time profile.

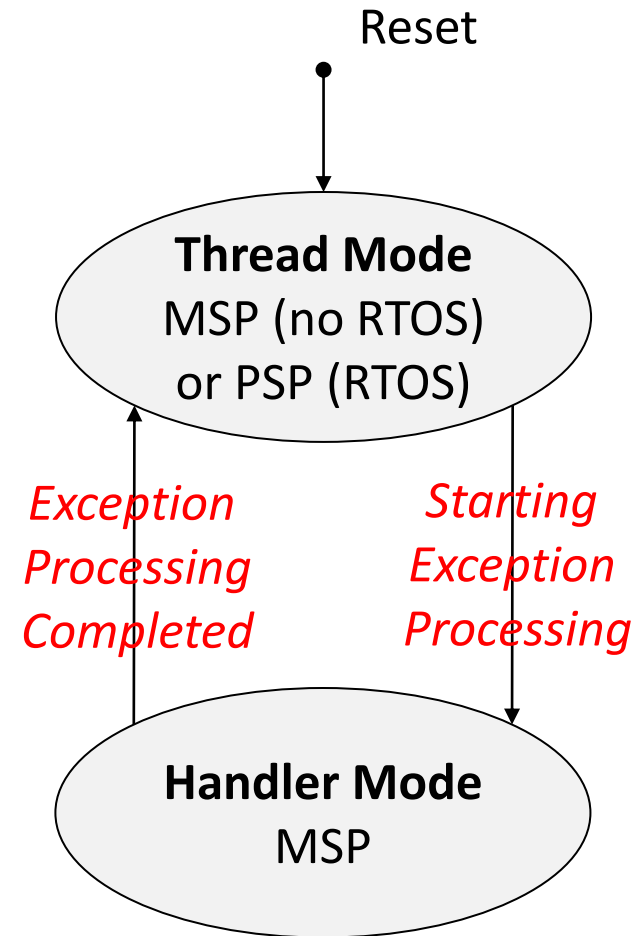
# RETRIEVING THE RETURN ADDRESS AND FINDING THE FUNCTION

# Identifying the Interrupted Function

- Demo code uses PIT to generate periodic interrupt
- Within timer ISR:
  - Read the return address from the stack
  - Examine table of region addresses to determine currently executing region  $i$
  - Increment entry  $i$  of execution count table
  - Extras
    - Increment `profile_samples` variable (to reveal out-of-range PC instances) – want to be able to double-check the measurements
    - Keep track of last address which was not found

# Which Stack? MSP or PSP?

- CPU can operate in two different modes
  - Handler mode for exception/interrupt handlers
  - Thread mode otherwise
- CPU has two stack pointers
  - Main Stack Pointer
  - Process Stack Pointer
- SPSEL flag in CONTROL register selects SP for thread mode
  - On reset, SPSEL = 0
- SP refers to either MSP or PSP, depending on mode and SPSEL
  - Handler mode uses MSP
  - Thread mode
    - If SPSEL is 0, uses MSP
    - If SPSEL is 1, uses PSP. This means handlers use a different stack than threads.





# Finding Return Address in ISR – No RTOS

*Where is MSP within exception handler ???*

*MSP upon entering exception handler →*

Address Offset	Contents
-8	Free space/Handler stack frame?
-4	
0	Saved R0
+4	Saved R1
+8	Saved R2
+12	Saved R3
+16	Saved R12
+20	Saved LR
+24	Saved PC
+28	Saved xPSR
+32	Foo's stack frame

*MSP before entering exception handler →*

- Function foo is running and is interrupted
- Where is the return address stored?
  - Not in LR, which holds exception return code in ISRs
  - Instead need to access saved PC value on stack
- Only main stack pointer (MSP) is used
- Where is it on the stack?
  - 24 bytes past MSP on entry to handler/ISR
  - But within handler, offset may be larger since handler may push more onto the stack

# Finding Return Address in ISR – Using RTOS

*PSP upon entering  
exception handler →*

Address Offset	Contents
0	Saved R0
+4	Saved R1
+8	Saved R2
+12	Saved R3
+16	Saved R12
+20	Saved LR
+24	Saved PC
+28	Saved xPSR
+32	Foo's stack frame

*PSP before entering  
exception handler →*

- Thread running Foo uses PSP
- Handler uses MSP
- Where is it on the stack?
  - 24 bytes past PSP on entry to handler/ISR

*MSP upon entering  
exception handler →*

Address Offset	Contents
0	
+4	
+8	
+12	
+16	
+20	
+24	
+28	
+32	Free space/Handler's stack frame

*MSP before entering  
exception handler →*

- Handler may push more onto the MSP stack, but PSP stack is not affected
- Within handler, SP refers to MSP, but we need to access PSP to get return address

# Code for Finding Return Address – Arm Compiler 5

- Read saved PC value from correct stack
  - Define constant values in profile.h
  - Use inline assembly code to access correct SP (PSP or MSP)
  - Add offset for context saved by hardware in response to interrupt (HW\_RET\_ADX\_OFFSET)
  - **Possibly** add offset for ISR's stack frame contents (FRAME\_SIZE)
  - Load the word from resulting memory address
- Resulting object code
  - Look at what that long line of C code turned into! Very efficient object code.
- Note: `__return_address()`
  - Compiler provides intrinsic command `__return_address()`.
  - Not useful because it returns value of link register, which doesn't hold return address for handlers.

```
#define HW_RET_ADX_OFFSET (24)
#define IRQ_FRAME_SIZE (8)
#define PPS_FRAME_SIZE (12)

#ifdef USING_RTOS // Don't need these since PC is on PSP, not MSP
#define FRAME_SIZE (0)
#define CUR_SP (_psp)
#else // Using MSP, so stack frames are also on the MSP stack
#define FRAME_SIZE (IRQ_FRAME_SIZE + PPS_FRAME_SIZE)
#define CUR_SP (_msp)
#endif
```

```
register unsigned int _psp __asm("psp");
register unsigned int _msp __asm("msp");
volatile unsigned int PC_val = 0;
```

```
PC_val = *((unsigned int *) (CUR_SP + FRAME_SIZE + HW_RET_ADX_OFFSET));
```

```
0x00001732 F3EF8008 MRS    r0,MSP
0x00001736 6AC1     LDR    r1,[r0,#0x2C]
0x00001738 4814     LDR    r0,[pc,#80]
0x0000173A 6101     STR    r1,[r0,#0x10]
```

# Code for Finding Return Address – Arm Compiler 6 (CLANG)

- Read return address (saved PC value) from correct stack (“RA\_SP”)
  - Define constant values in profile.h
  - Use inline assembly code to access correct SP (PSP or MSP)
  - Possibly** add offset for ISR’s stack frame contents (FRAME\_SIZE)
  - Add offset for context saved by hardware in response to interrupt (HW\_RET\_ADX\_OFFSET)
  - Load the word from resulting memory address
- Resulting object code
  - Look at what that long line of C code turned into! Very efficient object code.

```

profile.h
#ifdef USING_AC5 // register variables declared in profile.c
#define PROFILER_PSP (_psp)
#define PROFILER_MSP (_msp)
#else // USING_AC6 // intrinsics
#define PROFILER_PSP (__arm_rsr("PSP"))
#define PROFILER_MSP (__arm_rsr("MSP"))
#endif

#define HW_RET_ADX_OFFSET 24 // always 24, ISA-defined
#define IRQ_FRAME_SIZE (8) // may change based on handler code and compiler
#define PPS_FRAME_SIZE (20) // may change based on handler code and compiler

#ifdef USING_RTOS // PC is on PSP, not MSP
#define FRAME_SIZE (0)
#define RA_SP (PROFILER_PSP)
#else // Using MSP, so stack frames are also on the MSP stack
#define FRAME_SIZE (IRQ_FRAME_SIZE + PPS_FRAME_SIZE)
#define RA_SP (PROFILER_MSP)
#endif

```

```

profile.c: Process_Profile_Sample()
PC_val = *((unsigned int *) RA_SP + FRAME_SIZE + HW_RET_ADX_OFFSET);

```

## Disassembly

```

81:          PC_val = *((unsigned int *) (CUR_SP + FRAME_SIZE + HW_RET_ADX_OFFSET));
0x00002916 F3EF8009 MRS      r0,PSP
0x0000291A 6980      LDR      r0,[r0,#0x18]
0x0000291C 60E0      STR      r0,[r4,#0x0C]

```

# Return Address Summary

	Stack Pointer with Return Address (RA_SP)	Frames on that Stack	Offset within Hardware- Stacked Register Context
No RTOS	msp	PIT_IRQHandler + Process_Sample_Profile	24
RTOS	psp	(none)	24



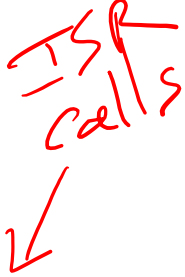
# Looking Up The Address

```
/* look up function in table and increment counter */
for (i=0; i<NumProfileRegions; i++) {
    s = RegionTable[i].Start & ~0x00000001; // Clear LSB of Thumb instruction address
    e = RegionTable[i].End;
    if ((PC_val >= s) && (PC_val <= e)) {
        RegionCount[i]++;
        break; // break out of the for loop
    }
}
if (i == NumProfileRegions) {
    adx_lost = PC_val;
    num_lost++;
}
```

## ■ By the way...

- This function executes frequently and could be optimized
- Could sort entries, starting with most frequently executed regions: “*Profile-driven profiler optimization!*”
- Uses a linear search: Change to binary search? Need to sort regions in address order

# Profiler API and Use

- 
- **void Init\_Profiling(void)**
    - Clears RegionCount table
    - Initializes and starts timer
  - **void Process\_Profile\_Sample(void)**
    - Reads return address from correct stack
    - Finds first containing region and increments its RegionCount entry
  - **void Disable\_Profiling(void)**
    - Clears profiling\_enabled flag
  - **void Enable\_Profiling(void)**
    - Sets profiling\_enabled flag
  - **void Sort\_Profile\_Regions(void)**
    - Creates a list of region numbers, sorted by highest values first
  - **void Display\_Profile(void)**
    - If PROFILER\_LCD\_SUPPORT defined, provides user interface to page through sorted regions
  - **void Serial\_Print\_Sorted\_Profile(void)**
    - If PROFILER\_SERIAL\_SUPPORT defined, sends sorted profile out UART to virtual serial port
  - **Configure settings in profile.h**
    - Sampling frequency
    - Whether RTOS is used or not
    - LCD and serial support
    - Stack frame sizes
  - **Program responsibilities**
    - Initialize profiling system and timers with Init\_Profiling()
    - Ensure interrupts are enabled (can call \_\_enable\_irq() to make sure)
    - Enable profiling
    - Run code in question
    - Disable profiling

# Disadvantages of Sampling

- Sampling is inexact - not guaranteed to get everything that runs
  - Code which disables interrupts (e.g. ISRs, OS code) is not measured
  - Rarely executed code may be missed
  - Takes time for numbers to settle down
  - Profile changes based on mode of program
- How long is enough?
  - The statistician I asked said “Well, it depends,” and changed the subject
  - A complex statistical question!
    - If this were a statistics class, we probably would never actually get to *running the program...*
  - So, run it until the digits you care about stop changing.
    - 9%? 9.7%? 9.73%? 9.731%? 9.731123452345245%?
    - Each additional digit will take more time to stabilize

# DEMONSTRATION

# ProfileDemo

- Where does this program spend most of its time?
  - f1, f2 and f3 are floats
- Each loop has
  - 2 FP adds
  - 2 FP multiplies
  - 1 FP sine
  - 1 FP cosine
- Build program, download, run
- Break at while (1) loop
- Examine RegionCount and profile\_samples

```
Init_Profiling();  
__enable_irq();  
Control_RGB_LEDs(1,0,0);  
Enable_Profiling();  
  
for (p=0; p<10000; p++) {  
    f1 *= 1.02;  
    f2 += f1;  
    f3 += sin(f1)*cos(f2);  
}  
Disable_Profiling();  
Control_RGB_LEDs(0,1,0);  
while (1)  
    ;
```

# Results

- `profile_samples` = 5329 (number of samples)
- Top region is #26
  - 1413 samples/5329 = 26.5%
  - Look up #26 in RegionTable – is `__aeabi_fmul` (floating-point multiply)
- 2<sup>nd</sup> region is #23
  - 1345 / 5329 = 25.2%
  - `__aeabi_fadd` (fp add)
- 3<sup>rd</sup> region is #39
  - 494 / 5329 = 9.3%
  - `__mathlib_rredf2` (?)
- Better to sort the regions with call to `Sort_Profile_Regions`, then examine `SortedRegions[]`

## RegionCount[]

[21]	0
[22]	219
[23]	1345
[24]	20
[25]	43
[26]	1413
[27]	46
[28]	65
[29]	205
[30]	446
[31]	0
[32]	16
[33]	7
[34]	0
[35]	53
[36]	111
[37]	121
[38]	12
[39]	494
[40]	0

## RegionTable[]

```

__aeabi_fadd", // 23
__aeabi_fsub", // 24
__aeabi_frsub", // 25
__aeabi_fmul", // 26
__ARM_scalbnf", // 27
__aeabi_i2f", // 28
_float_round", // 29
_float_epilogue", // 30
__aeabi_fdiv", // 31
_frnd", // 32
__aeabi_f2iz", // 33
__scatterload", // 34
__aeabi_ui2f", // 35
__ARM_common_ll_muluu", //
__ARM_fpclassifyf", // 37
__mathlib_flt_underflow",
__mathlib_rredf2", // 39

```



# How long do the sin and cos calls take?

- Look them up in RegionTable
  - Entries 45 (sinf) and 44 (cosf)
- Then look in RegionCount
  - Sinf:  $264/5329 = 4.95\%$
  - Cosf:  $248/5329 = 4.64\%$
- Are those functions *really* that fast? 2.5x faster than multiply or add?
  - No they aren't!
  - Look at object code for sinf in disassembly window
  - Many calls to fmul and other functions

## RegionTable

```
{0x00000ae5, 0x00000c28, "cosf"}, // 44
{0x00000c61, 0x00000db0, "sinf"}, // 45
```

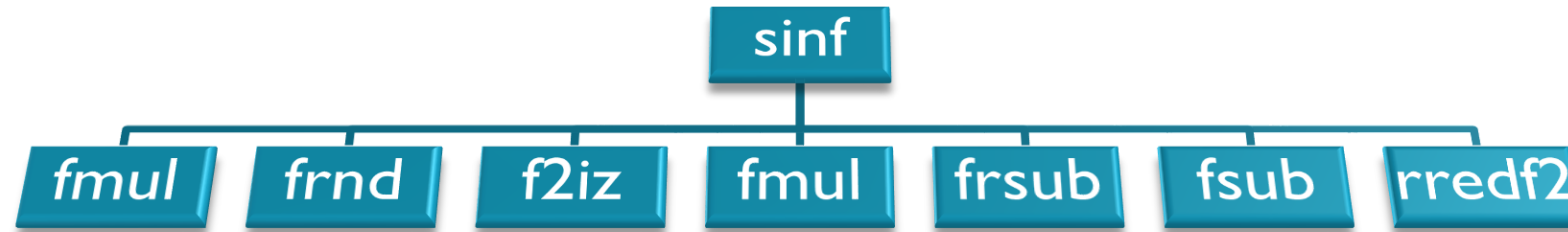
## RegionCount

[44]	248	unsigned int
[45]	264	unsigned int

## sinf Disassembly

```
BL.W    __aeabi_fmul (0x0000069E)
STR      r0,[sp,#0x04]
LDR      r1,[pc,#268] ; @0x00000DC0
MOV      r0,r7
BL.W    __aeabi_fmul (0x0000069E)
MOV      r6,r0
LDR      r1,[pc,#260] ; @0x00000DC4
MOV      r0,r7
BL.W    __aeabi_fmul (0x0000069E)
MOV      r5,r0
LDR      r1,[pc,#256] ; @0x00000DC8
MOV      r0,r7
BL.W    __aeabi_fmul (0x0000069E)
MOV      r1,r4
BL.W    __aeabi_frsb (0x00000696)
```

# Flat vs. Cumulative Profile



- Flat profile
  - A function F accumulates time only if the PC address sample is in function F
- Cumulative profile
  - A function F accumulates time
    - If the PC address sample is in function F
    - If the PC address sample is in a function G, which F called (directly or indirectly)
      - All functions above G in the call tree also accumulate time
  - This can be done by examining all return addresses on the call stack
    - Not implemented here (possible extra credit project?)
- Approximation
  - Modify function of interest to set an output bit on entry, clear it on exit
  - Examine bit with oscilloscope, measure duty cycle