

# Reverse Engineering for Beginners



Dennis Yurichev

---

# Reverse Engineering for Beginners

Dennis Yurichev

`<dennis(a)yurichev.com>`



©2013-2016, Dennis Yurichev.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license. To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>.

Text version (May 3, 2017).

The latest version (and Russian edition) of this text is accessible at [beginners.re](http://beginners.re).

The cover was made by Andy Nечаевский: [facebook](#).

---

# Call for translators!

You may want to help me with translating this work into languages other than English and Russian. Just send me any piece of translated text (no matter how short) and I'll put it into my LaTeX source code.

[Read here.](#)

Speed isn't important, because this is an open-source project, after all. Your name will be mentioned as a project contributor. Korean, Chinese, and Persian languages are reserved by publishers. English and Russian versions I do by myself, but my English is still that horrible, so I'm very grateful for any notes about grammar, etc. Even my Russian is flawed, so I'm grateful for notes about Russian text as well!

So do not hesitate to contact me: [dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com).

# Abridged contents

<b>1 Code patterns</b>	<b>1</b>
<b>2 Important fundamentals</b>	<b>448</b>
<b>3 Slightly more advanced examples</b>	<b>467</b>
<b>4 Java</b>	<b>666</b>
<b>5 Finding important/interesting stuff in the code</b>	<b>705</b>
<b>6 OS-specific</b>	<b>733</b>
<b>7 Tools</b>	<b>788</b>
<b>8 Case studies</b>	<b>792</b>
<b>9 Examples of reversing proprietary file formats</b>	<b>927</b>
<b>10 Other things</b>	<b>987</b>
<b>11 Books/blogs worth reading</b>	<b>995</b>
<b>12 Communities</b>	<b>998</b>
<b>Afterword</b>	<b>1000</b>
<b>Appendix</b>	<b>1002</b>
<b>Acronyms used</b>	<b>1031</b>
<b>Glossary</b>	<b>1036</b>
<b>Index</b>	<b>1038</b>

# Contents

<b>1</b>	<b>Code patterns</b>	<b>1</b>
1.1	The method . . . . .	1
1.2	Some basics . . . . .	2
1.2.1	A short introduction to the CPU	2
1.2.2	Numeral systems . . . . .	3
1.3	Empty function . . . . .	5
1.3.1	x86 . . . . .	5
1.3.2	ARM . . . . .	6
1.3.3	MIPS . . . . .	6
1.3.4	Empty functions in practice . . . . .	6
1.4	Returning value . . . . .	7
1.4.1	x86 . . . . .	7
1.4.2	ARM . . . . .	7
1.4.3	MIPS . . . . .	8
1.4.4	In practice . . . . .	8
1.5	Hello, world! . . . . .	8
1.5.1	x86 . . . . .	8
1.5.2	x86-64 . . . . .	14
1.5.3	GCC—one more thing . . . . .	18
1.5.4	ARM . . . . .	19
1.5.5	MIPS . . . . .	25
1.5.6	Conclusion . . . . .	30
1.5.7	Exercises . . . . .	30
1.6	Function prologue and epilogue . . . . .	30
1.6.1	Recursion . . . . .	30
1.7	Stack . . . . .	30
1.7.1	Why does the stack grow backwards? . . . . .	31
1.7.2	What is the stack used for? . . . . .	31
1.7.3	A typical stack layout . . . . .	38
1.7.4	Noise in stack . . . . .	38
1.7.5	Exercises . . . . .	42
1.8	printf() with several arguments . . . . .	42
1.8.1	x86 . . . . .	42
1.8.2	ARM . . . . .	53
1.8.3	MIPS . . . . .	59
1.8.4	Conclusion . . . . .	65
1.8.5	By the way . . . . .	66
1.9	scanf() . . . . .	66
1.9.1	Simple example . . . . .	66
1.9.2	Popular mistake . . . . .	75
1.9.3	Global variables . . . . .	76
1.9.4	scanf() . . . . .	85
1.9.5	Exercise . . . . .	97
1.10	Accessing passed arguments . . . . .	98
1.10.1	x86 . . . . .	98
1.10.2	x64 . . . . .	100
1.10.3	ARM . . . . .	103
1.10.4	MIPS . . . . .	106
1.11	More about results returning . . . . .	107
1.11.1	Attempt to use the result of a function returning void . . . . .	107
1.11.2	What if we do not use the function result? . . . . .	109
1.11.3	Returning a structure . . . . .	109

## CONTENTS

1.12 Pointers . . . . .	110
1.12.1 Swap input values . . . . .	110
1.12.2 Returning values . . . . .	111
1.13 GOTO operator . . . . .	121
1.13.1 Dead code . . . . .	124
1.13.2 Exercise . . . . .	125
1.14 Conditional jumps . . . . .	125
1.14.1 Simple example . . . . .	125
1.14.2 Calculating absolute value . . . . .	142
1.14.3 Ternary conditional operator . . . . .	144
1.14.4 Getting minimal and maximal values . . . . .	147
1.14.5 Conclusion . . . . .	152
1.14.6 Exercise . . . . .	153
1.15 switch() / case / default . . . . .	154
1.15.1 Small number of cases . . . . .	154
1.15.2 A lot of cases . . . . .	167
1.15.3 When there are several case statements in one block . . . . .	179
1.15.4 Fall-through . . . . .	183
1.15.5 Exercises . . . . .	184
1.16 Loops . . . . .	185
1.16.1 Simple example . . . . .	185
1.16.2 Memory blocks copying routine . . . . .	196
1.16.3 Conclusion . . . . .	199
1.16.4 Exercises . . . . .	200
1.17 More about strings . . . . .	201
1.17.1 strlen() . . . . .	201
1.17.2 Boundaries of strings . . . . .	212
1.18 Replacing arithmetic instructions to other ones . . . . .	212
1.18.1 Multiplication . . . . .	212
1.18.2 Division . . . . .	217
1.18.3 Exercise . . . . .	218
1.19 Floating-point unit . . . . .	218
1.19.1 IEEE 754 . . . . .	218
1.19.2 x86 . . . . .	218
1.19.3 ARM, MIPS, x86/x64 SIMD . . . . .	219
1.19.4 C/C++ . . . . .	219
1.19.5 Simple example . . . . .	219
1.19.6 Passing floating point numbers via arguments . . . . .	230
1.19.7 Comparison example . . . . .	233
1.19.8 Some constants . . . . .	267
1.19.9 Copying . . . . .	267
1.19.10 Stack, calculators and reverse Polish notation . . . . .	267
1.19.11 x64 . . . . .	267
1.19.12 Exercises . . . . .	267
1.20 Arrays . . . . .	268
1.20.1 Simple example . . . . .	268
1.20.2 Buffer overflow . . . . .	275
1.20.3 Buffer overflow protection methods . . . . .	283
1.20.4 One more word about arrays . . . . .	286
1.20.5 Array of pointers to strings . . . . .	287
1.20.6 Multidimensional arrays . . . . .	293
1.20.7 Pack of strings as a two-dimensional array . . . . .	300
1.20.8 Conclusion . . . . .	304
1.21 By the way . . . . .	304
1.21.1 Exercises . . . . .	304
1.22 Manipulating specific bit(s) . . . . .	304
1.22.1 Specific bit checking . . . . .	304
1.22.2 Setting and clearing specific bits . . . . .	308
1.22.3 Shifts . . . . .	317
1.22.4 Setting and clearing specific bits: FPU <sup>1</sup> example . . . . .	317
1.22.5 Counting bits set to 1 . . . . .	322
1.22.6 Conclusion . . . . .	337
1.22.7 Exercises . . . . .	339

<sup>1</sup>Floating-point unit

## CONTENTS

1.23	Linear congruential generator . . . . .	339
1.23.1	x86 . . . . .	340
1.23.2	x64 . . . . .	341
1.23.3	32-bit ARM . . . . .	341
1.23.4	MIPS . . . . .	342
1.23.5	Thread-safe version of the example . . . . .	344
1.24	Structures . . . . .	345
1.24.1	MSVC: SYSTEMTIME example . . . . .	345
1.24.2	Let's allocate space for a structure using malloc() . . . . .	349
1.24.3	UNIX: struct tm . . . . .	351
1.24.4	Fields packing in structure . . . . .	360
1.24.5	Nested structures . . . . .	367
1.24.6	Bit fields in a structure . . . . .	370
1.24.7	Exercises . . . . .	377
1.25	Unions . . . . .	377
1.25.1	Pseudo-random number generator example . . . . .	377
1.25.2	Calculating machine epsilon . . . . .	381
1.26	FSCALE replacement . . . . .	383
1.26.1	Fast square root calculation . . . . .	385
1.27	Pointers to functions . . . . .	385
1.27.1	MSVC . . . . .	386
1.27.2	GCC . . . . .	393
1.27.3	Danger of pointers to functions . . . . .	397
1.28	64-bit values in 32-bit environment . . . . .	397
1.28.1	Returning of 64-bit value . . . . .	397
1.28.2	Arguments passing, addition, subtraction . . . . .	398
1.28.3	Multiplication, division . . . . .	401
1.28.4	Shifting right . . . . .	405
1.28.5	Converting 32-bit value into 64-bit one . . . . .	406
1.29	SIMD . . . . .	407
1.29.1	Vectorization . . . . .	408
1.29.2	SIMD <code>strlen()</code> implementation . . . . .	417
1.30	64 bits . . . . .	421
1.30.1	x86-64 . . . . .	421
1.30.2	ARM . . . . .	427
1.30.3	Float point numbers . . . . .	428
1.30.4	64-bit architecture criticism . . . . .	428
1.31	Working with floating point numbers using SIMD . . . . .	428
1.31.1	Simple example . . . . .	428
1.31.2	Passing floating point number via arguments . . . . .	436
1.31.3	Comparison example . . . . .	437
1.31.4	Calculating machine epsilon: x64 and SIMD . . . . .	439
1.31.5	Pseudo-random number generator example revisited . . . . .	440
1.31.6	Summary . . . . .	440
1.32	ARM-specific details . . . . .	441
1.32.1	Number sign (#) before number . . . . .	441
1.32.2	Addressing modes . . . . .	441
1.32.3	Loading a constant into a register . . . . .	442
1.32.4	Relocs in ARM64 . . . . .	444
1.33	MIPS-specific details . . . . .	445
1.33.1	Loading a 32-bit constant into register . . . . .	445
1.33.2	Further reading about MIPS . . . . .	447
<b>2</b>	<b>Important fundamentals</b> . . . . .	<b>448</b>
2.1	Integral datatypes . . . . .	449
2.1.1	Bit . . . . .	449
2.1.2	Nibble AKA nybble . . . . .	449
2.1.3	Byte . . . . .	450
2.1.4	Wide char . . . . .	450
2.1.5	Signed integer vs unsigned . . . . .	451
2.1.6	Word . . . . .	451
2.1.7	Address register . . . . .	452
2.1.8	Numbers . . . . .	452
2.2	Signed number representations . . . . .	454

## CONTENTS

2.2.1 Using IMUL over MUL . . . . .	456
2.2.2 Couple of additions about two's complement form . . . . .	456
2.3 AND . . . . .	457
2.3.1 Checking if a value is on $2^n$ boundary . . . . .	457
2.3.2 KOI-8R Cyrillic encoding . . . . .	458
2.4 AND and OR as subtraction and addition . . . . .	459
2.4.1 ZX Spectrum ROM text strings . . . . .	459
2.5 XOR (exclusive OR) . . . . .	461
2.5.1 Everyday speech . . . . .	461
2.5.2 Encryption . . . . .	461
2.5.3 RAID <sup>2</sup> 4 . . . . .	461
2.5.4 XOR swap algorithm . . . . .	461
2.5.5 XOR linked list . . . . .	462
2.5.6 By the way . . . . .	462
2.5.7 AND/OR/XOR as MOV . . . . .	463
2.6 Population count . . . . .	463
2.7 Endianness . . . . .	463
2.7.1 Big-endian . . . . .	463
2.7.2 Little-endian . . . . .	464
2.7.3 Example . . . . .	464
2.7.4 Bi-endian . . . . .	464
2.7.5 Converting data . . . . .	464
2.8 Memory . . . . .	465
2.9 CPU . . . . .	465
2.9.1 Branch predictors . . . . .	465
2.9.2 Data dependencies . . . . .	465
2.10 Hash functions . . . . .	465
2.10.1 How do one-way functions work? . . . . .	466
<b>3 Slightly more advanced examples</b> . . . . .	<b>467</b>
3.1 Double negation . . . . .	467
3.2 strstr() example . . . . .	468
3.3 Temperature converting . . . . .	468
3.3.1 Integer values . . . . .	468
3.3.2 Floating-point values . . . . .	470
3.4 Fibonacci numbers . . . . .	472
3.4.1 Example #1 . . . . .	473
3.4.2 Example #2 . . . . .	476
3.4.3 Summary . . . . .	480
3.5 CRC32 calculation example . . . . .	481
3.6 Network address calculation example . . . . .	484
3.6.1 calc_network_address() . . . . .	485
3.6.2 form_IP() . . . . .	485
3.6.3 print_as_IP() . . . . .	487
3.6.4 form_netmask() and set_bit() . . . . .	488
3.6.5 Summary . . . . .	489
3.7 Loops: several iterators . . . . .	489
3.7.1 Three iterators . . . . .	489
3.7.2 Two iterators . . . . .	490
3.7.3 Intel C++ 2011 case . . . . .	492
3.8 Duff's device . . . . .	493
3.8.1 Should one use unrolled loops? . . . . .	496
3.9 Division using multiplication . . . . .	496
3.9.1 x86 . . . . .	496
3.9.2 How it works . . . . .	497
3.9.3 ARM . . . . .	498
3.9.4 MIPS . . . . .	499
3.9.5 Exercise . . . . .	499
3.10 String to number conversion (atoi()) . . . . .	499
3.10.1 Simple example . . . . .	500
3.10.2 A slightly advanced example . . . . .	503
3.10.3 Exercise . . . . .	506
3.11 Inline functions . . . . .	506

<sup>2</sup>Redundant Array of Independent Disks

## CONTENTS

3.11.1 Strings and memory functions . . . . .	507
3.12 C99 restrict . . . . .	515
3.13 Branchless <i>abs()</i> function . . . . .	517
3.13.1 Optimizing GCC 4.9.1 x64 . . . . .	517
3.13.2 Optimizing GCC 4.9 ARM64 . . . . .	518
3.14 Variadic functions . . . . .	518
3.14.1 Computing arithmetic mean . . . . .	518
3.14.2 <i>vprintf()</i> function case . . . . .	522
3.14.3 <i>Pin</i> case . . . . .	523
3.14.4 Format string exploit . . . . .	524
3.15 Strings trimming . . . . .	525
3.15.1 x64: Optimizing MSVC 2013 . . . . .	526
3.15.2 x64: Non-optimizing GCC 4.9.1 . . . . .	527
3.15.3 x64: Optimizing GCC 4.9.1 . . . . .	528
3.15.4 ARM64: Non-optimizing GCC (Linaro) 4.9 . . . . .	529
3.15.5 ARM64: Optimizing GCC (Linaro) 4.9 . . . . .	530
3.15.6 ARM: Optimizing Keil 6/2013 (ARM mode) . . . . .	531
3.15.7 ARM: Optimizing Keil 6/2013 (Thumb mode) . . . . .	532
3.15.8 MIPS . . . . .	532
3.16 <i>toupper()</i> function . . . . .	533
3.16.1 x64 . . . . .	534
3.16.2 ARM . . . . .	536
3.16.3 Summary . . . . .	537
3.17 Incorrectly disassembled code . . . . .	537
3.17.1 Disassembling from an incorrect start (x86) . . . . .	537
3.17.2 How does random noise looks disassembled? . . . . .	538
3.18 Obfuscation . . . . .	542
3.18.1 Text strings . . . . .	542
3.18.2 Executable code . . . . .	542
3.18.3 Virtual machine / pseudo-code . . . . .	544
3.18.4 Other things to mention . . . . .	544
3.18.5 Exercise . . . . .	544
3.19 C++ . . . . .	545
3.19.1 Classes . . . . .	545
3.19.2 ostream . . . . .	561
3.19.3 References . . . . .	562
3.19.4 STL . . . . .	563
3.19.5 Memory . . . . .	596
3.20 Negative array indices . . . . .	597
3.20.1 Addressing string from the end . . . . .	597
3.20.2 Addressing some kind of block from the end . . . . .	597
3.20.3 Arrays started at 1 . . . . .	598
3.21 Packing 12-bit values into array . . . . .	600
3.21.1 Introduction . . . . .	600
3.21.2 Data structure . . . . .	600
3.21.3 The algorithm . . . . .	601
3.21.4 The C/C++ code . . . . .	601
3.21.5 How it works . . . . .	603
3.21.6 Optimizing GCC 4.8.2 for x86-64 . . . . .	604
3.21.7 Optimizing Keil 5.05 (Thumb mode) . . . . .	606
3.21.8 Optimizing Keil 5.05 (ARM mode) . . . . .	609
3.21.9 (32-bit ARM) Comparison of code density in Thumb and ARM modes . . . . .	610
3.21.10 Optimizing GCC 4.9.3 for ARM64 . . . . .	610
3.21.11 Optimizing GCC 4.4.5 for MIPS . . . . .	612
3.21.12 Difference from the real FAT12 . . . . .	614
3.21.13 Exercise . . . . .	615
3.21.14 Summary . . . . .	615
3.21.15 Conclusion . . . . .	615
3.22 More about pointers . . . . .	615
3.22.1 Working with addresses instead of pointers . . . . .	616
3.22.2 Passing values as pointers; tagged unions . . . . .	618
3.22.3 Pointers abuse in Windows kernel . . . . .	619
3.22.4 Null pointers . . . . .	623
3.22.5 Array as function argument . . . . .	627

## CONTENTS

3.22.6 Pointer to function . . . . .	628
3.22.7 Pointer as object identifier . . . . .	628
3.23 Loop optimizations . . . . .	629
3.23.1 Weird loop optimization . . . . .	629
3.23.2 Another loop optimization . . . . .	631
3.24 More about structures . . . . .	633
3.24.1 Sometimes a C structure can be used instead of array . . . . .	633
3.24.2 Unsized array in C structure . . . . .	634
3.24.3 Version of C structure . . . . .	635
3.24.4 High-score file in “Block out” game and primitive serialization . . . . .	637
3.25 memmove() and memcpy() . . . . .	641
3.25.1 Anti-debugging trick . . . . .	642
3.26 setjmp/longjmp . . . . .	643
3.27 Other weird stack hacks . . . . .	645
3.27.1 Accessing arguments/local variables of caller . . . . .	645
3.27.2 Returning string . . . . .	647
3.28 OpenMP . . . . .	648
3.28.1 MSVC . . . . .	650
3.28.2 GCC . . . . .	652
3.29 Another heisenbug . . . . .	653
3.30 Windows 16-bit . . . . .	654
3.30.1 Example #1 . . . . .	655
3.30.2 Example #2 . . . . .	655
3.30.3 Example #3 . . . . .	656
3.30.4 Example #4 . . . . .	657
3.30.5 Example #5 . . . . .	659
3.30.6 Example #6 . . . . .	663
<b>4 Java</b> . . . . .	<b>666</b>
4.1 Java . . . . .	666
4.1.1 Introduction . . . . .	666
4.1.2 Returning a value . . . . .	666
4.1.3 Simple calculating functions . . . . .	671
4.1.4 JVM <sup>3</sup> memory model . . . . .	673
4.1.5 Simple function calling . . . . .	674
4.1.6 Calling beep() . . . . .	675
4.1.7 Linear congruential PRNG <sup>4</sup> . . . . .	676
4.1.8 Conditional jumps . . . . .	677
4.1.9 Passing arguments . . . . .	679
4.1.10 Bitfields . . . . .	680
4.1.11 Loops . . . . .	681
4.1.12 switch() . . . . .	683
4.1.13 Arrays . . . . .	684
4.1.14 Strings . . . . .	692
4.1.15 Exceptions . . . . .	694
4.1.16 Classes . . . . .	697
4.1.17 Simple patching . . . . .	699
4.1.18 Summary . . . . .	704
<b>5 Finding important/interesting stuff in the code</b> . . . . .	<b>705</b>
5.1 Identification of executable files . . . . .	705
5.1.1 Microsoft Visual C++ . . . . .	705
5.1.2 GCC . . . . .	706
5.1.3 Intel Fortran . . . . .	706
5.1.4 Watcom, OpenWatcom . . . . .	706
5.1.5 Borland . . . . .	707
5.1.6 Other known DLLs . . . . .	708
5.2 Communication with outer world (function level) . . . . .	708
5.3 Communication with the outer world (win32) . . . . .	708
5.3.1 Often used functions in the Windows API . . . . .	709
5.3.2 Extending trial period . . . . .	709
5.3.3 Removing nag dialog box . . . . .	709

<sup>3</sup>Java virtual machine

<sup>4</sup>Pseudorandom number generator

## CONTENTS

5.3.4 tracer: Intercepting all functions in specific module . . . . .	709
5.4 Strings . . . . .	710
5.4.1 Text strings . . . . .	710
5.4.2 Finding strings in binary . . . . .	715
5.4.3 Error/debug messages . . . . .	716
5.4.4 Suspicious magic strings . . . . .	716
5.5 Calls to assert() . . . . .	717
5.6 Constants . . . . .	717
5.6.1 Magic numbers . . . . .	718
5.6.2 Specific constants . . . . .	720
5.6.3 Searching for constants . . . . .	720
5.7 Finding the right instructions . . . . .	720
5.8 Suspicious code patterns . . . . .	721
5.8.1 XOR instructions . . . . .	721
5.8.2 Hand-written assembly code . . . . .	722
5.9 Using magic numbers while tracing . . . . .	723
5.10 Loops . . . . .	723
5.11 Other things . . . . .	724
5.11.1 General idea . . . . .	724
5.11.2 Order of functions in binary code . . . . .	724
5.11.3 Tiny functions . . . . .	724
5.11.4 C++ . . . . .	724
5.11.5 Some binary file patterns . . . . .	724
5.11.6 Memory “snapshots” comparing . . . . .	731
<b>6 OS-specific</b> . . . . .	<b>733</b>
6.1 Arguments passing methods (calling conventions) . . . . .	733
6.1.1 cdecl . . . . .	733
6.1.2 stdcall . . . . .	733
6.1.3 fastcall . . . . .	734
6.1.4 thiscall . . . . .	736
6.1.5 x86-64 . . . . .	736
6.1.6 Return values of <i>float</i> and <i>double</i> type . . . . .	739
6.1.7 Modifying arguments . . . . .	739
6.1.8 Taking a pointer to function argument . . . . .	740
6.2 Thread Local Storage . . . . .	741
6.2.1 Linear congruential generator revisited . . . . .	742
6.3 System calls (syscall-s) . . . . .	746
6.3.1 Linux . . . . .	747
6.3.2 Windows . . . . .	747
6.4 Linux . . . . .	747
6.4.1 Position-independent code . . . . .	747
6.4.2 <i>LD_PRELOAD</i> hack in Linux . . . . .	750
6.5 Windows NT . . . . .	752
6.5.1 CRT (win32) . . . . .	752
6.5.2 Win32 PE . . . . .	756
6.5.3 Windows SEH . . . . .	763
6.5.4 Windows NT: Critical section . . . . .	786
<b>7 Tools</b> . . . . .	<b>788</b>
7.1 Binary analysis . . . . .	788
7.1.1 Disassemblers . . . . .	788
7.1.2 Decompilers . . . . .	789
7.1.3 Patch comparison/diffing . . . . .	789
7.2 Live analysis . . . . .	789
7.2.1 Debuggers . . . . .	789
7.2.2 Library calls tracing . . . . .	789
7.2.3 System calls tracing . . . . .	790
7.2.4 Network sniffing . . . . .	790
7.2.5 Sysinternals . . . . .	790
7.2.6 Valgrind . . . . .	790
7.2.7 Emulators . . . . .	790
7.3 Other tools . . . . .	791
7.4 Something missing here? . . . . .	791

## CONTENTS

<b>8 Case studies</b>	<b>792</b>
8.1 Task manager practical joke (Windows Vista) . . . . .	793
8.1.1 Using LEA to load values . . . . .	796
8.2 Color Lines game practical joke . . . . .	798
8.3 Minesweeper (Windows XP) . . . . .	801
8.3.1 Finding grid automatically . . . . .	806
8.3.2 Exercises . . . . .	807
8.4 Hacking Windows clock . . . . .	807
8.5 Dongles . . . . .	814
8.5.1 Example #1: MacOS Classic and PowerPC . . . . .	814
8.5.2 Example #2: SCO OpenServer . . . . .	821
8.5.3 Example #3: MS-DOS . . . . .	831
8.6 "QR9": Rubik's cube inspired amateur crypto-algorithm . . . . .	836
8.7 Encrypted database case #1 . . . . .	863
8.7.1 Base64 and entropy . . . . .	863
8.7.2 Is it compressed? . . . . .	866
8.7.3 Is it encrypted? . . . . .	867
8.7.4 CryptoPP . . . . .	867
8.7.5 Cipher Feedback mode . . . . .	869
8.7.6 Initializing Vector . . . . .	871
8.7.7 Structure of the buffer . . . . .	872
8.7.8 Noise at the end . . . . .	874
8.7.9 Conclusion . . . . .	874
8.7.10 Post Scriptum: brute-forcing IV . . . . .	875
8.8 Overclocking Cointerra Bitcoin miner . . . . .	875
8.9 Breaking simple executable cryptor . . . . .	879
8.9.1 Other ideas to consider . . . . .	884
8.10 SAP . . . . .	884
8.10.1 About SAP client network traffic compression . . . . .	884
8.10.2 SAP 6.0 password checking functions . . . . .	895
8.11 Oracle RDBMS . . . . .	899
8.11.1 V\$VERSION table in the Oracle RDBMS . . . . .	899
8.11.2 X\$KSMLRU table in Oracle RDBMS . . . . .	907
8.11.3 V\$TIMER table in Oracle RDBMS . . . . .	908
8.12 Handwritten assembly code . . . . .	912
8.12.1 EICAR test file . . . . .	912
8.13 Demos . . . . .	913
8.13.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10 . . . . .	913
8.13.2 Mandelbrot set . . . . .	916
8.14 Other examples . . . . .	926
<b>9 Examples of reversing proprietary file formats</b>	<b>927</b>
9.1 Primitive XOR-encryption . . . . .	927
9.1.1 Norton Guide: simplest possible 1-byte XOR encryption . . . . .	928
9.1.2 Simplest possible 4-byte XOR encryption . . . . .	931
9.1.3 Simple encryption using XOR mask . . . . .	935
9.1.4 Simple encryption using XOR mask, case II . . . . .	941
9.2 Analyzing using information entropy . . . . .	947
9.2.1 Analyzing entropy in Mathematica . . . . .	947
9.2.2 Conclusion . . . . .	954
9.2.3 Tools . . . . .	954
9.2.4 A word about primitive encryption like XORing . . . . .	954
9.2.5 More about entropy of executable code . . . . .	954
9.2.6 Random number generators . . . . .	955
9.2.7 Entropy of various files . . . . .	955
9.2.8 Making lower level of entropy . . . . .	956
9.3 Millenium game save file . . . . .	957
9.4 fortune program indexing file . . . . .	964
9.4.1 Hacking . . . . .	968
9.4.2 The files . . . . .	969
9.5 Oracle RDBMS: .SYM-files . . . . .	969
9.6 Oracle RDBMS: .MSB-files . . . . .	979
9.6.1 Summary . . . . .	986

## CONTENTS

9.7 Exercise . . . . .	986
9.8 Further reading . . . . .	986
<b>10 Other things</b>	<b>987</b>
10.1 Executable files patching . . . . .	987
10.1.1 Text strings . . . . .	987
10.1.2 x86 code . . . . .	987
10.2 Function arguments statistics . . . . .	988
10.3 Compiler intrinsic . . . . .	988
10.4 Compiler's anomalies . . . . .	989
10.4.1 Oracle RDBMS 11.2 and Intel C++ 10.1 . . . . .	989
10.4.2 MSVC 6.0 . . . . .	989
10.4.3 Summary . . . . .	990
10.5 Itanium . . . . .	990
10.6 8086 memory model . . . . .	992
10.7 Basic blocks reordering . . . . .	993
10.7.1 Profile-guided optimization . . . . .	993
<b>11 Books/blogs worth reading</b>	<b>995</b>
11.1 Books and other materials . . . . .	995
11.1.1 Reverse Engineering . . . . .	995
11.1.2 Windows . . . . .	995
11.1.3 C/C++ . . . . .	995
11.1.4 x86 / x86-64 . . . . .	996
11.1.5 ARM . . . . .	996
11.1.6 Assembly language . . . . .	996
11.1.7 Java . . . . .	996
11.1.8 UNIX . . . . .	996
11.1.9 Programming in general . . . . .	996
11.1.10 Cryptography . . . . .	997
<b>12 Communities</b>	<b>998</b>
<b>Afterword</b>	<b>1000</b>
12.1 Questions? . . . . .	1000
<b>Appendix</b>	<b>1002</b>
.1 x86 . . . . .	1002
.1.1 Terminology . . . . .	1002
.1.2 General purpose registers . . . . .	1002
.1.3 FPU registers . . . . .	1006
.1.4 SIMD registers . . . . .	1008
.1.5 Debugging registers . . . . .	1008
.1.6 Instructions . . . . .	1009
.1.7 npad . . . . .	1021
.2 ARM . . . . .	1022
.2.1 Terminology . . . . .	1022
.2.2 Versions . . . . .	1023
.2.3 32-bit ARM (AArch32) . . . . .	1023
.2.4 64-bit ARM (AArch64) . . . . .	1024
.2.5 Instructions . . . . .	1024
.3 MIPS . . . . .	1025
.3.1 Registers . . . . .	1025
.3.2 Instructions . . . . .	1026
.4 Some GCC library functions . . . . .	1026
.5 Some MSVC library functions . . . . .	1026
.6 Cheatsheets . . . . .	1027
.6.1 IDA . . . . .	1027
.6.2 OllyDbg . . . . .	1027
.6.3 MSVC . . . . .	1027
.6.4 GCC . . . . .	1028
.6.5 GDB . . . . .	1028

*CONTENTS*

---

<b>Acronyms used</b>	<b>1031</b>
<b>Glossary</b>	<b>1036</b>
<b>Index</b>	<b>1038</b>

There are several popular meanings of the term “reverse engineering”: 1) The reverse engineering of software: researching compiled programs; 2) The scanning of 3D structures and the subsequent digital manipulation required in order to duplicate them; 3) Recreating [DBMS<sup>5</sup>](#) structure. This book is about the first meaning.

## Topics discussed in-depth

x86/x64, ARM/ARM64, MIPS, Java/JVM.

## Topics touched upon

Oracle RDBMS ( [8.11 on page 899](#)), Itanium ( [10.5 on page 990](#)), copy-protection dongles ( [8.5 on page 814](#)), LD\_PRELOAD ( [6.4.2 on page 750](#)), stack overflow, [ELF<sup>6</sup>](#), win32 PE file format ( [6.5.2 on page 756](#)), x86-64 ( [1.30.1 on page 421](#)), critical sections ( [6.5.4 on page 786](#)), syscalls ( [6.3 on page 746](#)), [TLS<sup>7</sup>](#), position-independent code (PIC<sup>8</sup>) ( [6.4.1 on page 747](#)), profile-guided optimization ( [10.7.1 on page 993](#)), C++ STL ( [3.19.4 on page 563](#)), OpenMP ( [3.28 on page 648](#)), SEH ( [6.5.3 on page 763](#)).

## Prerequisites

Basic C [PL<sup>9</sup>](#) knowledge. Recommended reading: [11.1.3 on page 995](#).

## Exercises and tasks

...are all moved to the separate website: <http://challenges.re>.

## About the author



Dennis Yurichev is an experienced reverse engineer and programmer. He can be contacted by email: [dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com).

## Praise for *Reverse Engineering for Beginners*

- “Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education.” Richard M. Stallman, GNU founder, software freedom activist.
- “It’s very well done .. and for free .. amazing.”<sup>10</sup> Daniel Bilar, Siege Technologies, LLC.

<sup>5</sup>Database management systems

<sup>6</sup>Executable file format widely used in \*NIX systems including Linux

<sup>7</sup>Thread Local Storage

<sup>8</sup>Position Independent Code: [6.4.1 on page 747](#)

<sup>9</sup>Programming language

<sup>10</sup>[twitter.com/daniel\\_bilar/status/436578617221742593](https://twitter.com/daniel_bilar/status/436578617221742593)

## CONTENTS

---

- "... excellent and free"<sup>11</sup> Pete Finnigan, Oracle RDBMS security guru.
- "... book is interesting, great job!" Michael Sikorski, author of *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- "... my compliments for the very nice tutorial!" Herbert Bos, full professor at the Vrije Universiteit Amsterdam, co-author of *Modern Operating Systems (4th Edition)*.
- "... It is amazing and unbelievable." Luis Rocha, CISSP / ISSAP, Technical Manager, Network & Information Security at Verizon Business.
- "Thanks for the great work and your book." Joris van de Vis, SAP Netweaver & Security specialist.
- "... reasonable intro to some of the techniques."<sup>12</sup> Mike Stay, teacher at the Federal Law Enforcement Training Center, Georgia, US.
- "I love this book! I have several students reading it at the moment, plan to use it in graduate course."<sup>13</sup> Sergey Bratus , Research Assistant Professor at the Computer Science Department at Dartmouth College
- "Dennis @Yurichev has published an impressive (and free!) book on reverse engineering"<sup>14</sup> Tanel Poder, Oracle RDBMS performance tuning expert .
- "This book is some kind of Wikipedia to beginners..." Archer, Chinese Translator, IT Security Researcher.
- "First class reference for people wanting to learn reverse engineering. And it's free for all." Mikko Hypponen, F-Secure.

## Thanks

For patiently answering all my questions: Andrey "hermit" Baranovich, Slava "Avid" Kazakov.

For sending me notes about mistakes and inaccuracies: Stanislav "Beaver" Bobrytskyy, Alexander Lysenko, Alexander "Solar Designer" Peslyak, Federico Ramondino, Mark Wilson, Shell Rocket, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon<sup>15</sup>, Ben L., Etienne Khan, Norbert Szetei<sup>16</sup>, Marc Remy, Michael Hansen..

For helping me in other ways: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar.

For translating the book into Simplified Chinese: Antiy Labs ([antiy.cn](#)), Archer.

For translating the book into Korean: Byungho Min.

For translating the book into Dutch: Cedric Sambre (AKA Midas).

For translating the book into Spanish: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida.

For translating the book into Portuguese: Thales Stevan de A. Gois.

For translating the book into Italian: Federico Ramondino<sup>17</sup>, Paolo Stivanin<sup>18</sup>, twyK.

For translating the book into French: Florent Besnard<sup>19</sup>, Marc Remy<sup>20</sup>, Baudouin Landais, Téo Dacquet<sup>21</sup>.

For translating the book into German: Dennis Siekmeier<sup>22</sup>, Julius Angres<sup>23</sup>, Dirk Loser<sup>24</sup>.

For proofreading: Alexander "Lstar" Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

---

<sup>11</sup>[twitter.com/petefinnigan/status/400551705797869568](https://twitter.com/petefinnigan/status/400551705797869568)

<sup>12</sup>[reddit](#)

<sup>13</sup>[twitter.com/sergeybratus/status/505590326560833536](https://twitter.com/sergeybratus/status/505590326560833536)

<sup>14</sup>[twitter.com/TanelPoder/status/524668104065159169](https://twitter.com/TanelPoder/status/524668104065159169)

<sup>15</sup><https://github.com/pixjuan>

<sup>16</sup><https://github.com/73696e65>

<sup>17</sup><https://github.com/pinkrab>

<sup>18</sup><https://github.com/paolostivanin>

<sup>19</sup><https://github.com/besnardf>

<sup>20</sup><https://github.com/mremy>

<sup>21</sup><https://github.com/T30rix>

<sup>22</sup><https://github.com/DSiekmeier>

<sup>23</sup><https://github.com/JAngres>

<sup>24</sup><https://github.com/PolymathMonkey>

## CONTENTS

---

Vasil Kolev<sup>25</sup> did a great amount of work in proofreading and correcting many mistakes.

For illustrations and cover art: Andy Nechaevsky.

Thanks also to all the folks on [github.com](https://github.com) who have contributed notes and corrections<sup>26</sup>.

Many  $\text{\LaTeX}$  packages were used: I would like to thank the authors as well.

## Donors

Those who supported me during the time when I wrote significant part of the book:

2 \* Oleg Vygovsky (50+100 UAH), Daniel Bilar (\$50), James Truscott (\$4.5), Luis Rocha (\$63), Joris van de Vis (\$127), Richard S Shultz (\$20), Jang Minchang (\$20), Shade Atlas (5 AUD), Yao Xiao (\$10), Paweł Szczur (40 CHF), Justin Simms (\$20), Shawn the R0ck (\$27), Ki Chan Ahn (\$50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao (\$50), Philippe Teuwen (\$4), Martin Haeberli (\$10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai (\$15), Bayna AlZaabi (\$75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop (\$50), Nicolas Werner (€12), Jeremy Brown (\$100), Alexandre Borges (\$25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent (\$30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe (\$14), Maxim Dyakonov (\$3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov (\$100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei (\$40), Z0vsky (€10), Yu Dai (\$10).

Thanks a lot to every donor!

## mini-FAQ

Q: What are prerequisites for reading this book?

A: Basic understanding of C/C++ is desirable.

Q: Can I buy Russian/English hardcopy/paper book?

A: Unfortunately no, no publisher got interested in publishing Russian or English version so far. Meanwhile, you can ask your favorite copy shop to print/bind it.

Q: Is there epub/mobi version?

A: The book is highly dependent on TeX/LaTeX-specific hacks, so converting to HTML (epub/mobi is a set of HTMLs) will not be easy.

Q: Why should one learn assembly language these days?

A: Unless you are an OS<sup>27</sup> developer, you probably don't need to code in assembly—latest compilers (2010s) are much better at performing optimizations than humans<sup>28</sup>.

Also, latest CPU<sup>29</sup>s are very complex devices and assembly knowledge doesn't really help one to understand their internals.

That being said, there are at least two areas where a good understanding of assembly can be helpful: First and foremost, security/malware research. It is also a good way to gain a better understanding of your compiled code whilst debugging. This book is therefore intended for those who want to understand assembly language rather than to code in it, which is why there are many examples of compiler output contained within.

Q: I clicked on a hyperlink inside a PDF-document, how do I go back?

A: In Adobe Acrobat Reader click Alt+LeftArrow. In Evince click “<” button.

Q: May I print this book / use it for teaching?

A: Of course! That's why the book is licensed under the Creative Commons license (CC BY-SA 4.0).

Q: Why is this book free? You've done great job. This is suspicious, as many other free things.

---

<sup>25</sup><https://vasil.ludost.net/>

<sup>26</sup><https://github.com/dennis714/RE-for-beginners/graphs/contributors>

<sup>27</sup>Operating System

<sup>28</sup>A very good text about this topic: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

<sup>29</sup>Central processing unit

## CONTENTS

---

A: In my own experience, authors of technical literature do this mostly for self-advertisement purposes. It's not possible to get any decent money from such work.

Q: How does one get a job in reverse engineering?

A: There are hiring threads that appear from time to time on reddit, devoted to RE<sup>30</sup> (2016). Try looking there.

A somewhat related hiring thread can be found in the “netsec” subreddit: [2016](#).

Q: I have a question...

A: Send it to me by email ([dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com)).

## About the Korean translation

In January 2015, the Acorn publishing company ([www.acornpub.co.kr](http://www.acornpub.co.kr)) in South Korea did a huge amount of work in translating and publishing my book (as it was in August 2014) into Korean.

It's now available at [their website](#).

The translator is Byungho Min ([twitter/tais9](#)). The cover art was done by my artistic friend, Andy Nechaevsky: [facebook/andydinka](#). They also hold the copyright to the Korean translation.

So, if you want to have a *real* book on your shelf in Korean and want to support my work, it is now available for purchase.

## About the Persian/Farsi translation

In 2016 the book has been translated by Mohsen Mostafa Jokar (who is also known to Iranian community by his translation of Radare manual<sup>31</sup>). It is available on the publisher's website<sup>32</sup> (Pendare Pars).

40 page excerpt: <https://beginners.re/farsi.pdf>.

Registration of the book in National Library of Iran: <http://opac.nlai.ir/opac-prod/bibliographic/4473995>.

## About the Chinese translation

In April 2017, translation to Chinese has been finished by Chinese PTPress publisher. They are also the Chinese translation copyright holder.

It's available for order here: <http://www.epubit.com.cn/book/details/4174>. Some kind of review and history behind the translation: <http://www.cptoday.cn/news/detail/3155>.

Principal translator is Archer, to whom I owe so much. He was extremely meticulous (in good sense) and reported most of known mistakes and bugs, which is very important to literature like this book. I'll recommend his services to any other author!

Guys from [Antiy Labs](#) has also helped with translation. [Here is preface](#) written by them.

---

<sup>30</sup>[reddit.com/r/ReverseEngineering/](http://reddit.com/r/ReverseEngineering/)

<sup>31</sup><http://rada.re/get/radare2book-persian.pdf>

<sup>32</sup><http://goo.gl/2Tzx0H>

# Chapter 1

## Code patterns

Everything is comprehended through comparison

---

Author unknown

### 1.1 The method

When the author of this book first started learning C and, later, C++, he used to write small pieces of code, compile them, and then look at the assembly language output. This made it very easy for him to understand what was going on in the code that he had written.<sup>1</sup>. He did it so many times that the relationship between the C/C++ code and what the compiler produced was imprinted deeply in his mind. It's easy to imagine instantly a rough outline of C code's appearance and function. Perhaps this technique could be helpful for others.

Sometimes ancient compilers are used here, in order to get the shortest (or simplest) possible code snippet.

By the way, there is a great website where you can do the same, with various compilers, instead of installing them on your box. You can use it as well: <https://gcc.godbolt.org/>.

### Exercises

When the author of this book studied assembly language, he also often compiled small C-functions and then rewrote them gradually to assembly, trying to make their code as short as possible. This probably is not worth doing in real-world scenarios today, because it's hard to compete with latest compilers in terms of efficiency. It is, however, a very good way to gain a better understanding of assembly. Feel free, therefore, to take any assembly code from this book and try to make it shorter. However, don't forget to test what you have written.

### Optimization levels and debug information

Source code can be compiled by different compilers with various optimization levels. A typical compiler has about three such levels, where level zero means disable optimization. Optimization can also be targeted towards code size or code speed. A non-optimizing compiler is faster and produces more understandable (albeit verbose) code, whereas an optimizing compiler is slower and tries to produce code that runs faster (but is not necessarily more compact). In addition to optimization levels, a compiler can include in the resulting file some debug information, thus producing code for easy debugging. One of the important features of the 'debug' code is that it might contain links between each line of the source code and the respective machine code addresses. Optimizing compilers, on the other hand, tend to produce output where entire lines of source code can be optimized away and thus not even be present in the resulting machine code. Reverse engineers can encounter either version, simply because some developers

---

<sup>1</sup>In fact, he still does it when he can't understand what a particular bit of code does.

## 1.2. SOME BASICS

turn on the compiler's optimization flags and others do not. Because of this, we'll try to work on examples of both debug and release versions of the code featured in this book, where possible.

# 1.2 Some basics

## 1.2.1 A short introduction to the CPU

The **CPU** is the device that executes the machine code a program consists of.

### A short glossary:

**Instruction** : A primitive **CPU** command. The simplest examples include: moving data between registers, working with memory, primitive arithmetic operations. As a rule, each **CPU** has its own instruction set architecture (**ISA**<sup>2</sup>).

**Machine code** : Code that the **CPU** directly processes. Each instruction is usually encoded by several bytes.

**Assembly language** : Mnemonic code and some extensions like macros that are intended to make a programmer's life easier.

**CPU register** : Each **CPU** has a fixed set of general purpose registers (**GPR**<sup>3</sup>).  $\approx 8$  in x86,  $\approx 16$  in x86-64,  $\approx 16$  in ARM. The easiest way to understand a register is to think of it as an untyped temporary variable. Imagine if you were working with a high-level **PL** and could only use eight 32-bit (or 64-bit) variables. Yet a lot can be done using just these!

One might wonder why there needs to be a difference between machine code and a **PL**. The answer lies in the fact that humans and **CPU**s are not alike—it is much easier for humans to use a high-level **PL** like C/C++, Java, Python, etc., but it is easier for a **CPU** to use a much lower level of abstraction. Perhaps it would be possible to invent a **CPU** that can execute high-level **PL** code, but it would be many times more complex than the **CPU**s we know of today. In a similar fashion, it is very inconvenient for humans to write in assembly language, due to it being so low-level and difficult to write in without making a huge number of annoying mistakes. The program that converts the high-level **PL** code into assembly is called a *compiler*. <sup>4</sup>.

### A couple of words about different **ISAs**

The x86 **ISA** has always been one with variable-length instructions, so when the 64-bit era came, the x64 extensions did not impact the **ISA** very significantly. In fact, the x86 **ISA** still contains a lot of instructions that first appeared in 16-bit 8086 CPU, yet are still found in the CPUs of today. ARM is a **RISC**<sup>5</sup> **CPU** designed with constant-length instructions in mind, which had some advantages in the past. In the very beginning, all ARM instructions were encoded in 4 bytes<sup>6</sup>. This is now referred to as "ARM mode". Then they thought it wasn't as frugal as they first imagined. In fact, most used **CPU** instructions<sup>7</sup> in real world applications can be encoded using less information. They therefore added another **ISA**, called Thumb, where each instruction was encoded in just 2 bytes. This is now referred as "Thumb mode". However, not *all* ARM instructions can be encoded in just 2 bytes, so the Thumb instruction set is somewhat limited. It is worth noting that code compiled for ARM mode and Thumb mode may of course coexist within one single program. The ARM creators thought Thumb could be extended, giving rise to Thumb-2, which appeared in ARMv7. Thumb-2 still uses 2-byte instructions, but has some new instructions which have the size of 4 bytes. There is a common misconception that Thumb-2 is a mix of ARM and Thumb. This is incorrect. Rather, Thumb-2 was extended to fully support all processor features so it could compete with ARM mode—a goal that was clearly achieved, as the majority of applications for iPod/iPhone/iPad are compiled for the Thumb-2 instruction set (admittedly, largely due to the fact that Xcode does this by default). Later the 64-bit ARM came out. This **ISA** has 4-byte instructions, and lacked the need of any additional Thumb mode. However, the 64-bit requirements affected the **ISA**, resulting in us now having three ARM instruction sets: ARM mode, Thumb mode (including Thumb-2) and ARM64. These **ISAs** intersect partially, but it can be said that they are different **ISAs**, rather than variations of the same one. Therefore, we would try to add

<sup>2</sup>Instruction Set Architecture

<sup>3</sup>General Purpose Registers

<sup>4</sup>Old-school Russian literature also use term "translator".

<sup>5</sup>Reduced instruction set computing

<sup>6</sup>By the way, fixed-length instructions are handy because one can calculate the next (or previous) instruction address without effort. This feature will be discussed in the switch() operator ( [1.15.2 on page 174](#) ) section.

<sup>7</sup>These are MOV/PUSH/CALL/jcc

## 1.2. SOME BASICS

fragments of code in all three ARM ISAs in this book. There are, by the way, many other RISC ISAs with fixed length 32-bit instructions, such as MIPS, PowerPC and Alpha AXP.

### 1.2.2 Numeral systems

Humans accustomed to decimal numeral system probably because almost all ones has 10 fingers. Nevertheless, number 10 has no significant meaning in science and mathematics. Natural numeral system in digital electronics is binary: 0 is for absence of current in wire and 1 for presence. 10 in binary is 2 in decimal; 100 in binary is 4 in decimal and so on.

If the numeral system has 10 digits, it has *radix* (or *base*) of 10. Binary numeral system has *radix* of 2.

Important things to recall: 1) *number* is a number, while *digit* is a term of writing system and is usually one character; 2) number is not changed when converted to another radix: writing notation is (and way of representing it in RAM<sup>8</sup>).

How to convert a number from one radix to another?

Positional notation is used almost everywhere, this means, a digit has some weight depending on where it is placed inside of number. If 2 is placed at the rightmost place, it's 2. If it is placed at the place one digit before rightmost, it's 20.

What does 1234 stand for?

$$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234 \text{ or } 1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$$

Same story for binary numbers, but base is 2 instead of 10. What does 0b101011 stand for?

$$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43 \text{ or } 32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$$

Positional notation can be opposed to non-positional notation such as Roman numeric system <sup>9</sup>. Perhaps, humankind switched to positional notation because it's easier to do basic operations (addition, multiplication, etc.) on paper by hand.

Indeed, binary numbers can be added, subtracted and so on in the very same as taught in schools, but only 2 digits are available.

Binary numbers are bulky when represented in source code and dumps, so that is where hexadecimal numeral system can be used. Hexadecimal radix uses 0..9 digits and also 6 Latin characters: A..F. Each hexadecimal digit takes 4 bits or 4 binary digits, so it's very easy to convert from binary number to hexadecimal and back, even manually, in one's mind.

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

How to understand, which radix is used in a specific place?

Decimal numbers are usually written as is, i.e., 1234. But some assemblers allows to make emphasis on decimal radix and this number can be written with "d" suffix: 1234d.

<sup>8</sup>Random-access memory

<sup>9</sup>About numeric system evolution, see [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

## 1.2. SOME BASICS

Binary numbers sometimes prepended with "0b" prefix: 0b100110111 (GCC<sup>10</sup> has non-standard language extension for this<sup>11</sup>). There is also another way: "b" suffix, for example: 100110111b. I'll try to stick to "0b" prefix throughout the book for binary numbers.

Hexadecimal numbers are prepended with "0x" prefix in C/C++ and other PLs: 0x1234ABCD. Or they are has "h" suffix: 1234ABC Dh—this is common way of representing them in assemblers and debuggers. If the number is started with A..F digit, 0 is to be added before: 0ABCDEFh. There was also convention that was popular in 8-bit home computers era, using \$ prefix, like \$ABCD. I'll try to stick to "0x" prefix throughout the book for hexadecimal numbers.

Should one learn to convert numbers in mind? A table of 1-digit hexadecimal numbers can easily be memorized. As of larger numbers, probably, it's not worth to torment yourself.

Perhaps, the most visible to all people hexadecimal numbers are in URL<sup>12</sup>s. This is the way how non-Latin characters are encoded. For example: <https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9> is the URL of Wiktionary article about “naïveté” word.

## Octal radix

Another numeral system heavily used in past of computer programming is octal: there are 8 digits (0..7) and each is mapped to 3 bits, so it's easy to convert numbers back and forth. It has been superseded by hexadecimal system almost everywhere, but surprisingly, there is \*NIX utility used by many people often which takes octal number as argument: `chmod`.

As many \*NIX users know, `chmod` argument can be a number of 3 digits. The first digit is rights for owner of file, second is rights for group (to which file belongs), third is for everyone else. And each digit can be represented in binary form:

decimal	binary	meaning
7	111	<b>rwx</b>
6	110	<b>rw-</b>
5	101	<b>r-x</b>
4	100	<b>r--</b>
3	011	<b>-wx</b>
2	010	<b>-w-</b>
1	001	<b>--x</b>
0	000	<b>---</b>

So each bit is mapped to a flag: read/write/execute.

Now the reason why I'm talking about `chmod` here is that the whole number in argument can be represented as octal number. Let's take for example, 644. When you run `chmod 644 file`, you set read/write permissions for owner, read permissions for group and again, read permissions for everyone else. Let's convert 644 octal number to binary, this will be `110100100`, or (in groups of 3 bits) `110 100 100`.

Now we see that each triplet describe permissions for owner/group/others: first is `rw-`, second is `r--` and third is `r--`.

Octal numeral system was also popular on old computers like PDP-8, because word there could be 12, 24 or 36 bits, and these numbers are divisible by 3, so octal system was natural on that environment. Nowadays, all popular computers employs word/address size of 16, 32 or 64 bits, and these numbers are divisible by 4, so hexadecimal system is more natural here.

Octal numeral system is supported by all standard C/C++ compilers. This is source of confusion sometimes, because octal numbers are encoded with zero prepended, for example, 0377 is 255. And sometimes, you may make a typo and write "09" instead of 9, and the compiler would report error. GCC may report something like that:

```
error: invalid digit "9" in octal constant.
```

<sup>10</sup>GNU Compiler Collection

<sup>11</sup><https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html>

<sup>12</sup>Uniform Resource Locator

## Divisibility

When you see a decimal number like 120, you can quickly deduce that it's divisible by 10, because the last digit is zero. In the same way, 123400 is divisible by 100, because two last digits are zeros.

Likewise, hexadecimal number 0x1230 is divisible by 0x10 (or 16), 0x123000 is divisible by 0x1000 (or 4096), etc.

Binary number 0b1000101000 is divisible by 0b1000 (8), etc.

This property can be used often to realize quickly if a size of some block in memory is padded to some boundary. For example, sections in PE<sup>13</sup> files are almost always started at addresses ending with 3 hexadecim zeros: 0x41000, 0x10001000, etc. The reason behind this is in the fact that almost all PE sections are padded to boundary of 0x1000 (4096) bytes.

## Multi-precision arithmetic and radix

Multi-precision arithmetic can use huge numbers, and each one may be stored in several bytes. For example, RSA keys, both public and private, are spanning up to 4096 bits and maybe even more.

In [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] we can find the following idea: when you store multi-precision number in several bytes, the whole number can be represented as having a radix of  $2^8 = 256$ , and each digit goes to corresponding byte. Likewise, if you store multi-precision number in several 32-bit integer values, each digit goes to each 32-bit slot, and you may think about this number as stored in radix of  $2^{32}$ .

## Pronouncement

Numbers in non-decimal base are usually pronounced by one digit: "one-zero-zero-one-one-...". Words like "ten", "thousand", etc, are usually not pronounced, because it will be confused with decimal base then.

## Floating point numbers

To distinguish floating point numbers from integer ones, they are usually written with ".0" at the end, like 0.0, 123.0, etc.

# 1.3 Empty function

The simplest possible function is arguably one that does nothing:

Listing 1.1: C/C++ Code

```
void f()
{
    return;
};
```

Lets compile it!

## 1.3.1 x86

Here's what both the optimizing GCC and MSVC compilers produce on the x86 platform:

Listing 1.2: Optimizing GCC/MSVC (assembly output)

```
f:
    ret
```

There is just one instruction: RET, which returns execution to the caller.

<sup>13</sup>Portable Executable

### 1.3.2 ARM

Listing 1.3: Optimizing Keil 6/2013 (ARM mode) assembly output

```
f      PROC
      BX      lr
      ENDP
```

The return address is not saved on the local stack in the ARM [ISA](#), but rather in the link register, so the `BX LR` instruction causes execution to jump to that address—effectively returning execution to the [caller](#).

### 1.3.3 MIPS

There are two naming conventions used in the world of MIPS when naming registers: by number (from \$0 to \$31) or by pseudo name (\$V0, \$A0, etc.).

The GCC assembly output below lists registers by number:

Listing 1.4: Optimizing GCC 4.4.5 (assembly output)

```
j      $31
nop
```

...while [IDA](#)<sup>14</sup> does it—by their pseudo names:

Listing 1.5: Optimizing GCC 4.4.5 (IDA)

```
j      $ra
nop
```

The first instruction is the jump instruction (J or JR) which returns the execution flow to the [caller](#), jumping to the address in the \$31 (or \$RA) register.

This is the register analogous to [LR](#)<sup>15</sup> in ARM.

The second instruction is [NOP](#)<sup>16</sup>, which does nothing. We can ignore it so far.

#### A note about MIPS instruction/register names

Register and instruction names in the world of MIPS are traditionally written in lowercase. However, for the sake of consistency, we'll stick to using uppercase letters, as it is the convention followed by all other [ISAs](#) featured this book.

### 1.3.4 Empty functions in practice

Despite the fact empty functions are useless, they are quite frequent in low-level code.

First of all, debugging functions are quite popular, like this one:

Listing 1.6: C/C++ Code

```
void dbg_print (const char *fmt, ...)
{
#ifndef _DEBUG
    // open log file
    // write to log file
    // close log file
#endif
};

void some_function()
{
    ...
}
```

<sup>14</sup> Interactive Disassembler and debugger developed by [Hex-Rays](#)

<sup>15</sup>Link Register

<sup>16</sup>No OPeration

## 1.4. RETURNING VALUE

```
    dbg_print ("we did something\n");
    ...
};
```

In non-debug build (e.g., “release”), `_DEBUG` is not defined, so `dbg_print()` function, despite still being called during execution, will be empty.

Another popular way of software protection is make several builds: one for legal customers, and a demo build. Demo build can lack some important functions, like this:

Listing 1.7: C/C++ Code

```
void save_file ()
{
#ifndef DEMO
    // actual saving code
#endif
};
```

`save_file()` function can be called when user click `File->Save` menu. Demo version may be delivered with this menu item disabled, but even if software cracker will enable it, empty function with no useful code will be called.

IDA marks such functions with names like `nullsub_00`, `nullsub_01`, etc.

## 1.4 Returning value

Another simple function is the one that simply returns a constant value:

Listing 1.8: C/C++ Code

```
int f()
{
    return 123;
};
```

Lets compile it.

### 1.4.1 x86

Here’s what both the optimizing GCC and MSVC compilers produce on the x86 platform:

Listing 1.9: Optimizing GCC/MSVC (assembly output)

```
f:
    mov     eax, 123
    ret
```

There are just two instructions: the first places the value 123 into the `EAX` register, which is used by convention for storing the return value and the second one is `RET`, which returns execution to the [caller](#).

The caller will take the result from the `EAX` register.

### 1.4.2 ARM

There are a few differences on the ARM platform:

Listing 1.10: Optimizing Keil 6/2013 (ARM mode) ASM Output

```
f      PROC
      MOV      r0,#0x7b ; 123
      BX      lr
ENDP
```

## 1.5. HELLO, WORLD!

ARM uses the register `R0` for returning the results of functions, so 123 is copied into `R0`.

It is worth noting that `MOV` is a misleading name for the instruction in both x86 and ARM [ISAs](#).

The data is not in fact *moved*, but *copied*.

### 1.4.3 MIPS

The GCC assembly output below lists registers by number:

Listing 1.11: Optimizing GCC 4.4.5 (assembly output)

```
j      $31  
li    $2,123          # 0x7b
```

...while [IDA](#) does it by their pseudo names:

Listing 1.12: Optimizing GCC 4.4.5 (IDA)

```
jr   $ra  
li   $v0, 0x7B
```

The `$2` (or `$V0`) register is used to store the function's return value. `LI` stands for "Load Immediate" and is the MIPS equivalent to `MOV`.

The other instruction is the jump instruction (`J` or `JR`) which returns the execution flow to the [caller](#).

You might be wondering why positions of the load instruction (`LI`) and the jump instruction (`J` or `JR`) are swapped. This is due to a [RISC](#) feature called "branch delay slot".

The reason this happens is a quirk in the architecture of some RISC [ISAs](#) and isn't important for our purposes—we just must keep in mind that in MIPS, the instruction following a jump or branch instruction is executed *before* the jump/branch instruction itself.

As a consequence, branch instructions always swap places with the instruction which must be executed beforehand.

### 1.4.4 In practice

Functions which merely returns 1 (*true*) or 0 (*false*) are very frequent in practice.

## 1.5 Hello, world!

Let's use the famous example from the book [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, (1988)]:

```
#include <stdio.h>  
  
int main()  
{  
    printf("hello, world\n");  
    return 0;  
}
```

### 1.5.1 x86

#### MSVC

Let's compile it in MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

## 1.5. HELLO, WORLD!

(`/Fa` option instructs the compiler to generate assembly listing file)

Listing 1.13: MSVC 2010

```
CONST SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_main PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG3830
    call    _printf
    add    esp, 4
    xor    eax, eax
    pop    ebp
    ret    0
_main ENDP
_TEXT ENDS
```

MSVC produces assembly listings in Intel-syntax. The difference between Intel-syntax and AT&T-syntax will be discussed in [1.5.1 on page 11](#).

The compiler generated the file, `1.obj`, which is to be linked into `1.exe`. In our case, the file contains two segments: `CONST` (for data constants) and `_TEXT` (for code).

The string `hello, world` in C/C++ has type `const char[]` [Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], but it does not have its own name. The compiler needs to deal with the string somehow so it defines the internal name `$SG3830` for it.

That is why the example may be rewritten as follows:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Let's go back to the assembly listing. As we can see, the string is terminated by a zero byte, which is standard for C/C++ strings. More about C/C++ strings: [5.4.1 on page 710](#).

In the code segment, `_TEXT`, there is only one function so far: `main()`. The function `main()` starts with prologue code and ends with epilogue code (like almost any function)<sup>17</sup>.

After the function prologue we see the call to the `printf()` function:

`CALL _printf`. Before the call, a string address (or a pointer to it) containing our greeting is placed on the stack with the help of the `PUSH` instruction.

When the `printf()` function returns the control to the `main()` function, the string address (or a pointer to it) is still on the stack. Since we do not need it anymore, the `stack pointer` (the `ESP` register) needs to be corrected.

`ADD ESP, 4` means add 4 to the `ESP` register value.

Why 4? Since this is a 32-bit program, we need exactly 4 bytes for address passing through the stack. If it was x64 code we would need 8 bytes. `ADD ESP, 4` is effectively equivalent to `POP register` but without using any register<sup>18</sup>.

<sup>17</sup>You can read more about it in the section about function prologues and epilogues ([1.6 on page 30](#)).

<sup>18</sup>CPU flags, however, are modified

## 1.5. HELLO, WORLD!

For the same purpose, some compilers (like the Intel C++ Compiler) may emit `POP ECX` instead of `ADD` (e.g., such a pattern can be observed in the Oracle RDBMS code as it is compiled with the Intel C++ compiler). This instruction has almost the same effect but the `ECX` register contents will be overwritten. The Intel C++ compiler supposedly uses `POP ECX` since this instruction's opcode is shorter than `ADD ESP, X` (1 byte for `POP` against 3 for `ADD`).

Here is an example of using `POP` instead of `ADD` from Oracle RDBMS:

Listing 1.14: Oracle RDBMS 10.2 Linux (app.o file)

```
.text:0800029A          push    ebx
.text:0800029B          call    qksfroChild
.text:080002A0          pop     ecx
```

After calling `printf()`, the original C/C++ code contains the statement `return 0` —return 0 as the result of the `main()` function.

In the generated code this is implemented by the instruction `XOR EAX, EAX`.

`XOR` is in fact just “eXclusive OR”<sup>19</sup> but the compilers often use it instead of `MOV EAX, 0`—again because it is a slightly shorter opcode (2 bytes for `XOR` against 5 for `MOV`).

Some compilers emit `SUB EAX, EAX`, which means *SUBtract the value in the EAX from the value in EAX*. That in any case will results in zero.

The last instruction `RET` returns the control to the `caller`. Usually, this is C/C++ `CRT`<sup>20</sup> code which in turn returns control to the `OS`.

## GCC

Now let's try to compile the same C/C++ code in the GCC 4.4.1 compiler in Linux: `gcc 1.c -o 1`. Next, with the assistance of the `IDA` disassembler, let's see how the `main()` function was created. `IDA`, like MSVC, uses Intel-syntax<sup>21</sup>.

Listing 1.15: code in IDA

```
main          proc near
var_10        = dword ptr -10h
              push    ebp
              mov     ebp, esp
              and    esp, 0FFFFFFF0h
              sub    esp, 10h
              mov    eax, offset aHelloWorld ; "hello, world\n"
              mov    [esp+10h+var_10], eax
              call   _printf
              mov    eax, 0
              leave
              retn
main          endp
```

The result is almost the same. The address of the `hello, world` string (stored in the data segment) is loaded in the `EAX` register first and then it is saved onto the stack.

In addition, the function prologue has `AND ESP, 0FFFFFFF0h`—this instruction aligns the `ESP` register value on a 16-byte boundary. This results in all values in the stack being aligned the same way (The CPU performs better if the values it is dealing with are located in memory at addresses aligned on a 4-byte or 16-byte boundary)<sup>22</sup>.

`SUB ESP, 10h` allocates 16 bytes on the stack. Although, as we can see hereafter, only 4 are necessary here.

<sup>19</sup>[wikipedia](#)

<sup>20</sup>C runtime library

<sup>21</sup>We could also have GCC produce assembly listings in Intel-syntax by applying the options `-S -masm=intel`.

<sup>22</sup>[Wikipedia: Data structure alignment](#)

## 1.5. HELLO, WORLD!

This is because the size of the allocated stack is also aligned on a 16-byte boundary.

The string address (or a pointer to the string) is then stored directly onto the stack without using the `PUSH` instruction. `var_10` —is a local variable and is also an argument for `printf()`. Read about it below.

Then the `printf()` function is called.

Unlike MSVC, when GCC is compiling without optimization turned on, it emits `MOV EAX, 0` instead of a shorter opcode.

The last instruction, `LEAVE` —is the equivalent of the `MOV ESP, EBP` and `POP EBP` instruction pair —in other words, this instruction sets the `stack pointer` (`ESP`) back and restores the `EBP` register to its initial state. This is necessary since we modified these register values (`ESP` and `EBP`) at the beginning of the function (by executing `MOV EBP, ESP / AND ESP, ...`).

### GCC: AT&T syntax

Let's see how this can be represented in assembly language AT&T syntax. This syntax is much more popular in the UNIX-world.

Listing 1.16: let's compile in GCC 4.7.3

```
gcc -S 1_1.c
```

We get this:

Listing 1.17: GCC 4.7.3

```
.file "1_1.c"
.section .rodata
.LC0:
.string "hello, world\n"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
andl $-16, %esp
subl $16, %esp
movl $.LC0, (%esp)
call printf
movl $0, %eax
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
.section .note.GNU-stack,"",@progbits
```

The listing contains many macros (beginning with dot). These are not interesting for us at the moment.

For now, for the sake of simplification, we can ignore them (except the `.string` macro which encodes a null-terminated character sequence just like a C-string). Then we'll see this <sup>23</sup>:

Listing 1.18: GCC 4.7.3

```
.LC0:
.string "hello, world\n"
```

<sup>23</sup>This GCC option can be used to eliminate “unnecessary” macros: `-fno-asynchronous-unwind-tables`

## 1.5. HELLO, WORLD!

main:

```
pushl %ebp
movl %esp, %ebp
andl $-16, %esp
subl $16, %esp
movl $.LC0, (%esp)
call printf
movl $0, %eax
leave
ret
```

Some of the major differences between Intel and AT&T syntax are:

- Source and destination operands are written in opposite order.

In Intel-syntax: <instruction> <destination operand> <source operand>.

In AT&T syntax: <instruction> <source operand> <destination operand>.

Here is an easy way to memorize the difference: when you deal with Intel-syntax, you can imagine that there is an equality sign (=) between operands and when you deal with AT&T-syntax imagine there is a right arrow (→)<sup>24</sup>.

- AT&T: Before register names, a percent sign must be written (%) and before numbers a dollar sign (\$). Parentheses are used instead of brackets.

- AT&T: A suffix is added to instructions to define the operand size:

- q — quad (64 bits)
- l — long (32 bits)
- w — word (16 bits)
- b — byte (8 bits)

Let's go back to the compiled result: it is identical to what we saw in [IDA](#). With one subtle difference: `0xFFFFFFFF0h` is presented as `$-16`. It is the same thing: `16` in the decimal system is `0x10` in hexadecimal. `-0x10` is equal to `0xFFFFFFFF0` (for a 32-bit data type).

One more thing: the return value is to be set to 0 by using the usual `MOV`, not `XOR`. `MOV` just loads a value to a register. Its name is a misnomer (data is not moved but rather copied). In other architectures, this instruction is named “LOAD” or “STORE” or something similar.

## String patching (Win32)

We can easily find “hello, world” string in executable file using Hiew:

The screenshot shows the Hiew debugger interface with the title "Hiew: hw\_spanish.exe". The memory dump window displays the file structure and the string "hello, world" at address 0x40003000. The string is highlighted in yellow.

Address	Hex Value	ASCII Value
0x400025E0	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00	
0x400025F0	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00	
0x40003000	68 65 6C 6C-6F 2C 20 77-6F 72 6C 64-0A 00 00 00	hello, world
0x40003010	01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00	
0x40003020	32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF	
0x40003030	20 -Ö+ =] ¶f L	
0x40003040	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00	

Figure 1.1: Hiew

And we can try to translate our message to Spanish language:

<sup>24</sup>By the way, in some C standard functions (e.g., `memcpy()`, `strcpy()`) the arguments are listed in the same way as in Intel-syntax: first the pointer to the destination memory block, and then the pointer to the source memory block.

## 1.5. HELLO, WORLD!

```
Hiew: hw_spanish.exe
C:\tmp\hw_spanish.exe FWO EDITMODE PE+ 00000000`0000120D Hiew 8.02
000011E0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000011F0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00001200: 68 6F 6C 61-2C 20 6D 75-6E 64 6F 0A-00 00 00 00 hola, mundo
00001210: 01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00
00001220: 32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF 2d-Ö+ =] TfL
00001230: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00001240: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

Figure 1.2: Hiew

Spanish text is one byte shorter than English, so we also add 0x0A byte at the end (\n) and zero byte. It works.

What if we want to insert longer message? There are some zero bytes after original English text. Hard to say if they can be overwritten: they may be used somewhere in [CRT](#) code, or maybe not. Anyway, you can overwrite them only if you really know what you are doing.

## String patching (Linux x64)

Let's try to patch Linux x64 executable using rada.re:

Listing 1.19: rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c
dennis@bigbox ~/tmp % radare2 a.out
-- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x004005c4 6865 6c6c 6f2c 2077 6f72 6c64 0000 0000 hello, world....
0x004005d4 011b 033b 3000 0000 0500 0000 1cfe ffff ...;0.....
0x004005e4 7c00 0000 5cfe ffff 4c00 0000 52ff ffff |...\.L..R...
0x004005f4 a400 0000 6cff ffff c400 0000 dcff ffff ....l.....
0x00400604 0c01 0000 1400 0000 0000 0000 017a 5200 .....zR.
0x00400614 0178 1001 1b0c 0708 9001 0710 1400 0000 .x.....
0x00400624 1c00 0000 08fe ffff 2a00 0000 0000 0000 .....*.....
0x00400634 0000 0000 1400 0000 0000 0000 017a 5200 .....zR.
0x00400644 0178 1001 1b0c 0708 9001 0000 2400 0000 .x.....$...
0x00400654 1c00 0000 98fd ffff 3000 0000 000e 1046 .....0.....F
0x00400664 0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422 ..J..w...?..;*3$""
0x00400674 0000 0000 1c00 0000 4400 0000 a6fe ffff .....D.....
0x00400684 1500 0000 0041 0e10 8602 430d 0650 0c07 .....A....C..P..
0x00400694 0800 0000 4400 0000 6400 0000 a0fe ffff .....D....d.....
0x004006a4 6500 0000 0042 0e10 8f02 420e 188e 0345 e....B....B....E
0x004006b4 0e20 8d04 420e 288c 0548 0e30 8606 480e . .B.(..H.0..H.

[0x004005c4]> oo+
File a.out reopened in read-write mode

[0x004005c4]> w hola, mundo\x00

[0x004005c4]> q
```

## 1.5. HELLO, WORLD!

```
dennis@bigbox ~ /tmp % ./a.out  
hola, mundo
```

What I do here: I search for “hello” string using `/` command, then I set *cursor* (or *seek* in rada.re terms) to that address. Then I want to be sure that this is really that place: `px` dumps bytes there. `oo+` switches rada.re to *read-write* mode. `w` writes ASCII string at the current seek. Note `\00` at the end—this is zero byte. `q` quits.

## Software localization of MS-DOS era

The way I described was a common way to translate MS-DOS software to Russian language back to 1980’s and 1990’s. Russian words and sentences are usually slightly longer than its English counterparts, so that is why *localized* software has a lot of weird acronyms and hardly readable abbreviations.

Perhaps, this also happened to other languages during that era.

### 1.5.2 x86-64

#### MSVC: x86-64

Let’s also try 64-bit MSVC:

Listing 1.20: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H  
  
main    PROC  
        sub     rsp, 40  
        lea     rcx, OFFSET FLAT:$SG2989  
        call    printf  
        xor    eax, eax  
        add    rsp, 40  
        ret    0  
main    ENDP
```

In x86-64, all registers were extended to 64-bit and now their names have an `R-` prefix. In order to use the stack less often (in other words, to access external memory/cache less often), there exists a popular way to pass function arguments via registers (*fastcall*) [6.1.3 on page 734](#). I.e., a part of the function arguments is passed in registers, the rest—via the stack. In Win64, 4 function arguments are passed in the `RCX`, `RDX`, `R8`, `R9` registers. That is what we see here: a pointer to the string for `printf()` is now passed not in the stack, but in the `RCX` register. The pointers are 64-bit now, so they are passed in the 64-bit registers (which have the `R-` prefix). However, for backward compatibility, it is still possible to access the 32-bit parts, using the `E-` prefix. This is how the `RAX / EAX / AX / AL` register looks like in x86-64:

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RAX <sup>x64</sup>							
				EAX			
				AX			
				AH		AL	

The `main()` function returns an *int*-typed value, which is, in C/C++, for better backward compatibility and portability, still 32-bit, so that is why the `EAX` register is cleared at the function end (i.e., the 32-bit part of the register) instead of `RAX`. There are also 40 bytes allocated in the local stack. This is called the “shadow space”, about which we are going to talk later: [1.10.2 on page 101](#).

#### GCC: x86-64

Let’s also try GCC in 64-bit Linux:

Listing 1.21: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
    sub    rsp, 8
    mov    edi, OFFSET FLAT:.LC0 ; "hello, world\n"
    xor    eax, eax ; number of vector registers passed
    call   printf
    xor    eax, eax
    add    rsp, 8
    ret
```

A method to pass function arguments in registers is also used in Linux, \*BSD and Mac OS X is [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>25</sup>.

The first 6 arguments are passed in the `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` registers, and the rest—via the stack.

So the pointer to the string is passed in `EDI` (the 32-bit part of the register). But why not use the 64-bit part, `rdi`?

It is important to keep in mind that all `MOV` instructions in 64-bit mode that write something into the lower 32-bit register part also clear the higher 32-bits (as stated in Intel manuals: [11.1.4 on page 996](#)).

I.e., the `MOV EAX, 011223344h` writes a value into `RAX` correctly, since the higher bits will be cleared.

If we open the compiled object file (.o), we can also see all the instructions' opcodes<sup>26</sup>:

Listing 1.22: GCC 4.4.6 x64

```
.text:00000000004004D0          main  proc near
.text:00000000004004D0 48 83 EC 08    sub    rsp, 8
.text:00000000004004D4 BF E8 05 40 00  mov    edi, offset format ; "hello, world\n"
.text:00000000004004D9 31 C0    xor    eax, eax
.text:00000000004004DB E8 D8 FE FF FF  call   _printf
.text:00000000004004E0 31 C0    xor    eax, eax
.text:00000000004004E2 48 83 C4 08    add    rsp, 8
.text:00000000004004E6 C3        retn
.text:00000000004004E6          main  endp
```

As we can see, the instruction that writes into `EDI` at `0x4004D4` occupies 5 bytes. The same instruction writing a 64-bit value into `rdi` occupies 7 bytes. Apparently, GCC is trying to save some space. Besides, it can be sure that the data segment containing the string will not be allocated at the addresses higher than 4GiB.

We also see that the `EAX` register has been cleared before the `printf()` function call. This is done because according to **ABI**<sup>27</sup> standard mentioned above, the number of used vector registers is passed in `EAX` in \*NIX systems on x86-64.

## Address patching (Win64)

When our example compiled in MSVC2013 using `\MD` switch (meaning smaller executable due to `MSVCR*.DLL` file linkage), the `main()` function came first and can be easily found:

<sup>25</sup>Also available as <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

<sup>26</sup>This must be enabled in **Options** → **Disassembly** → **Number of opcode bytes**

<sup>27</sup>**ABI!**

## 1.5. *HELLO, WORLD!*

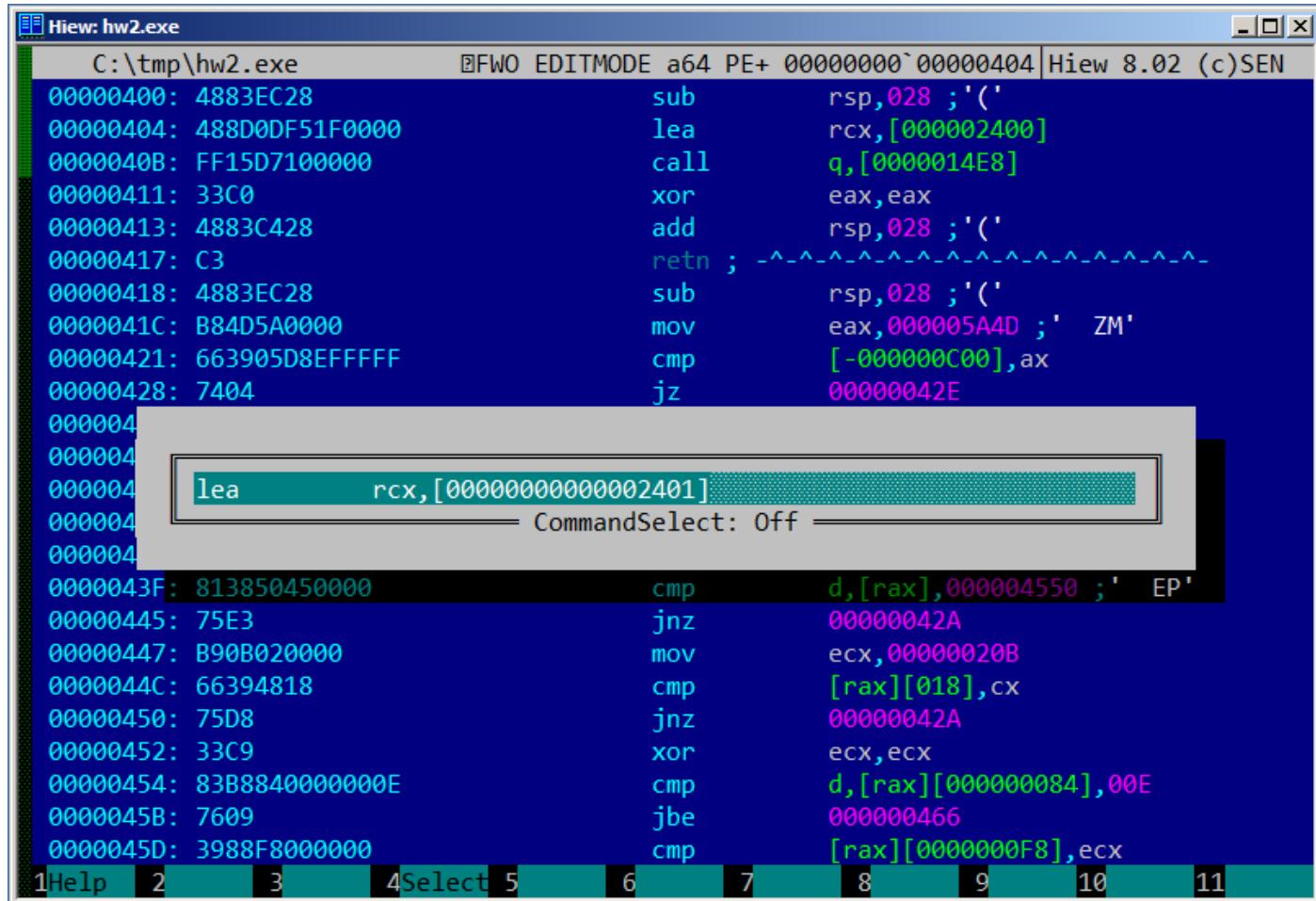
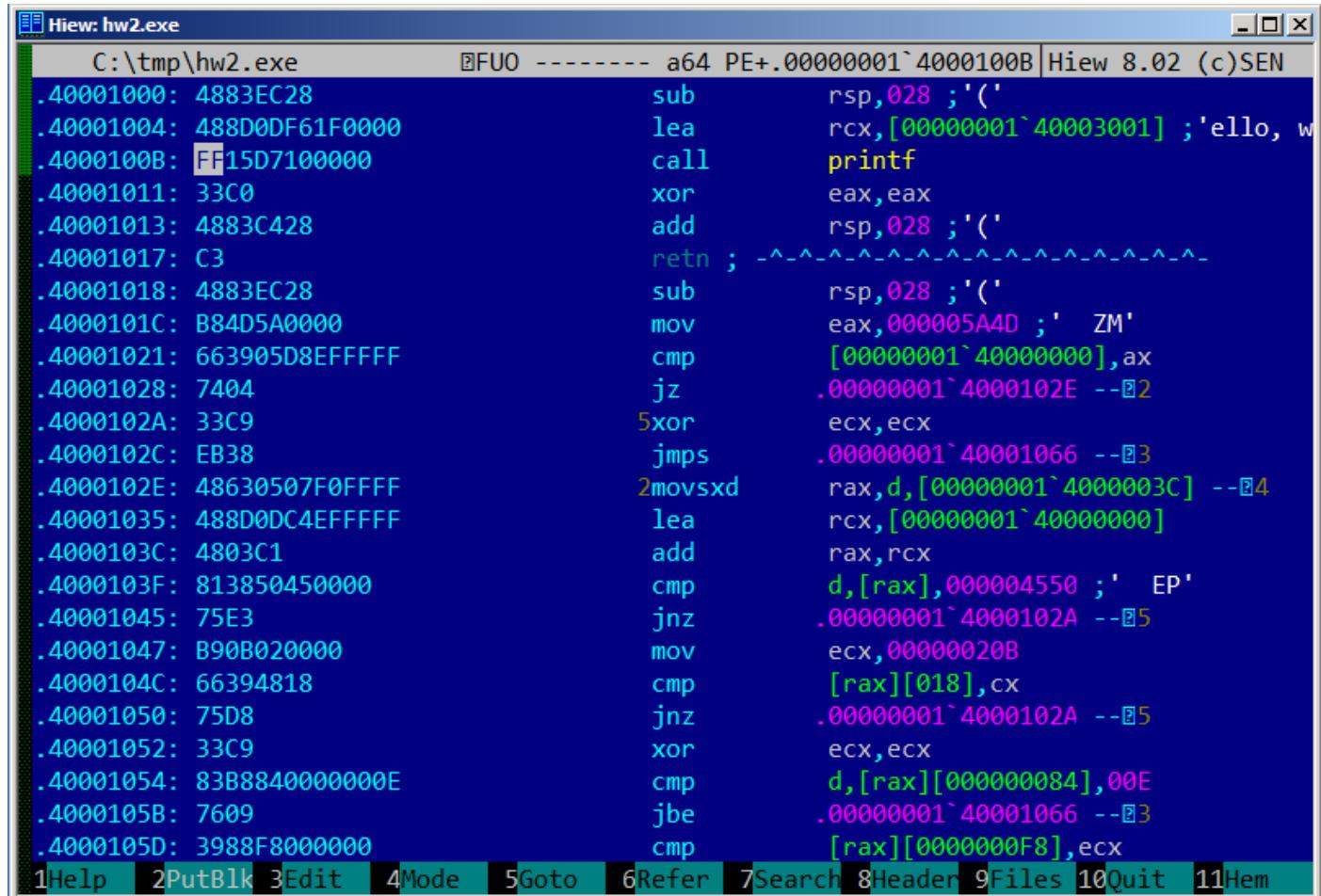


Figure 1.3: Hiew

As an experiment, we can [increment](#) address of the pointer by 1:

## 1.5. HELLO, WORLD!



The screenshot shows the Hiew debugger interface with the title bar "Hiew: hw2.exe". The main window displays assembly code for a 64-bit PE executable. The assembly listing includes several instructions, such as `sub`, `lea`, `call`, `xor`, `add`, and `ret`, along with memory addresses and register names. A portion of the string "Hello, world" is visible in the assembly output. The bottom of the window shows a menu bar with options 1 through 11.

Figure 1.4: Hiew

Hiew shows “ello, world” string. And when we run patched executable, this very string is printed.

### Pick another string from binary image (Linux x64)

The binary file I've got when I compile our example using GCC 5.4.0 on Linux x64 box has many other text strings: they are mostly imported function names and library names.

I run objdump to get contents of all sections of the compiled file:

```
$ objdump -s a.out

a.out:      file format elf64-x86-64

Contents of section .interp:
400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
400248 7838362d 36342e73 6f2e3200          x86-64.so.2.

Contents of section .note.ABI-tag:
400254 04000000 10000000 01000000 474e5500  ....GNU.
400264 00000000 02000000 06000000 20000000  .....

Contents of section .note.gnu.build-id:
400274 04000000 14000000 03000000 474e5500  ....GNU.
400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.....^...
400294 cf3f7ae4                      .?z.

...
```

It's not a problem to pass address of the text string “/lib64/ld-linux-x86-64.so.2” to `printf()` call:

```
#include <stdio.h>
```

## 1.5. HELLO, WORLD!

```
int main()
{
    printf(0x400238);
    return 0;
}
```

Hard to believe, this code prints mentioned string.

Change the address to `0x400260`, and the “GNU” string will be printed. The address is true for my specific GCC version, GNU toolset, etc. On your system, executable may be slightly different, and all addresses will also be different. Also, adding/removing code to/from this source code will probably shift all addresses back and forth.

### 1.5.3 GCC—one more thing

The fact that an *anonymous* C-string has *const* type ( [1.5.1 on page 9](#)), and that C-strings allocated in constants segment are guaranteed to be immutable, has an interesting consequence: the compiler may use a specific part of the string.

Let's try this example:

```
#include <stdio.h>

int f1()
{
    printf ("world\n");
}

int f2()
{
    printf ("hello world\n");
}

int main()
{
    f1();
    f2();
}
```

Common C/C++-compilers (including MSVC) allocate two strings, but let's see what GCC 4.8.1 does:

Listing 1.23: GCC 4.8.1 + IDA listing

```
f1          proc near
s           = dword ptr -1Ch
            sub    esp, 1Ch
            mov    [esp+1Ch+s], offset s ; "world\n"
            call   _puts
            add    esp, 1Ch
            retn
f1          endp

f2          proc near
s           = dword ptr -1Ch
            sub    esp, 1Ch
            mov    [esp+1Ch+s], offset aHello ; "hello "
            call   _puts
            add    esp, 1Ch
            retn
f2          endp

aHello      db 'hello '
s           db 'world',0xa,0
```

## 1.5. HELLO, WORLD!

Indeed: when we print the “hello world” string these two words are positioned in memory adjacently and `puts()` called from `f2()` function is not aware that this string is divided. In fact, it’s not divided; it’s divided only “virtually”, in this listing.

When `puts()` is called from `f1()`, it uses the “world” string plus a zero byte. `puts()` is not aware that there is something before this string!

This clever trick is often used by at least GCC and can save some memory. This is close to *string interning*.

Another related example is here: [3.2 on page 468](#).

### 1.5.4 ARM

For my experiments with ARM processors, several compilers were used:

- Popular in the embedded area: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE with the LLVM-GCC 4.2 compiler <sup>28</sup>.
- GCC 4.9 (Linaro) (for ARM64), available as win32-executables at <http://go.yurichev.com/17325>.

32-bit ARM code is used (including Thumb and Thumb-2 modes) in all cases in this book, if not mentioned otherwise. When we talk about 64-bit ARM here, we call it ARM64.

#### Non-optimizing Keil 6/2013 (ARM mode)

Let’s start by compiling our example in Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

The `armcc` compiler produces assembly listings in Intel-syntax, but it has high-level ARM-processor related macros <sup>29</sup>, but it is more important for us to see the instructions “as is” so let’s see the compiled result in [IDA](#).

Listing 1.24: Non-optimizing Keil 6/2013 (ARM mode) [IDA](#)

```
.text:00000000          main
.text:00000000 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2    ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB    BL      _2printf
.text:0000000C 00 00 A0 E3    MOV     R0, #0
.text:00000010 10 80 BD E8    LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0      ; DATA XREF: main+4
```

In the example, we can easily see each instruction has a size of 4 bytes. Indeed, we compiled our code for ARM mode, not for Thumb.

The very first instruction, `STMFD SP!, {R4,LR}` <sup>30</sup>, works as an x86 `PUSH` instruction, writing the values of two registers (`R4` and `LR`) into the stack.

Indeed, in the output listing from the `armcc` compiler, for the sake of simplification, actually shows the `PUSH {r4,lr}` instruction. But that is not quite precise. The `PUSH` instruction is only available in Thumb mode. So, to make things less confusing, we’re doing this in [IDA](#).

This instruction first [decrements](#) the `SP`<sup>32</sup> so it points to the place in the stack that is free for new entries, then it saves the values of the `R4` and `LR` registers at the address stored in the modified `SP`.

This instruction (like the `PUSH` instruction in Thumb mode) is able to save several register values at once which can be very useful. By the way, this has no equivalent in x86. It can also be noted that the `STMFD` instruction is a generalization of the `PUSH` instruction (extending its features), since it can work with any register, not just with `SP`. In other words, `STMFD` may be used for storing a set of registers at the specified memory address.

<sup>28</sup>It is indeed so: Apple Xcode 4.6.3 uses open-source GCC as front-end compiler and LLVM code generator

<sup>29</sup>e.g. ARM mode lacks `PUSH` / `POP` instructions

<sup>30</sup>`STMFD`<sup>31</sup>

<sup>32</sup>[stack pointer](#). `SP/ESP/RSP` in x86/x64. `SP` in ARM.

## 1.5. HELLO, WORLD!

The `ADR R0, aHelloWorld` instruction adds or subtracts the value in the `PC`<sup>33</sup> register to the offset where the `hello, world` string is located. How is the `PC` register used here, one might ask? This is called “position-independent code”<sup>34</sup>.

Such code can be executed at a non-fixed address in memory. In other words, this is `PC`-relative addressing. The `ADR` instruction takes into account the difference between the address of this instruction and the address where the string is located. This difference (offset) is always to be the same, no matter at what address our code is loaded by the `OS`. That’s why all we need is to add the address of the current instruction (from `PC`) in order to get the absolute memory address of our C-string.

`BL __2printf`<sup>35</sup> instruction calls the `printf()` function. Here’s how this instruction works:

- store the address following the `BL` instruction (`0xC`) into the `LR`;
- then pass the control to `printf()` by writing its address into the `PC` register.

When `printf()` finishes its execution it must have information about where it needs to return the control to. That’s why each function passes control to the address stored in the `LR` register.

That is a difference between “pure” `RISC`-processors like ARM and `CISC`<sup>36</sup>-processors like x86, where the return address is usually stored on the stack. Read more about this in next section ([1.7 on page 30](#)).

By the way, an absolute 32-bit address or offset cannot be encoded in the 32-bit `BL` instruction because it only has space for 24 bits. As we may recall, all ARM-mode instructions have a size of 4 bytes (32 bits). Hence, they can only be located on 4-byte boundary addresses. This implies that the last 2 bits of the instruction address (which are always zero bits) may be omitted. In summary, we have 26 bits for offset encoding. This is enough to encode  $current\_PC \pm \approx 32M$ .

Next, the `MOV R0, #0`<sup>37</sup> instruction just writes 0 into the `R0` register. That’s because our C-function returns 0 and the return value is to be placed in the `R0` register.

The last instruction `LDMFD SP!, R4,PC`<sup>38</sup>. It loads values from the stack (or any other memory place) in order to save them into `R4` and `PC`, and increments the `stack pointer SP`. It works like `POP` here.

N.B. The very first instruction `STMFD` saved the `R4` and `LR` registers pair on the stack, but `R4` and `PC` are restored during the `LDMFD` execution.

As we already know, the address of the place where each function must return control to is usually saved in the `LR` register. The very first instruction saves its value in the stack because the same register will be used by our `main()` function when calling `printf()`. In the function’s end, this value can be written directly to the `PC` register, thus passing control to where our function has been called.

Since `main()` is usually the primary function in C/C++, the control will be returned to the `OS` loader or to a point in a `CRT`, or something like that.

All that allows omitting the `BX LR` instruction at the end of the function.

`DCB` is an assembly language directive defining an array of bytes or ASCII strings, akin to the `DB` directive in the x86-assembly language.

### Non-optimizing Keil 6/2013 (Thumb mode)

Let’s compile the same example using Keil in Thumb mode:

```
armcc.exe --thumb --c90 -O0 1.c
```

We are getting (in `IDA`):

Listing 1.25: Non-optimizing Keil 6/2013 (Thumb mode) + `IDA`

```
.text:00000000          main
.text:00000000 10 B5      PUSH    {R4,LR}
```

<sup>33</sup>Program Counter. IP/EIP/RIP in x86/64. PC in ARM.

<sup>34</sup>Read more about it in relevant section ([6.4.1 on page 747](#))

<sup>35</sup>Branch with Link

<sup>36</sup>Complex instruction set computing

<sup>37</sup>Meaning MOVE

<sup>38</sup>`LDMFD`<sup>39</sup> is an inverse instruction of `STMFD`

## 1.5. HELLO, WORLD!

```
.text:00000002 C0 A0      ADR    R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9  BL     _2printf
.text:00000008 00 20      MOVS   R0, #0
.text:0000000A 10 BD      POP    {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0      ; DATA XREF: main+2
```

We can easily spot the 2-byte (16-bit) opcodes. This is, as was already noted, Thumb. The `BL` instruction, however, consists of two 16-bit instructions. This is because it is impossible to load an offset for the `printf()` function while using the small space in one 16-bit opcode. Therefore, the first 16-bit instruction loads the higher 10 bits of the offset and the second instruction loads the lower 11 bits of the offset.

As was noted, all instructions in Thumb mode have a size of 2 bytes (or 16 bits). This implies it is impossible for a Thumb-instruction to be at an odd address whatsoever. Given the above, the last address bit may be omitted while encoding instructions.

In summary, the `BL` Thumb-instruction can encode an address in  $current\_PC \pm \approx 2M$ .

As for the other instructions in the function: `PUSH` and `POP` work here just like the described `STMFD` / `LDMFD` only the `SP` register is not mentioned explicitly here. `ADR` works just like in the previous example. `MOVS` writes 0 into the `R0` register in order to return zero.

## Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Xcode 4.6.3 without optimization turned on produces a lot of redundant code so we'll study optimized output, where the instruction count is as small as possible, setting the compiler switch `-O3`.

Listing 1.26: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
_text:000028C4          _hello_world
_text:000028C4 80 40 2D E9  STMFD    SP!, {R7,LR}
_text:000028C8 86 06 01 E3  MOV       R0, #0x1686
_text:000028CC 0D 70 A0 E1  MOV       R7, SP
_text:000028D0 00 00 40 E3  MOVT      R0, #0
_text:000028D4 00 00 8F E0  ADD       R0, PC, R0
_text:000028D8 C3 05 00 EB  BL        _puts
_text:000028DC 00 00 A0 E3  MOV       R0, #0
_text:000028E0 80 80 BD E8  LDMFD    SP!, {R7,PC}

_cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world!",0
```

The instructions `STMFD` and `LDMFD` are already familiar to us.

The `MOV` instruction just writes the number `0x1686` into the `R0` register. This is the offset pointing to the "Hello world!" string.

The `R7` register (as it is standardized in [iOS ABI Function Call Guide, (2010)]<sup>40</sup>) is a frame pointer. More on that below.

The `MOVT R0, #0` (`MOVE Top`) instruction writes 0 into higher 16 bits of the register. The issue here is that the generic `MOV` instruction in ARM mode may write only the lower 16 bits of the register.

Keep in mind, all instruction opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving data between registers. That's why an additional instruction `MOVT` exists for writing into the higher bits (from 16 to 31 inclusive). Its usage here, however, is redundant because the `MOV R0, #0x1686` instruction above cleared the higher part of the register. This is supposedly a shortcoming of the compiler.

The `ADD R0, PC, R0` instruction adds the value in the `PC` to the value in the `R0`, to calculate the absolute address of the "Hello world!" string. As we already know, it is "position-independent code" so this correction is essential here.

The `BL` instruction calls the `puts()` function instead of `printf()`.

<sup>40</sup>Also available as <http://go.yurichev.com/17276>

## 1.5. HELLO, WORLD!

GCC replaced the first `printf()` call with `puts()`. Indeed: `printf()` with a sole argument is almost analogous to `puts()`.

Almost, because the two functions are producing the same result only in case the string does not contain printf format identifiers starting with %. In case it does, the effect of these two functions would be different <sup>41</sup>.

Why did the compiler replace the `printf()` with `puts()`? Presumably because `puts()` is faster <sup>42</sup>.

Because it just passes characters to `stdout` without comparing every one of them with the % symbol.

Next, we see the familiar `MOV R0, #0` instruction intended to set the `R0` register to 0.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

By default Xcode 4.6.3 generates code for Thumb-2 in this manner:

Listing 1.27: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
_text:00002B6C          _hello_world
__text:00002B6C 80 B5      PUSH      {R7,LR}
__text:00002B6E 41 F2 D8 30    MOVW     R0, #0x13D8
__text:00002B72 6F 46      MOV       R7, SP
__text:00002B74 C0 F2 00 00    MOVT.W   R0, #0
__text:00002B78 78 44      ADD       R0, PC
__text:00002B7A 01 F0 38 EA    BLX      _puts
__text:00002B7E 00 20      MOVS     R0, #0
__text:00002B80 80 BD      POP      {R7,PC}

...
__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

The `BL` and `BLX` instructions in Thumb mode, as we recall, are encoded as a pair of 16-bit instructions. In Thumb-2 these *surrogate* opcodes are extended in such a way so that new instructions may be encoded here as 32-bit instructions.

That is obvious considering that the opcodes of the Thumb-2 instructions always begin with `0xFx` or `0xEx`.

But in the IDA listing the opcode bytes are swapped because for ARM processor the instructions are encoded as follows: last byte comes first and after that comes the first one (for Thumb and Thumb-2 modes) or for instructions in ARM mode the fourth byte comes first, then the third, then the second and finally the first (due to different [endianness](#)).

So that is how bytes are located in IDA listings:

- for ARM and ARM64 modes: 4-3-2-1;
- for Thumb mode: 2-1;
- for 16-bit instructions pair in Thumb-2 mode: 2-1-4-3.

So as we can see, the `MOVW`, `MOVT.W` and `BLX` instructions begin with `0xFx`.

One of the Thumb-2 instructions is `MOVW R0, #0x13D8` —it stores a 16-bit value into the lower part of the `R0` register, clearing the higher bits.

Also, `MOVT.W R0, #0` works just like `MOVT` from the previous example only it works in Thumb-2.

Among the other differences, the `BLX` instruction is used in this case instead of the `BL`.

The difference is that, besides saving the `RA43` in the `LR` register and passing control to the `puts()` function, the processor is also switching from Thumb/Thumb-2 mode to ARM mode (or back).

This instruction is placed here since the instruction to which control is passed looks like (it is encoded in ARM mode):

<sup>41</sup>It has also to be noted the `puts()` does not require a '\n' new line symbol at the end of a string, so we do not see it here.

<sup>42</sup>[ciselant.de/projects/gcc\\_printf/gcc\\_printf.html](http://ciselant.de/projects/gcc_printf/gcc_printf.html)

<sup>43</sup>Return Address

## 1.5. HELLO, WORLD!

```
__symbolstub1:00003FEC _puts ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5 LDR PC, =__imp__puts
```

This is essentially a jump to the place where the address of `puts()` is written in the imports' section.

So, the observant reader may ask: why not call `puts()` right at the point in the code where it is needed?

Because it is not very space-efficient.

Almost any program uses external dynamic libraries (like DLL in Windows, .so in \*NIX or .dylib in Mac OS X). The dynamic libraries contain frequently used library functions, including the standard C-function `puts()`.

In an executable binary file (Windows PE .exe, ELF or Mach-O) an import section is present. This is a list of symbols (functions or global variables) imported from external modules along with the names of the modules themselves.

The [OS](#) loader loads all modules it needs and, while enumerating import symbols in the primary module, determines the correct addresses of each symbol.

In our case, `_imp_puts` is a 32-bit variable used by the [OS](#) loader to store the correct address of the function in an external library. Then the `LDR` instruction just reads the 32-bit value from this variable and writes it into the `PC` register, passing control to it.

So, in order to reduce the time the [OS](#) loader needs for completing this procedure, it is good idea to write the address of each symbol only once, to a dedicated place.

Besides, as we have already figured out, it is impossible to load a 32-bit value into a register while using only one instruction without a memory access.

Therefore, the optimal solution is to allocate a separate function working in ARM mode with the sole goal of passing control to the dynamic library and then to jump to this short one-instruction function (the so-called [thunk function](#)) from the Thumb-code.

By the way, in the previous example (compiled for ARM mode) the control is passed by the `BL` to the same [thunk function](#). The processor mode, however, is not being switched (hence the absence of an "X" in the instruction mnemonic).

### More about thunk-functions

Thunk-functions are hard to understand, apparently, because of a misnomer. The simplest way to understand it as adaptors or converters of one type of jack to another. For example, an adaptor allowing the insertion of a British power plug into an American wall socket, or vice-versa. Thunk functions are also sometimes called *wrappers*.

Here are a couple more descriptions of these functions:

"A piece of coding which provides an address:", according to P. Z. Ingberman, who invented thunks in 1961 as a way of binding actual parameters to their formal definitions in Algol-60 procedure calls. If a procedure is called with an expression in the place of a formal parameter, the compiler generates a thunk which computes the expression and leaves the address of the result in some standard location.

...  
Microsoft and IBM have both defined, in their Intel-based systems, a "16-bit environment" (with bletcherous segment registers and 64K address limits) and a "32-bit environment" (with flat addressing and semi-real memory management). The two environments can both be running on the same computer and OS (thanks to what is called, in the Microsoft world, WOW which stands for Windows On Windows). MS and IBM have both decided that the process of getting from 16- to 32-bit and vice versa is called a "thunk"; for Windows 95, there is even a tool, THUNK.EXE, called a "thunk compiler".

### ( The Jargon File )

Another example we can find in LAPACK library—a "Linear Algebra PACKage" written in FORTRAN. C/C++ developers also want to use LAPACK, but it's insane to rewrite it to C/C++ and then maintain several

## 1.5. HELLO, WORLD!

versions. So there are short C functions callable from C/C++ environment, which are, in turn, call FORTRAN functions, and do almost anything else:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Also, functions like that are called “wrappers”.

## ARM64

### GCC

Let's compile the example using GCC 4.8.1 in ARM64:

Listing 1.28: Non-optimizing GCC 4.8.1 + objdump

```
1 0000000000400590 <main>:
2  400590: a9bf7bfd      stp    x29, x30, [sp,#-16]!
3  400594: 910003fd      mov    x29, sp
4  400598: 90000000      adrp   x0, 4000000 <_init-0x3b8>
5  40059c: 91192000      add    x0, x0, #0x648
6  4005a0: 97ffffa0      bl     400420 <puts@plt>
7  4005a4: 52800000      mov    w0, #0x0           // #0
8  4005a8: a8c17bfd      ldp    x29, x30, [sp],#16
9  4005ac: d65f03c0      ret
10 ...
11 ...
12 ...
13 Contents of section .rodata:
14 400640 01000200 00000000 48656c6c 6f210a00 .....Hello!..
```

There are no Thumb and Thumb-2 modes in ARM64, only ARM, so there are 32-bit instructions only. The Register count is doubled: [2.4 on page 1024](#). 64-bit registers have **X-** prefixes, while its 32-bit parts—**W-**.

The **STP** instruction (*Store Pair*) saves two registers in the stack simultaneously: **X29** and **X30**.

Of course, this instruction is able to save this pair at an arbitrary place in memory, but the **SP** register is specified here, so the pair is saved in the stack.

ARM64 registers are 64-bit ones, each has a size of 8 bytes, so one needs 16 bytes for saving two registers.

The exclamation mark (“!”) after the operand means that 16 is to be subtracted from **SP** first, and only then are values from register pair to be written into the stack. This is also called *pre-index*. About the difference between *post-index* and *pre-index* read here: [1.32.2 on page 441](#).

Hence, in terms of the more familiar x86, the first instruction is just an analogue to a pair of **PUSH X29** and **PUSH X30**. **X29** is used as **FP<sup>44</sup>** in ARM64, and **X30** as **LR**, so that's why they are saved in the function prologue and restored in the function epilogue.

The second instruction copies **SP** in **X29** (or **FP**). This is made so to set up the function stack frame.

**ADRP** and **ADD** instructions are used to fill the address of the string “Hello!” into the **X0** register, because the first function argument is passed in this register. There are no instructions, whatsoever, in ARM that can store a large number into a register (because the instruction length is limited to 4 bytes, read more about it here: [1.32.3 on page 442](#)). So several instructions must be utilized. The first instruction (**ADRP**) writes the address of the 4KiB page, where the string is located, into **X0**, and the second one (**ADD**) just adds the remainder to the address. More about that in: [1.32.4 on page 444](#).

<sup>44</sup>Frame Pointer

## 1.5. HELLO, WORLD!

`0x400000 + 0x648 = 0x400648`, and we see our “Hello!” C-string in the `.rodata` data segment at this address.

`puts()` is called afterwards using the `BL` instruction. This was already discussed: [1.5.4 on page 21](#).

`MOV` writes 0 into `W0`. `W0` is the lower 32 bits of the 64-bit `X0` register:

High 32-bit part	low 32-bit part
	<code>X0</code>
	<code>W0</code>

The function result is returned via `X0` and `main()` returns 0, so that's how the return result is prepared. But why use the 32-bit part?

Because the `int` data type in ARM64, just like in x86-64, is still 32-bit, for better compatibility.

So if a function returns a 32-bit `int`, only the lower 32 bits of `X0` register have to be filled.

In order to verify this, let's change this example slightly and recompile it. Now `main()` returns a 64-bit value:

Listing 1.29: `main()` returning a value of `uint64_t` type

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
    printf ("Hello!\n");
    return 0;
}
```

The result is the same, but that's how `MOV` at that line looks like now:

Listing 1.30: Non-optimizing GCC 4.8.1 + objdump

```
4005a4: d2800000      mov      x0, #0x0      // #0
```

`LDP` (*Load Pair*) then restores the `X29` and `X30` registers.

There is no exclamation mark after the instruction: this implies that the values are first loaded from the stack, and only then is `SP` increased by 16. This is called *post-index*.

A new instruction appeared in ARM64: `RET`. It works just as `BX LR`, only a special *hint* bit is added, informing the `CPU` that this is a return from a function, not just another jump instruction, so it can execute it more optimally.

Due to the simplicity of the function, optimizing GCC generates the very same code.

## 1.5.5 MIPS

### A word about the “global pointer”

One important MIPS concept is the “global pointer”. As we may already know, each MIPS instruction has a size of 32 bits, so it's impossible to embed a 32-bit address into one instruction: a pair has to be used for this (like GCC did in our example for the text string address loading). It's possible, however, to load data from the address in the range of `register - 32768...register + 32767` using one single instruction (because 16 bits of signed offset could be encoded in a single instruction). So we can allocate some register for this purpose and also allocate a 64KiB area of most used data. This allocated register is called a “global pointer” and it points to the middle of the 64KiB area. This area usually contains global variables and addresses of imported functions like `printf()`, because the GCC developers decided that getting the address of some function must be as fast as a single instruction execution instead of two. In an ELF file this 64KiB area is located partly in sections `.sbss` (“small `BSS`<sup>45</sup>”) for uninitialized data and `.sdata` (“small `data`”) for initialized data. This implies that the programmer may choose what data he/she wants to be accessed fast and place it into `.sdata/.sbss`. Some old-school programmers may recall the MS-DOS

<sup>45</sup>Block Started by Symbol

## 1.5. HELLO, WORLD!

memory model [10.6 on page 992](#) or the MS-DOS memory managers like XMS/EMS where all memory was divided in 64KiB blocks.

This concept is not unique to MIPS. At least PowerPC uses this technique as well.

### Optimizing GCC

Lets consider the following example, which illustrates the “global pointer” concept.

Listing 1.31: Optimizing GCC 4.4.5 (assembly output)

```
1 $LC0:  
2 ; \000 is zero byte in octal base:  
3     .ascii  "Hello, world!\012\000"  
4 main:  
5 ; function prologue.  
6 ; set the GP:  
7     lui      $28,%hi(__gnu_local_gp)  
8     addiu   $sp,$sp,-32  
9     addiu   $28,$28,%lo(__gnu_local_gp)  
10 ; save the RA to the local stack:  
11    sw      $31,28($sp)  
12 ; load the address of the puts() function from the GP to $25:  
13    lw      $25,%call16(puts)($28)  
14 ; load the address of the text string to $4 ($a0):  
15    lui      $4,%hi($LC0)  
16 ; jump to puts(), saving the return address in the link register:  
17    jalr    $25  
18    addiu   $4,$4,%lo($LC0) ; branch delay slot  
19 ; restore the RA:  
20    lw      $31,28($sp)  
21 ; copy 0 from $zero to $v0:  
22    move    $2,$0  
23 ; return by jumping to the RA:  
24    j      $31  
25 ; function epilogue:  
26    addiu   $sp,$sp,32 ; branch delay slot + free local stack
```

As we see, the \$GP register is set in the function prologue to point to the middle of this area. The [RA](#) register is also saved in the local stack. [puts\(\)](#) is also used here instead of [printf\(\)](#). The address of the [puts\(\)](#) function is loaded into \$25 using [LW](#) the instruction (“Load Word”). Then the address of the text string is loaded to \$4 using [LUI](#) (“Load Upper Immediate”) and [ADDIU](#) (“Add Immediate Unsigned Word”) instruction pair. [LUI](#) sets the high 16 bits of the register (hence “upper” word in instruction name) and [ADDIU](#) adds the lower 16 bits of the address.

[ADDIU](#) follows [JALR](#) (haven’t you forgot *branch delay slots* yet?). The register \$4 is also called \$A0, which is used for passing the first function argument <sup>46</sup>.

[JALR](#) (“Jump and Link Register”) jumps to the address stored in the \$25 register (address of [puts\(\)](#)) while saving the address of the next instruction (LW) in [RA](#). This is very similar to ARM. Oh, and one important thing is that the address saved in [RA](#) is not the address of the next instruction (because it’s in a *delay slot* and is executed before the jump instruction), but the address of the instruction after the next one (after the *delay slot*). Hence,  $PC + 8$  is written to [RA](#) during the execution of [JALR](#), in our case, this is the address of the [LW](#) instruction next to [ADDIU](#).

[LW](#) (“Load Word”) at line 20 restores [RA](#) from the local stack (this instruction is actually part of the function epilogue).

[MOVE](#) at line 22 copies the value from the \$0 (\$ZERO) register to \$2 (\$V0).

MIPS has a *constant* register, which always holds zero. Apparently, the MIPS developers came up with the idea that zero is in fact the busiest constant in the computer programming, so let’s just use the \$0 register every time zero is needed.

<sup>46</sup>The MIPS registers table is available in appendix [3.1 on page 1025](#)

## 1.5. HELLO, WORLD!

Another interesting fact is that MIPS lacks an instruction that transfers data between registers. In fact, `MOVE DST, SRC` is `ADD DST, SRC, $ZERO` ( $DST = SRC + 0$ ), which does the same. Apparently, the MIPS developers wanted to have a compact opcode table. This does not mean an actual addition happens at each `MOVE` instruction. Most likely, the CPU optimizes these pseudo instructions and the ALU<sup>47</sup> is never used.

`J` at line 24 jumps to the address in `RA`, which is effectively performing a return from the function. `ADDIU` after `J` is in fact executed before `J` (remember *branch delay slots?*) and is part of the function epilogue. Here is also a listing generated by [IDA](#). Each register here has its own pseudo name:

Listing 1.32: Optimizing GCC 4.4.5 (IDA)

```
1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10          = -0x10
4 .text:00000000 var_4           = -4
5 .text:00000000
6 ; function prologue.
7 ; set the GP:
8 .text:00000000        lui      $gp, (__gnu_local_gp >> 16)
9 .text:00000004        addiu   $sp, -0x20
10 .text:00000008       la      $gp, (__gnu_local_gp & 0xFFFF)
11 ; save the RA to the local stack:
12 .text:0000000C       sw      $ra, 0x20+var_4($sp)
13 ; save the GP to the local stack:
14 ; for some reason, this instruction is missing in the GCC assembly output:
15 .text:00000010       sw      $gp, 0x20+var_10($sp)
16 ; load the address of the puts() function from the GP to $t9:
17 .text:00000014       lw      $t9, (puts & 0xFFFF)($gp)
18 ; form the address of the text string in $a0:
19 .text:00000018       lui      $a0, ($LC0 >> 16) # "Hello, world!"
20 ; jump to puts(), saving the return address in the link register:
21 .text:0000001C       jalr    $t9
22 .text:00000020       la      $a0, ($LC0 & 0xFFFF) # "Hello, world!"
23 ; restore the RA:
24 .text:00000024       lw      $ra, 0x20+var_4($sp)
25 ; copy 0 from $zero to $v0:
26 .text:00000028       move    $v0, $zero
27 ; return by jumping to the RA:
28 .text:0000002C       jr      $ra
29 ; function epilogue:
30 .text:00000030       addiu  $sp, 0x20
```

The instruction at line 15 saves the GP value into the local stack, and this instruction is missing mysteriously from the GCC output listing, maybe by a GCC error <sup>48</sup>. The GP value has to be saved indeed, because each function can use its own 64KiB data window. The register containing the `puts()` address is called `$T9`, because registers prefixed with T are called “temporaries” and their contents may not be preserved.

## Non-optimizing GCC

Non-optimizing GCC is more verbose.

Listing 1.33: Non-optimizing GCC 4.4.5 (assembly output)

```
1 $LC0:
2     .ascii  "Hello, world!\012\000"
3 main:
4 ; function prologue.
5 ; save the RA ($31) and FP in the stack:
6     addiu  $sp,$sp,-32
7     sw     $31,28($sp)
8     sw     $fp,24($sp)
9 ; set the FP (stack frame pointer):
10    move   $fp,$sp
11 ; set the GP:
```

<sup>47</sup>Arithmetic logic unit

<sup>48</sup>Apparently, functions generating listings are not so critical to GCC users, so some unfixed errors may still exist.

## 1.5. HELLO, WORLD!

```
12      lui      $28,%hi(__gnu_local_gp)
13      addiu   $28,$28,%lo(__gnu_local_gp)
14 ; load the address of the text string:
15      lui      $2,%hi($LC0)
16      addiu   $4,$2,%lo($LC0)
17 ; load the address of puts() using the GP:
18      lw       $2,%call16(puts)($28)
19      nop
20 ; call puts():
21      move    $25,$2
22      jalr    $25
23      nop ; branch delay slot
24
25 ; restore the GP from the local stack:
26      lw       $28,16($fp)
27 ; set register $2 ($V0) to zero:
28      move    $2,$0
29 ; function epilogue.
30 ; restore the SP:
31      move    $sp,$fp
32 ; restore the RA:
33      lw       $31,28($sp)
34 ; restore the FP:
35      lw       $fp,24($sp)
36      addiu   $sp,$sp,32
37 ; jump to the RA:
38      j       $31
39      nop ; branch delay slot
```

We see here that register FP is used as a pointer to the stack frame. We also see 3 NOPs. The second and third of which follow the branch instructions. Perhaps the GCC compiler always adds NOPs (because of *branch delay slots*) after branch instructions and then, if optimization is turned on, maybe eliminates them. So in this case they are left here.

Here is also [IDA](#) listing:

Listing 1.34: Non-optimizing GCC 4.4.5 ([IDA](#))

```
1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10          = -0x10
4 .text:00000000 var_8           = -8
5 .text:00000000 var_4           = -4
6 .text:00000000
7 ; function prologue.
8 ; save the RA and FP in the stack:
9 .text:00000000      addiu  $sp, -0x20
10 .text:00000004      sw     $ra, 0x20+var_4($sp)
11 .text:00000008      sw     $fp, 0x20+var_8($sp)
12 ; set the FP (stack frame pointer):
13 .text:0000000C      move   $fp, $sp
14 ; set the GP:
15 .text:00000010      la     $gp, __gnu_local_gp
16 .text:00000018      sw     $gp, 0x20+var_10($sp)
17 ; load the address of the text string:
18 .text:0000001C      lui    $v0, (aHelloWorld >> 16) # "Hello, world!"
19 .text:00000020      addiu $a0, $v0, (aHelloWorld & 0xFFFF) # "Hello, world!"
20 ; load the address of puts() using the GP:
21 .text:00000024      lw    $v0, (puts & 0xFFFF)($gp)
22 .text:00000028      or    $at, $zero ; NOP
23 ; call puts():
24 .text:0000002C      move  $t9, $v0
25 .text:00000030      jalr $t9
26 .text:00000034      or    $at, $zero ; NOP
27 ; restore the GP from local stack:
28 .text:00000038      lw    $gp, 0x20+var_10($fp)
29 ; set register $2 ($V0) to zero:
30 .text:0000003C      move  $v0, $zero
31 ; function epilogue.
32 ; restore the SP:
33 .text:00000040      move  $sp, $fp
```

## 1.5. HELLO, WORLD!

```
34 ; restore the RA:  
35 .text:00000044          lw      $ra, 0x20+var_4($sp)  
36 ; restore the FP:  
37 .text:00000048          lw      $fp, 0x20+var_8($sp)  
38 .text:0000004C          addiu $sp, 0x20  
39 ; jump to the RA:  
40 .text:00000050          jr      $ra  
41 .text:00000054          or      $at, $zero ; NOP
```

Interestingly, IDA recognized the LUI / ADDIU instructions pair and coalesced them into one LA ("Load Address") pseudo instruction at line 15. We may also see that this pseudo instruction has a size of 8 bytes! This is a pseudo instruction (or *macro*) because it's not a real MIPS instruction, but rather a handy name for an instruction pair.

Another thing is that IDA doesn't recognize NOP instructions, so here they are at lines 22, 26 and 41. It is OR \$AT, \$ZERO. Essentially, this instruction applies the OR operation to the contents of the \$AT register with zero, which is, of course, an idle instruction. MIPS, like many other ISAs, doesn't have a separate NOP instruction.

### Role of the stack frame in this example

The address of the text string is passed in the register. Why setup a local stack anyway? The reason for this lies in the fact that the values of registers RA and GP have to be saved somewhere (because printf() is called), and the local stack is used for this purpose. If this was a leaf function, it would have been possible to get rid of the function prologue and epilogue, for example: [1.4.3 on page 8](#).

### Optimizing GCC: load it into GDB

Listing 1.35: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw  
root@debian-mips:~# gdb hw  
GNU gdb (GDB) 7.0.1-debian  
...  
Reading symbols from /root/hw...(no debugging symbols found)...done.  
(gdb) b main  
Breakpoint 1 at 0x400654  
(gdb) run  
Starting program: /root/hw  
  
Breakpoint 1, 0x00400654 in main ()  
(gdb) set step-mode on  
(gdb) disas  
Dump of assembler code for function main:  
0x00400640 <main+0>: lui      gp,0x42  
0x00400644 <main+4>: addiu   sp,sp,-32  
0x00400648 <main+8>: addiu   gp,sp,-30624  
0x0040064c <main+12>: sw      ra,28(sp)  
0x00400650 <main+16>: sw      gp,16(sp)  
0x00400654 <main+20>: lw      t9,-32716(gp)  
0x00400658 <main+24>: lui      a0,0x40  
0x0040065c <main+28>: jalr    t9  
0x00400660 <main+32>: addiu   a0,a0,2080  
0x00400664 <main+36>: lw      ra,28(sp)  
0x00400668 <main+40>: move    v0,zero  
0x0040066c <main+44>: jr      ra  
0x00400670 <main+48>: addiu   sp,sp,32  
End of assembler dump.  
(gdb) s  
0x00400658 in main ()  
(gdb) s  
0x0040065c in main ()  
(gdb) s  
0x2ab2de60 in printf () from /lib/libc.so.6  
(gdb) x/s $a0
```

## 1.6. FUNCTION PROLOGUE AND EPILOGUE

```
0x400820:      "hello, world"
(gdb)
```

### 1.5.6 Conclusion

The main difference between x86/ARM and x64/ARM64 code is that the pointer to the string is now 64-bits in length. Indeed, modern CPUs are now 64-bit due to both the reduced cost of memory and the greater demand for it by modern applications. We can add much more memory to our computers than 32-bit pointers are able to address. As such, all pointers are now 64-bit.

### 1.5.7 Exercises

- <http://challenges.re/48>
- <http://challenges.re/49>

## 1.6 Function prologue and epilogue

A function prologue is a sequence of instructions at the start of a function. It often looks something like the following code fragment:

```
push    ebp
mov     ebp, esp
sub    esp, X
```

What these instruction do: save the value in the `EBP` register, set the value of the `EBP` register to the value of the `ESP` and then allocate space on the stack for local variables.

The value in the `EBP` stays the same over the period of the function execution and is to be used for local variables and arguments access. For the same purpose one can use `ESP`, but since it changes over time this approach is not too convenient.

The function epilogue frees the allocated space in the stack, returns the value in the `EBP` register back to its initial state and returns the control flow to the [caller](#):

```
mov    esp, ebp
pop    ebp
ret    0
```

Function prologues and epilogues are usually detected in disassemblers for function delimitation.

### 1.6.1 Recursion

Epilogues and prologues can negatively affect the recursion performance.

More about recursion in this book: [3.4.3 on page 480](#).

## 1.7 Stack

The stack is one of the most fundamental data structures in computer science <sup>49</sup>. AKA<sup>50</sup> LIFO<sup>51</sup>.

Technically, it is just a block of memory in process memory along with the `ESP` or `RSP` register in x86 or x64, or the `SP` register in ARM, as a pointer within that block.

<sup>49</sup>[wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

<sup>50</sup> Also Known As

<sup>51</sup>Last In First Out

## 1.7. STACK

The most frequently used stack access instructions are `PUSH` and `POP` (in both x86 and ARM Thumb-mode). `PUSH` subtracts from `ESP / RSP /SP` 4 in 32-bit mode (or 8 in 64-bit mode) and then writes the contents of its sole operand to the memory address pointed by `ESP / RSP /SP`.

`POP` is the reverse operation: retrieve the data from the memory location that `SP` points to, load it into the instruction operand (often a register) and then add 4 (or 8) to the `stack pointer`.

After stack allocation, the `stack pointer` points at the bottom of the stack. `PUSH` decreases the `stack pointer` and `POP` increases it. The bottom of the stack is actually at the beginning of the memory allocated for the stack block. It seems strange, but that's the way it is.

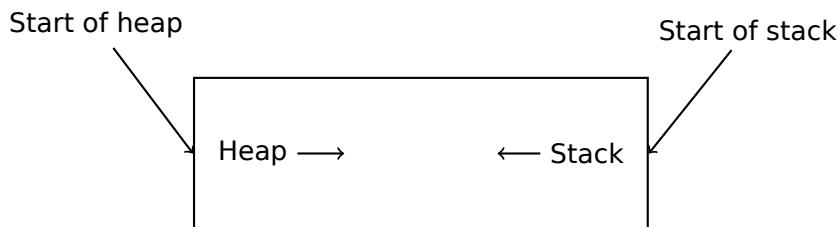
ARM supports both descending and ascending stacks.

For example the `STMFD/LDMFD`, `STMED52/LDMED53` instructions are intended to deal with a descending stack (grows downwards, starting with a high address and progressing to a lower one). The `STMFA54/LDMFA55`, `STMEA56/LDMEA57` instructions are intended to deal with an ascending stack (grows upwards, starting from a low address and progressing to a higher one).

### 1.7.1 Why does the stack grow backwards?

Intuitively, we might think that the stack grows upwards, i.e. towards higher addresses, like any other data structure.

The reason that the stack grows backward is probably historical. When the computers were big and occupied a whole room, it was easy to divide memory into two parts, one for the `heap` and one for the stack. Of course, it was unknown how big the `heap` and the stack would be during program execution, so this solution was the simplest possible.



In [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)]<sup>58</sup> we can read:

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

This reminds us how some students write two lecture notes using only one notebook: notes for the first lecture are written as usual, and notes for the second one are written from the end of notebook, by flipping it. Notes may meet each other somewhere in between, in case of lack of free space.

### 1.7.2 What is the stack used for?

#### Save the function's return address

x86

<sup>52</sup>Store Multiple Empty Descending (ARM instruction)

<sup>53</sup>Load Multiple Empty Descending (ARM instruction)

<sup>54</sup>Store Multiple Full Ascending (ARM instruction)

<sup>55</sup>Load Multiple Full Ascending (ARM instruction)

<sup>56</sup>Store Multiple Empty Ascending (ARM instruction)

<sup>57</sup>Load Multiple Empty Ascending (ARM instruction)

<sup>58</sup>Also available as <http://go.yurichev.com/17270>

## 1.7. STACK

When calling another function with a `CALL` instruction, the address of the point exactly after the `CALL` instruction is saved to the stack and then an unconditional jump to the address in the `CALL` operand is executed.

The `CALL` instruction is equivalent to a `PUSH address_after_call / JMP operand` instruction pair.

`RET` fetches a value from the stack and jumps to it —that is equivalent to a `POP tmp / JMP tmp` instruction pair.

Overflowing the stack is straightforward. Just run eternal recursion:

```
void f()
{
    f();
};
```

MSVC 2008 reports the problem:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause ↴
    runtime stack overflow
```

...but generates the right code anyway:

```
?f@YAXXZ PROC          ; f
; File c:\tmp6\ss.cpp
; Line 2
    push    ebp
    mov     ebp, esp
; Line 3
    call    ?f@YAXXZ      ; f
; Line 4
    pop    ebp
    ret    0
?f@YAXXZ ENDP          ; f
```

...Also if we turn on the compiler optimization (`/Ox` option) the optimized code will not overflow the stack and will work correctly<sup>59</sup> instead:

```
?f@YAXXZ PROC          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
    jmp    SHORT $LL3@f
?f@YAXXZ ENDP          ; f
```

GCC 4.4.1 generates similar code in both cases without, however, issuing any warning about the problem.

## ARM

ARM programs also use the stack for saving return addresses, but differently. As mentioned in “Hello, world!” (1.5.4 on page 19), the `RA` is saved to the `LR` (link register). If one needs, however, to call another function and use the `LR` register one more time, its value has to be saved. Usually it is saved in the function prologue.

Often, we see instructions like `PUSH R4-R7,LR` along with this instruction in epilogue `POP R4-R7,PC` —thus register values to be used in the function are saved in the stack, including `LR`.

<sup>59</sup>irony here

## 1.7. STACK

Nevertheless, if a function never calls any other function, in RISC terminology it is called a *leaf function*<sup>60</sup>. As a consequence, leaf functions do not save the `LR` register (because they don't modify it). If such function is small and uses a small number of registers, it may not use the stack at all. Thus, it is possible to call leaf functions without using the stack, which can be faster than on older x86 machines because external RAM is not used for the stack<sup>61</sup>. This can be also useful for situations when memory for the stack is not yet allocated or not available.

Some examples of leaf functions: [1.10.3 on page 104](#), [1.10.3 on page 104](#), [1.277 on page 315](#), [1.293 on page 334](#), [1.22.5 on page 334](#), [1.185 on page 210](#), [1.183 on page 208](#), [1.202 on page 226](#).

## Passing function arguments

The most popular way to pass parameters in x86 is called “cdecl”:

```
push arg3  
push arg2  
push arg1  
call f  
add esp, 12 ; 4*3=12
```

Callee functions get their arguments via the stack pointer.

Therefore, this is how the argument values are located in the stack before the execution of the `f()` function's very first instruction:

ESP	return address
ESP+4	argument#1, marked in IDA as <code>arg_0</code>
ESP+8	argument#2, marked in IDA as <code>arg_4</code>
ESP+0xC	argument#3, marked in IDA as <code>arg_8</code>
...	...

For more information on other calling conventions see also section ([6.1 on page 733](#)).

By the way, the `callee` function does not have any information about how many arguments were passed. C functions with a variable number of arguments (like `printf()`) determine their number using format string specifiers (which begin with the % symbol).

If we write something like:

```
printf("%d %d %d", 1234);
```

`printf()` will print 1234, and then two random numbers<sup>62</sup>, which were lying next to it in the stack.

That's why it is not very important how we declare the `main()` function: as `main()`, `main(int argc, char *argv[])` or `main(int argc, char *argv[], char *envp[])`.

In fact, the `CRT`-code is calling `main()` roughly as:

```
push envp  
push argv  
push argc  
call main  
...
```

If you declare `main()` as `main()` without arguments, they are, nevertheless, still present in the stack, but are not used. If you declare `main()` as `main(int argc, char *argv[])`, you will be able to use first two arguments, and the third will remain “invisible” for your function. Even more, it is possible to declare `main(int argc)`, and it will work.

<sup>60</sup>[infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html)

<sup>61</sup>Some time ago, on PDP-11 and VAX, the CALL instruction (calling other functions) was expensive; up to 50% of execution time might be spent on it, so it was considered that having a big number of small functions is an *anti-pattern* [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

<sup>62</sup>Not random in strict sense, but rather unpredictable: [1.7.4 on page 38](#)

## Alternative ways of passing arguments

It is worth noting that nothing obliges programmers to pass arguments through the stack. It is not a requirement. One could implement any other method without using the stack at all.

A somewhat popular way among assembly language newbies is to pass arguments via global variables, like:

Listing 1.36: Assembly code

```
...
    mov     X, 123
    mov     Y, 456
    call    do_something

...
X      dd      ?
Y      dd      ?

do_something proc near
    ; take X
    ; take Y
    ; do something
    retn
do_something endp
```

But this method has obvious drawback: `do_something()` function cannot call itself recursively (or via another function), because it has to zap its own arguments. The same story with local variables: if you hold them in global variables, the function couldn't call itself. And this is also not thread-safe <sup>63</sup>. A method to store such information in stack makes this easier—it can hold as many function arguments and/or values, as much space it has.

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] mentions even weirder schemes particularly convenient on IBM System/360.

MS-DOS had a way of passing all function arguments via registers, for example, this is piece of code for ancient 16-bit MS-DOS prints "Hello, world!":

```
mov dx, msg        ; address of message
mov ah, 9          ; 9 means "print string" function
int 21h           ; DOS "syscall"

mov ah, 4ch         ; "terminate program" function
int 21h           ; DOS "syscall"

msg db 'Hello, World!\$'
```

This is quite similar to [6.1.3 on page 734](#) method. And also it's very similar to calling syscalls in Linux ([6.3.1 on page 747](#)) and Windows.

If a MS-DOS function is going to return a boolean value (i.e., single bit, usually indicating error state), **CF** flag was often used.

For example:

```
mov ah, 3ch        ; create file
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error:
```

In case of error, **CF** flag is raised. Otherwise, handle of newly created file is returned via **AX**.

<sup>63</sup>Correctly implemented, each thread would have its own stack with its own arguments/variables.

## 1.7. STACK

This method is still used by assembly language programmers. In Windows Research Kernel source code (which is quite similar to Windows 2003) we can find something like this (file `base/ntos/ke/i386/cpu.asm`):

```
public Get386Stepping
Get386Stepping proc

    call MultiplyTest          ; Perform multiplication test
    jnc short G3s00            ; if nc, muttest is ok
    mov ax, 0
    ret

G3s00:
    call Check386B0           ; Check for B0 stepping
    jnc short G3s05            ; if nc, it's B1/later
    mov ax, 100h                ; It is B0/earlier stepping
    ret

G3s05:
    call Check386D1           ; Check for D1 stepping
    jc short G3s10              ; if c, it is NOT D1
    mov ax, 301h                ; It is D1/later stepping
    ret

G3s10:
    mov ax, 101h                ; assume it is B1 stepping
    ret

    ...

MultiplyTest proc

    xor cx,cx                  ; 64K times is a nice round number
mlt00: push cx
    call Multiply               ; does this chip's multiply work?
    pop cx
    jc short mltx              ; if c, No, exit
    loop mlt00                  ; if nc, YES, loop to try again
    clc
mltx:
    ret

MultiplyTest endp
```

## Local variable storage

A function could allocate space in the stack for its local variables just by decreasing the [stack pointer](#) towards the stack bottom.

Hence, it's very fast, no matter how many local variables are defined. It is also not a requirement to store local variables in the stack. You could store local variables wherever you like, but traditionally this is how it's done.

## x86: `alloca()` function

It is worth noting the `alloca()` function <sup>64</sup>. This function works like `malloc()`, but allocates memory directly on the stack. The allocated memory chunk does not have to be freed via a `free()` function call, since the function epilogue ([1.6 on page 30](#)) returns `ESP` back to its initial state and the allocated memory is just *dropped*. It is worth noting how `alloca()` is implemented. In simple terms, this function just shifts `ESP` downwards toward the stack bottom by the number of bytes you need and sets `ESP` as a pointer to the *allocated* block.

Let's try:

<sup>64</sup>In MSVC, the function implementation can be found in `alloca16.asm` and `chkstk.asm` in  
`C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

## 1.7. STACK

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    sprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

`_snprintf()` function works just like `printf()`, but instead of dumping the result into `stdout` (e.g., to terminal or console), it writes it to the `buf` buffer. Function `puts()` copies the contents of `buf` to `stdout`. Of course, these two function calls might be replaced by one `printf()` call, but we have to illustrate small buffer usage.

### MSVC

Let's compile (MSVC 2010):

Listing 1.37: MSVC 2010

```
...
mov    eax, 600 ; 00000258H
call   __alloca_probe_16
mov    esi, esp

push   3
push   2
push   1
push   OFFSET $SG2672
push   600      ; 00000258H
push   esi
call   __snprintf

push   esi
call   _puts
add    esp, 28

...
```

The sole `alloca()` argument is passed via `EAX` (instead of pushing it into the stack)<sup>65</sup>.

### GCC + Intel syntax

GCC 4.4.1 does the same without calling external functions:

Listing 1.38: GCC 4.7.3

```
.LC0:
```

<sup>65</sup>It is because `alloca()` is rather a compiler intrinsic ([10.3 on page 988](#)) than a normal function. One of the reasons we need a separate function instead of just a couple of instructions in the code, is because the [MSVC<sup>66</sup>](#) `alloca()` implementation also has code which reads from the memory just allocated, in order to let the [OS](#) map physical memory to this [VM<sup>67</sup>](#) region. After the `alloca()` call, `ESP` points to the block of 600 bytes and we can use it as memory for the `buf` array.

## 1.7. STACK

```
.string "hi! %d, %d, %d\n"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 660
    lea     ebx, [esp+39]
    and    ebx, -16           ; align pointer by 16-bit border
    mov     DWORD PTR [esp], ebx ; s
    mov     DWORD PTR [esp+20], 3
    mov     DWORD PTR [esp+16], 2
    mov     DWORD PTR [esp+12], 1
    mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov     DWORD PTR [esp+4], 600      ; maxlen
    call    _snprintf
    mov     DWORD PTR [esp], ebx      ; s
    call    puts
    mov     ebx, DWORD PTR [ebp-4]
    leave
    ret
```

## GCC + AT&T syntax

Let's see the same code, but in AT&T syntax:

Listing 1.39: GCC 4.7.3

```
.LC0:
.string "hi! %d, %d, %d\n"
f:
    pushl  %ebp
    movl   %esp, %ebp
    pushl  %ebx
    subl   $660, %esp
    leal   39(%esp), %ebx
    andl   $-16, %ebx
    movl   %ebx, (%esp)
    movl   $3, 20(%esp)
    movl   $2, 16(%esp)
    movl   $1, 12(%esp)
    movl   $.LC0, 8(%esp)
    movl   $600, 4(%esp)
    call   _snprintf
    movl   %ebx, (%esp)
    call   puts
    movl   -4(%ebp), %ebx
    leave
    ret
```

The code is the same as in the previous listing.

By the way, `movl $3, 20(%esp)` corresponds to `mov DWORD PTR [esp+20], 3` in Intel-syntax. In the AT&T syntax, the register+offset format of addressing memory looks like `offset(%register)`.

## (Windows) SEH

SEH<sup>68</sup> records are also stored on the stack (if they are present). Read more about it: ([6.5.3 on page 763](#)).

## Buffer overflow protection

More about it here ([1.20.2 on page 275](#)).

<sup>68</sup>Structured Exception Handling

## Automatic deallocation of data in stack

Perhaps the reason for storing local variables and SEH records in the stack is that they are freed automatically upon function exit, using just one instruction to correct the stack pointer (it is often ADD). Function arguments, as we could say, are also deallocated automatically at the end of function. In contrast, everything stored in the *heap* must be deallocated explicitly.

### 1.7.3 A typical stack layout

A typical stack layout in a 32-bit environment at the start of a function, before the first instruction execution looks like this:

...	...
ESP-0xC	local variable#2, marked in IDA as var_8
ESP-8	local variable#1, marked in IDA as var_4
ESP-4	saved value of EBP
ESP	Return Address
ESP+4	argument#1, marked in IDA as arg_0
ESP+8	argument#2, marked in IDA as arg_4
ESP+0xC	argument#3, marked in IDA as arg_8
...	...

### 1.7.4 Noise in stack

When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.

Stephen Wolfram, A New Kind of Science.

Often in this book “noise” or “garbage” values in the stack or memory are mentioned. Where do they come from? These are what has been left there after other functions’ executions. Short example:

```
#include <stdio.h>

void f1()
{
    int a=1, b=2, c=3;
}

void f2()
{
    int a, b, c;
    printf ("%d, %d, %d\n", a, b, c);
}

int main()
{
    f1();
    f2();
}
```

Compiling ...

Listing 1.40: Non-optimizing MSVC 2010

\$SG2752 DB	'%d, %d, %d', 0aH, 00H
_c\$ = -12	; size = 4
_b\$ = -8	; size = 4
_a\$ = -4	; size = 4
_f1 PROC	
push	ebp
mov	ebp, esp

## 1.7. STACK

```
sub    esp, 12
mov    DWORD PTR _a$[ebp], 1
mov    DWORD PTR _b$[ebp], 2
mov    DWORD PTR _c$[ebp], 3
mov    esp, ebp
pop    ebp
ret    0
_f1    ENDP

_c$ = -12      ; size = 4
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
_f2    PROC
push   ebp
mov    ebp, esp
sub    esp, 12
mov    eax, DWORD PTR _c$[ebp]
push   eax
mov    ecx, DWORD PTR _b$[ebp]
push   ecx
mov    edx, DWORD PTR _a$[ebp]
push   edx
push   OFFSET $SG2752 ; '%d, %d, %d'
call   DWORD PTR __imp_printf
add    esp, 16
mov    esp, ebp
pop    ebp
ret    0
_f2    ENDP

_main  PROC
push   ebp
mov    ebp, esp
call   _f1
call   _f2
xor    eax, eax
pop    ebp
ret    0
_main  ENDP
```

The compiler will grumble a little...

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:st.exe
st.obj
```

But when we run the compiled program ...

```
c:\Polygon\c>st
1, 2, 3
```

Oh, what a weird thing! We did not set any variables in `f2()`. These are “ghosts” values, which are still in the stack.

## 1.7. STACK

Let's load the example into OllyDbg:

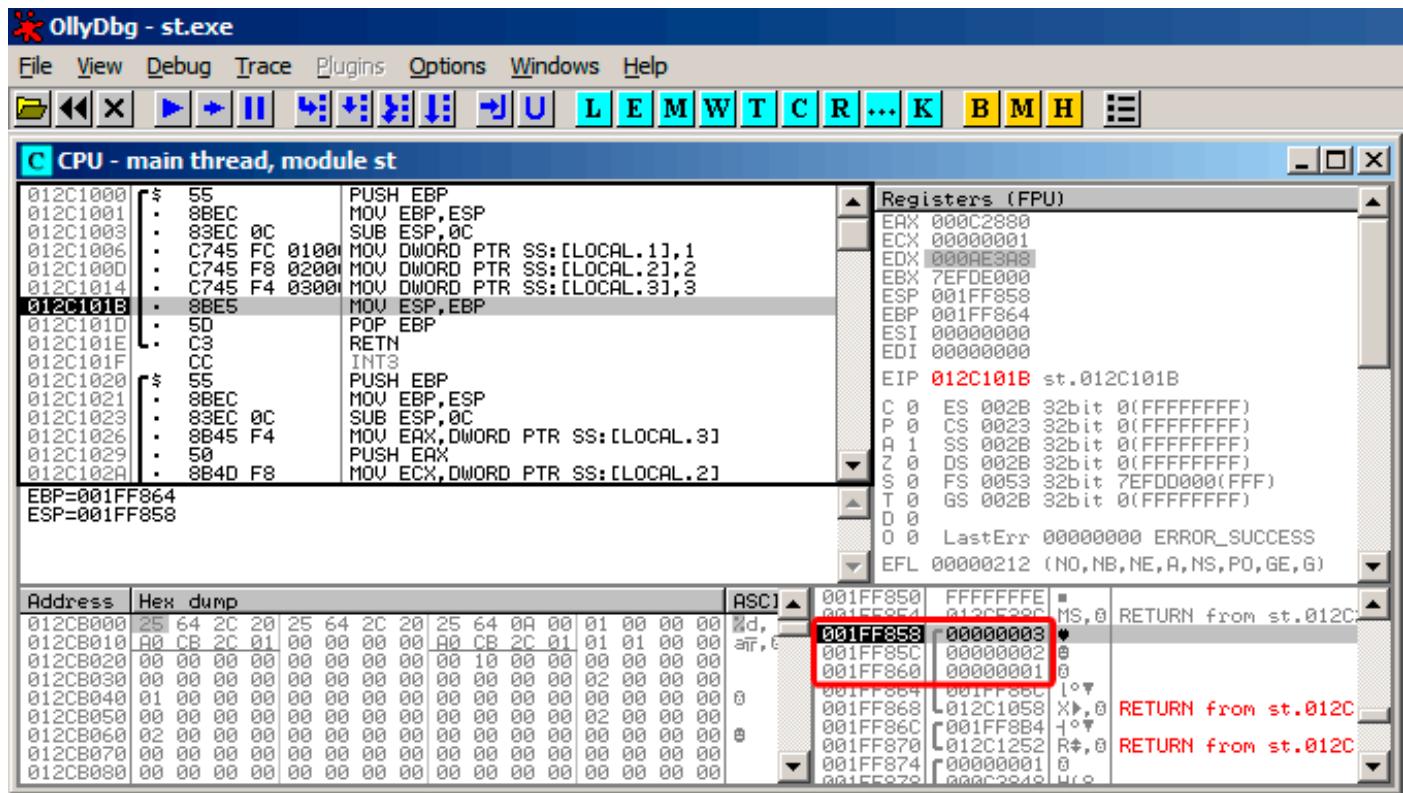


Figure 1.5: OllyDbg: `f1()`

When `f1()` assigns the variables `a`, `b` and `c`, their values are stored at the address `0x1FF860` and so on.

## 1.7. STACK

And when `f2()` executes:

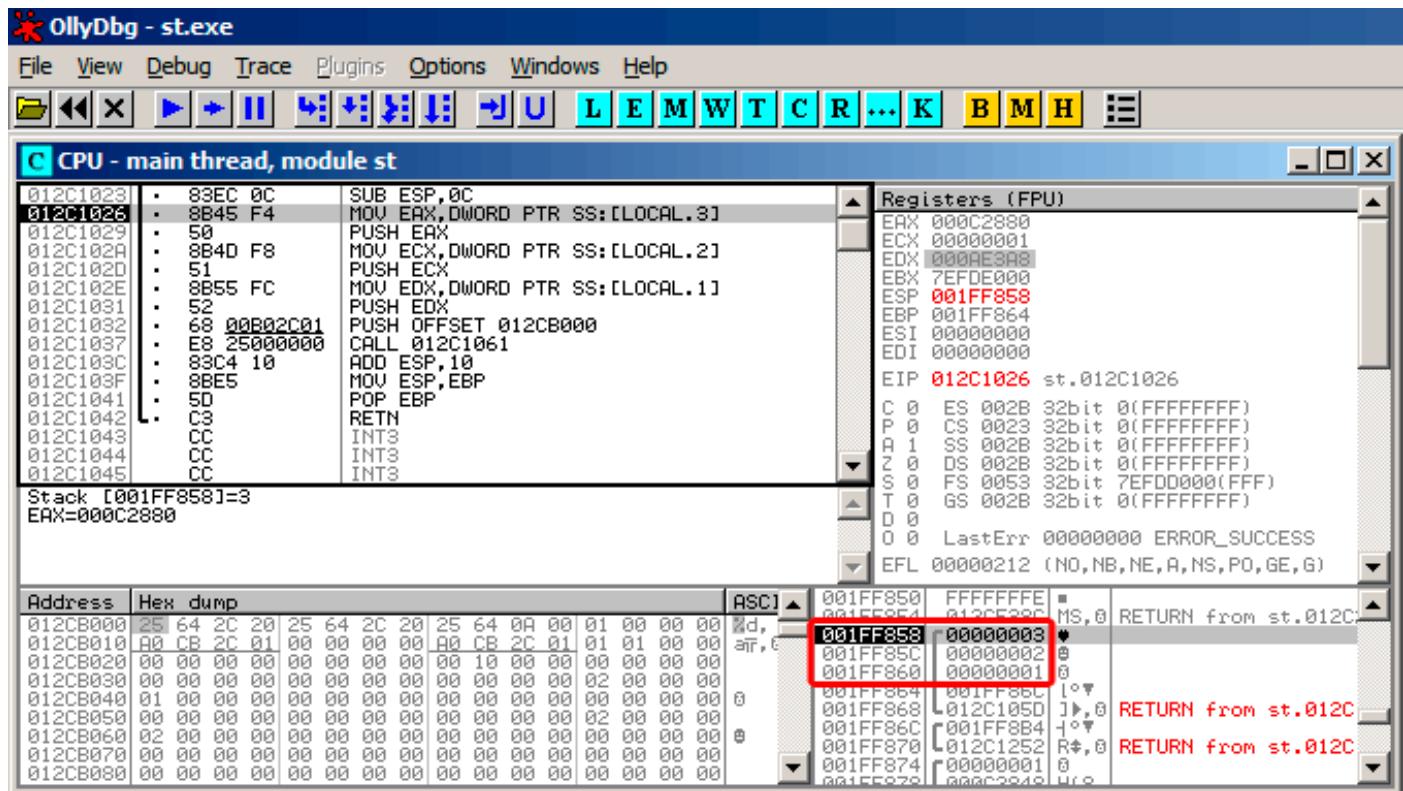


Figure 1.6: OllyDbg: `f2()`

... `a`, `b` and `c` of `f2()` are located at the same addresses! No one has overwritten the values yet, so at that point they are still untouched. So, for this weird situation to occur, several functions have to be called one after another and `SP` has to be the same at each function entry (i.e., they have the same number of arguments). Then the local variables will be located at the same positions in the stack. Summarizing, all values in the stack (and memory cells in general) have values left there from previous function executions. They are not random in the strict sense, but rather have unpredictable values. Is there another option? It would probably be possible to clear portions of the stack before each function execution, but that's too much extra (and unnecessary) work.

## MSVC 2013

The example was compiled by MSVC 2010. But the reader of this book made attempt to compile this example in MSVC 2013, ran it, and got all 3 numbers reversed:

```
c:\Polygon\c>st
3, 2, 1
```

Why? I also compiled this example in MSVC 2013 and saw this:

Listing 1.41: MSVC 2013

```
_a$ = -12      ; size = 4
_b$ = -8      ; size = 4
_c$ = -4      ; size = 4
_f2    PROC
...
_f2    ENDP

_c$ = -12      ; size = 4
_b$ = -8      ; size = 4
_a$ = -4      ; size = 4
_f1    PROC
```

```
...
_f1      ENDP
```

Unlike MSVC 2010, MSVC 2013 allocated a/b/c variables in function `f2()` in reverse order. And this is completely correct, because C/C++ standards has no rule, in which order local variables must be allocated in the local stack, if at all. The reason of difference is because MSVC 2010 has one way to do it, and MSVC 2013 has supposedly something changed inside of compiler guts, so it behaves slightly different.

### 1.7.5 Exercises

- <http://challenges.re/51>
- <http://challenges.re/52>

## 1.8 printf() with several arguments

Now let's extend the *Hello, world!* ( [1.5 on page 8](#)) example, replacing `printf()` in the `main()` function body with this:

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};
```

### 1.8.1 x86

#### x86: 3 arguments

##### MSVC

When we compile it with MSVC 2010 Express we get:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H
...
push    3
push    2
push    1
push    OFFSET $SG3830
call    _printf
add    esp, 16          ; 00000010H
```

Almost the same, but now we can see the `printf()` arguments are pushed onto the stack in reverse order. The first argument is pushed last.

By the way, variables of *int* type in 32-bit environment have 32-bit width, that is 4 bytes.

So, we have 4 arguments here.  $4 * 4 = 16$  —they occupy exactly 16 bytes in the stack: a 32-bit pointer to a string and 3 numbers of type *int*.

When the `stack pointer` (`ESP` register) has changed back by the `ADD ESP, X` instruction after a function call, often, the number of function arguments could be deduced by simply dividing `X` by 4.

Of course, this is specific to the `cdecl` calling convention, and only for 32-bit environment.

## 1.8. *PRINTF()* WITH SEVERAL ARGUMENTS

---

See also the calling conventions section ([6.1 on page 733](#)).

In certain cases where several functions return right after one another, the compiler could merge multiple "ADD ESP, X" instructions into one, after the last call:

```
push a1  
push a2  
call ...  
...  
push a1  
call ...  
...  
push a1  
push a2  
push a3  
call ...  
add esp, 24
```

Here is a real-world example:

Listing 1.42: x86

```
.text:100113E7    push    3  
.text:100113E9    call     sub_100018B0 ; takes one argument (3)  
.text:100113EE    call     sub_100019D0 ; takes no arguments at all  
.text:100113F3    call     sub_10006A90 ; takes no arguments at all  
.text:100113F8    push    1  
.text:100113FA    call     sub_100018B0 ; takes one argument (1)  
.text:100113FF    add     esp, 8      ; drops two arguments from stack at once
```

**MSVC and OllyDbg**

Now let's try to load this example in OllyDbg. It is one of the most popular user-land win32 debuggers. We can compile our example in MSVC 2012 with `/MD` option, which means to link with `MSVCR*.DLL`, so we can see the imported functions clearly in the debugger.

Then load the executable in OllyDbg. The very first breakpoint is in `ntdll.dll`, press F9 (run). The second breakpoint is in `CRT`-code. Now we have to find the `main()` function.

Find this code by scrolling the code to the very top (MSVC allocates the `main()` function at the very beginning of the code section):

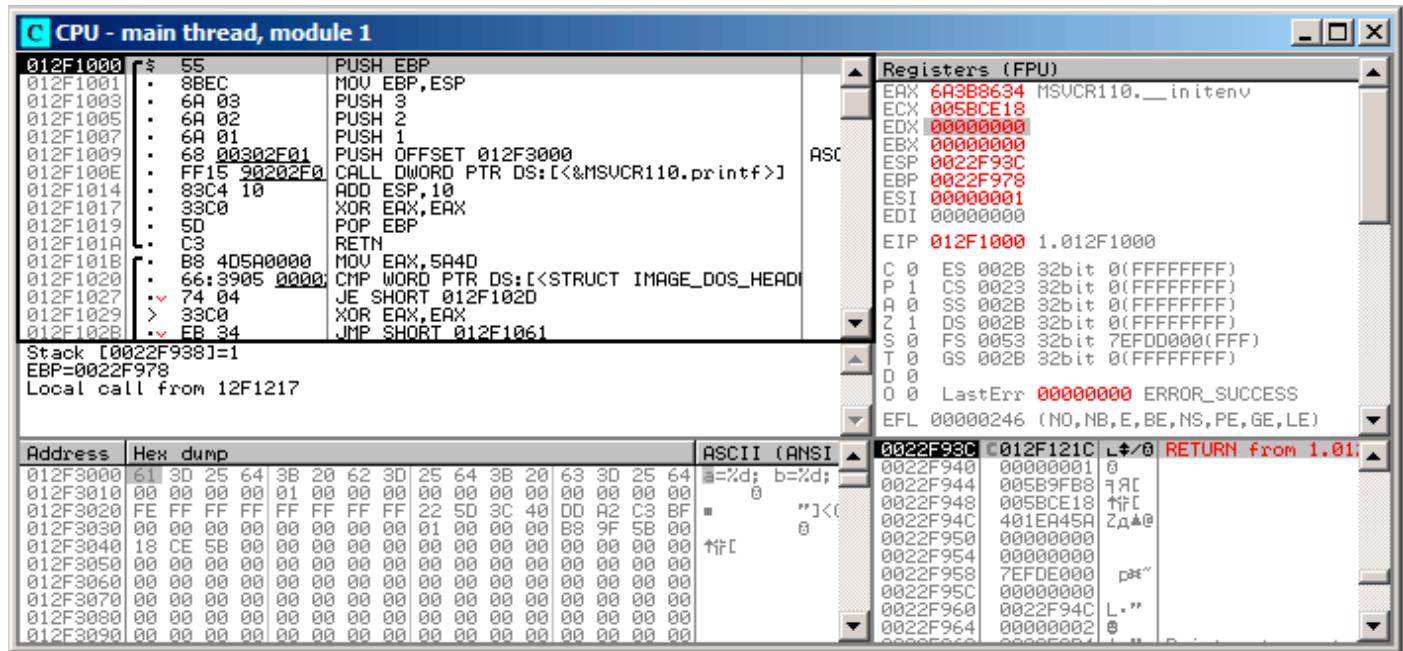


Figure 1.7: OllyDbg: the very start of the `main()` function

Click on the `PUSH EBP` instruction, press F2 (set breakpoint) and press F9 (run). We have to perform these actions in order to skip `CRT`-code, because we aren't really interested in it yet.

## 1.8. `PRINTF()` WITH SEVERAL ARGUMENTS

Press F8 (step over) 6 times, i.e. skip 6 instructions:

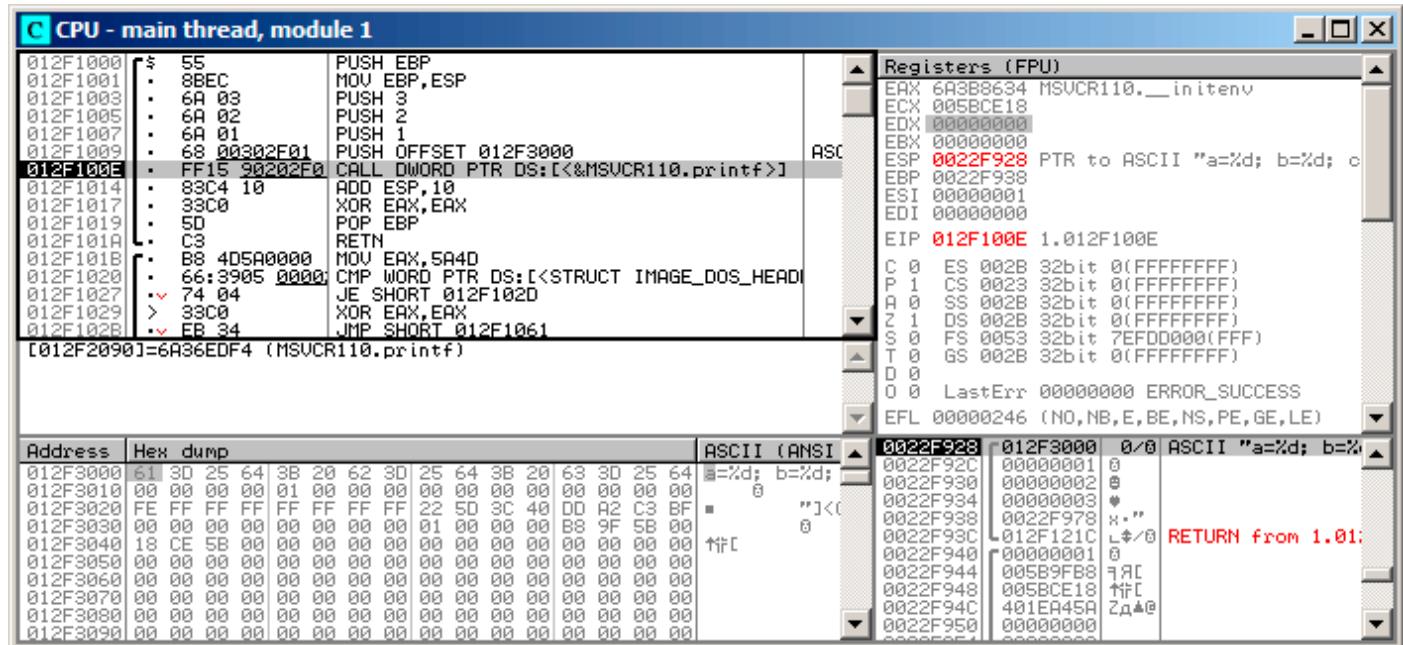


Figure 1.8: OllyDbg: before `printf()` execution

Now the PC points to the `CALL printf` instruction. OllyDbg, like other debuggers, highlights the value of the registers which were changed. So each time you press F8, EIP changes and its value is displayed in red. ESP changes as well, because the arguments values are pushed into the stack.

Where are the values in the stack? Take a look at the right bottom debugger window:

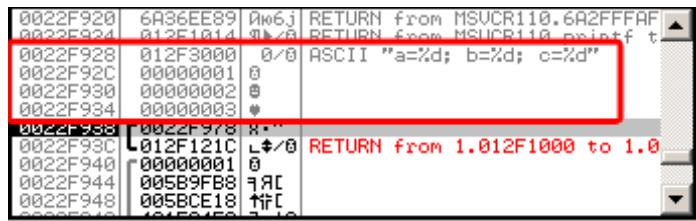


Figure 1.9: OllyDbg: stack after the argument values have been pushed (The red rectangular border was added by the author in a graphics editor)

We can see 3 columns there: address in the stack, value in the stack and some additional OllyDbg comments. OllyDbg understands `printf()`-like strings, so it reports the string here and the 3 values attached to it.

It is possible to right-click on the format string, click on “Follow in dump”, and the format string will appear in the debugger left-bottom window, which always displays some part of the memory. These memory values can be edited. It is possible to change the format string, in which case the result of our example would be different. It is not very useful in this particular case, but it could be good as an exercise so you start building a feel of how everything works here.

## 1.8. `PRINTF()` WITH SEVERAL ARGUMENTS

Press F8 (step over).

We see the following output in the console:

```
a=1; b=2; c=3
```

Let's see how the registers and stack state have changed:

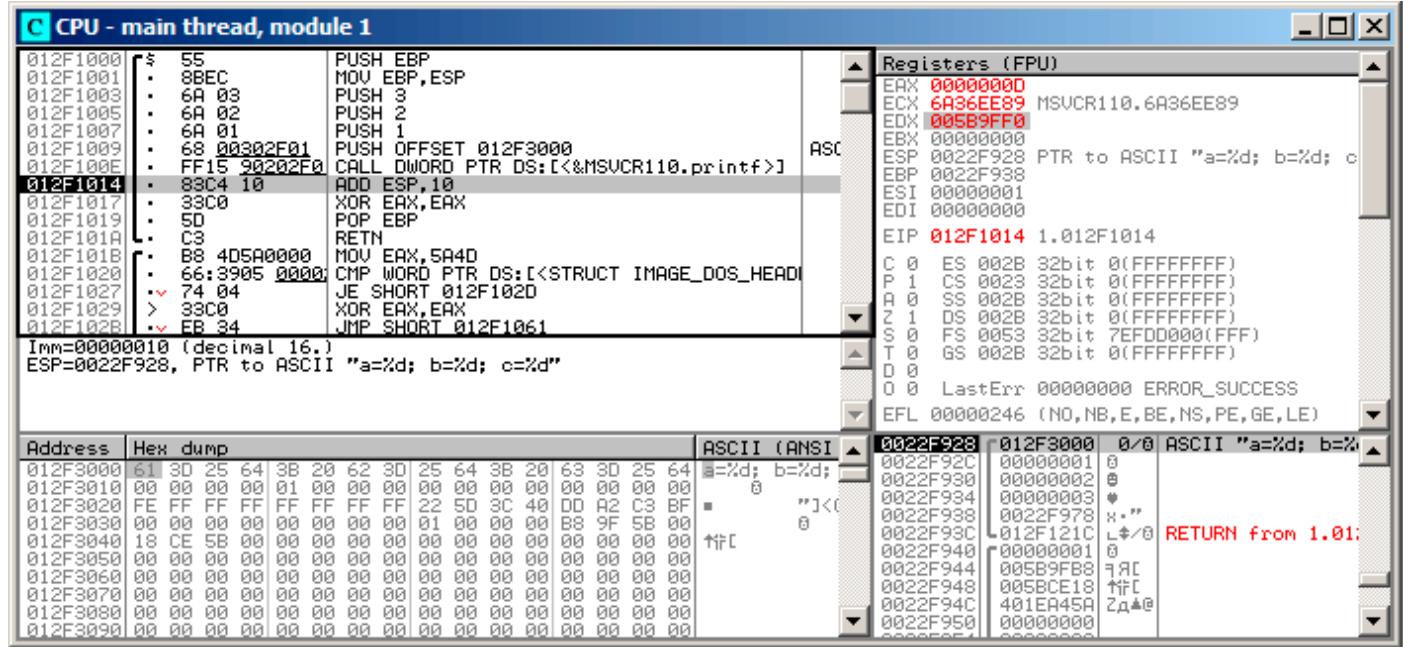


Figure 1.10: OllyDbg after `printf()` execution

Register `EAX` now contains `0xD` (13). That is correct, since `printf()` returns the number of characters printed. The value of `EIP` has changed: indeed, now it contains the address of the instruction coming after `CALL printf`. `ECX` and `EDX` values have changed as well. Apparently, the `printf()` function's hidden machinery used them for its own needs.

A very important fact is that neither the `ESP` value, nor the stack state have been changed! We clearly see that the format string and corresponding 3 values are still there. This is indeed the `cdecl` calling convention behavior: `callee` does not return `ESP` back to its previous value. The `caller` is responsible to do so.

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

Press F8 again to execute ADD ESP, 10 instruction:

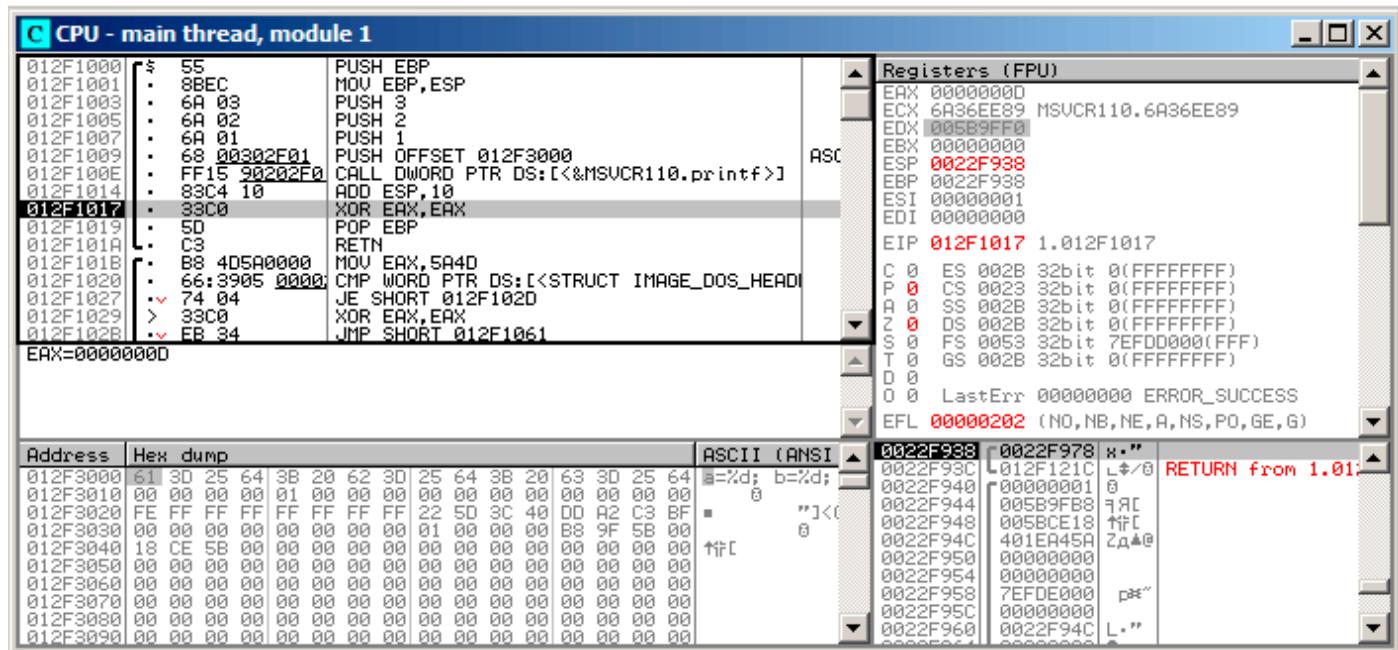


Figure 1.11: OllyDbg: after ADD ESP, 10 instruction execution

ESP has changed, but the values are still in the stack! Yes, of course; no one needs to set these values to zeros or something like that. Everything above the stack pointer (SP) is *noise* or *garbage* and has no meaning at all. It would be time consuming to clear the unused stack entries anyway, and no one really needs to.

## GCC

Now let's compile the same program in Linux using GCC 4.4.1 and take a look at what we have got in IDA:

```

main          proc near

var_10        = dword ptr -10h
var_C         = dword ptr -0Ch
var_8         = dword ptr -8
var_4         = dword ptr -4

push    ebp
mov     ebp, esp
and    esp, 0FFFFFFF0h
sub    esp, 10h
mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
mov     [esp+10h+var_4], 3
mov     [esp+10h+var_8], 2
mov     [esp+10h+var_C], 1
mov     [esp+10h+var_10], eax
call    _printf
mov     eax, 0
leave
retn
main          endp
  
```

It's noticeable that the difference between the MSVC code and the GCC code is only in the way the arguments are stored on the stack. Here the GCC is working directly with the stack without the use of PUSH / POP.

## GCC and GDB

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

Let's try this example also in [GDB](#)<sup>69</sup> in Linux.

`-g` option instructs the compiler to include debug information in the executable file.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.43: let's set breakpoint on `printf()`

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Run. We don't have the `printf()` function source code here, so [GDB](#) can't show it, but may do so.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Print 10 stack elements. The most left column contains addresses on the stack.

```
(gdb) x/10w $esp
0xbffff11c:    0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c:    0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c:    0xb7e29905      0x00000001      0x00000000      0x00000000
```

The very first element is the [RA](#) (`0x0804844a`). We can verify this by disassembling the memory at this address:

```
(gdb) x/5i 0x0804844a
0x804844a <main+45>: mov    $0x0,%eax
0x804844f <main+50>: leave
0x8048450 <main+51>: ret
0x8048451:   xchg   %ax,%ax
0x8048453:   xchg   %ax,%ax
```

The two `XCHG` instructions are idle instructions, analogous to [NOPs](#).

The second element (`0x080484f0`) is the format string address:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

Next 3 elements (1, 2, 3) are the `printf()` arguments. The rest of the elements could be just "garbage" on the stack, but could also be values from other functions, their local variables, etc. We can ignore them for now.

Run "finish". The command instructs GDB to "execute all instructions until the end of the function". In this case: execute till the end of `printf()`.

```
(gdb) finish
Run till exit from #0 __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
main () at 1.c:6
6           return 0;
Value returned is $2 = 13
```

[GDB](#) shows what `printf()` returned in `EAX` (13). This is the number of characters printed out, just like in the OllyDbg example.

We also see "return 0;" and the information that this expression is in the `1.c` file at the line 6. Indeed, the `1.c` file is located in the current directory, and [GDB](#) finds the string there. How does [GDB](#) know

<sup>69</sup>GNU debugger

## 1.8. `PRINTF()` WITH SEVERAL ARGUMENTS

which C-code line is being currently executed? This is due to the fact that the compiler, while generating debugging information, also saves a table of relations between source code line numbers and instruction addresses. GDB is a source-level debugger, after all.

Let's examine the registers. 13 in `EAX`:

```
(gdb) info registers
eax          0xd      13
ecx          0x0      0
edx          0x0      0
ebx 0xb7fc0000      -1208221696
esp 0xbfffff120      0xbfffff120
ebp 0xbfffff138      0xbfffff138
esi          0x0      0
edi          0x0      0
eip 0x804844a <main+45>
...
...
```

Let's disassemble the current instructions. The arrow points to the instruction to be executed next.

```
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>:    push    %ebp
0x0804841e <+1>:    mov     %esp,%ebp
0x08048420 <+3>:    and    $0xffffffff0,%esp
0x08048423 <+6>:    sub    $0x10,%esp
0x08048426 <+9>:    movl   $0x3,0xc(%esp)
0x0804842e <+17>:   movl   $0x2,0x8(%esp)
0x08048436 <+25>:   movl   $0x1,0x4(%esp)
0x0804843e <+33>:   movl   $0x80484f0,(%esp)
0x08048445 <+40>:   call    0x80482f0 <printf@plt>
=> 0x0804844a <+45>: mov     $0x0,%eax
0x0804844f <+50>:   leave
0x08048450 <+51>:   ret
End of assembler dump.
```

GDB uses AT&T syntax by default. But it is possible to switch to Intel syntax:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>:    push    ebp
0x0804841e <+1>:    mov     ebp,esp
0x08048420 <+3>:    and    esp,0xffffffff0
0x08048423 <+6>:    sub    esp,0x10
0x08048426 <+9>:    mov    DWORD PTR [esp+0xc],0x3
0x0804842e <+17>:   mov    DWORD PTR [esp+0x8],0x2
0x08048436 <+25>:   mov    DWORD PTR [esp+0x4],0x1
0x0804843e <+33>:   mov    DWORD PTR [esp],0x80484f0
0x08048445 <+40>:   call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>: mov    eax,0x0
0x0804844f <+50>:   leave
0x08048450 <+51>:   ret
End of assembler dump.
```

Execute next instruction. GDB shows ending bracket, meaning, it ends the block.

```
(gdb) step
7      ;
```

Let's examine the registers after the `MOV EAX, 0` instruction execution. Indeed `EAX` is zero at that point.

```
(gdb) info registers
eax          0x0      0
ecx          0x0      0
edx          0x0      0
ebx 0xb7fc0000      -1208221696
esp 0xbfffff120      0xbfffff120
ebp 0xbfffff138      0xbfffff138
esi          0x0      0
```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```
edi          0x0      0
eip          0x804844f      0x804844f <main+50>
...
...
```

### x64: 8 arguments

To see how other arguments are passed via the stack, let's change our example again by increasing the number of arguments to 9 (`printf()` format string + 8 `int` variables):

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
};
```

### MSVC

As it was mentioned earlier, the first 4 arguments has to be passed through the `RCX`, `RDX`, `R8`, `R9` registers in Win64, while all the rest—via the stack. That is exactly what we see here. However, the `MOV` instruction, instead of `PUSH`, is used for preparing the stack, so the values are stored to the stack in a straightforward manner.

Listing 1.44: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main PROC
    sub    rsp, 88

    mov    DWORD PTR [rsp+64], 8
    mov    DWORD PTR [rsp+56], 7
    mov    DWORD PTR [rsp+48], 6
    mov    DWORD PTR [rsp+40], 5
    mov    DWORD PTR [rsp+32], 4
    mov    r9d, 3
    mov    r8d, 2
    mov    edx, 1
    lea    rcx, OFFSET FLAT:$SG2923
    call   printf

    ; return 0
    xor    eax, eax

    add    rsp, 88
    ret    0
main ENDP
_TEXT ENDS
END
```

The observant reader may ask why are 8 bytes allocated for `int` values, when 4 is enough? Yes, one has to recall: 8 bytes are allocated for any data type shorter than 64 bits. This is established for the convenience's sake: it makes it easy to calculate the address of arbitrary argument. Besides, they are all located at aligned memory addresses. It is the same in the 32-bit environments: 4 bytes are reserved for all data types.

### GCC

The picture is similar for x86-64 \*NIX OS-es, except that the first 6 arguments are passed through the `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` registers. All the rest—via the stack. GCC generates the code storing the string pointer into `EDI` instead of `RDI`—we noted that previously: [1.5.2 on page 15](#).

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

We also noted earlier that the `EAX` register has been cleared before a `printf()` call: [1.5.2 on page 15](#).

Listing 1.45: Optimizing GCC 4.4.6 x64

```
.LC0:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
    sub    rsp, 40

    mov    r9d, 5
    mov    r8d, 4
    mov    ecx, 3
    mov    edx, 2
    mov    esi, 1
    mov    edi, OFFSET FLAT:.LC0
    xor    eax, eax ; number of vector registers passed
    mov    DWORD PTR [rsp+16], 8
    mov    DWORD PTR [rsp+8], 7
    mov    DWORD PTR [rsp], 6
    call   printf

    ; return 0

    xor    eax, eax
    add    rsp, 40
    ret
```

## GCC + GDB

Let's try this example in [GDB](#).

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 1.46: let's set the breakpoint to `printf()`, and run

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at ↴
  ↴ printf.c:29
29      printf.c: No such file or directory.
```

Registers `RSI` / `RDX` / `RCX` / `R8` / `R9` have the expected values. `RIP` has the address of the very first instruction of the `printf()` function.

```
(gdb) info registers
rax          0x0      0
rbx          0x0      0
rcx          0x3      3
rdx          0x2      2
rsi          0x1      1
rdi          0x400628 4195880
rbp          0x7fffffffdf60 0x7fffffffdf60
rsp          0x7fffffffdf38 0x7fffffffdf38
r8           0x4      4
r9           0x5      5
r10          0x7fffffffdfce0 140737488346336
r11          0x7fffff7a65f60 140737348263776
r12          0x400440 4195392
```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```
r13          0x7fffffff040  140737488347200
r14          0x0          0
r15          0x0          0
rip          0x7ffff7a65f60  0x7ffff7a65f60 <__printf>
...
```

Listing 1.47: let's inspect the format string

```
(gdb) x/s $rdi
0x400628: "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Let's dump the stack with the `x/g` command this time—`g` stands for *giant words*, i.e., 64-bit words.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x00000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007fffffe048      0x0000000100000000
```

The very first stack element, just like in the previous case, is the `RA`. 3 values are also passed through the stack: 6, 7, 8. We also see that 8 is passed with the high 32-bits not cleared: `0x00007fff00000008`. That's OK, because the values are of `int` type, which is 32-bit. So, the high register or stack element part may contain “random garbage”.

If you take a look at where the control will return after the `printf()` execution, `GDB` will show the entire `main()` function:

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x00000000000400576
Dump of assembler code for function main:
0x0000000000040052d <+0>:    push   rbp
0x0000000000040052e <+1>:    mov    rbp,rs
0x00000000000400531 <+4>:    sub    rsp,0x20
0x00000000000400535 <+8>:    mov    DWORD PTR [rsp+0x10],0x8
0x0000000000040053d <+16>:   mov    DWORD PTR [rsp+0x8],0x7
0x00000000000400545 <+24>:   mov    DWORD PTR [rsp],0x6
0x0000000000040054c <+31>:   mov    r9d,0x5
0x00000000000400552 <+37>:   mov    r8d,0x4
0x00000000000400558 <+43>:   mov    ecx,0x3
0x0000000000040055d <+48>:   mov    edx,0x2
0x00000000000400562 <+53>:   mov    esi,0x1
0x00000000000400567 <+58>:   mov    edi,0x400628
0x0000000000040056c <+63>:   mov    eax,0x0
0x00000000000400571 <+68>:   call   0x400410 <printf@plt>
0x00000000000400576 <+73>:   mov    eax,0x0
0x0000000000040057b <+78>:   leave 
0x0000000000040057c <+79>:   ret
End of assembler dump.
```

Let's finish executing `printf()`, execute the instruction zeroing `EAX`, and note that the `EAX` register has a value of exactly zero. `RIP` now points to the `LEAVE` instruction, i.e., the penultimate one in the `main()` function.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at printf.c:29
  ↳ d\n")
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6          return 0;
Value returned is $1 = 39
(gdb) next
7      };
(gdb) info registers
rax          0x0          0
rbx          0x0          0
rcx          0x26        38
rdx          0x7ffff7dd59f0  140737351866864
```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```
rsi          0x7fffffd9      2147483609
rdi          0x0          0
rbp          0x7fffffffdf60    0x7fffffffdf60
rsp          0x7fffffffdf40    0x7fffffffdf40
r8           0x7fffff7dd26a0    140737351853728
r9           0x7fffff7a60134    140737348239668
r10          0x7fffffffdf5b0    140737488344496
r11          0x7fffff7a95900   140737348458752
r12          0x400440 4195392
r13          0x7fffffff040     140737488347200
r14          0x0          0
r15          0x0          0
rip          0x40057b 0x40057b <main+78>
...
```

### 1.8.2 ARM

#### ARM: 3 arguments

ARM's traditional scheme for passing arguments (calling convention) behaves as follows: the first 4 arguments are passed through the R0 - R3 registers; the remaining arguments via the stack. This resembles the arguments passing scheme in fastcall ([6.1.3 on page 734](#)) or win64 ([6.1.5 on page 736](#)).

#### 32-bit ARM

#### Non-optimizing Keil 6/2013 (ARM mode)

Listing 1.48: Non-optimizing Keil 6/2013 (ARM mode)

```
.text:00000000 main
.text:00000000 10 40 2D E9  STMFD  SP!, {R4,LR}
.text:00000004 03 30 A0 E3  MOV     R3, #3
.text:00000008 02 20 A0 E3  MOV     R2, #2
.text:0000000C 01 10 A0 E3  MOV     R1, #1
.text:00000010 08 00 8F E2  ADR     R0, aADBD_CD ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB  BL      _2printf
.text:00000018 00 00 A0 E3  MOV     R0, #0       ; return 0
.text:0000001C 10 80 BD E8  LDMFD  SP!, {R4,PC}
```

So, the first 4 arguments are passed via the R0 - R3 registers in this order: a pointer to the printf() format string in R0, then 1 in R1, 2 in R2 and 3 in R3. The instruction at 0x18 writes 0 to R0 —this is return 0 C-statement. There is nothing unusual so far.

Optimizing Keil 6/2013 generates the same code.

#### Optimizing Keil 6/2013 (Thumb mode)

Listing 1.49: Optimizing Keil 6/2013 (Thumb mode)

```
.text:00000000 main
.text:00000000 10 B5      PUSH    {R4,LR}
.text:00000002 03 23      MOVS    R3, #3
.text:00000004 02 22      MOVS    R2, #2
.text:00000006 01 21      MOVS    R1, #1
.text:00000008 02 A0      ADR     R0, aADBD_CD ; "a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8  BL      _2printf
.text:0000000E 00 20      MOVS    R0, #0
.text:00000010 10 BD      POP     {R4,PC}
```

There is no significant difference from the non-optimized code for ARM mode.

**Optimizing Keil 6/2013 (ARM mode) + let's remove return**

Let's rework example slightly by removing *return 0*:

```
#include <stdio.h>

void main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
}
```

The result is somewhat unusual:

Listing 1.50: Optimizing Keil 6/2013 (ARM mode)

```
.text:00000014 main
.text:00000014 03 30 A0 E3    MOV      R3, #3
.text:00000018 02 20 A0 E3    MOV      R2, #2
.text:0000001C 01 10 A0 E3    MOV      R1, #1
.text:00000020 1E 0E 8F E2    ADR      R0, aADBDLCD ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA    B       __2printf
```

This is the optimized (-O3) version for ARM mode and this time we see **B** as the last instruction instead of the familiar **BL**. Another difference between this optimized version and the previous one (compiled without optimization) is the lack of function prologue and epilogue (instructions preserving the **R0** and **LR** registers values). The **B** instruction just jumps to another address, without any manipulation of the **LR** register, similar to **JMP** in x86. Why does it work? Because this code is, in fact, effectively equivalent to the previous. There are two main reasons: 1) neither the stack nor **SP** (the **stack pointer**) is modified; 2) the call to **printf()** is the last instruction, so there is nothing going on afterwards. On completion, the **printf()** function simply returns the control to the address stored in **LR**. Since the **LR** currently stores the address of the point from where our function has been called then the control from **printf()** will be returned to that point. Therefore we do not have to save **LR** because we do not have necessity to modify **LR**. And we do not have necessity to modify **LR** because there are no other function calls except **printf()**. Furthermore, after this call we do not do anything else! That is the reason such optimization is possible.

This optimization is often used in functions where the last statement is a call to another function. A similar example is presented here: [1.15.1 on page 156](#).

**ARM64****Non-optimizing GCC (Linaro) 4.9**

Listing 1.51: Non-optimizing GCC (Linaro) 4.9

```
.LC1:
    .string "a=%d; b=%d; c=%d"
f2:
; save FP and LR in stack frame:
    stp    x29, x30, [sp, -16]!
; set stack frame (FP=SP):
    add    x29, sp, 0
    adrp   x0, .LC1
    add    x0, x0, :lo12:.LC1
    mov    w1, 1
    mov    w2, 2
    mov    w3, 3
    bl     printf
    mov    w0, 0
; restore FP and LR
    ldp    x29, x30, [sp], 16
    ret
```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

The first instruction `STP` (*Store Pair*) saves `FP` (X29) and `LR` (X30) in the stack. The second `ADD X29, SP, 0` instruction forms the stack frame. It is just writing the value of `SP` into X29.

Next, we see the familiar `ADRP / ADD` instruction pair, which forms a pointer to the string. *l012* meaning low 12 bits, i.e., linker will write low 12 bits of LC1 address into the opcode of `ADD` instruction. `%d` in `printf()` string format is a 32-bit *int*, so the 1, 2 and 3 are loaded into 32-bit register parts.

Optimizing GCC (Linaro) 4.9 generates the same code.

### ARM: 8 arguments

Let's use again the example with 9 arguments from the previous section: [1.8.1 on page 50](#).

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
}
```

### Optimizing Keil 6/2013: ARM mode

```
.text:00000028          main
.text:00000028
.text:00000028          var_18 = -0x18
.text:00000028          var_14 = -0x14
.text:00000028          var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR   LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB   SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV    R3, #8
.text:00000034 07 20 A0 E3  MOV    R2, #7
.text:00000038 06 10 A0 E3  MOV    R1, #6
.text:0000003C 05 00 A0 E3  MOV    R0, #5
.text:00000040 04 C0 8D E2  ADD   R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV    R0, #4
.text:0000004C 00 00 8D E5  STR   R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV    R3, #3
.text:00000054 02 20 A0 E3  MOV    R2, #2
.text:00000058 01 10 A0 E3  MOV    R1, #1
.text:0000005C 6E 0F 8F E2  ADR   R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d"
     ↴ =%"...
.text:00000060 BC 18 00 EB  BL    __2printf
.text:00000064 14 D0 8D E2  ADD   SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR   PC, [SP+4+var_4],#4
```

This code can be divided into several parts:

- Function prologue:

The very first `STR LR, [SP,#var_4]!` instruction saves `LR` on the stack, because we are going to use this register for the `printf()` call. Exclamation mark at the end indicates *pre-index*.

This implies that `SP` is to be decreased by 4 first, and then `LR` will be saved at the address stored in `SP`. This is similar to `PUSH` in x86. Read more about it at: [1.32.2 on page 441](#).

The second `SUB SP, SP, #0x14` instruction decreases `SP` (the *stack pointer*) in order to allocate `0x14` (20) bytes on the stack. Indeed, we have to pass 5 32-bit values via the stack to the `printf()` function, and each one occupies 4 bytes, which is exactly  $5 \times 4 = 20$ . The other 4 32-bit values are to be passed through registers.

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

- Passing 5, 6, 7 and 8 via the stack: they are stored in the R0, R1, R2 and R3 registers respectively.

Then, the ADD R12, SP, #0x18+var\_14 instruction writes the stack address where these 4 variables are to be stored, into the R12 register. var\_14 is an assembly macro, equal to -0x14, created by IDA to conveniently display the code accessing the stack. The var\_? macros generated by IDA reflect local variables in the stack.

So, SP+4 is to be stored into the R12 register.

The next STMIA R12, R0-R3 instruction writes registers R0 - R3 contents to the memory pointed by R12. STMIA abbreviates *Store Multiple Increment After*. "Increment After" implies that R12 is to be increased by 4 after each register value is written.

- Passing 4 via the stack: 4 is stored in R0 and then this value, with the help of the STR R0, [SP,#0x18+var\_18] instruction is saved on the stack. var\_18 is -0x18, so the offset is to be 0, thus the value from the R0 register (4) is to be written to the address written in SP.
- Passing 1, 2 and 3 via registers: The values of the first 3 numbers (a, b, c) (1, 2, 3 respectively) are passed through the R1, R2 and R3 registers right before the printf() call, and the other 5 values are passed via the stack:
- printf() call.
- Function epilogue:

The ADD SP, SP, #0x14 instruction restores the SP pointer back to its former value, thus annulling everything what has been stored on the stack. Of course, what has been stored on the stack will stay there, but it will all be rewritten during the execution of subsequent functions.

The LDR PC, [SP+4+var\_4],#4 instruction loads the saved LR value from the stack into the PC register, thus causing the function to exit. There is no exclamation mark—indeed, PC is loaded first from the address stored in SP ( $4+var\_4 = 4 + (-4) = 0$ ), so this instruction is analogous to LDR PC, [SP],#4, and then SP is increased by 4. This is referred as *post-index*<sup>70</sup>. Why does IDA display the instruction like that? Because it wants to illustrate the stack layout and the fact that var\_4 is allocated for saving the LR value in the local stack. This instruction is somewhat similar to POP PC in x86<sup>71</sup>.

## Optimizing Keil 6/2013: Thumb mode

```
.text:0000001C          printf_main2
.text:0000001C
.text:0000001C          var_18 = -0x18
.text:0000001C          var_14 = -0x14
.text:0000001C          var_8  = -8
.text:0000001C
.text:0000001C 00 B5    PUSH   {LR}
.text:0000001E 08 23    MOVS   R3, #8
.text:00000020 85 B0    SUB    SP, SP, #0x14
.text:00000022 04 93    STR    R3, [SP,#0x18+var_8]
.text:00000024 07 22    MOVS   R2, #7
.text:00000026 06 21    MOVS   R1, #6
.text:00000028 05 20    MOVS   R0, #5
.text:0000002A 01 AB    ADD    R3, SP, #0x18+var_14
.text:0000002C 07 C3    STMIA  R3!, {R0-R2}
.text:0000002E 04 20    MOVS   R0, #4
.text:00000030 00 90    STR    R0, [SP,#0x18+var_18]
.text:00000032 03 23    MOVS   R3, #3
.text:00000034 02 22    MOVS   R2, #2
.text:00000036 01 21    MOVS   R1, #1
.text:00000038 A0 A0    ADR    R0, aADBDCDDDEFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; ↴
     g=%"...
.text:0000003A 06 F0 D9 F8    BL     __2printf
.text:0000003E          loc_3E  ; CODE XREF: example13_f+16
```

<sup>70</sup>Read more about it: [1.32.2 on page 441](#).

<sup>71</sup>It is impossible to set IP/EIP/RIP value using POP in x86, but anyway, you got the analogy right.

## 1.8. `PRINTF()` WITH SEVERAL ARGUMENTS

```
.text:0000003E 05 B0      ADD     SP, SP, #0x14
.text:00000040 00 BD      POP    {PC}
```

The output is almost like in the previous example. However, this is Thumb code and the values are packed into stack differently: 8 goes first, then 5, 6, 7, and 4 goes third.

### Optimizing Xcode 4.6.3 (LLVM): ARM mode

```
_text:0000290C          _printf_main2
_text:0000290C
_text:0000290C          var_1C = -0x1C
_text:0000290C          var_C  = -0xC
_text:0000290C
_text:0000290C 80 40 2D E9  STMFD  SP!, {R7,LR}
_text:00002910 0D 70 A0 E1  MOV     R7, SP
_text:00002914 14 D0 4D E2  SUB    SP, SP, #0x14
_text:00002918 70 05 01 E3  MOV     R0, #0x1570
_text:0000291C 07 C0 A0 E3  MOV     R12, #7
_text:00002920 00 00 40 E3  MOVT   R0, #0
_text:00002924 04 20 A0 E3  MOV     R2, #4
_text:00002928 00 00 8F E0  ADD    R0, PC, R0
_text:0000292C 06 30 A0 E3  MOV     R3, #6
_text:00002930 05 10 A0 E3  MOV     R1, #5
_text:00002934 00 20 8D E5  STR    R2, [SP,#0x1C+var_1C]
_text:00002938 0A 10 8D E9  STMFA  SP, {R1,R3,R12}
_text:0000293C 08 90 A0 E3  MOV     R9, #8
_text:00002940 01 10 A0 E3  MOV     R1, #1
_text:00002944 02 20 A0 E3  MOV     R2, #2
_text:00002948 03 30 A0 E3  MOV     R3, #3
_text:0000294C 10 90 8D E5  STR    R9, [SP,#0x1C+var_C]
_text:00002950 A4 05 00 EB  BL     _printf
_text:00002954 07 D0 A0 E1  MOV     SP, R7
_text:00002958 80 80 BD E8  LDMFD  SP!, {R7,PC}
```

Almost the same as what we have already seen, with the exception of `STMFA` (Store Multiple Full Ascending) instruction, which is a synonym of `STMIB` (Store Multiple Increment Before) instruction. This instruction increases the value in the `SP` register and only then writes the next register value into the memory, rather than performing those two actions in the opposite order.

Another thing that catches the eye is that the instructions are arranged seemingly random. For example, the value in the `R0` register is manipulated in three places, at addresses `0x2918`, `0x2920` and `0x2928`, when it would be possible to do it in one point.

However, the optimizing compiler may have its own reasons on how to order the instructions so to achieve higher efficiency during the execution.

Usually, the processor attempts to simultaneously execute instructions located side-by-side. For example, instructions like `MOVT R0, #0` and `ADD R0, PC, R0` cannot be executed simultaneously since they both modify the `R0` register. On the other hand, `MOVT R0, #0` and `MOV R2, #4` instructions can be executed simultaneously since the effects of their execution are not conflicting with each other. Presumably, the compiler tries to generate code in such a manner (wherever it is possible).

### Optimizing Xcode 4.6.3 (LLVM): Thumb-2 mode

```
_text:00002BA0          _printf_main2
_text:00002BA0
_text:00002BA0          var_1C = -0x1C
_text:00002BA0          var_18 = -0x18
_text:00002BA0          var_C  = -0xC
_text:00002BA0
_text:00002BA0 80 B5      PUSH    {R7,LR}
_text:00002BA2 6F 46      MOV     R7, SP
_text:00002BA4 85 B0      SUB    SP, SP, #0x14
_text:00002BA6 41 F2 D8 20  MOVW   R0, #0x12D8
```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```
__text:00002BAA 4F F0 07 0C    MOV.W   R12, #7
__text:00002BAE C0 F2 00 00    MOVT.W  R0, #0
__text:00002BB2 04 22          MOVS    R2, #4
__text:00002BB4 78 44          ADD     R0, PC ; char *
__text:00002BB6 06 23          MOVS    R3, #6
__text:00002BB8 05 21          MOVS    R1, #5
__text:00002BBA 0D F1 04 0E    ADD.W   LR, SP, #0x1C+var_18
__text:00002BBE 00 92          STR     R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09    MOV.W   R9, #8
__text:00002BC4 8E E8 0A 10    STMIA.W LR, {R1,R3,R12}
__text:00002BC8 01 21          MOVS    R1, #1
__text:00002BCA 02 22          MOVS    R2, #2
__text:00002BCC 03 23          MOVS    R3, #3
__text:00002BCE CD F8 10 90    STR.W   R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA    BLX    _printf
__text:00002BD6 05 B0          ADD    SP, SP, #0x14
__text:00002BD8 80 BD          POP    {R7,PC}
```

The output is almost the same as in the previous example, with the exception that Thumb-instructions are used instead.

## ARM64

### Non-optimizing GCC (Linaro) 4.9

Listing 1.52: Non-optimizing GCC (Linaro) 4.9

```
.LC2:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; grab more space in stack:
    sub    sp, sp, #32
; save FP and LR in stack frame:
    stp    x29, x30, [sp,16]
; set stack frame (FP=SP):
    add    x29, sp, 16
    adrp   x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
    add    x0, x0, :lo12:.LC2
    mov    w1, 8           ; 9th argument
    str    w1, [sp]        ; store 9th argument in the stack
    mov    w1, 1
    mov    w2, 2
    mov    w3, 3
    mov    w4, 4
    mov    w5, 5
    mov    w6, 6
    mov    w7, 7
    bl     printf
    sub    sp, x29, #16
; restore FP and LR
    ldp    x29, x30, [sp,16]
    add    sp, sp, 32
    ret
```

The first 8 arguments are passed in X- or W-registers: [Procedure Call Standard for the ARM 64-bit Architecture (AArch64), (2013)]<sup>72</sup>. A string pointer requires a 64-bit register, so it's passed in X0. All other values have a *int* 32-bit type, so they are stored in the 32-bit part of the registers (W-). The 9th argument (8) is passed via the stack. Indeed: it's not possible to pass large number of arguments through registers, because the number of registers is limited.

Optimizing GCC (Linaro) 4.9 generates the same code.

<sup>72</sup>Also available as <http://go.yurichev.com/17287>

**1.8.3 MIPS****3 arguments****Optimizing GCC 4.4.5**

The main difference with the “Hello, world!” example is that in this case `printf()` is called instead of `puts()` and 3 more arguments are passed through the registers \$5...\$7 (or \$A0...\$A2). That is why these registers are prefixed with A-, which implies they are used for function arguments passing.

Listing 1.53: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
    .ascii  "a=%d; b=%d; c=%d\000"
main:
; function prologue:
    lui      $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-32
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw      $31,28($sp)
; load address of printf():
    lw      $25,%call16(sprintf)($28)
; load address of the text string and set 1st argument of printf():
    lui      $4,%hi($LC0)
    addiu   $4,$4,%lo($LC0)
; set 2nd argument of printf():
    li      $5,1                      # 0x1
; set 3rd argument of printf():
    li      $6,2                      # 0x2
; call printf():
    jalr   $25
; set 4th argument of printf() (branch delay slot):
    li      $7,3                      # 0x3

; function epilogue:
    lw      $31,28($sp)
; set return value to 0:
    move   $2,$0
; return
    j      $31
    addiu $sp,$sp,32 ; branch delay slot
```

Listing 1.54: Optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.text:00000000          lui      $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu   $sp, -0x20
.text:00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw      $ra, 0x20+var_4($sp)
.text:00000010          sw      $gp, 0x20+var_10($sp)
; load address of printf():
.text:00000014          lw      $t9, (printf & 0xFFFF)($gp)
; load address of the text string and set 1st argument of printf():
.text:00000018          la      $a0, $LC0      # "a=%d; b=%d; c=%d"
; set 2nd argument of printf():
.text:00000020          li      $a1, 1
; set 3rd argument of printf():
.text:00000024          li      $a2, 2
; call printf():
.text:00000028          jalr   $t9
; set 4th argument of printf() (branch delay slot):
.text:0000002C          li      $a3, 3
; function epilogue:
.text:00000030          lw      $ra, 0x20+var_4($sp)
```

## 1.8. `PRINTF()` WITH SEVERAL ARGUMENTS

```
; set return value to 0:  
.text:00000034          move    $v0, $zero  
;  
; return  
.text:00000038          jr     $ra  
.text:0000003C          addiu  $sp, 0x20 ; branch delay slot
```

IDA has coalesced pair of `LUI` and `ADDIU` instructions into one `LA` pseudo instruction. That's why there are no instruction at address 0x1C: because `LA` occupies 8 bytes.

### Non-optimizing GCC 4.4.5

Non-optimizing GCC is more verbose:

Listing 1.55: Non-optimizing GCC 4.4.5 (assembly output)

```
$LC0:  
    .ascii  "a=%d; b=%d; c=%d\000"  
main:  
; function prologue:  
    addiu  $sp,$sp,-32  
    sw     $31,28($sp)  
    sw     $fp,24($sp)  
    move   $fp,$sp  
    lui    $28,%hi(__gnu_local_gp)  
    addiu  $28,$28,%lo(__gnu_local_gp)  
; load address of the text string:  
    lui    $2,%hi($LC0)  
    addiu  $2,$2,%lo($LC0)  
; set 1st argument of printf():  
    move   $4,$2  
; set 2nd argument of printf():  
    li     $5,1                  # 0x1  
; set 3rd argument of printf():  
    li     $6,2                  # 0x2  
; set 4th argument of printf():  
    li     $7,3                  # 0x3  
; get address of printf():  
    lw     $2,%call16(sprintf)($28)  
    nop  
; call printf():  
    move   $25,$2  
    jalr   $25  
    nop  
  
; function epilogue:  
    lw     $28,16($fp)  
; set return value to 0:  
    move   $2,$0  
    move   $sp,$fp  
    lw     $31,28($sp)  
    lw     $fp,24($sp)  
    addiu $sp,$sp,32  
; return  
    j      $31  
    nop
```

Listing 1.56: Non-optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:  
.text:00000000  
.text:00000000 var_10      = -0x10  
.text:00000000 var_8       = -8  
.text:00000000 var_4       = -4  
.text:00000000  
; function prologue:  
.text:00000000          addiu  $sp, -0x20  
.text:00000004          sw     $ra, 0x20+var_4($sp)  
.text:00000008          sw     $fp, 0x20+var_8($sp)
```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```
.text:0000000C          move   $fp, $sp
.text:00000010          la     $gp, __gnu_local_gp
.text:00000018          sw     $gp, 0x20+var_10($sp)
; load address of the text string:
.text:0000001C          la     $v0, aADBD_CD      # "a=%d; b=%d; c=%d"
; set 1st argument of printf():
.text:00000024          move   $a0, $v0
; set 2nd argument of printf():
.text:00000028          li     $a1, 1
; set 3rd argument of printf():
.text:0000002C          li     $a2, 2
; set 4th argument of printf():
.text:00000030          li     $a3, 3
; get address of printf():
.text:00000034          lw     $v0, (printf & 0xFFFF)($gp)
.text:00000038          or     $at, $zero
; call printf():
.text:0000003C          move   $t9, $v0
.text:00000040          jalr   $t9
.text:00000044          or     $at, $zero ; NOP
; function epilogue:
.text:00000048          lw     $gp, 0x20+var_10($fp)
; set return value to 0:
.text:0000004C          move   $v0, $zero
.text:00000050          move   $sp, $fp
.text:00000054          lw     $ra, 0x20+var_4($sp)
.text:00000058          lw     $fp, 0x20+var_8($sp)
.text:0000005C          addiu $sp, 0x20
; return
.text:00000060          jr     $ra
.or    $at, $zero ; NOP
```

## 8 arguments

Let's use again the example with 9 arguments from the previous section: [1.8.1 on page 50](#).

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
}
```

## Optimizing GCC 4.4.5

Only the first 4 arguments are passed in the \$A0 ...\$A3 registers, the rest are passed via the stack.

This is the O32 calling convention (which is the most common one in the MIPS world). Other calling conventions (like N32) may use the registers for different purposes.

**SW** abbreviates “Store Word” (from register to memory). MIPS lacks instructions for storing a value into memory, so an instruction pair has to be used instead (**LI / SW**).

Listing 1.57: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
.ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
    lui    $28,%hi(__gnu_local_gp)
    addiu $sp,$sp,-56
    addiu $28,$28,%lo(__gnu_local_gp)
    sw    $31,52($sp)
; pass 5th argument in stack:
    li    $2,4           # 0x4
```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```

        sw      $2,16($sp)
; pass 6th argument in stack:
        li      $2,5                      # 0x5
        sw      $2,20($sp)
; pass 7th argument in stack:
        li      $2,6                      # 0x6
        sw      $2,24($sp)
; pass 8th argument in stack:
        li      $2,7                      # 0x7
        lw      $25,%call16(sprintf)($28)
        sw      $2,28($sp)
; pass 1st argument in $a0:
        lui     $4,%hi($LC0)
; pass 9th argument in stack:
        li      $2,8                      # 0x8
        sw      $2,32($sp)
        addiu $4,$4,%lo($LC0)
; pass 2nd argument in $a1:
        li      $5,1                      # 0x1
; pass 3rd argument in $a2:
        li      $6,2                      # 0x2
; call printf():
        jalr   $25
; pass 4th argument in $a3 (branch delay slot):
        li      $7,3                      # 0x3

; function epilogue:
        lw      $31,52($sp)
; set return value to 0:
        move   $2,$0
; return
        j      $31
        addiu $sp,$sp,56 ; branch delay slot

```

Listing 1.58: Optimizing GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.text:00000000                 lui      $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu  $sp, -0x38
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x38+var_4($sp)
.text:00000010                 sw      $gp, 0x38+var_10($sp)
; pass 5th argument in stack:
.text:00000014                 li      $v0, 4
.text:00000018                 sw      $v0, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000001C                 li      $v0, 5
.text:00000020                 sw      $v0, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000024                 li      $v0, 6
.text:00000028                 sw      $v0, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000002C                 li      $v0, 7
.text:00000030                 lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034                 sw      $v0, 0x38+var_1C($sp)
; prepare 1st argument in $a0:
.text:00000038                 lui     $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d"
    ↴ ; g=%"...
; pass 9th argument in stack:
.text:0000003C                 li      $v0, 8

```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```
.text:00000040          sw      $v0, 0x38+var_18($sp)
; pass 1st argument in $a0:      la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d"...
.text:00000044          la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d"...
; pass 2nd argument in $a1:      li      $a1, 1
; pass 3rd argument in $a2:      li      $a2, 2
.text:0000004C          li      $a2, 2
; call printf():              jalr   $t9
; pass 4th argument in $a3 (branch delay slot):      li      $a3, 3
; function epilogue:          lw      $ra, 0x38+var_4($sp)
; set return value to 0:       move   $v0, $zero
; return                      jr      $ra
.text:00000060          addiu  $sp, 0x38 ; branch delay slot
.text:00000064
```

## Non-optimizing GCC 4.4.5

Non-optimizing GCC is more verbose:

Listing 1.59: Non-optimizing GCC 4.4.5 (assembly output)

```
$LC0:
.ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
    addiu $sp,$sp,-56
    sw    $31,52($sp)
    sw    $fp,48($sp)
    move  $fp,$sp
    lui   $28,%hi(__gnu_local_gp)
    addiu $28,$28,%lo(__gnu_local_gp)
    lui   $2,%hi($LC0)
    addiu $2,$2,%lo($LC0)
; pass 5th argument in stack:
    li    $3,4                  # 0x4
    sw    $3,16($sp)
; pass 6th argument in stack:
    li    $3,5                  # 0x5
    sw    $3,20($sp)
; pass 7th argument in stack:
    li    $3,6                  # 0x6
    sw    $3,24($sp)
; pass 8th argument in stack:
    li    $3,7                  # 0x7
    sw    $3,28($sp)
; pass 9th argument in stack:
    li    $3,8                  # 0x8
    sw    $3,32($sp)
; pass 1st argument in $a0:
    move  $4,$2
; pass 2nd argument in $a1:
    li    $5,1                  # 0x1
; pass 3rd argument in $a2:
    li    $6,2                  # 0x2
; pass 4th argument in $a3:
    li    $7,3                  # 0x3
; call printf():
    lw    $2,%call16(sprintf)($28)
    nop
    move  $25,$2
    jalr $25
    nop
; function epilogue:
```

## 1.8. *PRINTF()* WITH SEVERAL ARGUMENTS

```

lw      $28,40($fp)
; set return value to 0:
move   $2,$0
move   $sp,$fp
lw      $31,52($sp)
lw      $fp,48($sp)
addiu $sp,$sp,56
; return
j       $31
nop

```

Listing 1.60: Non-optimizing GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.text:00000000 addiu  $sp, -0x38
.text:00000004 sw     $ra, 0x38+var_4($sp)
.text:00000008 sw     $fp, 0x38+var_8($sp)
.text:0000000C move   $fp, $sp
.text:00000010 la     $gp, __gnu_local_gp
.text:00000018 sw     $gp, 0x38+var_10($sp)
.text:0000001C la     $v0, aABDCDDDEDFFGD # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
; pass 5th argument in stack:
.text:00000024 li     $v1, 4
.text:00000028 sw     $v1, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000002C li     $v1, 5
.text:00000030 sw     $v1, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000034 li     $v1, 6
.text:00000038 sw     $v1, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000003C li     $v1, 7
.text:00000040 sw     $v1, 0x38+var_1C($sp)
; pass 9th argument in stack:
.text:00000044 li     $v1, 8
.text:00000048 sw     $v1, 0x38+var_18($sp)
; pass 1st argument in $a0:
.text:0000004C move   $a0, $v0
; pass 2nd argument in $a1:
.text:00000050 li     $a1, 1
; pass 3rd argument in $a2:
.text:00000054 li     $a2, 2
; pass 4th argument in $a3:
.text:00000058 li     $a3, 3
; call printf():
.text:0000005C lw     $v0, (printf & 0xFFFF)($gp)
.text:00000060 or     $at, $zero
.text:00000064 move   $t9, $v0
.text:00000068 jalr  $t9
.text:0000006C or     $at, $zero ; NOP
; function epilogue:
.text:00000070 lw     $gp, 0x38+var_10($fp)
; set return value to 0:
.text:00000074 move   $v0, $zero
.text:00000078 move   $sp, $fp
.text:0000007C lw     $ra, 0x38+var_4($sp)
.text:00000080 lw     $fp, 0x38+var_8($sp)
.text:00000084 addiu $sp, 0x38

```

## 1.8. PRINTF() WITH SEVERAL ARGUMENTS

```
; return
.text:00000088          jr      $ra
.text:0000008C          or      $at, $zero ; NOP
```

### 1.8.4 Conclusion

Here is a rough skeleton of the function call:

Listing 1.61: x86

```
...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL function
; modify stack pointer (if needed)
```

Listing 1.62: x64 (MSVC)

```
MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5th, 6th argument, etc. (if needed)
CALL function
; modify stack pointer (if needed)
```

Listing 1.63: x64 (GCC)

```
MOV RDI, 1st argument
MOV RSI, 2nd argument
MOV RDX, 3rd argument
MOV RCX, 4th argument
MOV R8, 5th argument
MOV R9, 6th argument
...
PUSH 7th, 8th argument, etc. (if needed)
CALL function
; modify stack pointer (if needed)
```

Listing 1.64: ARM

```
MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; pass 5th, 6th argument, etc., in stack (if needed)
BL function
; modify stack pointer (if needed)
```

Listing 1.65: ARM64

```
MOV X0, 1st argument
MOV X1, 2nd argument
MOV X2, 3rd argument
MOV X3, 4th argument
MOV X4, 5th argument
MOV X5, 6th argument
MOV X6, 7th argument
MOV X7, 8th argument
; pass 9th, 10th argument, etc., in stack (if needed)
BL function
; modify stack pointer (if needed)
```

```

LI $4, 1st argument ; AKA $A0
LI $5, 2nd argument ; AKA $A1
LI $6, 3rd argument ; AKA $A2
LI $7, 4th argument ; AKA $A3
; pass 5th, 6th argument, etc., in stack (if needed)
LW temp_reg, address of function
JALR temp_reg

```

## 1.8.5 By the way

By the way, this difference between the arguments passing in x86, x64, fastcall, ARM and MIPS is a good illustration of the fact that the CPU is oblivious to how the arguments are passed to functions. It is also possible to create a hypothetical compiler able to pass arguments via a special structure without using stack at all.

MIPS \$A0 ...\$A3 registers are labeled this way only for convenience (that is in the O32 calling convention). Programmers may use any other register (well, maybe except \$ZERO) to pass data or use any other calling convention.

The [CPU](#) is not aware of calling conventions whatsoever.

We may also recall how new coming assembly language programmers passing arguments into other functions: usually via registers, without any explicit order, or even via global variables. Of course, it works fine.

## 1.9 scanf()

Now let's use `scanf()`.

### 1.9.1 Simple example

```

#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
}

```

It's not clever to use `scanf()` for user interactions nowadays. But we can, however, illustrate passing a pointer to a variable of type `int`.

### About pointers

Pointers are one of the fundamental concepts in computer science. Often, passing a large array, structure or object as an argument to another function is too expensive, while passing their address is much cheaper. For example, if you going to print a text string to console, it's much easier to pass its address into [OS](#) kernel.

In addition if the [callee](#) function needs to modify something in the large array or structure received as a parameter and return back the entire structure then the situation is close to absurd. So the simplest thing to do is to pass the address of the array or structure to the [callee](#) function, and let it change what needs to be changed.

A pointer in C/C++—is simply an address of some memory location.

## 1.9. SCANF()

In x86, the address is represented as a 32-bit number (i.e., it occupies 4 bytes), while in x86-64 it is a 64-bit number (occupying 8 bytes). By the way, that is the reason behind some people's indignation related to switching to x86-64—all pointers in the x64-architecture require twice as much space, including cache memory, which is “expensive” memory.

It is possible to work with untyped pointers only, given some effort; e.g. the standard C function `memcpy()`, that copies a block from one memory location to another, takes 2 pointers of type `void*` as arguments, since it is impossible to predict the type of the data you would like to copy. Data types are not important, only the block size matters.

Pointers are also widely used when a function needs to return more than one value (we are going to get back to this later ([3.22 on page 615](#))).

`scanf()` function—is such a case.

Besides the fact that the function needs to indicate how many values were successfully read, it also needs to return all these values.

In C/C++ the pointer type is only needed for compile-time type checking.

Internally, in the compiled code there is no information about pointer types at all.

## x86

### MSVC

Here is what we get after compiling with MSVC 2010:

```
CONST SEGMENT
$SG3831    DB      'Enter X:', 0aH, 00H
$SG3832    DB      '%d', 00H
$SG3833    DB      'You entered %d...', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_x$ = -4           ; size = 4
_main PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    OFFSET $SG3831 ; 'Enter X:'
    call    _printf
    add     esp, 4
    lea     eax, DWORD PTR _x$[ebp]
    push    eax
    push    OFFSET $SG3832 ; '%d'
    call    _scanf
    add     esp, 8
    mov     ecx, DWORD PTR _x$[ebp]
    push    ecx
    push    OFFSET $SG3833 ; 'You entered %d...'
    call    _printf
    add     esp, 8

    ; return 0
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main ENDP
_TEXT ENDS
```

`x` is a local variable.

## 1.9. SCANF()

According to the C/C++ standard it must be visible only in this function and not from any other external scope. Traditionally, local variables are stored on the stack. There are probably other ways to allocate them, but in x86 that is the way it is.

The goal of the instruction following the function prologue, `PUSH ECX`, is not to save the `ECX` state (notice the absence of corresponding `POP ECX` at the function's end).

In fact it allocates 4 bytes on the stack for storing the `x` variable.

`x` is to be accessed with the assistance of the `_x$` macro (it equals to -4) and the `EBP` register pointing to the current frame.

Over the span of the function's execution, `EBP` is pointing to the current [stack frame](#) making it possible to access local variables and function arguments via `EBP+offset`.

It is also possible to use `ESP` for the same purpose, although that is not very convenient since it changes frequently. The value of the `EBP` could be perceived as a *frozen state* of the value in `ESP` at the start of the function's execution.

Here is a typical [stack frame](#) layout in 32-bit environment:

...	...
EBP-8	local variable #2, marked in IDA as var_8
EBP-4	local variable #1, marked in IDA as var_4
EBP	saved value of EBP
EBP+4	return address
EBP+8	argument#1, marked in IDA as arg_0
EBP+0xC	argument#2, marked in IDA as arg_4
EBP+0x10	argument#3, marked in IDA as arg_8
...	...

The `scanf()` function in our example has two arguments.

The first one is a pointer to the string containing `%d` and the second is the address of the `x` variable.

First, the `x` variable's address is loaded into the `EAX` register by the `lea eax, DWORD PTR _x$[ebp]` instruction.

`LEA` stands for *load effective address*, and is often used for forming an address ([.1.6 on page 1011](#)).

We could say that in this case `LEA` simply stores the sum of the `EBP` register value and the `_x$` macro in the `EAX` register.

This is the same as `lea eax, [ebp-4]`.

So, 4 is being subtracted from the `EBP` register value and the result is loaded in the `EAX` register. Next the `EAX` register value is pushed into the stack and `scanf()` is being called.

`printf()` is being called after that with its first argument — a pointer to the string: `You entered %d...\\n`.

The second argument is prepared with: `mov ecx, [ebp-4]`. The instruction stores the `x` variable value and not its address, in the `ECX` register.

Next the value in the `ECX` is stored on the stack and the last `printf()` is being called.

## 1.9. SCANF()

**MSVC + OllyDbg**

Let's try this example in OllyDbg. Let's load it and keep pressing F8 (step over) until we reach our executable file instead of `ntdll.dll`. Scroll up until `main()` appears.

Click on the first instruction (`PUSH EBP`), press F2 (set a breakpoint), then F9 (Run). The breakpoint will be triggered when `main()` begins.

Let's trace to the point where the address of the variable `x` is calculated:

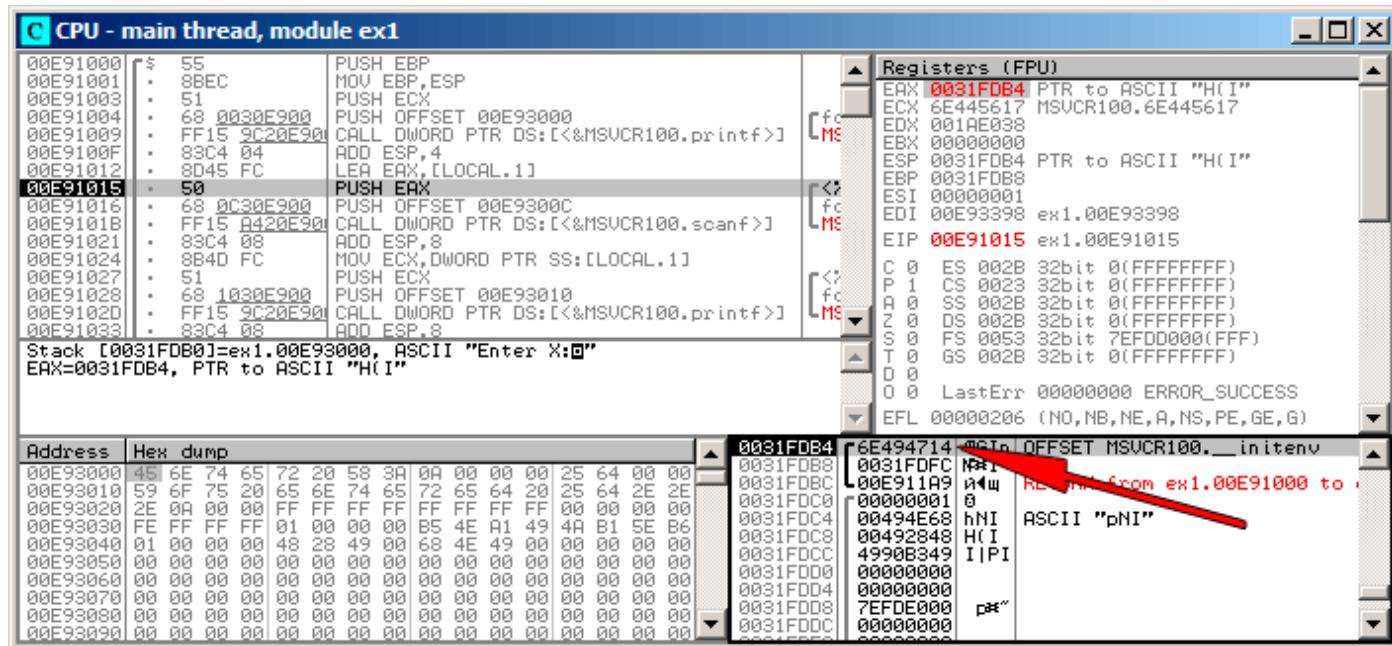


Figure 1.12: OllyDbg: The address of the local variable is calculated

Right-click the `EAX` in the registers window and then select “Follow in stack”.

This address will appear in the stack window. The red arrow has been added, pointing to the variable in the local stack. At that moment this location contains some garbage (`0x6E494714`). Now with the help of `PUSH` instruction the address of this stack element is going to be stored to the same stack on the next position. Let's trace with F8 until the `scanf()` execution completes. During the `scanf()` execution, we input, for example, 123, in the console window:

```
Enter X:  
123
```

## 1.9. SCANF()

scanf() completed its execution already:

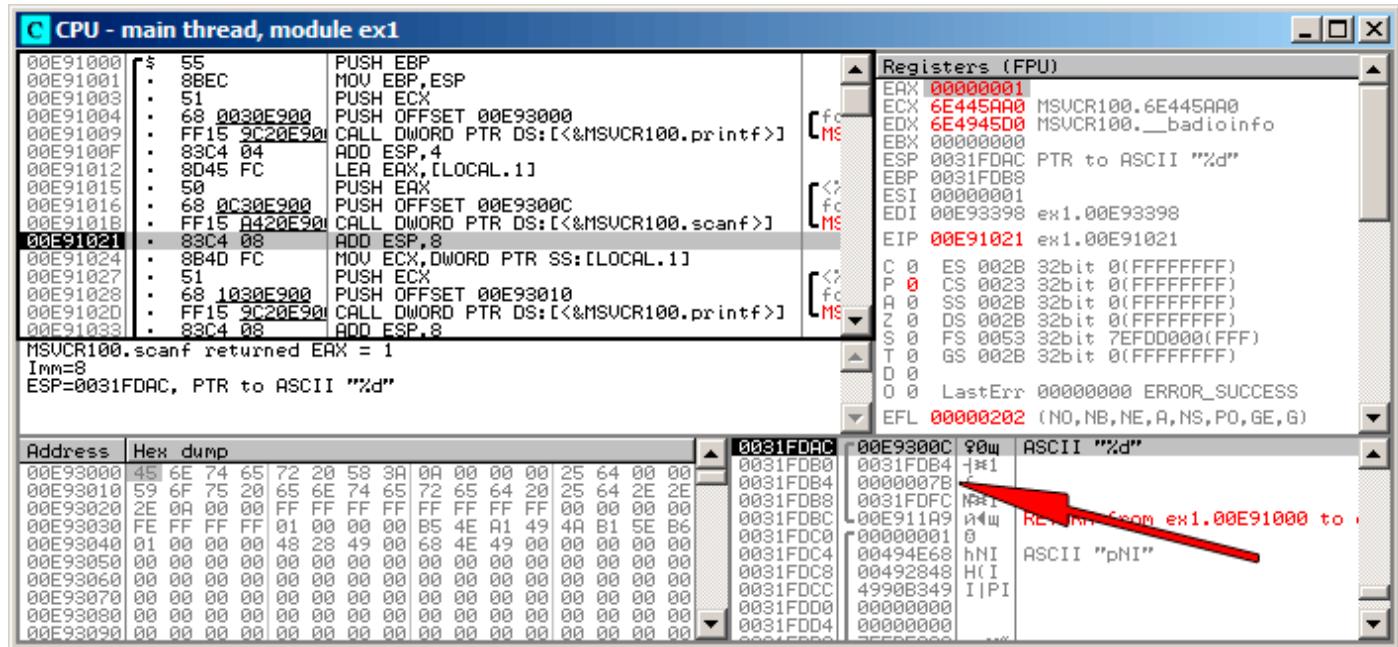


Figure 1.13: OllyDbg: scanf() executed

scanf() returns 1 in EAX, which implies that it has read successfully one value. If we look again at the stack element corresponding to the local variable it now contains 0x7B (123).

## 1.9. SCANF()

Later this value is copied from the stack to the ECX register and passed to `printf()`:

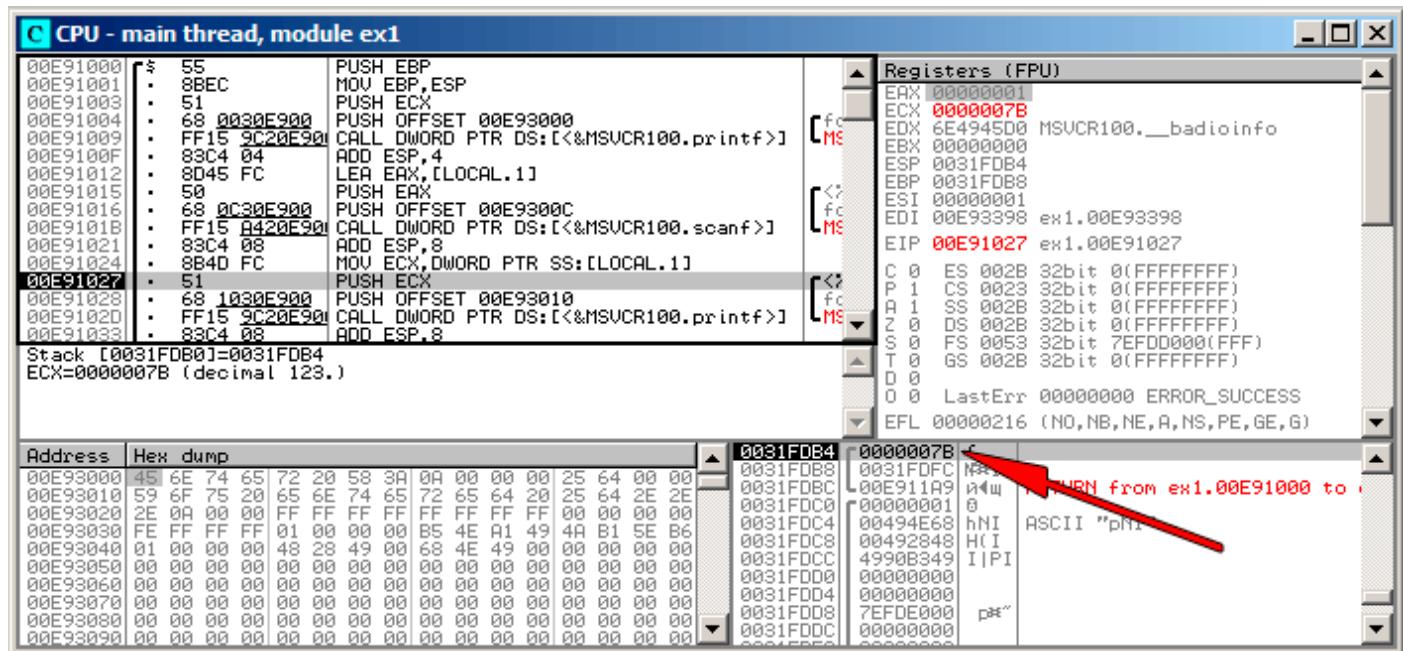


Figure 1.14: OllyDbg: preparing the value for passing to `printf()`

## GCC

Let's try to compile this code in GCC 4.4.1 under Linux:

```

main          proc near
var_20        = dword ptr -20h
var_1C        = dword ptr -1Ch
var_4         = dword ptr -4

push    ebp
mov     ebp, esp
and    esp, 0FFFFFFF0h
sub    esp, 20h
mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
call   _puts
mov     eax, offset aD    ; "%d"
lea     edx, [esp+20h+var_4]
mov     [esp+20h+var_1C], edx
mov     [esp+20h+var_20], eax
call   __isoc99_scanf
mov     edx, [esp+20h+var_4]
mov     eax, offset aYouEnteredD__ ; "You entered %d...\n"
mov     [esp+20h+var_1C], edx
mov     [esp+20h+var_20], eax
call   _printf
mov     eax, 0
leave
retn
endp

```

GCC replaced the `printf()` call with call to `puts()`. The reason for this was explained in ([1.5.4 on page 21](#)).

As in the MSVC example—the arguments are placed on the stack using the `MOV` instruction.

**By the way**

By the way, this simple example is a demonstration of the fact that compiler translates list of expressions in C/C++-block into sequential list of instructions. There are nothing between expressions in C/C++, and so in resulting machine code, there are nothing between, control flow slips from one expression to the next one.

**x64**

The picture here is similar with the difference that the registers, rather than the stack, are used for arguments passing.

**MSVC**

Listing 1.67: MSVC 2012 x64

```

_DATA SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
x$ = 32
main PROC
$LN3:
    sub    rsp, 56
    lea    rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
    call   printf
    lea    rdx, QWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG1291 ; '%d'
    call   scanf
    mov    edx, DWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
    call   printf

    ; return 0
    xor    eax, eax
    add    rsp, 56
    ret    0
main ENDP
_TEXT ENDS

```

**GCC**

Listing 1.68: Optimizing GCC 4.4.6 x64

```

.LC0:
.string "Enter X:"
.LC1:
.string "%d"
.LC2:
.string "You entered %d...\n"

main:
    sub    rsp, 24
    mov    edi, OFFSET FLAT:.LC0 ; "Enter X:"
    call   puts
    lea    rsi, [rsp+12]
    mov    edi, OFFSET FLAT:.LC1 ; "%d"
    xor    eax, eax
    call   __isoc99_scanf
    mov    esi, DWORD PTR [rsp+12]

```

## 1.9. SCANF()

```
    mov    edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
    xor    eax, eax
    call   printf

    ; return 0
    xor    eax, eax
    add    rsp, 24
    ret
```

## ARM

### Optimizing Keil 6/2013 (Thumb mode)

```
.text:00000042          scanf_main
.text:00000042
.text:00000042          var_8        = -8
.text:00000042
.text:00000042 08 B5      PUSH   {R3,LR}
.text:00000044 A9 A0      ADR    R0, aEnterX ; "Enter X:\n"
.text:00000046 06 F0 D3 F8 BL     _2printf
.text:0000004A 69 46      MOV    R1, SP
.text:0000004C AA A0      ADR    R0, aD ; "%d"
.text:0000004E 06 F0 CD F8 BL     _0scanf
.text:00000052 00 99      LDR    R1, [SP,#8+var_8]
.text:00000054 A9 A0      ADR    R0, aYouEnteredD__ ; "You entered %d...\n"
.text:00000056 06 F0 CB F8 BL     _2printf
.text:0000005A 00 20      MOVS   R0, #0
.text:0000005C 08 BD      POP    {R3,PC}
```

In order for `scanf()` to be able to read item it needs a parameter—pointer to an `int`. `int` is 32-bit, so we need 4 bytes to store it somewhere in memory, and it fits exactly in a 32-bit register. A place for the local variable `x` is allocated in the stack and IDA has named it `var_8`. It is not necessary, however, to allocate a such since `SP (stack pointer)` is already pointing to that space and it can be used directly.

So, `SP`'s value is copied to the `R1` register and, together with the format-string, passed to `scanf()`. Later, with the help of the `LDR` instruction, this value is moved from the stack to the `R1` register in order to be passed to `printf()`.

## ARM64

Listing 1.69: Non-optimizing GCC 4.9.1 ARM64

```
1 .LC0:
2     .string "Enter X:"
3 .LC1:
4     .string "%d"
5 .LC2:
6     .string "You entered %d...\n"
7 scanf_main:
8 ; subtract 32 from SP, then save FP and LR in stack frame:
9     stp    x29, x30, [sp, -32]!
10 ; set stack frame (FP=SP)
11     add    x29, sp, 0
12 ; load pointer to the "Enter X:" string:
13     adrp   x0, .LC0
14     add    x0, x0, :lo12:.LC0
15 ; X0=pointer to the "Enter X:" string
16 ; print it:
17     bl     puts
18 ; load pointer to the "%d" string:
19     adrp   x0, .LC1
20     add    x0, x0, :lo12:.LC1
21 ; find a space in stack frame for "x" variable (X1=FP+28):
22     add    x1, x29, 28
```

## 1.9. SCANF()

```
23 ; X1=address of "x" variable
24 ; pass the address to scanf() and call it:
25     bl    __isoc99_scanf
26 ; load 32-bit value from the variable in stack frame:
27     ldr    w1, [x29,28]
28 ; W1=x
29 ; load pointer to the "You entered %d...\n" string
30 ; printf() will take text string from X0 and "x" variable from X1 (or W1)
31     adrp   x0, .LC2
32     add    x0, x0, :lo12:.LC2
33     bl    printf
34 ; return 0
35     mov    w0, 0
36 ; restore FP and LR, then add 32 to SP:
37     ldp    x29, x30, [sp], 32
38     ret
```

There are 32 bytes allocated for stack frame, which is bigger than needed. Perhaps some memory aligning issue? The most interesting part is finding space for the *x* variable in the stack frame (line 22). Why 28? Somehow, compiler decided to place this variable at the end of stack frame instead of beginning. The address is passed to `scanf()`, which just stores the user input value in the memory at that address. This is 32-bit value of type *int*. The value is fetched at line 27 and then passed to `printf()`.

## MIPS

A place in the local stack is allocated for the *x* variable, and it is referred as  $\$sp + 24$ .

Its address is passed to `scanf()`, and the user input values is loaded using the `LW` ("Load Word") instruction and then passed to `printf()`.

Listing 1.70: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
    .ascii  "Enter X:\000"
$LC1:
    .ascii  "%d\000"
$LC2:
    .ascii  "You entered %d...\\012\000"
main:
; function prologue:
    lui    $28,%hi(__gnu_local_gp)
    addiu $sp,$sp,-40
    addiu $28,$28,%lo(__gnu_local_gp)
    sw    $31,36($sp)
; call puts():
    lw    $25,%call16(puts)($28)
    lui    $4,%hi($LC0)
    jalr   $25
    addiu $4,$4,%lo($LC0) ; branch delay slot
; call scanf():
    lw    $28,16($sp)
    lui    $4,%hi($LC1)
    lw    $25,%call16(__isoc99_scanf)($28)
; set 2nd argument of scanf(), $a1=$sp+24:
    addiu $5,$sp,24
    jalr   $25
    addiu $4,$4,%lo($LC1) ; branch delay slot

; call printf():
    lw    $28,16($sp)
; set 2nd argument of printf(),
; load word at address $sp+24:
    lw    $5,24($sp)
    lw    $25,%call16(printf)($28)
    lui    $4,%hi($LC2)
    jalr   $25
    addiu $4,$4,%lo($LC2) ; branch delay slot
```

## 1.9. SCANF()

```
; function epilogue:  
    lw      $31,36($sp)  
; set return value to 0:  
    move   $2,$0  
; return:  
    j      $31  
    addiu $sp,$sp,40      ; branch delay slot
```

IDA displays the stack layout as follows:

Listing 1.71: Optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:  
.text:00000000  
.text:00000000 var_18    = -0x18  
.text:00000000 var_10    = -0x10  
.text:00000000 var_4     = -4  
.text:00000000  
; function prologue:  
.text:00000000        lui      $gp, (_gnu_local_gp >> 16)  
.text:00000004        addiu   $sp, -0x28  
.text:00000008        la      $gp, (_gnu_local_gp & 0xFFFF)  
.text:0000000C        sw      $ra, 0x28+var_4($sp)  
.text:00000010        sw      $gp, 0x28+var_18($sp)  
; call puts():  
.text:00000014        lw      $t9, (puts & 0xFFFF)($gp)  
.text:00000018        lui      $a0, ($LC0 >> 16) # "Enter X:"  
.text:0000001C        jalr   $t9  
.text:00000020        la      $a0, ($LC0 & 0xFFFF) # "Enter X:" ; branch delay slot  
; call scanf():  
.text:00000024        lw      $gp, 0x28+var_18($sp)  
.text:00000028        lui      $a0, ($LC1 >> 16) # "%d"  
.text:0000002C        lw      $t9, (_isoc99_scanf & 0xFFFF)($gp)  
; set 2nd argument of scanf(), $a1=$sp+24:  
.text:00000030        addiu   $a1, $sp, 0x28+var_10  
.text:00000034        jalr   $t9 ; branch delay slot  
.text:00000038        la      $a0, ($LC1 & 0xFFFF) # "%d"  
; call printf():  
.text:0000003C        lw      $gp, 0x28+var_18($sp)  
; set 2nd argument of printf(),  
; load word at address $sp+24:  
.text:00000040        lw      $a1, 0x28+var_10($sp)  
.text:00000044        lw      $t9, (printf & 0xFFFF)($gp)  
.text:00000048        lui      $a0, ($LC2 >> 16) # "You entered %d...\n"  
.text:0000004C        jalr   $t9  
.text:00000050        la      $a0, ($LC2 & 0xFFFF) # "You entered %d...\n" ; branch delay ↴  
    ↴ slot  
; function epilogue:  
.text:00000054        lw      $ra, 0x28+var_4($sp)  
; set return value to 0:  
.text:00000058        move   $v0, $zero  
; return:  
.text:0000005C        jr      $ra  
.text:00000060        addiu $sp, 0x28 ; branch delay slot
```

### 1.9.2 Popular mistake

It's a very popular mistake (and/or typo) to pass value of *x* instead of pointer to *x*:

```
#include <stdio.h>  
  
int main()  
{  
    int x;  
    printf ("Enter X:\n");  
  
    scanf ("%d", x); // BUG
```

## 1.9. SCANF()

```
    printf ("You entered %d...\n", x);

    return 0;
};
```

So what happens here? `x` is not uninitialized and contains some random noise from local stack. When `scanf()` called, it takes string from user, parses it into number and tries to write it into `x`, treating it as an address in memory. But there is a random noise, so `scanf()` will try to write at random address. Most likely, the process will crash.

Interestingly enough, some [CRT](#) libraries in debug build, put visually distinctive patterns into memory just allocated, like `0xCCCCCCCC` or `0xBADF00D` and so on. In this case, `x` may contain `0xCCCCCCCC`, and `scanf()` would try to write at address `0xCCCCCCCC`. And if you'll notice that something in your process tries to write at address `0xCCCCCCCC`, you'll know that uninitialized variable (or pointer) gets used without prior initialization. This is better than as if newly allocated memory is just cleared.

### 1.9.3 Global variables

What if the `x` variable from the previous example isn't local but a global one? Then it would have been accessible from any point, not only from the function body. Global variables are considered [anti-pattern](#), but for the sake of the experiment, we could do this.

```
#include <stdio.h>

// now x is global variable
int x;

int main()
{
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};
```

## MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB      'Enter X:', 0aH, 00H
$SG2457    DB      '%d', 00H
$SG2458    DB      'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC   _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2456
    call    _printf
    add    esp, 4
    push    OFFSET _x
    push    OFFSET $SG2457
    call    _scanf
    add    esp, 8
    mov     eax, DWORD PTR _x
    push    eax
    push    OFFSET $SG2458
```

## 1.9. SCANF()

```
call  _printf
add   esp, 8
xor   eax, eax
pop   ebp
ret   0
_main  ENDP
_TEXT  ENDS
```

In this case the `x` variable is defined in the `_DATA` segment and no memory is allocated in the local stack. It is accessed directly, not through the stack. Uninitialized global variables take no space in the executable file (indeed, why one needs to allocate space for variables initially set to zero?), but when someone accesses their address, the OS will allocate a block of zeros there<sup>73</sup>.

Now let's explicitly assign a value to the variable:

```
int x=10; // default value
```

We got:

```
_DATA  SEGMENT
_x     DD      0AH
...
```

Here we see a value `0xA` of DWORD type (DD stands for DWORD = 32 bit) for this variable.

If you open the compiled .exe in [IDA](#), you can see the `x` variable placed at the beginning of the `_DATA` segment, and after it you can see text strings.

If you open the compiled .exe from the previous example in [IDA](#), where the value of `x` hasn't been set, you would see something like this:

Listing 1.72: [IDA](#)

```
.data:0040FA80 _x          dd ? ; DATA XREF: _main+10
.data:0040FA80                 ; _main+22
.data:0040FA84 dword_40FA84  dd ? ; DATA XREF: _memset+1E
.data:0040FA84                 ; unknown_libname_1+28
.data:0040FA88 dword_40FA88  dd ? ; DATA XREF: __sbh_find_block+5
.data:0040FA88                 ; __sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem         dd ? ; DATA XREF: __sbh_find_block+B
.data:0040FA8C                 ; __sbh_free_block+2CA
.data:0040FA90 dword_40FA90  dd ? ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                 ; _calloc_impl+72
.data:0040FA94 dword_40FA94  dd ? ; DATA XREF: __sbh_free_block+2FE
```

`_x` is marked with `?` with the rest of the variables that do not need to be initialized. This implies that after loading the .exe to the memory, a space for all these variables is to be allocated and filled with zeros [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.8p10]. But in the .exe file these uninitialized variables do not occupy anything. This is convenient for large arrays, for example.

<sup>73</sup>That is how a [VM](#) behaves

## 1.9. SCANF()

**MSVC: x86 + OllyDbg**

Things are even simpler here:

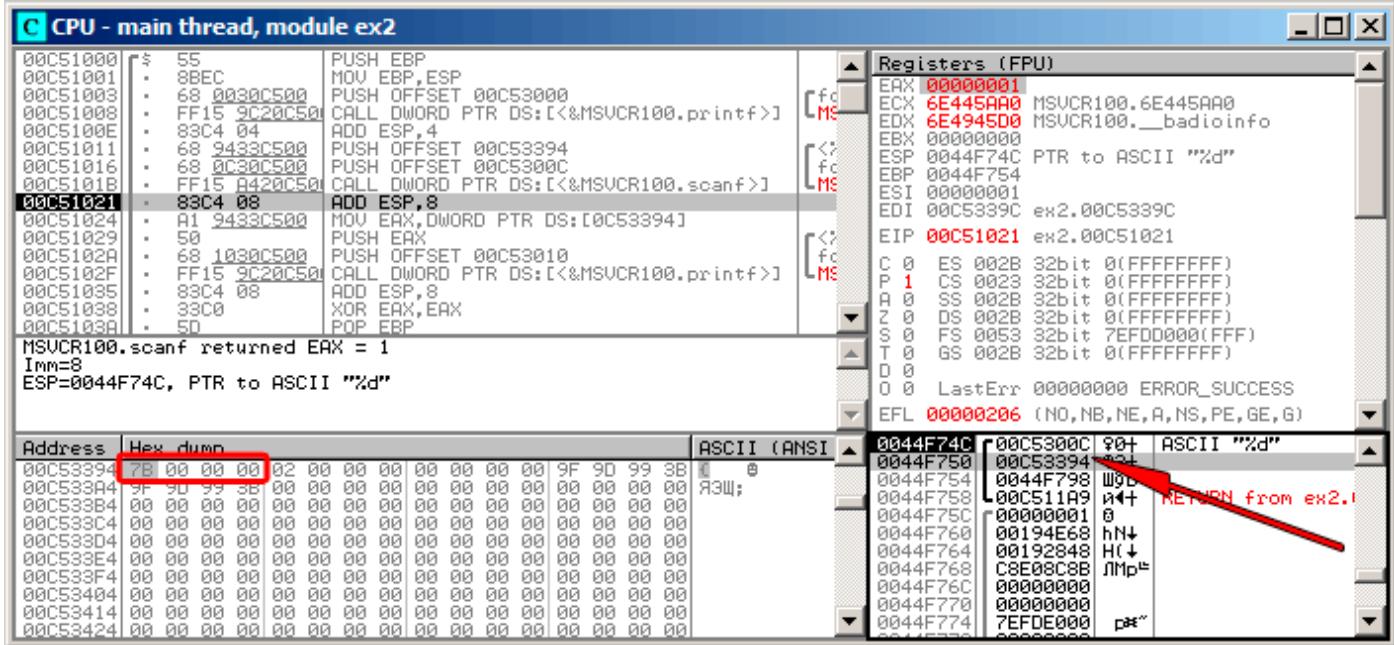


Figure 1.15: OllyDbg: after `scanf()` execution

The variable is located in the data segment. After the `PUSH` instruction (pushing the address of *x*) gets executed, the address appears in the stack window. Right-click on that row and select “Follow in dump”. The variable will appear in the memory window on the left. After we have entered 123 in the console, `0x7B` appears in the memory window (see the highlighted screenshot regions).

But why is the first byte `7B`? Thinking logically, `00 00 00 7B` must be there. The cause for this is referred as [endianness](#), and x86 uses *little-endian*. This implies that the lowest byte is written first, and the highest written last. Read more about it at: [2.7 on page 463](#). Back to the example, the 32-bit value is loaded from this memory address into `EAX` and passed to `printf()`.

The memory address of *x* is `0x00C53394`.

## 1.9. SCANF()

In OllyDbg we can review the process memory map (Alt-M) and we can see that this address is inside the `.data` PE-segment of our program:

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00070000	00067000			Heap	Map	R	R	C:\Windows\System32\locale.nls
00190000	00005000				Priv	RW	RW	
00280000	00007000				Priv	RW	Guar	RW
00440000	00001000			Stack of main thread	Priv	RW	Guar	RW
0044D000	00003000				Priv	RW	Guar	RW
00590000	00007000				Priv	RW	Guar	RW
00750000	0000C000			Default heap	Priv	RW	Guar	RW
00C50000	00001000	ex2		PE header	Img	R	RWE	Cop
00C51000	00001000	ex2	.text	Code	Img	R E	RWE	Cop
00C52000	00001000	ex2	.rdata	Imports	Img	R	RWE	Cop
00C53000	00001000	ex2	.data	Data	Img	RW	RWE	Cop
00C54000	00001000	ex2	.reloc	Relocations	Img	R	RWE	Cop
6E3E0000	00001000	MSVCR100		PE header	Img	R	RWE	Cop
6E3E1000	00062000	MSVCR100	.text	Code, imports, exports	Img	R E	RWE	Cop
6E493000	00006000	MSVCR100	.data	Data	Img	RW	Cop	RWE
6E499000	00001000	MSVCR100	.rsrc	Resources	Img	R	RWE	Cop
6E49A000	00005000	MSVCR100	.reloc	Relocations	Img	R	RWE	Cop
755D0000	00001000	Mod_755D		PE header	Img	R E	RWE	Cop
755D1000	00003000				Img	R	RWE	Cop
755D4000	00001000				Img	RW	RWE	Cop
755D5000	00003000				Img	R	RWE	Cop
755E0000	00001000	Mod_755E		PE header	Img	R	RWE	Cop
755E1000	00040000				Img	R E	RWE	Cop
7562E000	00005000				Img	RW	Cop	RWE
75633000	00009000				Img	R	RWE	Cop
75640000	00001000	Mod_7564		PE header	Img	R	RWE	Cop
75641000	00038000				Img	R E	RWE	Cop
75679000	00002000				Img	RW	RWE	Cop
7567B000	00004000				Img	R	RWE	Cop
76F50000	00010000	kernel32	.text	PE header	Img	R	RWE	Cop
76F60000	00000000	kernel32		Code, imports, exports	Img	R E	RWE	Cop
77030000	00010000	kernel32	.data	Data	Img	RW	Cop	RWE
77040000	00010000	kernel32	.rsrc	Resources	Img	R	RWE	Cop
77050000	0000B000	kernel32	.reloc	Relocations	Img	R	RWE	Cop
77810000	00001000	KERNELBASE		PE header	Img	R	RWE	Cop
77811000	00040000	KERNELBASE	.text	Code, imports, exports	Img	R E	RWE	Cop
77851000	00002000	KERNELBASE	.data	Data	Img	RW	RWE	Cop
77853000	00001000	KERNELBASE	.rsrc	Resources	Img	R	RWE	Cop
77854000	00003000	KERNELBASE	.reloc	Relocations	Img	R	RWE	Cop
77B20000	00001000	Mod_77B2		PE header	Img	R	RWE	Cop
77B21000	00102000				Img	R E	RWE	Cop
77C23000	0002F000				Img	R	RWE	Cop
77C52000	0000C000				Img	RW	Cop	RWE
77C5E000	0006B000				Img	R	RWE	Cop
77D00000	00001000	ntdll	.text	PE header	Img	R	RWE	Cop
77D10000	00006000	ntdll		Code, exports	Img	R E	RWE	Cop
77DF0000	00001000	ntdll	.rdata	Code	Img	R E	RWE	Cop
77E00000	00009000	ntdll	.data	Data	Img	RW	Cop	RWE

Figure 1.16: OllyDbg: process memory map

## GCC: x86

The picture in Linux is near the same, with the difference that the uninitialized variables are located in the `_bss` segment. In [ELF](#) file this segment has the following attributes:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

If you, however, initialize the variable with some value e.g. 10, it is to be placed in the `_data` segment, which has the following attributes:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

## MSVC: x64

Listing 1.73: MSVC 2012 x64

```
_DATA SEGMENT
COMM x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA ENDS
```

## 1.9. SCANF()

```
_TEXT SEGMENT
main PROC
$LN3:
    sub    rsp, 40

    lea    rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call   printf
    lea    rdx, OFFSET FLAT:x
    lea    rcx, OFFSET FLAT:$SG2925 ; '%d'
    call   scanf
    mov    edx, DWORD PTR x
    lea    rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
    call   printf

; return 0
    xor    eax, eax

    add    rsp, 40
    ret    0
main ENDP
_TEXT ENDS
```

The code is almost the same as in x86. Please note that the address of the *x* variable is passed to `scanf()` using a `LEA` instruction, while the variable's value is passed to the second `printf()` using a `MOV` instruction. `DWORD PTR`—is a part of the assembly language (no relation to the machine code), indicating that the variable data size is 32-bit and the `MOV` instruction has to be encoded accordingly.

## ARM: Optimizing Keil 6/2013 (Thumb mode)

Listing 1.74: IDA

```
.text:00000000 ; Segment type: Pure code
.text:00000000     AREA .text, CODE
...
.text:00000000 main
.text:00000000     PUSH   {R4,LR}
.text:00000002     ADR    R0, aEnterX ; "Enter X:\n"
.text:00000004     BL     _2printf
.text:00000008     LDR    R1, =x
.text:0000000A     ADR    R0, aD      ; "%d"
.text:0000000C     BL     _0scanf
.text:00000010     LDR    R0, =x
.text:00000012     LDR    R1, [R0]
.text:00000014     ADR    R0, aYouEnteredD__ ; "You entered %d...\n"
.text:00000016     BL     _2printf
.text:0000001A     MOVS   R0, #0
.text:0000001C     POP    {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A     DCB    0
.text:0000002B     DCB    0
.text:0000002C off_2C DCD x                 ; DATA XREF: main+8
.text:0000002C             ; main+10
.text:00000030 aD      DCB "%d",0          ; DATA XREF: main+A
.text:00000033     DCB    0
.text:00000034 aYouEnteredD__ DCB "You entered %d...",0xA,0 ; DATA XREF: main+14
.text:00000047     DCB    0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048     AREA .data, DATA
.data:00000048     ; ORG 0x48
.data:00000048     EXPORT x
.data:00000048 x      DCD 0xA            ; DATA XREF: main+8
.data:00000048             ; main+10
.data:00000048 ; .data ends
```

## 1.9. SCANF()

So, the `x` variable is now global and for this reason located in another segment, namely the data segment (`.data`). One could ask, why are the text strings located in the code segment (`.text`) and `x` is located right here? Because it is a variable and by definition its value could change. Moreover it could possibly change often. While text strings has constant type, they will not be changed, so they are located in the `.text` segment.

The code segment might sometimes be located in a **ROM**<sup>74</sup> chip (keep in mind, we now deal with embedded microelectronics, and memory scarcity is common here), and changeable variables —in **RAM**.

It is not very economical to store constant variables in RAM when you have ROM.

Furthermore, constant variables in RAM must be initialized, because after powering on, the RAM, obviously, contains random information.

Moving forward, we see a pointer to the `x` (`off_2C`) variable in the code segment, and that all operations with the variable occur via this pointer.

That is because the `x` variable could be located somewhere far from this particular code fragment, so its address must be saved somewhere in close proximity to the code.

The `LDR` instruction in Thumb mode can only address variables in a range of 1020 bytes from its location, and in ARM-mode —variables in range of  $\pm 4095$  bytes.

And so the address of the `x` variable must be located somewhere in close proximity, because there is no guarantee that the linker would be able to accommodate the variable somewhere nearby the code, it may well be even in an external memory chip!

One more thing: if a variable is declared as `const`, the Keil compiler allocates it in the `.constdata` segment.

Perhaps thereafter, the linker could place this segment in ROM too, along with the code segment.

## ARM64

Listing 1.75: Non-optimizing GCC 4.9.1 ARM64

```
1      .comm    x,4,4
2 .LC0:
3     .string "Enter X:"
4 .LC1:
5     .string "%d"
6 .LC2:
7     .string "You entered %d...\n"
8 f5:
9 ; save FP and LR in stack frame:
10    stp      x29, x30, [sp, -16]!
11 ; set stack frame (FP=SP)
12    add      x29, sp, 0
13 ; load pointer to the "Enter X:" string:
14    adrp    x0, .LC0
15    add     x0, x0, :lo12:.LC0
16    bl      puts
17 ; load pointer to the "%d" string:
18    adrp    x0, .LC1
19    add     x0, x0, :lo12:.LC1
20 ; form address of x global variable:
21    adrp    x1, x
22    add     x1, x1, :lo12:x
23    bl      __isoc99_scanf
24 ; form address of x global variable again:
25    adrp    x0, x
26    add     x0, x0, :lo12:x
27 ; load value from memory at this address:
28    ldr     w1, [x0]
29 ; load pointer to the "You entered %d...\n" string:
30    adrp    x0, .LC2
31    add     x0, x0, :lo12:.LC2
32    bl      printf
```

<sup>74</sup>Read-only memory

## 1.9. SCANF()

```
33 ; return 0
34     mov    w0, 0
35 ; restore FP and LR:
36     ldp    x29, x30, [sp], 16
37     ret
```

In this case the *x* variable is declared as global and its address is calculated using the `ADRP / ADD` instruction pair (lines 21 and 25).

## MIPS

### Uninitialized global variable

So now the *x* variable is global. Let's compile to executable file rather than object file and load it into **IDA**. IDA displays the *x* variable in the `.sbss` ELF section (remember the “Global Pointer”? [1.5.5 on page 25](#)), since the variable is not initialized at the start.

Listing 1.76: Optimizing GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10          = -0x10
.text:004006C0 var_4           = -4
.text:004006C0
; function prologue:
.text:004006C0             lui    $gp, 0x42
.text:004006C4             addiu $sp, -0x20
.text:004006C8             li    $gp, 0x418940
.text:004006CC             sw    $ra, 0x20+var_4($sp)
.text:004006D0             sw    $gp, 0x20+var_10($sp)
; call puts():
.text:004006D4             la    $t9, puts
.text:004006D8             lui    $a0, 0x40
.text:004006DC             jalr $t9 ; puts
.text:004006E0             la    $a0, aEnterX      # "Enter X:" ; branch delay slot
; call scanf():
.text:004006E4             lw    $gp, 0x20+var_10($sp)
.text:004006E8             lui    $a0, 0x40
.text:004006EC             la    $t9, __isoc99_scanf
; prepare address of x:
.text:004006F0             la    $a1, x
.text:004006F4             jalr $t9 ; __isoc99_scanf
.text:004006F8             la    $a0, aD          # "%d"           ; branch delay slot
; call printf():
.text:004006FC             lw    $gp, 0x20+var_10($sp)
.text:00400700             lui    $a0, 0x40
; get address of x:
.text:00400704             la    $v0, x
.text:00400708             la    $t9, printf
; load value from "x" variable and pass it to printf() in $a1:
.text:0040070C             lw    $a1, (x - 0x41099C)($v0)
.text:00400710             jalr $t9 ; printf
.text:00400714             la    $a0, aYouEnteredD__ # "You entered %d...\n" ; branch ↴
    ↴ delay slot
; function epilogue:
.text:00400718             lw    $ra, 0x20+var_4($sp)
.text:0040071C             move $v0, $zero
.text:00400720             jr    $ra
.text:00400724             addiu $sp, 0x20 ; branch delay slot

...
.sbss:0041099C # Segment type: Uninitialized
.sbss:0041099C             .sbss
.sbss:0041099C             .globl x
.sbss:0041099C x:           .space 4
.sbss:0041099C
```

## 1.9. SCANF()

IDA reduces the amount of information, so we'll also do a listing using objdump and comment it:

Listing 1.77: Optimizing GCC 4.4.5 (objdump)

```
1 004006c0 <main>:
2 ; function prologue:
3 4006c0: 3c1c0042 lui      gp,0x42
4 4006c4: 27bdffe0 addiu   sp,sp,-32
5 4006c8: 279c8940 addiu   gp,gp,-30400
6 4006cc: afbf001c sw       ra,28(sp)
7 4006d0: afbc0010 sw       gp,16(sp)
8 ; call puts():
9 4006d4: 8f998034 lw       t9,-32716(gp)
10 4006d8: 3c040040 lui     a0,0x40
11 4006dc: 0320f809 jalr    t9
12 4006e0: 248408f0 addiu   a0,a0,2288 ; branch delay slot
13 ; call scanf():
14 4006e4: 8fb0010 lw       gp,16(sp)
15 4006e8: 3c040040 lui     a0,0x40
16 4006ec: 8f998038 lw       t9,-32712(gp)
17 ; prepare address of x:
18 4006f0: 8f858044 lw       a1,-32700(gp)
19 4006f4: 0320f809 jalr    t9
20 4006f8: 248408fc addiu   a0,a0,2300 ; branch delay slot
21 ; call printf():
22 4006fc: 8fb0010 lw       gp,16(sp)
23 400700: 3c040040 lui     a0,0x40
24 ; get address of x:
25 400704: 8f828044 lw       v0,-32700(gp)
26 400708: 8f99803c lw       t9,-32708(gp)
27 ; load value from "x" variable and pass it to printf() in $a1:
28 40070c: 8c450000 lw       a1,0(v0)
29 400710: 0320f809 jalr    t9
30 400714: 24840900 addiu   a0,a0,2304 ; branch delay slot
31 ; function epilogue:
32 400718: 8fb001c lw       ra,28(sp)
33 40071c: 00001021 move   v0,zero
34 400720: 03e00008 jr     ra
35 400724: 27bd0020 addiu   sp,sp,32 ; branch delay slot
36 ; pack of NOPs used for aligning next function start on 16-byte boundary:
37 400728: 00200825 move   at,at
38 40072c: 00200825 move   at,at
```

Now we see the *x* variable address is read from a 64KiB data buffer using GP and adding negative offset to it (line 18). More than that, the addresses of the three external functions which are used in our example (`puts()`, `scanf()`, `printf()`), are also read from the 64KiB global data buffer using GP (lines 9, 16 and 26). GP points to the middle of the buffer, and such offset suggests that all three function's addresses, and also the address of the *x* variable, are all stored somewhere at the beginning of that buffer. That make sense, because our example is tiny.

Another thing worth mentioning is that the function ends with two **NOPs** (`MOVE $AT,$AT` — an idle instruction), in order to align next function's start on 16-byte boundary.

## Initialized global variable

Let's alter our example by giving the *x* variable a default value:

```
int x=10; // default value
```

Now IDA shows that the *x* variable is residing in the `.data` section:

Listing 1.78: Optimizing GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10      = -0x10
.text:004006A0 var_8       = -8
.text:004006A0 var_4       = -4
```

## 1.9. SCANF()

```

.text:004006A0          lui      $gp, 0x42
.text:004006A0          addiu   $sp, -0x20
.text:004006A4          li       $gp, 0x418930
.text:004006AC          sw      $ra, 0x20+var_4($sp)
.text:004006B0          sw      $s0, 0x20+var_8($sp)
.text:004006B4          sw      $gp, 0x20+var_10($sp)
.text:004006B8          la      $t9, puts
.text:004006BC          lui      $a0, 0x40
.text:004006C0          jalr   $t9 ; puts
.text:004006C4          la      $a0, aEnterX    # "Enter X:"
.text:004006C8          lw      $gp, 0x20+var_10($sp)
; prepare high part of x address:
.text:004006CC          lui      $s0, 0x41
.text:004006D0          la      $t9, __isoc99_scanf
.text:004006D4          lui      $a0, 0x40
; add low part of x address:
.text:004006D8          addiu   $a1, $s0, (x - 0x410000)
; now address of x is in $a1.
.text:004006DC          jalr   $t9 ; __isoc99_scanf
.text:004006E0          la      $a0, aD          # "%d"
.text:004006E4          lw      $gp, 0x20+var_10($sp)
; get a word from memory:
.text:004006E8          lw      $a1, x
; value of x is now in $a1.
.text:004006EC          la      $t9, printf
.text:004006F0          lui      $a0, 0x40
.text:004006F4          jalr   $t9 ; printf
.text:004006F8          la      $a0, aYouEnteredD__ # "You entered %d...\n"
.text:004006FC          lw      $ra, 0x20+var_4($sp)
.text:00400700          move   $v0, $zero
.text:00400704          lw      $s0, 0x20+var_8($sp)
.text:00400708          jr      $ra
.text:0040070C          addiu   $sp, 0x20

...
.data:00410920          .globl x
.data:00410920 x:        .word 0xA

```

Why not .sdata? Perhaps that depends on some GCC option?

Nevertheless, now *x* is in .data, which is a general memory area, and we can take a look how to work with variables there.

The variable's address must be formed using a pair of instructions.

In our case those are **LUI** ("Load Upper Immediate") and **ADDIU** ("Add Immediate Unsigned Word").

Here is also the objdump listing for close inspection:

Listing 1.79: Optimizing GCC 4.4.5 (objdump)

```

004006a0 <main>:
 4006a0: 3c1c0042    lui    gp,0x42
 4006a4: 27bdffe0    addiu   sp,sp,-32
 4006a8: 279c8930    addiu   gp,sp,-30416
 4006ac: afbf001c    sw     ra,28(sp)
 4006b0: afb00018    sw     s0,24(sp)
 4006b4: afbc0010    sw     gp,16(sp)
 4006b8: 8f998034    lw     t9,-32716(gp)
 4006bc: 3c040040    lui    a0,0x40
 4006c0: 0320f809    jalr   t9
 4006c4: 248408d0    addiu  a0,a0,2256
 4006c8: 8fbcc0010   lw     gp,16(sp)
; prepare high part of x address:
 4006cc: 3c100041    lui    s0,0x41
 4006d0: 8f998038    lw     t9,-32712(gp)
 4006d4: 3c040040    lui    a0,0x40
; add low part of x address:
 4006d8: 26050920    addiu  a1,s0,2336
; now address of x is in $a1.

```

## 1.9. SCANF()

```
4006dc: 0320f809      jalr    t9
4006e0: 248408dc      addiu   a0,a0,2268
4006e4: 8fb00010      lw       gp,16(sp)
; high part of x address is still in $s0.
; add low part to it and load a word from memory:
4006e8: 8e050920      lw       a1,2336($0)
; value of x is now in $a1.
4006ec: 8f99803c      lw       t9,-32708(gp)
4006f0: 3c040040      lui     a0,0x40
4006f4: 0320f809      jalr    t9
4006f8: 248408e0      addiu   a0,a0,2272
4006fc: 8fb0001c      lw       ra,28(sp)
400700: 00001021      move    v0,zero
400704: 8fb00018      lw       s0,24(sp)
400708: 03e00008      jr     ra
40070c: 27bd0020      addiu   sp,sp,32
```

We see that the address is formed using **LUI** and **ADDIU**, but the high part of address is still in the **\$S0** register, and it is possible to encode the offset in a **LW** ("Load Word") instruction, so one single **LW** is enough to load a value from the variable and pass it to **printf()**.

Registers holding temporary data are prefixed with T-, but here we also see some prefixed with S-, the contents of which must be preserved before use in other functions (i.e., saved somewhere).

That is why the value of **\$S0** has been set at address 0x4006cc and has been used again at address 0x4006e8, after the **scanf()** call. The **scanf()** function does not change its value.

### 1.9.4 scanf()

As was noted before, it is slightly old-fashioned to use **scanf()** today. But if we have to, we have to check if **scanf()** finishes correctly without an error.

```
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    if (scanf ("%d", &x)==1)
        printf ("You entered %d...\n", x);
    else
        printf ("What you entered? Huh?\n");

    return 0;
}
```

By standard, the **scanf()**<sup>75</sup> function returns the number of fields it has successfully read.

In our case, if everything goes fine and the user enters a number **scanf()** returns 1, or in case of error (or **EOF**<sup>76</sup>) — 0.

Let's add some C code to check the **scanf()** return value and print error message in case of an error.

This works as expected:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...
```

```
C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

<sup>75</sup>scanf, wscanf: [MSDN](#)

<sup>76</sup>End of file

Here is what we get in the assembly output (MSVC 2010):

```

    lea      eax, DWORD PTR _x$[ebp]
    push     eax
    push     OFFSET $SG3833 ; '%d', 00H
    call     _scanf
    add     esp, 8
    cmp     eax, 1
    jne     SHORT $LN2@main
    mov     ecx, DWORD PTR _x$[ebp]
    push     ecx
    push     OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
    call     _printf
    add     esp, 8
    jmp     SHORT $LN1@main
$LN2@main:
    push     OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
    call     _printf
    add     esp, 4
$LN1@main:
    xor     eax, eax

```

The **caller** function (`main()`) needs the **callee** function (`scanf()`) result, so the **callee** returns it in the **EAX** register.

We check it with the help of the instruction `CMP EAX, 1` (*CoMPare*). In other words, we compare the value in the **EAX** register with 1.

A `JNE` conditional jump follows the `CMP` instruction. `JNE` stands for *Jump if Not Equal*.

So, if the value in the **EAX** register is not equal to 1, the **CPU** will pass the execution to the address mentioned in the `JNE` operand, in our case `$LN2@main`. Passing the control to this address results in the **CPU** executing `printf()` with the argument `What you entered? Huh?`. But if everything is fine, the conditional jump is not be taken, and another `printf()` call is to be executed, with two arguments: `'You entered %d...'` and the value of `x`.

Since in this case the second `printf()` has not to be executed, there is a `JMP` preceding it (unconditional jump). It passes the control to the point after the second `printf()` and just before the `XOR EAX, EAX` instruction, which implements `return 0`.

So, it could be said that comparing a value with another is *usually* implemented by `CMP / Jcc` instruction pair, where *cc* is *condition code*. `CMP` compares two values and sets processor flags<sup>77</sup>. `Jcc` checks those flags and decides to either pass the control to the specified address or not.

This could sound paradoxical, but the `CMP` instruction is in fact `SUB` (subtract). All arithmetic instructions set processor flags, not just `CMP`. If we compare 1 and 1,  $1 - 1$  is 0 so the `ZF` flag would be set (meaning that the last result is 0). In no other circumstances `ZF` can be set, except when the operands are equal. `JNE` checks only the `ZF` flag and jumps only if it is not set. `JNE` is in fact a synonym for `JNZ` (*Jump if Not Zero*). Assembler translates both `JNE` and `JNZ` instructions into the same opcode. So, the `CMP` instruction can be replaced with a `SUB` instruction and almost everything will be fine, with the difference that `SUB` alters the value of the first operand. `CMP` is *SUB without saving the result, but affecting flags*.

## MSVC: x86: IDA

It is time to run **IDA** and try to do something in it. By the way, for beginners it is good idea to use `/MD` option in MSVC, which means that all these standard functions are not be linked with the executable file, but are to be imported from the `MSVCR*.DLL` file instead. Thus it will be easier to see which standard function are used and where.

<sup>77</sup>x86 flags, see also: [wikipedia](#).

## 1.9. SCANF()

While analyzing code in [IDA](#), it is very helpful to leave notes for oneself (and others). In instance, analyzing this example, we see that **JNZ** is to be triggered in case of an error. So it is possible to move the cursor to the label, press “n” and rename it to “error”. Create another label—into “exit”. Here is my result:

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr 8
.text:00401000 argv  = dword ptr 0Ch
.text:00401000 envp  = dword ptr 10h
.text:00401000
.text:00401000     push    ebp
.text:00401001     mov     ebp, esp
.text:00401003     push    ecx
.text:00401004     push    offset Format ; "Enter X:\n"
.text:00401009     call    ds:printf
.text:0040100F     add    esp, 4
.text:00401012     lea    eax, [ebp+var_4]
.text:00401015     push    eax
.text:00401016     push    offset aD ; "%d"
.text:0040101B     call    ds:scanf
.text:00401021     add    esp, 8
.text:00401024     cmp    eax, 1
.text:00401027     jnz    short error
.text:00401029     mov    ecx, [ebp+var_4]
.text:0040102C     push    ecx
.text:0040102D     push    offset aYou ; "You entered %d...\n"
.text:00401032     call    ds:printf
.text:00401038     add    esp, 8
.text:0040103B     jmp    short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D     push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042     call    ds:printf
.text:00401048     add    esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B     xor    eax, eax
.text:0040104D     mov    esp, ebp
.text:0040104F     pop    ebp
.text:00401050     retn
.text:00401050 _main endp
```

Now it is slightly easier to understand the code. However, it is not a good idea to comment on every instruction.

You could also hide(collapse) parts of a function in [IDA](#). To do that mark the block, then press “-” on the numerical pad and enter the text to be displayed instead.

Let's hide two blocks and give them names:

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000     assume cs:_text
.text:00401000     ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024     cmp    eax, 1
.text:00401027     jnz    short error
.text:00401029 ; print result
.text:0040103B     jmp    short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D     push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042     call    ds:printf
.text:00401048     add    esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B     xor    eax, eax
.text:0040104D     mov    esp, ebp
.text:0040104F     pop    ebp
.text:00401050     retn
```

### 1.9. *SCANF()*

```
.text:00401050 _main endp
```

To expand previously collapsed parts of the code, use “+” on the numerical pad.

## 1.9. SCANF()

By pressing "space", we can see how IDA represents a function as a graph:

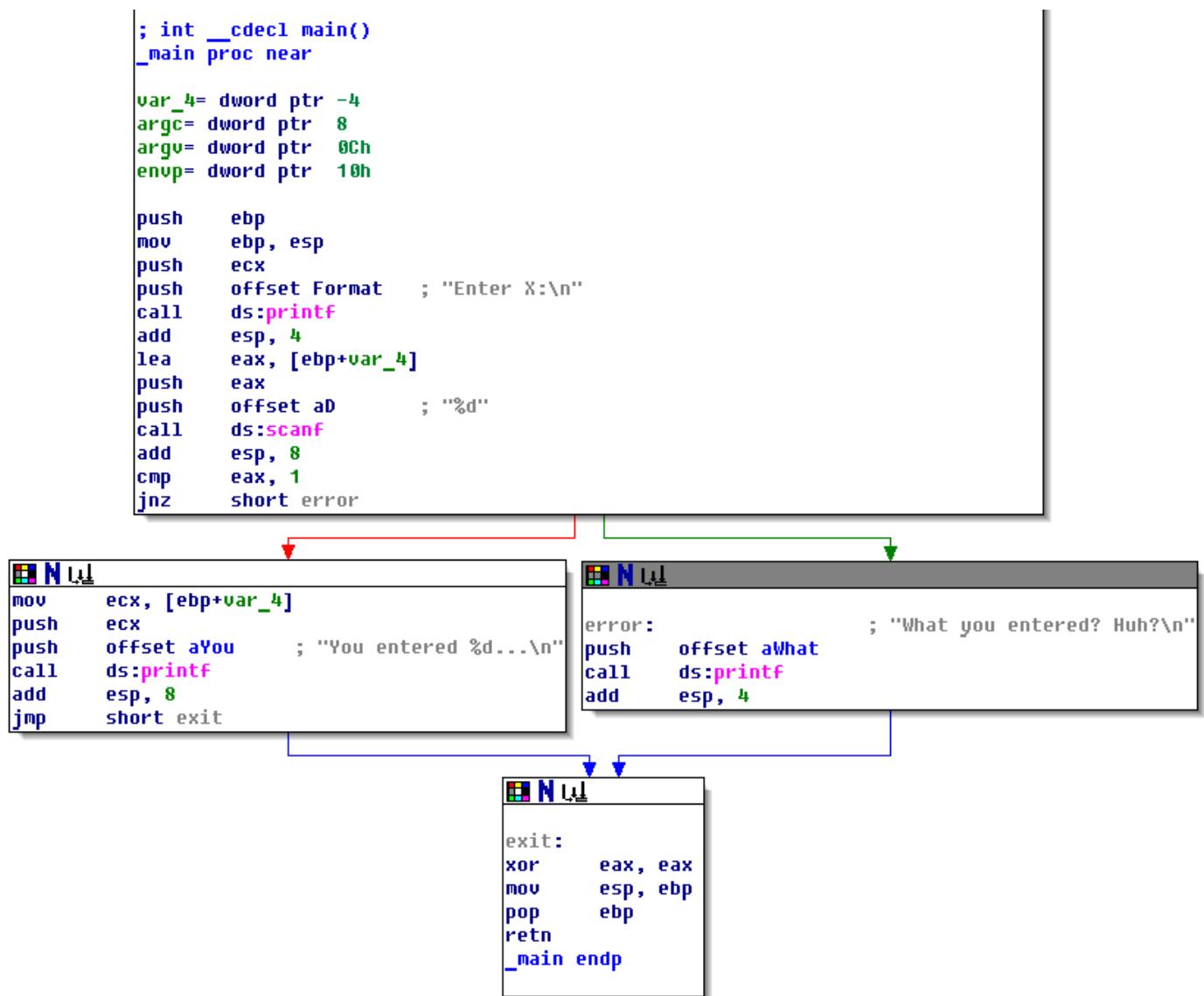


Figure 1.17: Graph mode in IDA

There are two arrows after each conditional jump: green and red. The green arrow points to the block which executes if the jump is triggered, and red if otherwise.

## 1.9. SCANF()

It is possible to fold nodes in this mode and give them names as well ("group nodes"). Let's do it for 3 blocks:

```
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
push    ecx
push    offset Format    ; "Enter X:\n"
call    ds:printf
add    esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD          ; "%d"
call    ds:scanf
add    esp, 8
cmp    eax, 1
jnz    short error
```

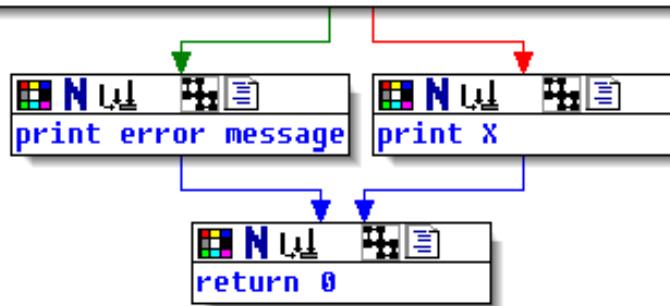


Figure 1.18: Graph mode in IDA with 3 nodes folded

That is very useful. It could be said that a very important part of the reverse engineers' job (and any other researcher as well) is to reduce the amount of information they deal with.

## 1.9. SCANF()

### MSVC: x86 + OllyDbg

Let's try to hack our program in OllyDbg, forcing it to think `scanf()` always works without error. When an address of a local variable is passed into `scanf()`, the variable initially contains some random garbage, in this case `0x6E494714`:

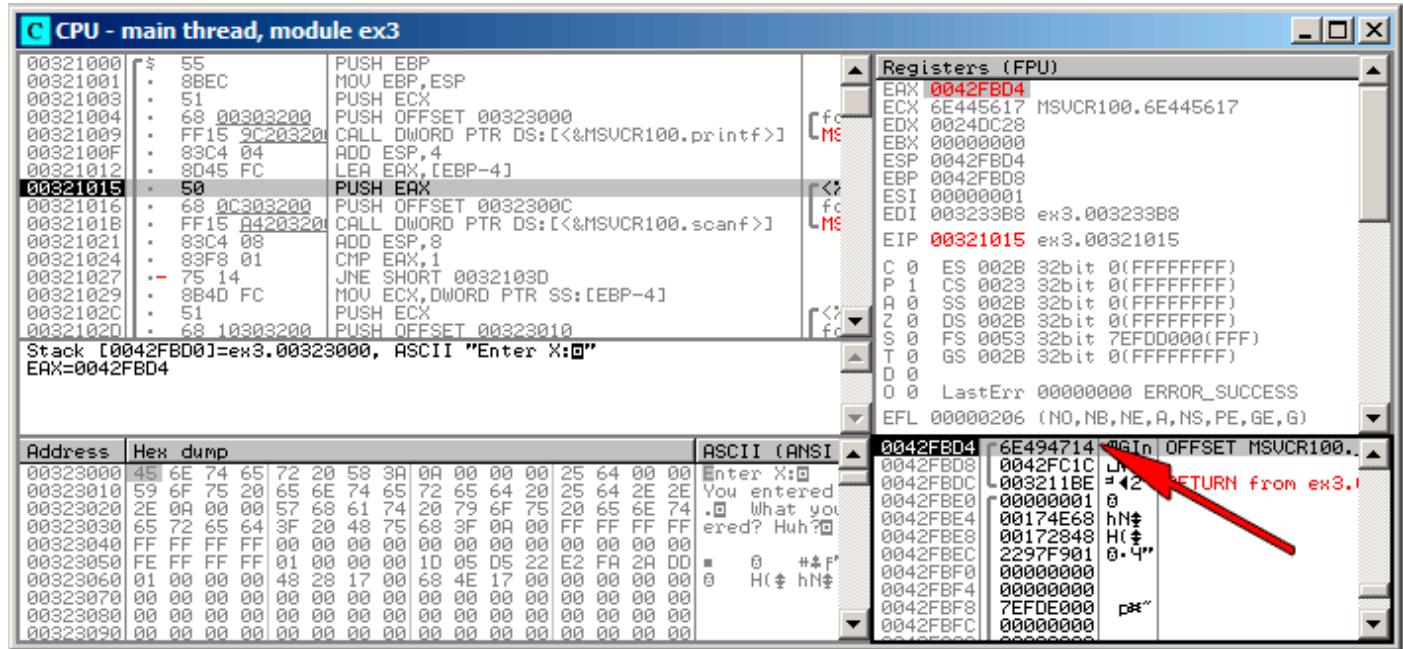


Figure 1.19: OllyDbg: passing variable address into `scanf()`

## 1.9. SCANF()

While `scanf()` executes, in the console we enter something that is definitely not a number, like "asdasd". `scanf()` finishes with 0 in `EAX`, which indicates that an error has occurred:

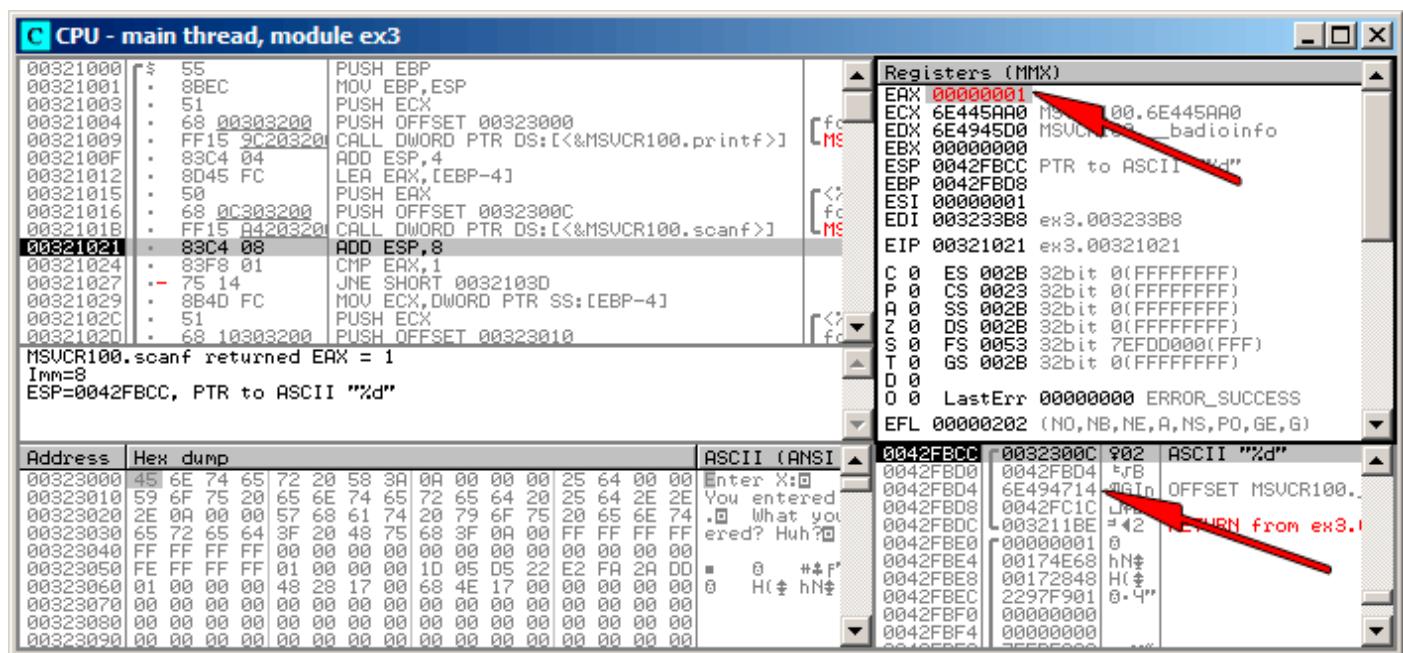


Figure 1.20: OllyDbg: `scanf()` returning error

We can also check the local variable in the stack and note that it has not changed. Indeed, what would `scanf()` write there? It simply did nothing except returning zero.

Let's try to "hack" our program. Right-click on `EAX`, Among the options there is "Set to 1". This is what we need.

We now have 1 in `EAX`, so the following check is to be executed as intended, and `printf()` will print the value of the variable in the stack.

When we run the program (F9) we can see the following in the console window:

Listing 1.80: console window

```
Enter X:  
asdasd  
You entered 1850296084...
```

Indeed, 1850296084 is a decimal representation of the number in the stack (`0x6E494714`)!

## 1.9. *SCANF()*

---

**MSVC: x86 + Hiew**

This can also be used as a simple example of executable file patching. We may try to patch the executable so the program would always print the input, no matter what we enter.

Assuming that the executable is compiled against external `MSVCR*.DLL` (i.e., with `/MD` option) <sup>78</sup>, we see the `main()` function at the beginning of the `.text` section. Let's open the executable in Hiew and find the beginning of the `.text` section (Enter, F8, F6, Enter, Enter).

We can see this:

Figure 1.21: Hiew: main() function

Hiew finds ASCII<sup>79</sup> strings and displays them, as it does with the imported functions' names.

<sup>78</sup>that's what also called "dynamic linking"

79 ASCII Zero (null-terminated ASCII string )

### 1.9. *SCANF()*

Move the cursor to address .00401027 (where the JNZ instruction, we have to bypass, is located), press F3, and then type “9090” (meaning two NOPs):

Figure 1.22: Hiew: replacing `JNZ` with two `NOPs`

Then press F9 (update). Now the executable is saved to the disk. It will behave as we wanted.

Two [NOPs](#) are probably not the most æsthetic approach. Another way to patch this instruction is to write just 0 to the second opcode byte ([jump offset](#)), so that [JNZ](#) will always jump to the next instruction.

We could also do the opposite: replace first byte with `EB` while not touching the second byte ([jump offset](#)). We would get an unconditional jump that is always triggered. In this case the error message would be printed every time, no matter the input.

MSVC: x64

MSVC: x64

Since we work here with *int*-typed variables, which are still 32-bit in x86-64, we see how the 32-bit part of the registers (prefixed with E- ) are used here as well. While working with pointers, however, 64-bit register parts are used, prefixed with R- .

## 1.9. SCANF()

Listing 1.81: MSVC 2012 x64

```
_DATA SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
x$ = 32
main PROC
$LN5:
    sub    rsp, 56
    lea    rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call   printf
    lea    rdx, QWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG2926 ; '%d'
    call   scanf
    cmp    eax, 1
    jne    SHORT $LN2@main
    mov    edx, DWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
    call   printf
    jmp    SHORT $LN1@main
$LN2@main:
    lea    rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
$LN1@main:
    ; return 0
    xor    eax, eax
    add    rsp, 56
    ret    0
main ENDP
_TEXT ENDS
END
```

## ARM

### ARM: Optimizing Keil 6/2013 (Thumb mode)

Listing 1.82: Optimizing Keil 6/2013 (Thumb mode)

```
var_8 = -8

        PUSH {R3,LR}
        ADR R0, aEnterX    ; "Enter X:\n"
        BL __2printf
        MOV R1, SP
        ADR R0, aD          ; "%d"
        BL __0scanf
        CMP R0, #1
        BEQ loc_1E
        ADR R0, aWhatYouEntered ; "What you entered? Huh?\n"
        BL __2printf

loc_1A                         ; CODE XREF: main+26
        MOVS R0, #0
        POP {R3,PC}

loc_1E                         ; CODE XREF: main+12
        LDR R1, [SP,#8+var_8]
        ADR R0, aYouEnteredDD__ ; "You entered %d...\n"
        BL __2printf
        B loc_1A
```

## 1.9. SCANF()

The new instructions here are **CMP** and **BEQ**<sup>80</sup>.

**CMP** is analogous to the x86 instruction with the same name, it subtracts one of the arguments from the other and updates the conditional flags if needed.

**BEQ** jumps to another address if the operands were equal to each other, or, if the result of the last computation has been 0, or if the Z flag is 1. It behaves as **JZ** in x86.

Everything else is simple: the execution flow forks in two branches, then the branches converge at the point where 0 is written into the **R0** as a function return value, and then the function ends.

## ARM64

Listing 1.83: Non-optimizing GCC 4.9.1 ARM64

```
1 .LC0:
2     .string "Enter X:"
3 .LC1:
4     .string "%d"
5 .LC2:
6     .string "You entered %d...\n"
7 .LC3:
8     .string "What you entered? Huh?"
9 f6:
10 ; save FP and LR in stack frame:
11     stp    x29, x30, [sp, -32]!
12 ; set stack frame (FP=SP)
13     add    x29, sp, 0
14 ; load pointer to the "Enter X:" string:
15     adrp   x0, .LC0
16     add    x0, x0, :lo12:.LC0
17     bl     puts
18 ; load pointer to the "%d" string:
19     adrp   x0, .LC1
20     add    x0, x0, :lo12:.LC1
21 ; calculate address of x variable in the local stack
22     add    x1, x29, 28
23     bl     __isoc99_scanf
24 ; scanf() returned result in W0.
25 ; check it:
26     cmp    w0, 1
27 ; BNE is Branch if Not Equal
28 ; so if W0<>0, jump to L2 will be occurred
29     bne   .L2
30 ; at this moment W0=1, meaning no error
31 ; load x value from the local stack
32     ldr    w1, [x29,28]
33 ; load pointer to the "You entered %d...\n" string:
34     adrp   x0, .LC2
35     add    x0, x0, :lo12:.LC2
36     bl     printf
37 ; skip the code, which print the "What you entered? Huh?" string:
38     b     .L3
39 .L2:
40 ; load pointer to the "What you entered? Huh?" string:
41     adrp   x0, .LC3
42     add    x0, x0, :lo12:.LC3
43     bl     puts
44 .L3:
45 ; return 0
46     mov    w0, 0
47 ; restore FP and LR:
48     ldp    x29, x30, [sp], 32
49     ret
```

Code flow in this case forks with the use of **CMP / BNE** (Branch if Not Equal) instructions pair.

<sup>80</sup>(PowerPC, ARM) Branch if Equal

Listing 1.84: Optimizing GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18      = -0x18
.text:004006A0 var_10      = -0x10
.text:004006A0 var_4       = -4
.text:004006A0
.text:004006A0     lui      $gp, 0x42
.text:004006A4     addiu   $sp, -0x28
.text:004006A8     li      $gp, 0x418960
.text:004006AC     sw      $ra, 0x28+var_4($sp)
.text:004006B0     sw      $gp, 0x28+var_18($sp)
.text:004006B4     la      $t9, puts
.text:004006B8     lui      $a0, 0x40
.text:004006BC     jalr    $t9 ; puts
.text:004006C0     la      $a0, aEnterX      # "Enter X:"
.text:004006C4     lw      $gp, 0x28+var_18($sp)
.text:004006C8     lui      $a0, 0x40
.text:004006CC     la      $t9, __isoc99_scanf
.text:004006D0     la      $a0, aD          # "%d"
.text:004006D4     jalr    $t9 ; __isoc99_scanf
.text:004006D8     addiu   $a1, $sp, 0x28+var_10  # branch delay slot
.text:004006DC     li      $v1, 1
.text:004006E0     lw      $gp, 0x28+var_18($sp)
.text:004006E4     beq    $v0, $v1, loc_40070C
.text:004006E8     or      $at, $zero        # branch delay slot, NOP
.text:004006EC     la      $t9, puts
.text:004006F0     lui      $a0, 0x40
.text:004006F4     jalr    $t9 ; puts
.text:004006F8     la      $a0, aWhatYouEntered  # "What you entered? Huh?"
.text:004006FC     lw      $ra, 0x28+var_4($sp)
.text:00400700     move    $v0, $zero
.text:00400704     jr      $ra
.text:00400708     addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C     la      $t9, printf
.text:00400710     lw      $a1, 0x28+var_10($sp)
.text:00400714     lui      $a0, 0x40
.text:00400718     jalr    $t9 ; printf
.text:0040071C     la      $a0, aYouEnteredD__  # "You entered %d...\n"
.text:00400720     lw      $ra, 0x28+var_4($sp)
.text:00400724     move    $v0, $zero
.text:00400728     jr      $ra
.text:0040072C     addiu   $sp, 0x28
```

`scanf()` returns the result of its work in register `$V0`. It is checked at address `0x004006E4` by comparing the values in `$V0` with `$V1` (`1` has been stored in `$V1` earlier, at `0x004006DC`). `BEQ` stands for “Branch Equal”. If the two values are equal (i.e., success), the execution jumps to address `0x0040070C`.

## Exercise

As we can see, the `JNE / JNZ` instruction can be easily replaced by the `JE / JZ` and vice versa (or `BNE` by `BEQ` and vice versa). But then the basic blocks must also be swapped. Try to do this in some of the examples.

### 1.9.5 Exercise

- <http://challenges.re/53>

## 1.10 Accessing passed arguments

Now we figured out that the `caller` function is passing arguments to the `callee` via the stack. But how does the `callee` access them?

Listing 1.85: simple example

```
#include <stdio.h>

int f (int a, int b, int c)
{
    return a*b+c;
}

int main()
{
    printf ("%d\n", f(1, 2, 3));
    return 0;
}
```

### 1.10.1 x86

#### MSVC

Here is what we get after compilation (MSVC 2010 Express):

Listing 1.86: MSVC 2010 Express

```
_TEXT SEGMENT
_a$ = 8      ; size = 4
_b$ = 12     ; size = 4
_c$ = 16     ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    imul   eax, DWORD PTR _b$[ebp]
    add    eax, DWORD PTR _c$[ebp]
    pop    ebp
    ret    0
_f ENDP

_main PROC
    push   ebp
    mov    ebp, esp
    push  3 ; 3rd argument
    push  2 ; 2nd argument
    push  1 ; 1st argument
    call   _f
    add    esp, 12
    push   eax
    push  OFFSET $SG2463 ; '%d', 0aH, 00H
    call   _printf
    add    esp, 8
    ; return 0
    xor    eax, eax
    pop    ebp
    ret    0
_main ENDP
```

What we see is that the `main()` function pushes 3 numbers onto the stack and calls `f(int,int,int)`.

Argument access inside `f()` is organized with the help of macros like:

`_a$ = 8`, in the same way as local variables, but with positive offsets (addressed with *plus*). So, we are addressing the *outer* side of the `stack frame` by adding the `_a$` macro to the value in the `EBP` register.

## 1.10. ACCESSING PASSED ARGUMENTS

Then the value of `a` is stored into `EAX`. After `IMUL` instruction execution, the value in `EAX` is a product of the value in `EAX` and the content of `_b`.

After that, `ADD` adds the value in `_c` to `EAX`.

The value in `EAX` does not need to be moved: it is already where it must be. On returning to `caller`, it takes the `EAX` value and use it as an argument to `printf()`.

## MSVC + OllyDbg

Let's illustrate this in OllyDbg. When we trace to the first instruction in `f()` that uses one of the arguments (first one), we see that `EBP` is pointing to the `stack frame`, which is marked with a red rectangle.

The first element of the `stack frame` is the saved value of `EBP`, the second one is `RA`, the third is the first function argument, then the second and third ones.

To access the first function argument, one needs to add exactly 8 (2 32-bit words) to `EBP`.

OllyDbg is aware about this, so it has added comments to the stack elements like

"RETURN from" and "Arg1 = ...", etc.

N.B.: Function arguments are not members of the function's stack frame, they are rather members of the stack frame of the `caller` function.

Hence, OllyDbg marked "Arg" elements as members of another stack frame.

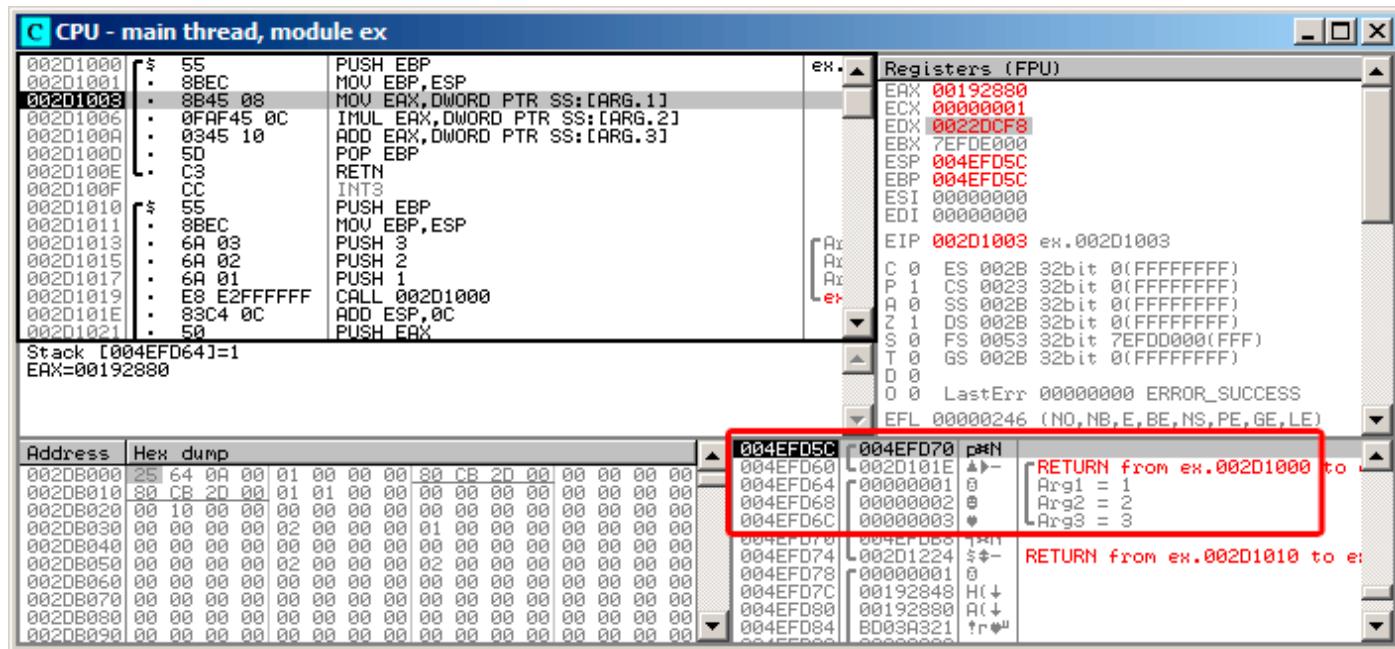


Figure 1.23: OllyDbg: inside of `f()` function

## GCC

Let's compile the same in GCC 4.4.1 and see the results in IDA:

Listing 1.87: GCC 4.4.1

```

public f
f proc near

arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h

push    ebp

```

## 1.10. ACCESSING PASSED ARGUMENTS

```
    mov    ebp, esp
    mov    eax, [ebp+arg_0] ; 1st argument
    imul   eax, [ebp+arg_4] ; 2nd argument
    add    eax, [ebp+arg_8] ; 3rd argument
    pop    ebp
    retn
f      endp

        public main
main   proc near

var_10 = dword ptr -10h
var_C  = dword ptr -0Ch
var_8  = dword ptr -8

    push   ebp
    mov    ebp, esp
    and    esp, 0FFFFFFF0h
    sub    esp, 10h
    mov    [esp+10h+var_8], 3 ; 3rd argument
    mov    [esp+10h+var_C], 2 ; 2nd argument
    mov    [esp+10h+var_10], 1 ; 1st argument
    call   f
    mov    edx, offset aD ; "%d\n"
    mov    [esp+10h+var_C], eax
    mov    [esp+10h+var_10], edx
    call   _printf
    mov    eax, 0
    leave
    retn
main   endp
```

The result is almost the same with some minor differences discussed earlier.

The [stack pointer](#) is not set back after the two function calls(f and printf), because the penultimate [LEAVE](#) (.1.6 on page 1011) instruction takes care of this at the end.

## 1.10.2 x64

The story is a bit different in x86-64. Function arguments (first 4 or first 6 of them) are passed in registers i.e. the [callee](#) reads them from registers instead of reading them from the stack.

### MSVC

Optimizing MSVC:

Listing 1.88: Optimizing MSVC 2012 x64

```
$SG2997 DB      '%d', 0aH, 00H

main  PROC
    sub    rsp, 40
    mov    edx, 2
    lea    r8d, QWORD PTR [rdx+1] ; R8D=3
    lea    ecx, QWORD PTR [rdx-1] ; ECX=1
    call   f
    lea    rcx, OFFSET FLAT:$SG2997 ; '%d'
    mov    edx, eax
    call   printf
    xor    eax, eax
    add    rsp, 40
    ret    0
main  ENDP

f      PROC
; ECX - 1st argument
; EDX - 2nd argument
```

## 1.10. ACCESSING PASSED ARGUMENTS

```
; R8D - 3rd argument  
imul    ecx, edx  
lea     eax, DWORD PTR [r8+rcx]  
ret    0  
f      ENDP
```

As we can see, the compact function `f()` takes all its arguments from the registers.

The `LEA` instruction here is used for addition, apparently the compiler considered it faster than `ADD`.

`LEA` is also used in the `main()` function to prepare the first and third `f()` arguments. The compiler must have decided that this would work faster than the usual way of loading values into a register using `MOV` instruction.

Let's take a look at the non-optimizing MSVC output:

Listing 1.89: MSVC 2012 x64

```
f          proc near  
  
; shadow space:  
arg_0      = dword ptr  8  
arg_8      = dword ptr  10h  
arg_10     = dword ptr  18h  
  
        ; ECX - 1st argument  
        ; EDX - 2nd argument  
        ; R8D - 3rd argument  
mov        [rsp+arg_10], r8d  
mov        [rsp+arg_8], edx  
mov        [rsp+arg_0], ecx  
mov        eax, [rsp+arg_0]  
imul      eax, [rsp+arg_8]  
add       eax, [rsp+arg_10]  
retn  
f          endp  
  
main      proc near  
sub        rsp, 28h  
mov        r8d, 3 ; 3rd argument  
mov        edx, 2 ; 2nd argument  
mov        ecx, 1 ; 1st argument  
call       f  
mov        edx, eax  
lea         rcx, $SG2931    ; "%d\n"  
call       printf  
  
        ; return 0  
xor        eax, eax  
add        rsp, 28h  
retn  
main      endp
```

It looks somewhat puzzling because all 3 arguments from the registers are saved to the stack for some reason. This is called “shadow space”<sup>81</sup>: every Win64 may (but is not required to) save all 4 register values there. This is done for two reasons: 1) it is too lavish to allocate a whole register (or even 4 registers) for an input argument, so it will be accessed via stack; 2) the debugger is always aware where to find the function arguments at a break<sup>82</sup>.

So, some large functions can save their input arguments in the “shadows space” if they want to use them during execution, but some small functions (like ours) may not do this.

It is a **caller** responsibility to allocate “shadow space” in the stack.

<sup>81</sup>[MSDN](#)

<sup>82</sup>[MSDN](#)

Optimizing GCC generates more or less understandable code:

Listing 1.90: Optimizing GCC 4.4.6 x64

```
f:
; EDI - 1st argument
; ESI - 2nd argument
; EDX - 3rd argument
imul    esi, edi
lea     eax, [rdx+rsi]
ret

main:
sub    rsp, 8
mov   edx, 3
mov   esi, 2
mov   edi, 1
call  f
mov   edi, OFFSET FLAT:.LC0 ; "%d\n"
mov   esi, eax
xor   eax, eax ; number of vector registers passed
call  printf
xor   eax, eax
add   rsp, 8
ret
```

Non-optimizing GCC:

Listing 1.91: GCC 4.4.6 x64

```
f:
; EDI - 1st argument
; ESI - 2nd argument
; EDX - 3rd argument
push  rbp
mov   rbp, rsp
mov   DWORD PTR [rbp-4], edi
mov   DWORD PTR [rbp-8], esi
mov   DWORD PTR [rbp-12], edx
mov   eax, DWORD PTR [rbp-4]
imul  eax, DWORD PTR [rbp-8]
add   eax, DWORD PTR [rbp-12]
leave
ret

main:
push  rbp
mov   rbp, rsp
mov   edx, 3
mov   esi, 2
mov   edi, 1
call  f
mov   edx, eax
mov   eax, OFFSET FLAT:.LC0 ; "%d\n"
mov   esi, edx
mov   rdi, rax
mov   eax, 0 ; number of vector registers passed
call  printf
mov   eax, 0
leave
ret
```

There are no “shadow space” requirements in System V \*NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>83</sup>), but the callee may want to save its arguments somewhere in case of registers shortage.

<sup>83</sup>Also available as <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

## 1.10. ACCESSING PASSED ARGUMENTS

### GCC: `uint64_t` instead of `int`

Our example works with 32-bit `int`, that is why 32-bit register parts are used (prefixed by `E-`).

It can be altered slightly in order to use 64-bit values:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
}

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                         0x1111111222222222,
                         0x3333333444444444));
    return 0;
}
```

Listing 1.92: Optimizing GCC 4.4.6 x64

```
f      proc near
    imul    rsi, rdi
    lea     rax, [rdx+rsi]
    retn
f      endp

main   proc near
    sub    rsp, 8
    mov    rdx, 333333344444444h ; 3rd argument
    mov    rsi, 111111122222222h ; 2nd argument
    mov    rdi, 1122334455667788h ; 1st argument
    call   f
    mov    edi, offset format ; "%lld\n"
    mov    rsi, rax
    xor    eax, eax ; number of vector registers passed
    call   _printf
    xor    eax, eax
    add    rsp, 8
    retn
main   endp
```

The code is the same, but this time the *full size* registers (prefixed by `R-`) are used.

## 1.10.3 ARM

### Non-optimizing Keil 6/2013 (ARM mode)

```
.text:000000A4 00 30 A0 E1      MOV      R3, R0
.text:000000A8 93 21 20 E0      MLA      R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX       LR
...
.text:000000B0          main
.text:000000B0 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV      R2, #3
.text:000000B8 02 10 A0 E3      MOV      R1, #2
.text:000000BC 01 00 A0 E3      MOV      R0, #1
.text:000000C0 F7 FF FF EB      BL       f
.text:000000C4 00 40 A0 E1      MOV      R4, R0
.text:000000C8 04 10 A0 E1      MOV      R1, R4
.text:000000CC 5A 0F 8F E2      ADR     R0, aD_0      ; "%d\n"
.text:000000D0 E3 18 00 EB      BL       __2printf
.text:000000D4 00 00 A0 E3      MOV      R0, #0
.text:000000D8 10 80 BD E8      LDMFD  SP!, {R4,PC}
```

## 1.10. ACCESSING PASSED ARGUMENTS

The `main()` function simply calls two other functions, with three values passed to the first one —(`f()`).

As was noted before, in ARM the first 4 values are usually passed in the first 4 registers (`R0 - R3`).

The `f()` function, as it seems, uses the first 3 registers (`R0 - R2`) as arguments.

The `MLA` (*Multiply Accumulate*) instruction multiplies its first two operands (`R3` and `R1`), adds the third operand (`R2`) to the product and stores the result into the zeroth register (`R0`), via which, by standard, functions return values.

Multiplication and addition at once<sup>84</sup> (*Fused multiply-add*) is a very useful operation. By the way, there was no such instruction in x86 before FMA-instructions appeared in SIMD<sup>85</sup>.

The very first `MOV R3, R0`, instruction is, apparently, redundant (a single `MLA` instruction could be used here instead). The compiler has not optimized it, since this is non-optimizing compilation.

The `BX` instruction returns the control to the address stored in the `LR` register and, if necessary, switches the processor mode from Thumb to ARM or vice versa. This can be necessary since, as we can see, function `f()` is not aware from what kind of code it may be called, ARM or Thumb. Thus, if it gets called from Thumb code, `BX` is not only returns control to the calling function, but also switches the processor mode to Thumb. Or not switch, if the function has been called from ARM code [ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition, (2012)A2.3.2].

### **Optimizing Keil 6/2013 (ARM mode)**

```
.text:00000098          f
.text:00000098 91 20 20 E0      MLA      R0, R1, R0, R2
.text:0000009C 1E FF 2F E1      BX       LR
```

And here is the `f()` function compiled by the Keil compiler in full optimization mode (-O3).

The `MOV` instruction was optimized out (or reduced) and now `MLA` uses all input registers and also places the result right into `R0`, exactly where the calling function will read and use it.

### **Optimizing Keil 6/2013 (Thumb mode)**

```
.text:0000005E 48 43      MULS    R0, R1
.text:00000060 80 18      ADDS    R0, R0, R2
.text:00000062 70 47      BX     LR
```

The `MLA` instruction is not available in Thumb mode, so the compiler generates the code doing these two operations (multiplication and addition) separately.

First the `MULS` instruction multiplies `R0` by `R1`, leaving the result in register `R0`. The second instruction (`ADDS`) adds the result and `R2` leaving the result in register `R0`.

## **ARM64**

### **Optimizing GCC (Linaro) 4.9**

Everything here is simple. `MADD` is just an instruction doing fused multiply/add (similar to the `MLA` we already saw). All 3 arguments are passed in the 32-bit parts of X-registers. Indeed, the argument types are 32-bit *int*'s. The result is returned in `W0`.

Listing 1.93: Optimizing GCC (Linaro) 4.9

```
f:
    madd    w0, w0, w1, w2
    ret
```

<sup>84</sup>[Wikipedia: Multiply-accumulate operation](#)

<sup>85</sup>[wikipedia](#)

## 1.10. ACCESSING PASSED ARGUMENTS

```
main:  
; save FP and LR to stack frame:  
    stp      x29, x30, [sp, -16]!  
    mov      w2, 3  
    mov      w1, 2  
    add      x29, sp, 0  
    mov      w0, 1  
    bl       f  
    mov      w1, w0  
    adrp    x0, .LC7  
    add      x0, x0, :lo12:.LC7  
    bl       printf  
; return 0  
    mov      w0, 0  
; restore FP and LR  
    ldp      x29, x30, [sp], 16  
    ret  
  
.LC7:  
    .string "%d\n"
```

Let's also extend all data types to 64-bit `uint64_t` and test:

```
#include <stdio.h>  
#include <stdint.h>  
  
uint64_t f (uint64_t a, uint64_t b, uint64_t c)  
{  
    return a*b+c;  
}  
  
int main()  
{  
    printf ("%lld\n", f(0x1122334455667788,  
                      0x1111111122222222,  
                      0x3333333444444444));  
    return 0;  
}
```

```
f:  
    madd    x0, x0, x1, x2  
    ret  
main:  
    mov     x1, 13396  
    adrp   x0, .LC8  
    stp    x29, x30, [sp, -16]!  
    movk   x1, 0x27d0, lsl 16  
    add    x0, x0, :lo12:.LC8  
    movk   x1, 0x122, lsl 32  
    add    x29, sp, 0  
    movk   x1, 0x58be, lsl 48  
    bl     printf  
    mov    w0, 0  
    ldp    x29, x30, [sp], 16  
    ret  
  
.LC8:  
    .string "%lld\n"
```

The `f()` function is the same, only the whole 64-bit X-registers are now used. Long 64-bit values are loaded into the registers by parts, this is also described here: [1.32.3 on page 442](#).

## Non-optimizing GCC (Linaro) 4.9

The non-optimizing compiler is more redundant:

```
f:
    sub    sp,  sp, #16
    str    w0, [sp,12]
    str    w1, [sp,8]
    str    w2, [sp,4]
    ldr    w1, [sp,12]
    ldr    w0, [sp,8]
    mul    w1, w1, w0
    ldr    w0, [sp,4]
    add    w0, w1, w0
    add    sp, sp, 16
    ret
```

The code saves its input arguments in the local stack, in case someone (or something) in this function needs using the `W0...W2` registers. This prevents overwriting the original function arguments, which may be needed again in the future.

This is called *Register Save Area*. ([*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]<sup>86</sup>). The callee, however, is not obliged to save them. This is somewhat similar to “Shadow Space”: [1.10.2 on page 101](#).

Why did the optimizing GCC 4.9 drop this argument saving code? Because it did some additional optimizing work and concluded that the function arguments will not be needed in the future and also that the registers `W0...W2` will not be used.

We also see a `MUL / ADD` instruction pair instead of single a `MADD`.

## 1.10.4 MIPS

Listing 1.94: Optimizing GCC 4.4.5

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000      mult   $a1, $a0
.text:00000004      mflo   $v0
.text:00000008      jr     $ra
.text:0000000C      addu   $v0, $a2, $v0      ; branch delay slot
; result is in $v0 upon return
.text:00000010 main:
.text:00000010
.text:00000010 var_10 = -0x10
.text:00000010 var_4  = -4
.text:00000010
.text:00000010      lui    $gp, (__gnu_local_gp >> 16)
.text:00000014      addiu $sp, -0x20
.text:00000018      la    $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C      sw    $ra, 0x20+var_4($sp)
.text:00000020      sw    $gp, 0x20+var_10($sp)
; set c:
.text:00000024      li    $a2, 3
; set a:
.text:00000028      li    $a0, 1
.text:0000002C      jal   f
; set b:
.text:00000030      li    $a1, 2      ; branch delay slot
; result in $v0 now
.text:00000034      lw    $gp, 0x20+var_10($sp)
.text:00000038      lui   $a0, ($LC0 >> 16)
.text:0000003C      lw    $t9, (printf & 0xFFFF)($gp)
.text:00000040      la    $a0, ($LC0 & 0xFFFF)
.text:00000044      jalr $t9
; take result of f() function and pass it as a second argument to printf():
.text:00000048      move  $a1, $v0      ; branch delay slot
.text:0000004C      lw    $ra, 0x20+var_4($sp)
```

<sup>86</sup>Also available as <http://go.yurichev.com/17287>

## 1.11. MORE ABOUT RESULTS RETURNING

```
.text:00000050      move    $v0, $zero
.text:00000054      jr     $ra
.text:00000058      addiu   $sp, 0x20 ; branch delay slot
```

The first four function arguments are passed in four registers prefixed by A-.

There are two special registers in MIPS: HI and LO which are filled with the 64-bit result of the multiplication during the execution of the `MULT` instruction.

These registers are accessible only by using the `MFL0` and `MFHI` instructions. `MFL0` here takes the low-part of the multiplication result and stores it into `$V0`. So the high 32-bit part of the multiplication result is dropped (the HI register content is not used). Indeed: we work with 32-bit *int* data types here.

Finally, `ADDU` ("Add Unsigned") adds the value of the third argument to the result.

There are two different addition instructions in MIPS: `ADD` and `ADDU`. The difference between them is not related to signedness, but to exceptions. `ADD` can raise an exception on overflow, which is sometimes useful<sup>87</sup> and supported in Ada `PL`, for instance. `ADDU` does not raise exceptions on overflow.

Since C/C++ does not support this, in our example we see `ADDU` instead of `ADD`.

The 32-bit result is left in `$V0`.

There is a new instruction for us in `main(): JAL` ("Jump and Link").

The difference between `JAL` and `JALR` is that a relative offset is encoded in the first instruction, while `JALR` jumps to the absolute address stored in a register ("Jump and Link Register").

Both `f()` and `main()` functions are located in the same object file, so the relative address of `f()` is known and fixed.

## 1.11 More about results returning

In x86, the result of function execution is usually returned<sup>88</sup> in the `EAX` register. If it is byte type or a character (`char`), then the lowest part of register `EAX` (`AL`) is used. If a function returns a *float* number, the FPU register `ST(0)` is used instead. In ARM, the result is usually returned in the `R0` register.

### 1.11.1 Attempt to use the result of a function returning `void`

So, what if the `main()` function return value was declared of type `void` and not `int`? The so-called startup-code is calling `main()` roughly as follows:

```
push envp
push argv
push argc
call main
push eax
call exit
```

In other words:

```
exit(main(argc, argv, envp));
```

If you declare `main()` as `void`, nothing is to be returned explicitly (using the `return` statement), then something random, that has been stored in the `EAX` register at the end of `main()` becomes the sole argument of the `exit()` function. Most likely, there will be a random value, left from your function execution, so the exit code of program is pseudorandom.

We can illustrate this fact. Please note that here the `main()` function has a `void` return type:

<sup>87</sup> <http://go.yurichev.com/17326>

<sup>88</sup> See also: MSDN: Return Values (C++): [MSDN](#)

## 1.11. MORE ABOUT RESULTS RETURNING

```
#include <stdio.h>

void main()
{
    printf ("Hello, world!\n");
}
```

Let's compile it in Linux.

GCC 4.8.1 replaced `printf()` with `puts()` (we have seen this before: [1.5.4 on page 21](#)), but that's OK, since `puts()` returns the number of characters printed out, just like `printf()`. Please notice that `EAX` is not zeroed before `main()`'s end.

This implies that the value of `EAX` at the end of `main()` contains what `puts()` has left there.

Listing 1.95: GCC 4.8.1

```
.LC0:
.string "Hello, world!"
main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0
    call    puts
    leave
    ret
```

Let's write a bash script that shows the exit status:

Listing 1.96: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

And run it:

```
$ ./tst.sh
Hello, world!
14
```

14 is the number of characters printed. The number of characters printed is *slips* from `printf()` through `EAX / RAX` into “exit code”.

By the way, when we decompile C++ in Hex-Rays, we can often encounter a function which terminated with destructor of some class:

```
...
call    ??1CString@@QAE@XZ ; CString::~CString(void)
mov     ecx, [esp+30h+var_C]
pop    edi
pop    ebx
mov     large fs:0, ecx
add    esp, 28h
retn
```

By C++ standard, destructor doesn't return anything, but when Hex-Rays don't know about it, and thinks that both destructor and this function returns `int`, we can see something like that in output:

```
...
        return CString::~CString(&Str);
}
```

## 1.11.2 What if we do not use the function result?

`printf()` returns the count of characters successfully output, but the result of this function is rarely used in practice.

It is also possible to call a function whose essence is in returning a value, and not use it:

```
int f()
{
    // skip first 3 random values:
    rand();
    rand();
    rand();
    // and use 4th:
    return rand();
};
```

The result of the `rand()` function is left in `EAX`, in all four cases.

But in the first 3 cases, the value in `EAX` is just not used.

## 1.11.3 Returning a structure

Let's go back to the fact that the return value is left in the `EAX` register.

That is why old C compilers cannot create functions capable of returning something that does not fit in one register (usually `int`), but if one needs it, one have to return information via pointers passed as function's arguments.

So, usually, if a function needs to return several values, it returns only one, and all the rest—via pointers.

Now it has become possible to return, let's say, an entire structure, but that is still not very popular. If a function has to return a large structure, the `caller` must allocate it and pass a pointer to it via the first argument, transparently for the programmer. That is almost the same as to pass a pointer in the first argument manually, but the compiler hides it.

Small example:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

...what we got (MSVC 2010 /0x):

```
$T3853 = 8          ; size = 4
_a$ = 12           ; size = 4
?get_some_values@@YA?AUa@H@Z PROC      ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
```

## 1.12. POINTERS

```
ret    0  
?get_some_values@@YA?AUs@H@Z ENDP           ; get_some_values
```

The macro name for internal passing of pointer to a structure here is `$T3853`.

This example can be rewritten using the C99 language extensions:

```
struct s  
{  
    int a;  
    int b;  
    int c;  
};  
  
struct s get_some_values (int a)  
{  
    return (struct s){.a=a+1, .b=a+2, .c=a+3};  
};
```

Listing 1.97: GCC 4.8.1

```
_get_some_values proc near  
  
ptr_to_struct    = dword ptr  4  
a                = dword ptr  8  
  
    mov     edx, [esp+a]  
    mov     eax, [esp+ptr_to_struct]  
    lea     ecx, [edx+1]  
    mov     [eax], ecx  
    lea     ecx, [edx+2]  
    add     edx, 3  
    mov     [eax+4], ecx  
    mov     [eax+8], edx  
    retn  
  
_get_some_values endp
```

As we see, the function is just filling the structure's fields allocated by the caller function, as if a pointer to the structure has been passed. So there are no performance drawbacks.

## 1.12 Pointers

### 1.12.1 Swap input values

This will do the job:

```
#include <memory.h>  
#include <stdio.h>  
  
void swap_bytes (unsigned char* first, unsigned char* second)  
{  
    unsigned char tmp1;  
    unsigned char tmp2;  
  
    tmp1=*first;  
    tmp2=*second;  
  
    *first=tmp2;  
    *second=tmp1;  
}  
  
int main()  
{  
    // copy string into heap, so we will be able to modify it  
    char *s=strdup("string");
```

## 1.12. POINTERS

```
// swap 2nd and 3rd characters
swap_bytes (s+1, s+2);

printf ("%s\n", s);
};
```

As we can see, bytes are loaded into lower 8-bit parts of **ECX** and **EBX** using **MOVZX** (so higher parts of these registers will be cleared) and then bytes are written back swapped.

Listing 1.98: Optimizing GCC 5.4

```
swap_bytes:
    push    ebx
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+12]
    movzx   ecx, BYTE PTR [edx]
    movzx   ebx, BYTE PTR [eax]
    mov     BYTE PTR [edx], bl
    mov     BYTE PTR [eax], cl
    pop    ebx
    ret
```

Addresses of both bytes are taken from arguments and through execution of the function are located in **EDX** and **EAX**.

So we use pointers: probably, there is no better way to solve this task without them.

### 1.12.2 Returning values

Pointers are often used to return values from functions (recall `scanf()` case ([1.9 on page 66](#))).

For example, when a function needs to return two values.

#### Global variables example

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
    *sum=x+y;
    *product=x*y;
}

int sum, product;

void main()
{
    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
}
```

This compiles to:

Listing 1.99: Optimizing MSVC 2010 (/Ob0)

```
COMM  _product:DWORD
COMM  _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8          ; size = 4
_y$ = 12         ; size = 4
_sum$ = 16        ; size = 4
_product$ = 20      ; size = 4
_f1    PROC
    mov     ecx, DWORD PTR _y$[esp-4]
    mov     eax, DWORD PTR _x$[esp-4]
    lea    edx, DWORD PTR [eax+ecx]
```

## 1.12. POINTERS

```
    imul    eax,  ecx
    mov     ecx,  DWORD PTR _product$[esp-4]
    push    esi
    mov     esi,  DWORD PTR _sum$[esp]
    mov     DWORD PTR [esi], edx
    mov     DWORD PTR [ecx], eax
    pop    esi
    ret    0
_f1    ENDP

_main PROC
push    OFFSET _product
push    OFFSET _sum
push    456      ; 0000001c8H
push    123      ; 00000007bH
call    _f1
mov     eax,  DWORD PTR _product
mov     ecx,  DWORD PTR _sum
push    eax
push    ecx
push    OFFSET $SG2803
call    DWORD PTR __imp__printf
add    esp, 28
xor    eax, eax
ret    0
_main ENDP
```

## 1.12. POINTERS

Let's see this in OllyDbg:

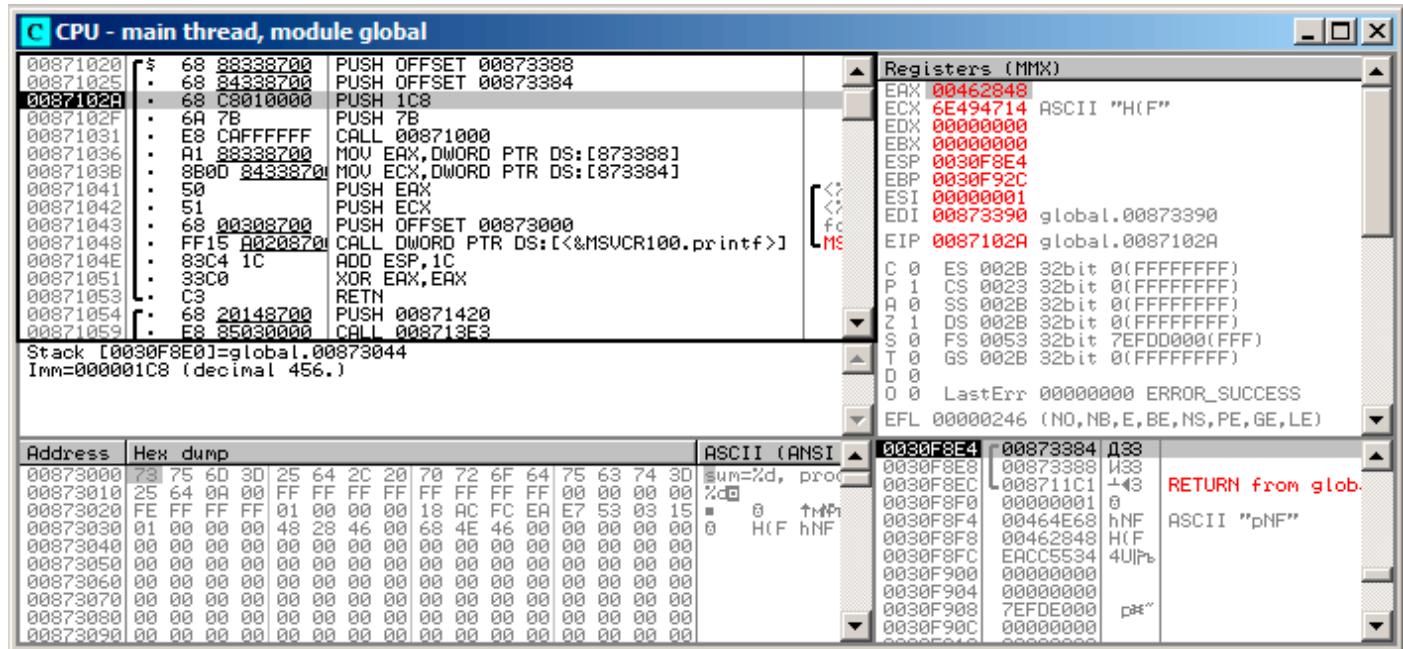


Figure 1.24: OllyDbg: global variables addresses are passed to `f1()`

First, global variables' addresses are passed to `f1()`. We can click “Follow in dump” on the stack element, and we can see the place in the data segment allocated for the two variables.

## 1.12. POINTERS

These variables are zeroed, because non-initialized data (from [BSS](#)) is cleared before the execution begins, [see ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.8p10].

They reside in the data segment, we can verify this by pressing Alt-M and reviewing the memory map:

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00050000	00004000				Map	R	R	
00068000	00001000				Priv	RW	RW	
00070000	00067000				Map	R	R	
00159000	00007000				Priv	RW	Guar	RW
00380000	00001000				Priv	RW	Guar	RW
0038E000	00002000			Stack of main thread	Priv	RW	Guar	RW
00460000	00005000			Heap	Priv	RW	Guar	RW
00490000	00007000				Priv	RW	Guar	RW
006B0000	0000C000			Default heap	Priv	RW	RW	
00870000	00001000	global		PE header	Img	R	RWE	Cop
00871000	00001000	global	.text	Code	Img	R E	RWE	Cop
00872000	00001000	global	.rdata	Imports	Img	R	RWE	Cop
00873000	00001000	global	.data	Data	Img	RW	RWE	Cop
00874000	00001000	global	.reloc	Relocations	Img	R	RWE	Cop
6E3E0000	00001000	MSVCR100		PE header	Img	R	RWE	Cop
6E3E1000	000B2000	MSVCR100	.text	Code, imports, exports	Img	R E	RWE	Cop
6E493000	00006000	MSVCR100	.data	Data	Img	RW	Cop	RWE
6E499000	00001000	MSVCR100	.rsrc	Resources	Img	R	RWE	Cop
6E49A000	00005000	MSVCR100	.reloc	Relocations	Img	R	RWE	Cop
755D0000	00001000	Mod_755D		PE header	Img	R	RWE	Cop
755D1000	00003000				Img	R E	RWE	Cop
755D4000	00001000				Img	RW	RWE	Cop
755D5000	00003000				Img	R	RWE	Cop
755E0000	00001000	Mod_755E		PE header	Img	R	RWE	Cop
755E1000	0004D000				Img	R E	RWE	Cop
7562E000	00005000				Img	RW	Cop	RWE
75633000	00009000				Img	R	RWE	Cop

Figure 1.25: OllyDbg: memory map

## 1.12. POINTERS

Let's trace (F7) to the start of `f1()`:

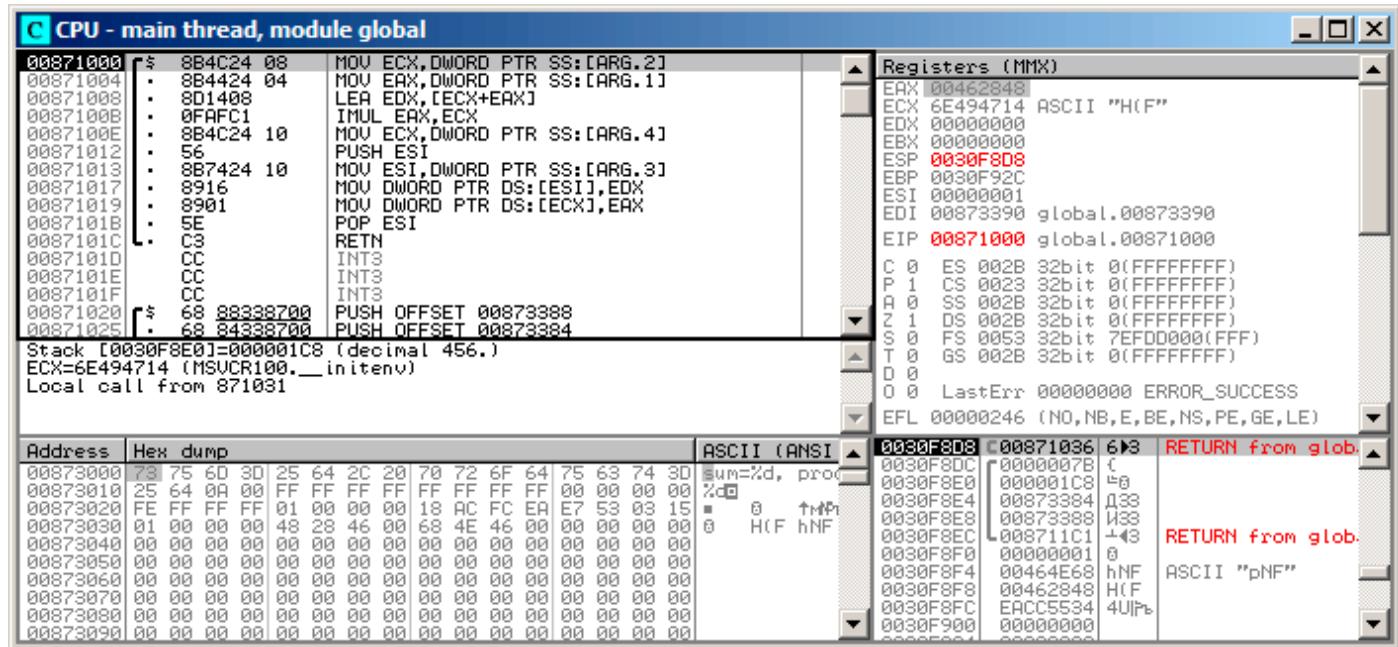


Figure 1.26: OllyDbg: `f1()` starts

Two values are visible in the stack 456 (0x1C8) and 123 (0x7B), and also the addresses of the two global variables.

## 1.12. POINTERS

Let's trace until the end of `f1()`. In the left bottom window we see how the results of the calculation appear in the global variables:

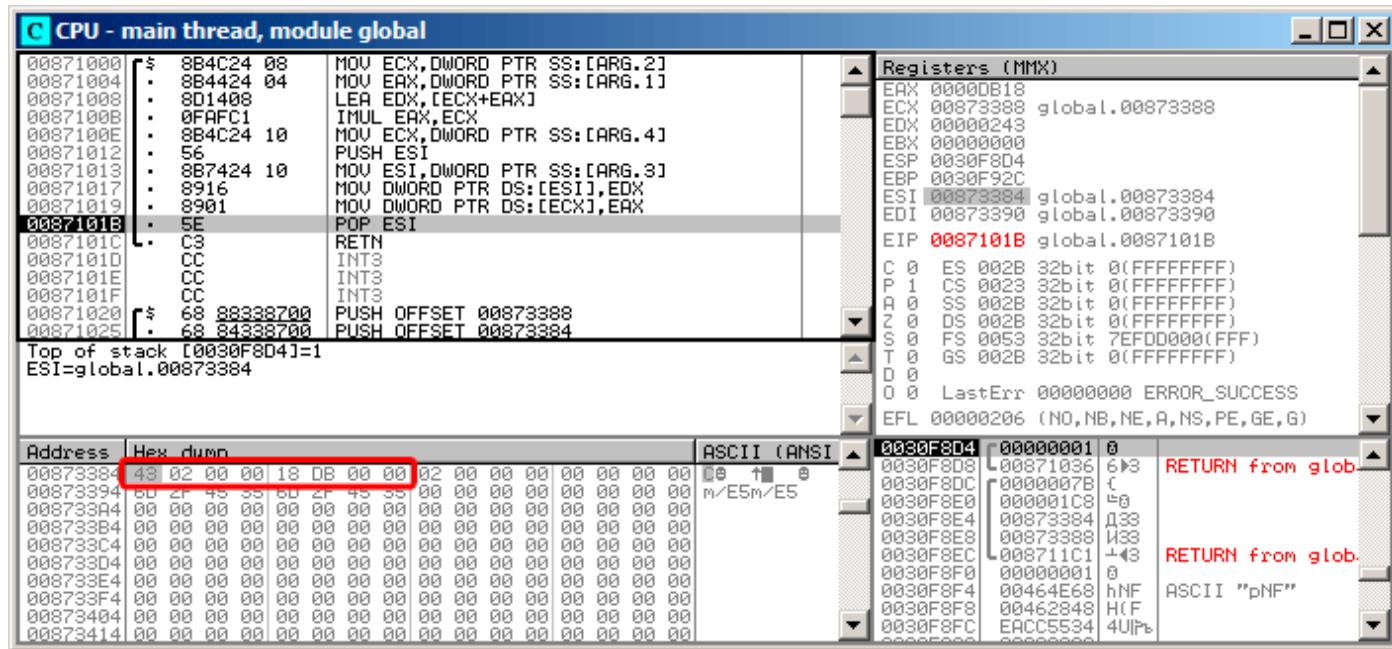


Figure 1.27: OllyDbg: `f1()` execution completed

## 1.12. POINTERS

Now the global variables' values are loaded into registers ready for passing to `printf()` (via the stack):

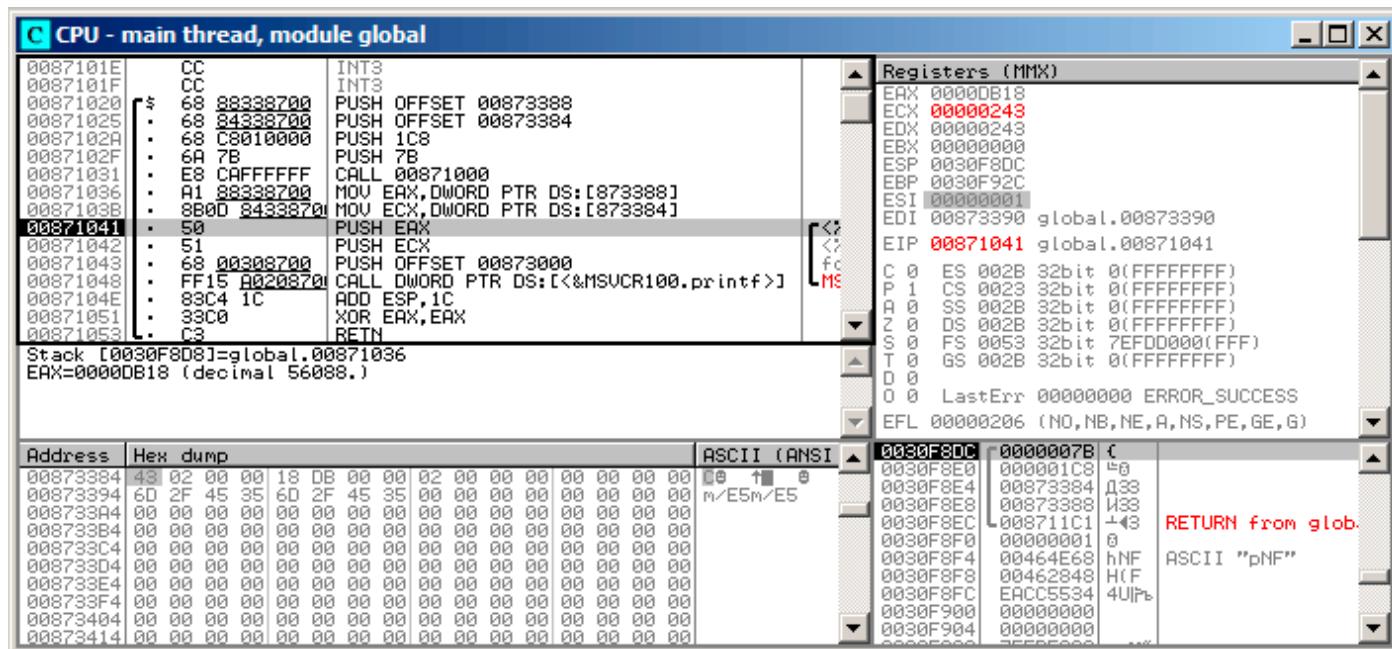


Figure 1.28: OllyDbg: global variables' values are passed into `printf()`

## Local variables example

Let's rework our example slightly:

Listing 1.100: now the `sum` and `product` variables are local

```
void main()
{
    int sum, product; // now variables are local in this function

    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
}
```

`f1()` code will not change. Only the code of `main()` will do:

Listing 1.101: Optimizing MSVC 2010 (/Ob0)

```
_product$ = -8          ; size = 4
_sum$ = -4            ; size = 4
_main    PROC
; Line 10
    sub     esp, 8
; Line 13
    lea      eax, DWORD PTR _product$[esp+8]
    push    eax
    lea      ecx, DWORD PTR _sum$[esp+12]
    push    ecx
    push    456      ; 000001c8H
    push    123      ; 0000007bh
    call    _f1
; Line 14
    mov      edx, DWORD PTR _product$[esp+24]
    mov      eax, DWORD PTR _sum$[esp+24]
    push    edx
    push    eax
    push    OFFSET $SG2803
    call    DWORD PTR __imp__printf
; Line 15
    xor      eax, eax
```

## 1.12. POINTERS

```
add    esp, 36  
ret    0
```

## 1.12. POINTERS

Let's look again with OllyDbg. The addresses of the local variables in the stack are `0x2EF854` and `0x2EF858`. We see how these are pushed into the stack:

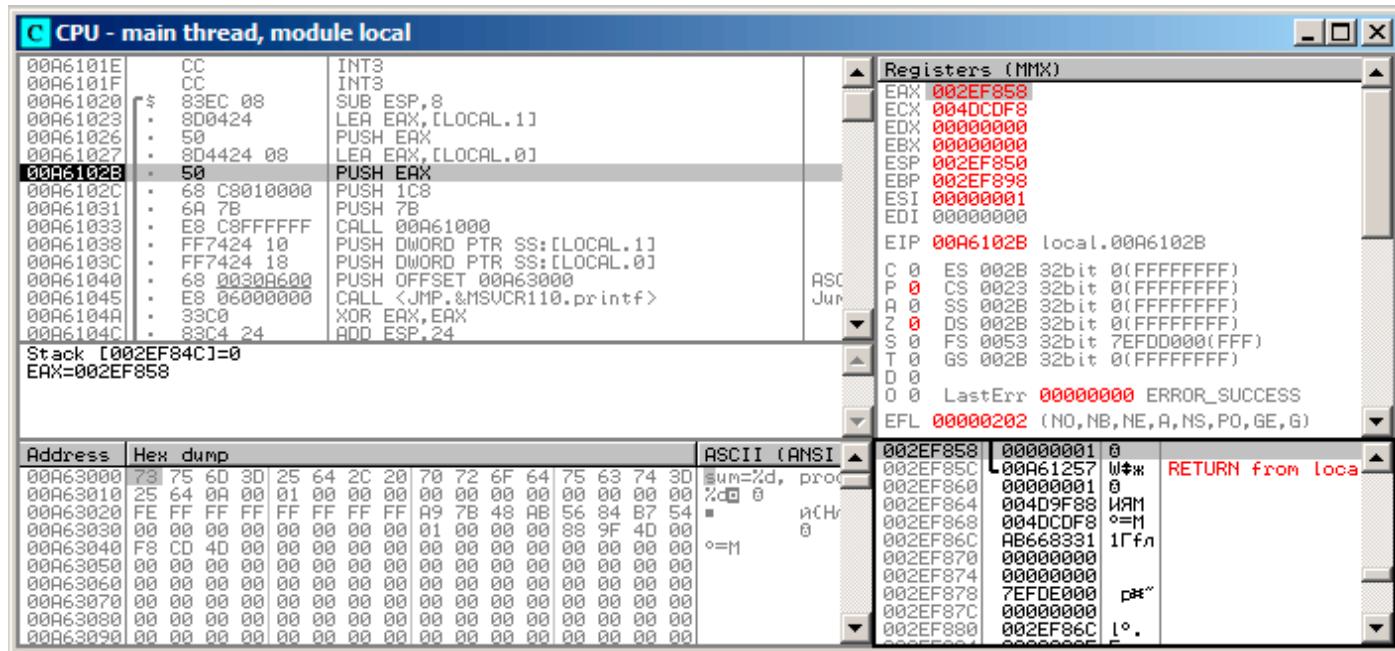


Figure 1.29: OllyDbg: local variables' addresses are pushed into the stack

## 1.12. POINTERS

f1() starts. So far there is only random garbage in the stack at 0x2EF854 and 0x2EF858 :

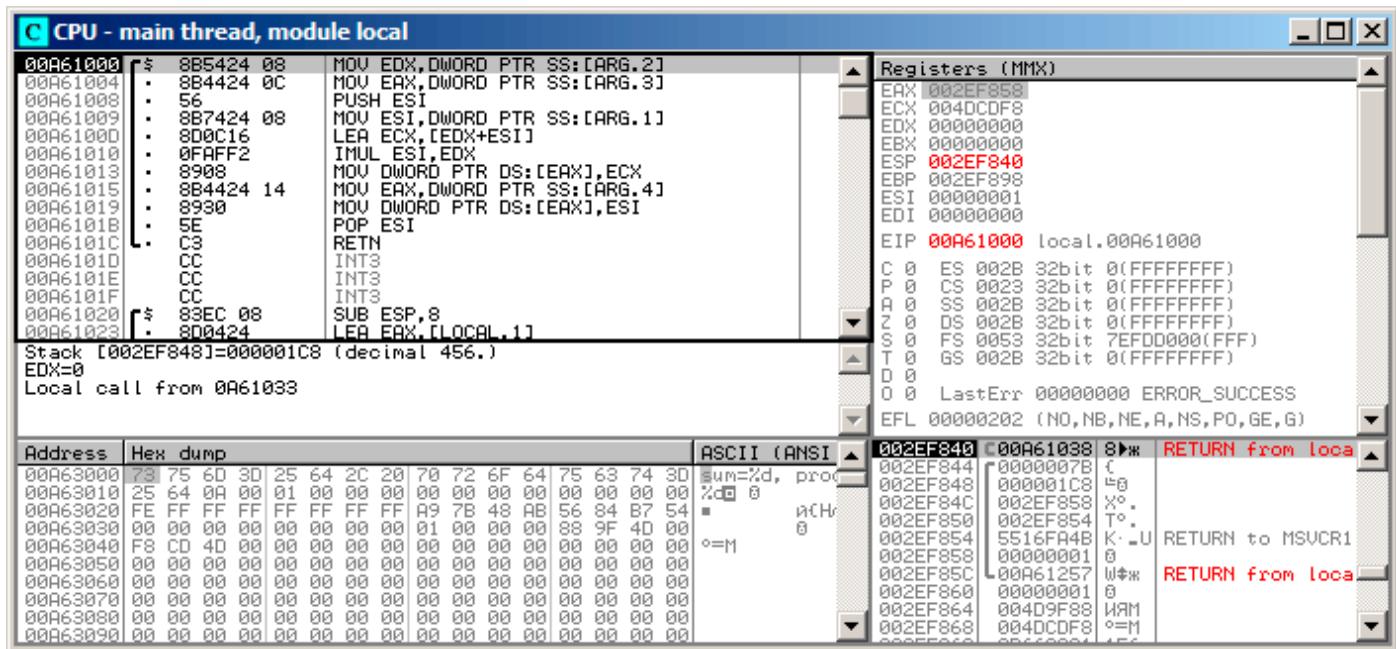


Figure 1.30: OllyDbg: f1() starting

## 1.13. GOTO OPERATOR

`f1()` completes:

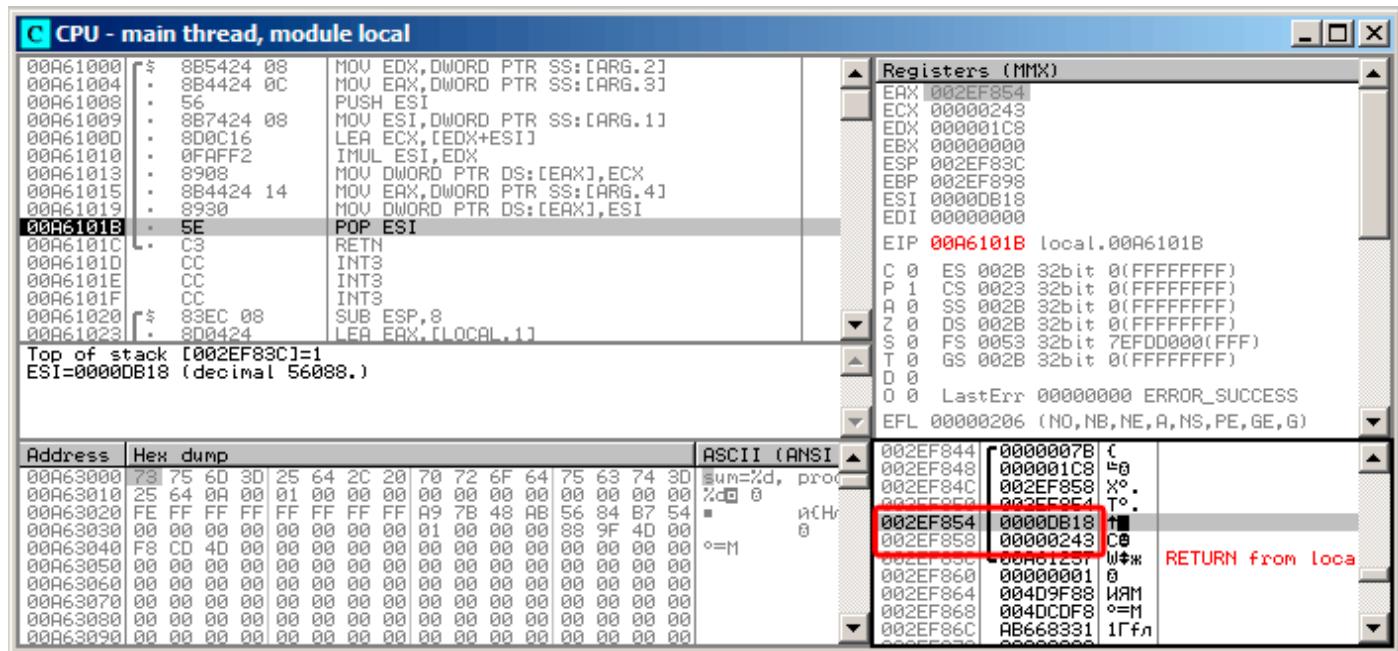


Figure 1.31: OllyDbg: `f1()` completes execution

We now find `0xDB18` and `0x243` at addresses `0x2EF854` and `0x2EF858`. These values are the `f1()` results.

## Conclusion

`f1()` could return pointers to any place in memory, located anywhere.

This is in essence the usefulness of the pointers.

By the way, C++ references work exactly the same way. Read more about them: ([3.19.3 on page 562](#)).

## 1.13 GOTO operator

The GOTO operator is generally considered as anti-pattern, see [Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)<sup>89</sup>]. Nevertheless, it can be used reasonably, see [Donald E. Knuth, *Structured Programming with go to Statements* (1974)<sup>90</sup>] <sup>91</sup>.

Here is a very simple example:

```
#include <stdio.h>

int main()
{
    printf ("begin\n");
    goto exit;
    printf ("skip me!\n");
exit:
    printf ("end\n");
}
```

Here is what we have got in MSVC 2012:

<sup>89</sup><http://yurichev.com/mirrors/Dijkstra68.pdf>

<sup>90</sup><http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf>

<sup>91</sup>[Dennis Yurichev, *C/C++ programming language notes*] also has some examples.

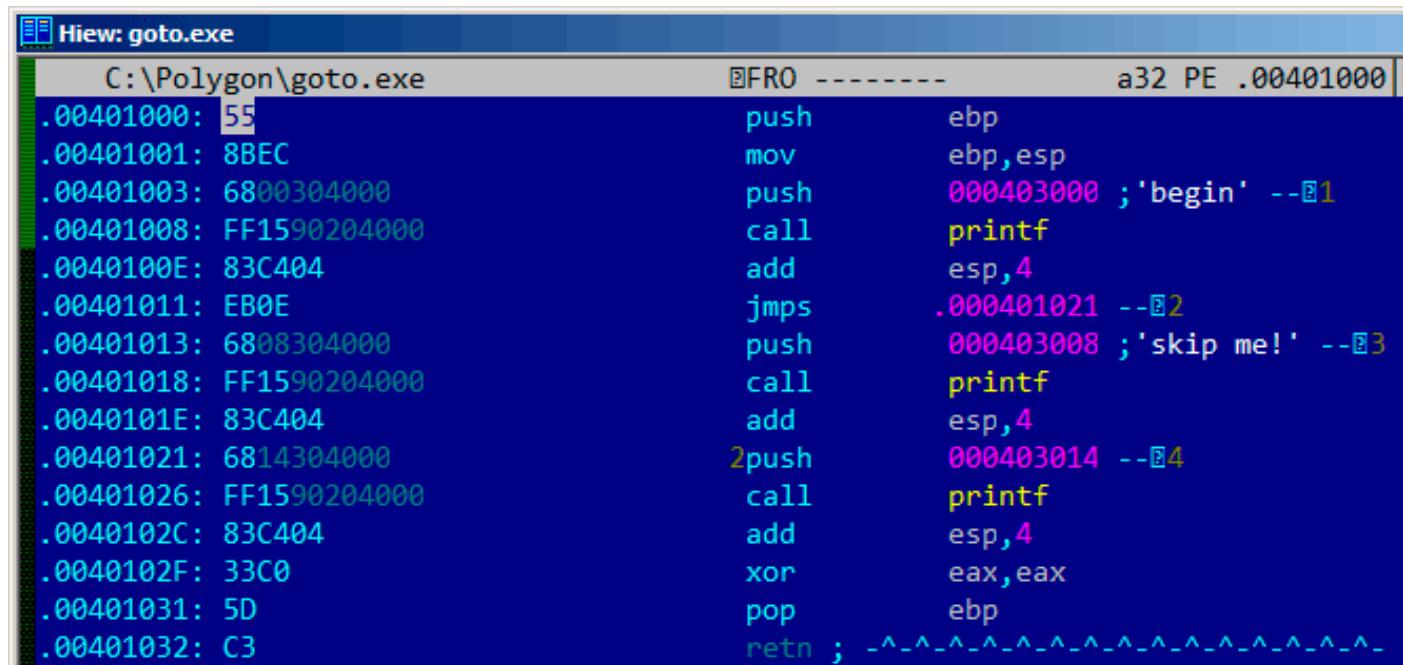
```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2934 ; 'begin'
    call    _printf
    add    esp, 4
    jmp    SHORT $exit$3
    push    OFFSET $SG2936 ; 'skip me!'
    call    _printf
    add    esp, 4
$exit$3:
    push    OFFSET $SG2937 ; 'end'
    call    _printf
    add    esp, 4
    xor    eax, eax
    pop    ebp
    ret    0
_main ENDP
```

The `goto` statement has been simply replaced by a `JMP` instruction, which has the same effect: unconditional jump to another place. The second `printf()` could be executed only with human intervention, by using a debugger or by patching the code.

### 1.13. GOTO OPERATOR

This could also be useful as a simple patching exercise. Let's open the resulting executable in Hiew:



The screenshot shows the Hiew debugger interface with the title "Hiew: goto.exe". The assembly code is displayed in three columns: address, assembly instructions, and comments. The code starts at address .00401000 and includes instructions like push, mov, call, add, jmps, push, call, add, push, call, add, xor, pop, and ret. There are also several comments in blue and green, such as 'begin' at address .000403000 and 'skip me!' at address .000403008. The assembly code is color-coded with syntax highlighting.

Address	Assembly	Comments
.00401000	55	
.00401001	8BEC	
.00401003	6800304000	
.00401008	FF1590204000	
.0040100E	83C404	
.00401011	EB0E	
.00401013	6808304000	
.00401018	FF1590204000	
.0040101E	83C404	
.00401021	6814304000	
.00401026	FF1590204000	
.0040102C	83C404	
.0040102F	33C0	
.00401031	5D	
.00401032	C3	

Figure 1.32: Hiew

### **1.13. GOTO OPERATOR**

Place the cursor to address `JMP (0x410)`, press F3 (edit), press zero twice, so the opcode becomes `EB 00`:

Figure 1.33: Hiew

The second byte of the `JMP` opcode denotes the relative offset for the jump, 0 means the point right after the current instruction.

So now JMP not skipping the second printf() call.

Press F9 (save) and exit. Now if we run the executable we will see this:

Listing 1.103: Patched executable output

C:\...>goto.exe

```
begin  
skip me!  
end
```

The same result could be achieved by replacing the `JMP` instruction with 2 `NOP` instructions.

`NOP` has an opcode of `0x90` and length of 1 byte, so we need 2 instructions as `JMP` replacement (which is 2 bytes in size).

### 1.13.1 Dead code

The second `printf()` call is also called “dead code” in compiler terms.

This means that the code will never be executed. So when you compile this example with optimizations, the compiler removes “dead code”, leaving no trace of it:

Listing 1.104: Optimizing MSVC 2012

```
$SG2981 DB      'begin', 0Ah, 00H
$SG2983 DB      'skip me!', 0Ah, 00H
$SG2984 DB      'end', 0Ah, 00H

_main    PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
```

## 1.14. CONDITIONAL JUMPS

```
call    _printf
add     esp, 8
xor     eax, eax
ret     0
_main  ENDP
```

However, the compiler forgot to remove the “skip me!” string.

### 1.13.2 Exercise

Try to achieve the same result using your favorite compiler and debugger.

## 1.14 Conditional jumps

### 1.14.1 Simple example

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

## x86

### x86 + MSVC

Here is how the `f_signed()` function looks like:

Listing 1.105: Non-optimizing MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cmp     eax, DWORD PTR _b$[ebp]
    jle     SHORT $LN3@f_signed
    push    OFFSET $SG737           ; 'a>b'
```

## 1.14. CONDITIONAL JUMPS

```
call    _printf
add    esp, 4
$LN3@f_signed:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739      ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741      ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

The first instruction, `JLE`, stands for *Jump if Less or Equal*. In other words, if the second operand is larger or equal to the first one, the control flow will be passed to the address or label specified in the instruction. If this condition does not trigger because the second operand is smaller than the first one, the control flow would not be altered and the first `printf()` would be executed. The second check is `JNE`: *Jump if Not Equal*. The control flow will not change if the operands are equal.

The third check is `JGE`: *Jump if Greater or Equal*—jump if the first operand is larger than the second or if they are equal. So, if all three conditional jumps are triggered, none of the `printf()` calls would be executed whatsoever. This is impossible without special intervention. Now let's take a look at the `f_unsigned()` function. The `f_unsigned()` function is the same as `f_signed()`, with the exception that the `JBE` and `JAE` instructions are used instead of `JLE` and `JGE`, as follows:

Listing 1.106: GCC

```
a$ = 8    ; size = 4
b$ = 12   ; size = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761      ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763      ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765      ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_unsigned:
    pop    ebp
    ret    0
_f_unsigned ENDP
```

As already mentioned, the branch instructions are different: `JBE`—*Jump if Below or Equal* and `JAE`—*Jump if Above or Equal*. These instructions (`JA` / `JAE` / `JB` / `JBE`) differ from `JG` / `JGE` / `JL` / `JLE` in the fact that they work with unsigned numbers.

#### 1.14. CONDITIONAL JUMPS

See also the section about signed number representations ([2.2 on page 454](#)). That is why if we see **JG / JL** in use instead of **JA / JB** or vice-versa, we can be almost sure that the variables are signed or unsigned, respectively. Here is also the `main()` function, where there is nothing much new to us:

Listing 1.107: `main()`

```
_main PROC
    push    ebp
    mov     ebp, esp
    push    2
    push    1
    call    _f_signed
    add    esp, 8
    push    2
    push    1
    call    _f_unsigned
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main ENDP
```

We can see how flags are set by running this example in OllyDbg. Let's begin with `f_unsigned()`, which works with unsigned numbers.

`CMP` is executed thrice here, but for the same arguments, so the flags are the same each time.

Result of the first comparison:

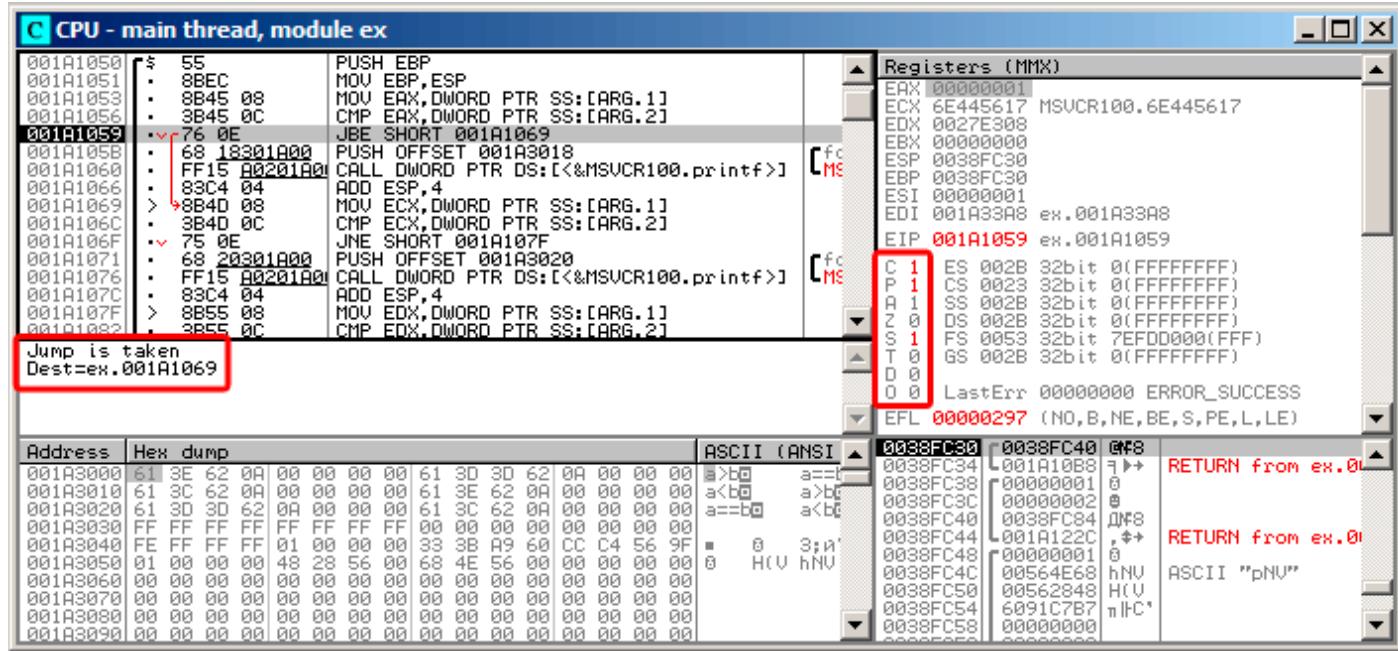


Figure 1.34: OllyDbg: `f_unsigned()`: first conditional jump

So, the flags are: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

They are named with one character for brevity in OllyDbg.

OllyDbg gives a hint that the (`JBE`) jump is to be triggered now. Indeed, if we take a look into Intel manuals ([11.1.4 on page 996](#)), we can read there that `JBE` is triggering if CF=1 or ZF=1. The condition is true here, so the jump is triggered.

## 1.14. CONDITIONAL JUMPS

The next conditional jump:

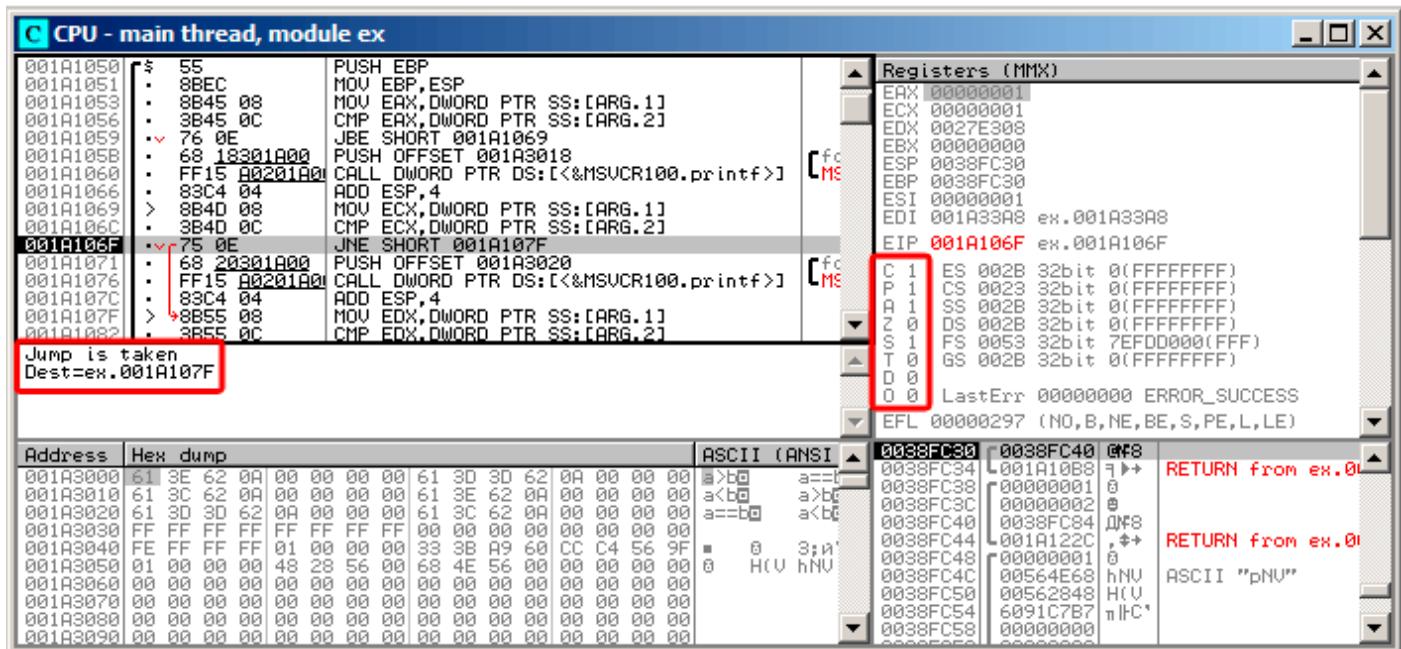


Figure 1.35: OllyDbg: `f_unsigned()`: second conditional jump

OllyDbg gives a hint that `JNZ` is to be triggered now. Indeed, `JNZ` triggering if  $ZF=0$  (zero flag).

## 1.14. CONDITIONAL JUMPS

The third conditional jump, `JNB` :

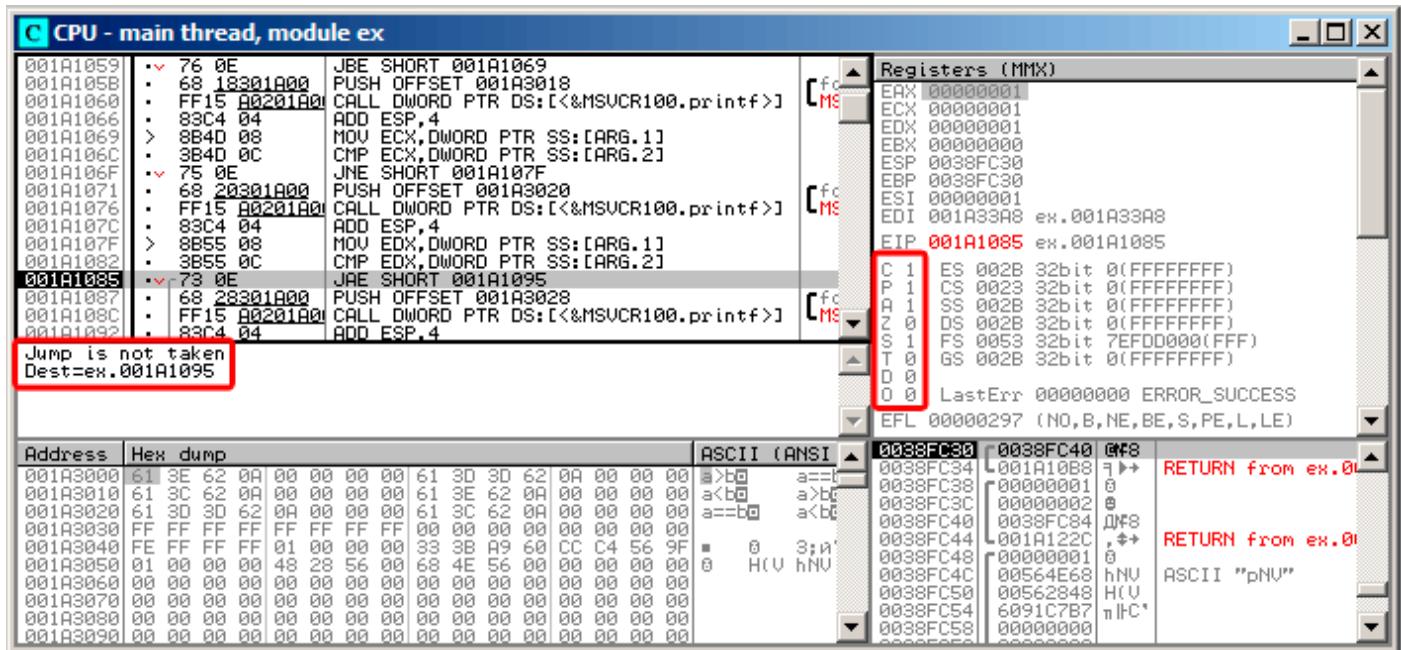


Figure 1.36: OllyDbg: `f_unsigned()` : third conditional jump

In Intel manuals ([11.1.4 on page 996](#)) we can see that `JNB` triggers if CF=0 (carry flag). That is not true in our case, so the third `printf()` will execute.

## 1.14. CONDITIONAL JUMPS

Now let's review the `f_signed()` function, which works with signed values, in OllyDbg. Flags are set in the same way: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. The first conditional jump `JLE` is to be triggered:

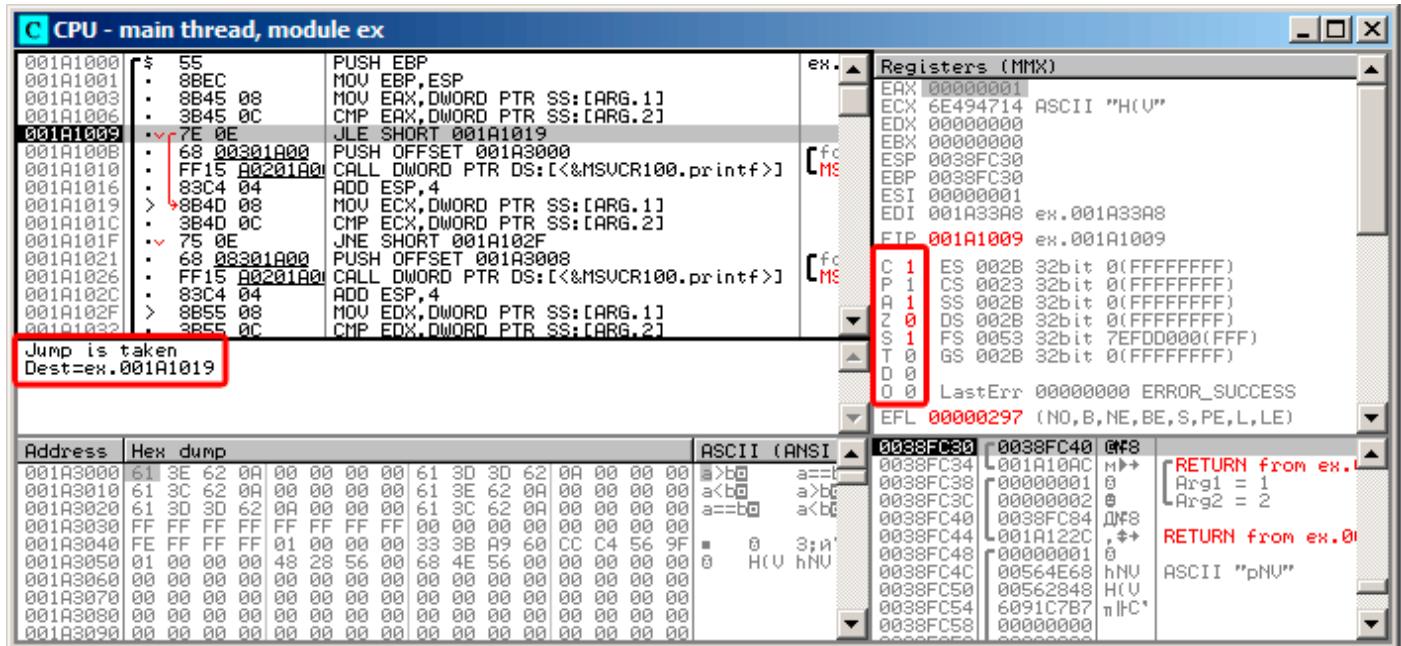


Figure 1.37: OllyDbg: `f_signed()`: first conditional jump

In Intel manuals ([11.1.4 on page 996](#)) we find that this instruction is triggered if  $ZF=1$  or  $SF \neq OF$ .  $SF \neq OF$  in our case, so the jump triggers.

## 1.14. CONDITIONAL JUMPS

The second `JNZ` conditional jump triggering: if `ZF=0` (zero flag):

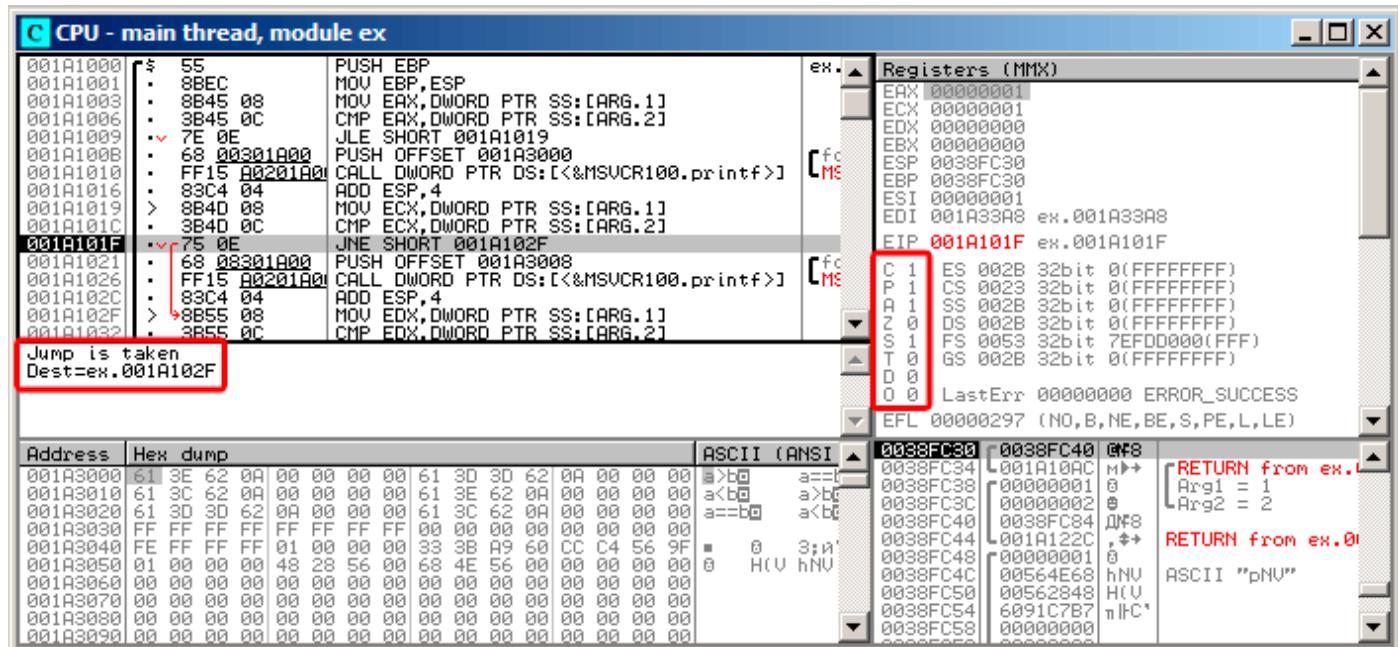


Figure 1.38: OllyDbg: `f_signed()`: second conditional jump

## 1.14. CONDITIONAL JUMPS

The third conditional jump `JGE` will not trigger because it would only do so if SF=OF, and that is not true in our case:

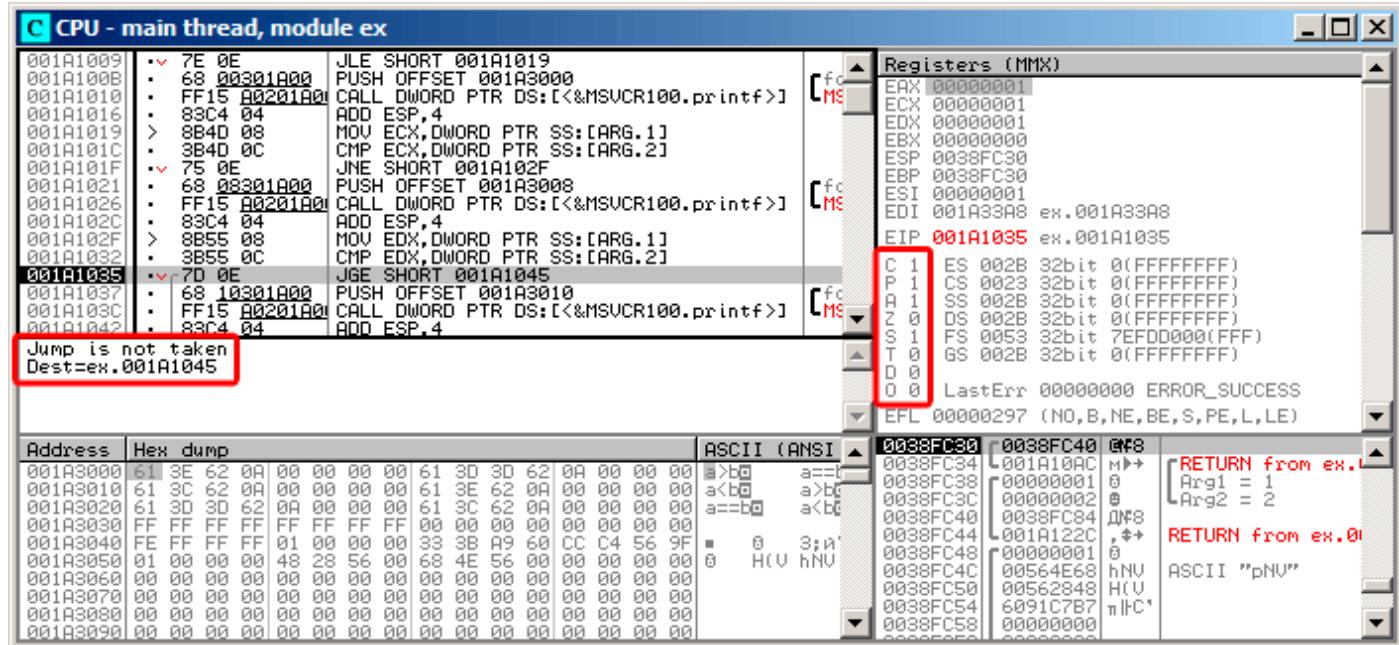


Figure 1.39: OllyDbg: `f_signed()`: third conditional jump

We can try to patch the executable file in a way that the `f_unsigned()` function would always print "a==b", no matter the input values. Here is how it looks in Hiew:

Figure 1.40: Hiew: f\_unsigned() function

Essentially, we have to accomplish three tasks:

- force the first jump to always trigger;
  - force the second jump to never trigger;
  - force the third jump to always trigger.

Thus we can direct the code flow to always pass through the second `printf()`, and output "a==b".

Three instructions (or bytes) has to be patched:

- The first jump becomes `JMP`, but the `jump offset` would remain the same.
  - The second jump might be triggered sometimes, but in any case it will jump to the next instruction, because, we set the `jump offset` to 0.

In these instructions the **jump offset** is added to the address for the next instruction. So if the offset is 0, the jump will transfer the control to the next instruction.

- The third jump we replace with `JMP` just as we do with the first one, so it will always trigger.

## 1.14. CONDITIONAL JUMPS

Here is the modified code:

The screenshot shows the Hiew 8.02 debugger interface with the assembly code for the `f_unsigned()` function. The code is color-coded by register (push, mov, cmp, jmps, add, call, etc.) and memory address. The assembly code consists of several loops and conditional jumps, primarily using `EB0D` (jmps) and `EB0A` (jnz) instructions to control the flow. The code is located at address `00000434` and ends with a `ret` instruction.

```
C:\Polygon\ollydbg\7_1.exe 0FWO EDITMODE a32 PE 00000434 Hiew 8.02 (c)SEN  
00000400: 55 push    ebp  
00000401: 8BEC mov     ebp,esp  
00000403: 8B4508 mov     eax,[ebp][8]  
00000406: 3B450C cmp     eax,[ebp][00C]  
00000409: EB0D jmps    00000418  
0000040B: 6800B04000 push    00040B000 ;' @  
00000410: E8AA000000 call    0000004BF  
00000415: 83C404 add    esp,4  
00000418: 8B4D08 mov     ecx,[ebp][8]  
0000041B: 3B4D0C cmp     ecx,[ebp][00C]  
0000041E: 7500 jnz    00000420  
00000420: 6808B04000 push    00040B008 ;' @  
00000425: E895000000 call    0000004BF  
0000042A: 83C404 add    esp,4  
0000042D: 8B5508 mov     edx,[ebp][8]  
00000430: 3B550C cmp     edx,[ebp][00C]  
00000433: EB0D jmps    00000442  
00000435: 6810B04000 push    00040B010 ;' @  
0000043A: E880000000 call    0000004BF  
0000043F: 83C404 add    esp,4  
00000442: 5D pop    ebp  
00000443: C3 retn   ; -^--^--^--^--^--^--^--^--^--^--^--^--  
00000444: CC int    3  
00000445: CC int    3  
00000446: CC int    3  
00000447: CC int    3  
00000448: CC int    3
```

Figure 1.41: Hiew: let's modify the `f_unsigned()` function

If we miss to change any of these jumps, then several `printf()` calls may execute, while we want to execute only one.

### Non-optimizing GCC

Non-optimizing GCC 4.4.1 produces almost the same code, but with `puts()` ([1.5.4 on page 21](#)) instead of `printf()`.

### Optimizing GCC

An observant reader may ask, why execute `CMP` several times, if the flags has the same values after each execution?

Perhaps optimizing MSVC cannot do this, but optimizing GCC 4.8.1 can go deeper:

Listing 1.108: GCC 4.8.1 `f_signed()`

```
f_signed:  
    mov    eax, DWORD PTR [esp+8]  
    cmp    DWORD PTR [esp+4], eax  
    jg     .L6  
    je     .L7  
    jge    .L1  
    mov    DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"  
    jmp    puts  
.L6:
```

## 1.14. CONDITIONAL JUMPS

```
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
    jmp     puts
.L1:
    rep    ret
.L7:
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
    jmp     puts
```

We also see `JMP puts` here instead of `CALL puts / RETN`.

This kind of trick will have explained later: [1.15.1 on page 155](#).

This type of x86 code is somewhat rare. MSVC 2012 as it seems, can't generate such code. On the other hand, assembly language programmers are fully aware of the fact that `Jcc` instructions can be stacked.

So if you see such stacking somewhere, it is highly probable that the code was hand-written.

The `f_unsigned()` function is not that aesthetically short:

Listing 1.109: GCC 4.8.1 `f_unsigned()`

```
f_unsigned:
    push    esi
    push    ebx
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    mov     ebx, DWORD PTR [esp+36]
    cmp     esi, ebx
    ja     .L13
    cmp     esi, ebx ; this instruction could be removed
    je     .L14
.L10:
    jb     .L15
    add    esp, 20
    pop    ebx
    pop    esi
    ret
.L15:
    mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
    add    esp, 20
    pop    ebx
    pop    esi
    jmp     puts
.L13:
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
    call    puts
    cmp     esi, ebx
    jne    .L10
.L14:
    mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
    add    esp, 20
    pop    ebx
    pop    esi
    jmp     puts
```

Nevertheless, there are two `CMP` instructions instead of three.

So optimization algorithms of GCC 4.8.1 are probably not perfect yet.

## ARM

### 32-bit ARM

## Optimizing Keil 6/2013 (ARM mode)

## 1.14. CONDITIONAL JUMPS

Listing 1.110: Optimizing Keil 6/2013 (ARM mode)

```
.text:000000B8          EXPORT f_signed
.text:000000B8          f_signed           ; CODE XREF: main+C
.text:000000B8 70 40 2D E9    STMF D SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1    MOV    R4, R1
.text:000000C0 04 00 50 E1    CMP    R0, R4
.text:000000C4 00 50 A0 E1    MOV    R5, R0
.text:000000C8 1A 0E 8F C2    ADRGT R0, aAB      ; "a>b\n"
.text:000000CC A1 18 00 CB    BLGT   __2printf
.text:000000D0 04 00 55 E1    CMP    R5, R4
.text:000000D4 67 0F 8F 02    ADREQ R0, aAB_0    ; "a==b\n"
.text:000000D8 9E 18 00 0B    BLEQ   __2printf
.text:000000DC 04 00 55 E1    CMP    R5, R4
.text:000000E0 70 80 BD A8    LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8    LDMFD  SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2    ADR    R0, aAB_1    ; "a<b\n"
.text:000000EC 99 18 00 EA    B      __2printf
.text:000000EC          ; End of function f_signed
```

Many instructions in ARM mode could be executed only when specific flags are set. E.g. this is often used when comparing numbers.

For instance, the `ADD` instruction is in fact named `ADDAL` internally, where `AL` stands for *Always*, i.e., execute always. The predicates are encoded in 4 high bits of the 32-bit ARM instructions (*condition field*). The `B` instruction for unconditional jumping is in fact conditional and encoded just like any other conditional jump, but has `AL` in the *condition field*, and it implies execute *ALways*, ignoring flags.

The `ADRG` instruction works just like `ADR` but executes only in case the previous `CMP` instruction founds one of the numbers greater than the another, while comparing the two (*Greater Than*).

The next `BLGT` instruction behaves exactly as `BL` and is triggered only if the result of the comparison has been (*Greater Than*). `ADRG` writes a pointer to the string `a>b\n` into `R0` and `BLGT` calls `printf()`. Therefore, instructions suffixed with `-GT` are to execute only in case the value in `R0` (which is *a*) is bigger than the value in `R4` (which is *b*).

Moving forward we see the `ADREQ` and `BLEQ` instructions. They behave just like `ADR` and `BL`, but are to be executed only if operands were equal to each other during the last comparison. Another `CMP` is located before them (because the `printf()` execution may have tampered the flags).

Then we see `LDMGEFD`, this instruction works just like `LDMFD`<sup>92</sup>, but is triggered only when one of the values is greater or equal than the other (*Greater or Equal*). The `LDMGEFD SP!, {R4-R6,PC}` instruction acts like a function epilogue, but it will be triggered only if *a*  $\geq$  *b*, and only then the function execution will finish.

But if that condition is not satisfied, i.e., *a*  $<$  *b*, then the control flow will continue to the next “`LDMFD SP!, {R4-R6,LR}`” instruction, which is one more function epilogue. This instruction restores not only the `R4-R6` registers state, but also `LR` instead of `PC`, thus, it does not return from the function. The last two instructions call `printf()` with the string «*a* $<$ *b*\n» as a sole argument. We already examined an unconditional jump to the `printf()` function instead of function return in «`printf()` with several arguments» section ([1.8.2 on page 54](#)).

`f_unsigned` is similar, only the `ADRHI`, `BLHI`, and `LDMCSFD` instructions are used there, these predicates (*HI* = *Unsigned higher*, *CS* = *Carry Set (greater than or equal)*) are analogous to those examined before, but for unsigned values.

There is not much new in the `main()` function for us:

Listing 1.111: `main()`

```
.text:00000128          EXPORT main
.text:00000128          main
.text:00000128 10 40 2D E9    STMF D SP!, {R4,LR}
.text:0000012C 02 10 A0 E3    MOV    R1, #2
.text:00000130 01 00 A0 E3    MOV    R0, #1
```

<sup>92</sup>[LDMFD](#)

## 1.14. CONDITIONAL JUMPS

```
.text:00000134 DF FF FF EB      BL      f_signed
.text:00000138 02 10 A0 E3      MOV     R1, #2
.text:0000013C 01 00 A0 E3      MOV     R0, #1
.text:00000140 EA FF FF EB      BL      f_unsigned
.text:00000144 00 00 A0 E3      MOV     R0, #0
.text:00000148 10 80 BD E8      LDMFD   SP!, {R4,PC}
.text:00000148                 ; End of function main
```

That is how you can get rid of conditional jumps in ARM mode.

Why is this so good? Read here: [2.9.1 on page 465](#).

There is no such feature in x86, except the `CMOVcc` instruction, it is the same as `MOV`, but triggered only when specific flags are set, usually set by `CMP`.

## Optimizing Keil 6/2013 (Thumb mode)

Listing 1.112: Optimizing Keil 6/2013 (Thumb mode)

```
.text:00000072          f_signed ; CODE XREF: main+6
.text:00000072 70 B5      PUSH    {R4-R6,LR}
.text:00000074 0C 00      MOVS    R4, R1
.text:00000076 05 00      MOVS    R5, R0
.text:00000078 A0 42      CMP     R0, R4
.text:0000007A 02 DD      BLE    loc_82
.text:0000007C A4 A0      ADR     R0, aAB           ; "a>b\n"
.text:0000007E 06 F0 B7 F8  BL      __2printf
.text:00000082
.loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42      CMP     R5, R4
.text:00000084 02 D1      BNE    loc_8C
.text:00000086 A4 A0      ADR     R0, aAB_0         ; "a==b\n"
.text:00000088 06 F0 B2 F8  BL      __2printf
.text:0000008C
.loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42      CMP     R5, R4
.text:0000008E 02 DA      BGE    locret_96
.text:00000090 A3 A0      ADR     R0, aAB_1         ; "a<b\n"
.text:00000092 06 F0 AD F8  BL      __2printf
.text:00000096
.locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD      POP    {R4-R6,PC}
.text:00000096 ; End of function f_signed
```

Only `B` instructions in Thumb mode may be supplemented by *condition codes*, so the Thumb code looks more ordinary.

`BLE` is a normal conditional jump *Less than or Equal*, `BNE`—*Not Equal*, `BGE`—*Greater than or Equal*.

`f_unsigned` is similar, only other instructions are used while dealing with unsigned values: `BLS` (*Un-signed lower or same*) and `BCS` (*Carry Set (Greater than or equal)*).

## ARM64: Optimizing GCC (Linaro) 4.9

Listing 1.113: `f_signed()`

```
f_signed:
; W0=a, W1=b
    cmp    w0, w1
    bgt   .L19    ; Branch if Greater Than (a>b)
    beq   .L20    ; Branch if Equal (a==b)
    bge   .L15    ; Branch if Greater than or Equal (a>=b) (impossible here)
; a<b
    add    x0, x0, :lo12:.LC11
    add    x0, x0, :lo12:.LC11
    puts
```

#### 1.14. CONDITIONAL JUMPS

```
.L19:
    adrp    x0, .LC9      ; "a>b"
    add     x0, x0, :lo12:.LC9
    b      puts
.L15: ; impossible to get here
    ret
.L20:
    adrp    x0, .LC10     ; "a==b"
    add     x0, x0, :lo12:.LC10
    b      puts
```

Listing 1.114: f\_unsigned()

```
f_unsigned:
    stp    x29, x30, [sp, -48]!
; W0=a, W1=b
    cmp    w0, w1
    add    x29, sp, 0
    str    x19, [sp,16]
    mov    w19, w0
    bhi   .L25      ; Branch if HIgher (a>b)
    cmp    w19, w1
    beq   .L26      ; Branch if Equal (a==b)
.L23:
    bcc   .L27      ; Branch if Carry Clear (if less than) (a<b)
; function epilogue, impossible to be here
    ldr    x19, [sp,16]
    ldp    x29, x30, [sp], 48
    ret
.L27:
    ldr    x19, [sp,16]
    adrp  x0, .LC11      ; "a<b"
    ldp    x29, x30, [sp], 48
    add    x0, x0, :lo12:.LC11
    b      puts
.L25:
    adrp  x0, .LC9      ; "a>b"
    str   x1, [x29,40]
    add   x0, x0, :lo12:.LC9
    bl    puts
    ldr   x1, [x29,40]
    cmp   w19, w1
    bne  .L23      ; Branch if Not Equal
.L26:
    ldr   x19, [sp,16]
    adrp  x0, .LC10     ; "a==b"
    ldp   x29, x30, [sp], 48
    add   x0, x0, :lo12:.LC10
    b      puts
```

The comments were added by the author of this book. What is striking is that the compiler is not aware that some conditions are not possible at all, so there is dead code at some places, which can never be executed.

#### Exercise

Try to optimize these functions manually for size, removing redundant instructions, without adding new ones.

#### MIPS

One distinctive MIPS feature is the absence of flags. Apparently, it was done to simplify the analysis of data dependencies.

## 1.14. CONDITIONAL JUMPS

There are instructions similar to `SETcc` in x86: `SLT` ("Set on Less Than": signed version) and `SLTU` (unsigned version). These instructions sets destination register value to 1 if the condition is true or to 0 if otherwise.

The destination register is then checked using `BEQ` ("Branch on Equal") or `BNE` ("Branch on Not Equal") and a jump may occur. So, this instruction pair has to be used in MIPS for comparison and branch. Let's first start with the signed version of our function:

Listing 1.115: Non-optimizing GCC 4.4.5 (IDA)

```
.text:00000000 f_signed:                                # CODE XREF: main+18
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000 arg_0           = 0
.text:00000000 arg_4           = 4
.text:00000000
.text:00000000 addiu   $sp, -0x20
.text:00000004 sw      $ra, 0x20+var_4($sp)
.text:00000008 sw      $fp, 0x20+var_8($sp)
.text:0000000C move    $fp, $sp
.text:00000010 la      $gp, __gnu_local_gp
.text:00000018 sw      $gp, 0x20+var_10($sp)
; store input values into local stack:
.text:0000001C sw      $a0, 0x20+arg_0($fp)
.text:00000020 sw      $a1, 0x20+arg_4($fp)
; reload them.
.text:00000024 lw      $v1, 0x20+arg_0($fp)
.text:00000028 lw      $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C or      $at, $zero ; NOP
; this is pseudoinstruction. in fact, "slt $v0,$v0,$v1" is there.
; so $v0 will be set to 1 if $v0<$v1 (b<a) or to 0 if otherwise:
.text:00000030 slt     $v0, $v1
; jump to loc_5c, if condition is not true.
; this is pseudoinstruction. in fact, "beq $v0,$zero,loc_5c" is there:
.text:00000034 beqz   $v0, loc_5C
; print "a>b" and finish
.text:00000038 or      $at, $zero ; branch delay slot, NOP
.text:0000003C lui     $v0, (unk_230 >> 16) # "a>b"
.text:00000040 addiu  $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044 lw      $v0, (puts & 0xFFFF)($gp)
.text:00000048 or      $at, $zero ; NOP
.text:0000004C move    $t9, $v0
.text:00000050 jalr   $t9
.text:00000054 or      $at, $zero ; branch delay slot, NOP
.text:00000058 lw      $gp, 0x20+var_10($fp)
.text:0000005C
.text:0000005C loc_5C:                                # CODE XREF: f_signed+34
.lw      $v1, 0x20+arg_0($fp)
.text:00000060 lw      $v0, 0x20+arg_4($fp)
.text:00000064 or      $at, $zero ; NOP
; check if a==b, jump to loc_90 if its not true':
.text:00000068 bne    $v1, $v0, loc_90
.text:0000006C or      $at, $zero ; branch delay slot, NOP
; condition is true, so print "a==b" and finish:
.text:00000070 lui     $v0, (aAB >> 16) # "a==b"
.text:00000074 addiu  $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000078 lw      $v0, (puts & 0xFFFF)($gp)
.text:0000007C or      $at, $zero ; NOP
.text:00000080 move    $t9, $v0
.text:00000084 jalr   $t9
.text:00000088 or      $at, $zero ; branch delay slot, NOP
.text:0000008C lw      $gp, 0x20+var_10($fp)
.text:00000090
.text:00000090 loc_90:                                # CODE XREF: f_signed+68
.lw      $v1, 0x20+arg_0($fp)
.text:00000094 lw      $v0, 0x20+arg_4($fp)
.text:00000098 or      $at, $zero ; NOP
```

## 1.14. CONDITIONAL JUMPS

```
; check if $v1<$v0 (a<b), set $v0 to 1 if condition is true:
.text:0000009C           slt    $v0, $v1, $v0
; if condition is not true (i.e., $v0==0), jump to loc_c8:
.text:000000A0           beqz  $v0, loc_C8
.text:000000A4           or     $at, $zero ; branch delay slot, NOP
; condition is true, print "a<b" and finish
.text:000000A8           lui    $v0, (aAB_0 >> 16) # "a<b"
.text:000000AC           addiu $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:000000B0           lw     $v0, (puts & 0xFFFF)($gp)
.text:000000B4           or     $at, $zero ; NOP
.text:000000B8           move   $t9, $v0
.text:000000BC           jalr   $t9
.text:000000C0           or     $at, $zero ; branch delay slot, NOP
.text:000000C4           lw     $gp, 0x20+var_10($fp)
.text:000000C8
; all 3 conditions were false, so just finish:
.text:000000C8 loc_C8:                                # CODE XREF: f_signed+A0
.text:000000C8           move   $sp, $fp
.text:000000CC           lw     $ra, 0x20+var_4($sp)
.text:000000D0           lw     $fp, 0x20+var_8($sp)
.text:000000D4           addiu $sp, 0x20
.text:000000D8           jr     $ra
.text:000000DC           or     $at, $zero ; branch delay slot, NOP
.text:000000DC # End of function f_signed
```

SLT REG0, REG0, REG1 is reduced by IDA to its shorter form:

SLT REG0, REG1.

We also see there BEQZ pseudo instruction ("Branch if Equal to Zero"), which are in fact BEQ REG, \$ZERO, LABEL.

The unsigned version is just the same, but SLTU (unsigned version, hence "U" in name) is used instead of SLT:

Listing 1.116: Non-optimizing GCC 4.4.5 (IDA)

```
.text:000000E0 f_unsigned:                                # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10          = -0x10
.text:000000E0 var_8           = -8
.text:000000E0 var_4           = -4
.text:000000E0 arg_0           = 0
.text:000000E0 arg_4           = 4
.text:000000E0
.text:000000E0           addiu  $sp, -0x20
.text:000000E4           sw     $ra, 0x20+var_4($sp)
.text:000000E8           sw     $fp, 0x20+var_8($sp)
.text:000000EC           move   $fp, $sp
.text:000000F0           la     $gp, __gnu_local_gp
.text:000000F8           sw     $gp, 0x20+var_10($sp)
.text:000000FC           sw     $a0, 0x20+arg_0($fp)
.text:00000100           sw     $a1, 0x20+arg_4($fp)
.text:00000104           lw     $v1, 0x20+arg_0($fp)
.text:00000108           lw     $v0, 0x20+arg_4($fp)
.text:0000010C           or     $at, $zero
.text:00000110           sltu   $v0, $v1
.text:00000114           beqz  $v0, loc_13C
.text:00000118           or     $at, $zero
.text:0000011C           lui    $v0, (unk_230 >> 16)
.text:00000120           addiu $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124           lw     $v0, (puts & 0xFFFF)($gp)
.text:00000128           or     $at, $zero
.text:0000012C           move   $t9, $v0
.text:00000130           jalr   $t9
.text:00000134           or     $at, $zero
.text:00000138           lw     $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C:                                # CODE XREF: f_unsigned+34
.text:0000013C           lw     $v1, 0x20+arg_0($fp)
```

## 1.14. CONDITIONAL JUMPS

```
.text:00000140          lw      $v0, 0x20+arg_4($fp)
.text:00000144          or      $at, $zero
.text:00000148          bne   $v1, $v0, loc_170
.text:0000014C          or      $at, $zero
.text:00000150          lui    $v0, (aAB >> 16) # "a==b"
.text:00000154          addiu $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000158          lw     $v0, (puts & 0xFFFF)($gp)
.text:0000015C          or      $at, $zero
.text:00000160          move   $t9, $v0
.text:00000164          jalr   $t9
.text:00000168          or      $at, $zero
.text:0000016C          lw     $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:           # CODE XREF: f_unsigned+68
.text:00000170          lw     $v1, 0x20+arg_0($fp)
.text:00000174          lw     $v0, 0x20+arg_4($fp)
.text:00000178          or      $at, $zero
.text:0000017C          sltu   $v0, $v1, $v0
.text:00000180          beqz   $v0, loc_1A8
.text:00000184          or      $at, $zero
.text:00000188          lui    $v0, (aAB_0 >> 16) # "a<b"
.text:0000018C          addiu $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:00000190          lw     $v0, (puts & 0xFFFF)($gp)
.text:00000194          or      $at, $zero
.text:00000198          move   $t9, $v0
.text:0000019C          jalr   $t9
.text:000001A0          or      $at, $zero
.text:000001A4          lw     $gp, 0x20+var_10($fp)
.text:000001A8
.text:000001A8 loc_1A8:           # CODE XREF: f_unsigned+A0
.text:000001A8          move   $sp, $fp
.text:000001AC          lw     $ra, 0x20+var_4($sp)
.text:000001B0          lw     $fp, 0x20+var_8($sp)
.text:000001B4          addiu $sp, 0x20
.text:000001B8          jr     $ra
.text:000001BC          or      $at, $zero
.text:000001BC # End of function f_unsigned
```

### 1.14.2 Calculating absolute value

A simple function:

```
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
};
```

### Optimizing MSVC

This is how the code is usually generated:

Listing 1.117: Optimizing MSVC 2012 x64

```
i$ = 8
my_abs PROC
; ECX = input
    test    ecx, ecx
; check for sign of input value
; skip NEG instruction if sign is positive
    jns     SHORT $LN2@my_abs
; negate value
    neg    ecx
$LN2@my_abs:
```

#### 1.14. CONDITIONAL JUMPS

```
; prepare result in EAX:  
    mov     eax, ecx  
    ret     0  
my_abs ENDP
```

GCC 4.9 does mostly the same.

#### Optimizing Keil 6/2013: Thumb mode

Listing 1.118: Optimizing Keil 6/2013: Thumb mode

```
my_abs PROC  
    CMP     r0,#0  
; is input value equal to zero or greater than zero?  
; skip RSBS instruction then  
    BGE     |L0.6|  
; subtract input value from 0:  
    RSBS   r0,r0,#0  
|L0.6|  
    BX     lr  
ENDP
```

ARM lacks a negate instruction, so the Keil compiler uses the “Reverse Subtract” instruction, which just subtracts with reversed operands.

#### Optimizing Keil 6/2013: ARM mode

It is possible to add condition codes to some instructions in ARM mode, so that is what the Keil compiler does:

Listing 1.119: Optimizing Keil 6/2013: ARM mode

```
my_abs PROC  
    CMP     r0,#0  
; execute "Reverse Subtract" instruction only if input value is less than 0:  
    RSBLT   r0,r0,#0  
    BX     lr  
ENDP
```

Now there are no conditional jumps and this is good: [2.9.1 on page 465](#).

#### Non-optimizing GCC 4.9 (ARM64)

ARM64 has instruction `NEG` for negating:

Listing 1.120: Optimizing GCC 4.9 (ARM64)

```
my_abs:  
    sub    sp, sp, #16  
    str   w0, [sp,12]  
    ldr   w0, [sp,12]  
; compare input value with contents of WZR register  
; (which always holds zero)  
    cmp    w0, wzr  
    bge   .L2  
    ldr   w0, [sp,12]  
    neg   w0, w0  
    b     .L3  
.L2:  
    ldr   w0, [sp,12]  
.L3:  
    add   sp, sp, 16  
    ret
```

Listing 1.121: Optimizing GCC 4.4.5 (IDA)

```

my_abs:
; jump if $a0<0:
    bltz    $a0, locret_10
; just return input value ($a0) in $v0:
    move    $v0, $a0
    jr     $ra
    or     $at, $zero ; branch delay slot, NOP
locret_10:
; negate input value and store it in $v0:
    jr     $ra
; this is pseudoinstruction. in fact, this is "subu $v0,$zero,$a0" ($v0=0-$a0)
    negu    $v0, $a0

```

Here we see a new instruction: `BLTZ` (“Branch if Less Than Zero”).

There is also the `NEGU` pseudo instruction, which just does subtraction from zero. The “U” suffix in both `SUBU` and `NEGU` implies that no exception to be raised in case of integer overflow.

### Branchless version?

You could have also a branchless version of this code. This we will review later: [3.13 on page 517](#).

### 1.14.3 Ternary conditional operator

The ternary conditional operator in C/C++ has the following syntax:

```
expression ? expression : expression
```

Here is an example:

```

const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
}

```

### x86

Old and non-optimizing compilers generate assembly code just as if an `if/else` statement was used:

Listing 1.122: Non-optimizing MSVC 2008

```

$SG746 DB      'it is ten', 00H
$SG747 DB      'it is not ten', 00H

tv65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
; compare input value with 10
    cmp     DWORD PTR _a$[ebp], 10
; jump to $LN3@f if not equal
    jne    SHORT $LN3@f
; store pointer to the string into temporary variable:
    mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; jump to exit
    jmp    SHORT $LN4@f
$LN3@f:
; store pointer to the string into temporary variable:

```

#### 1.14. CONDITIONAL JUMPS

```
    mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; this is exit. copy pointer to the string from temporary variable to EAX.
    mov     eax, DWORD PTR tv65[ebp]
    mov     esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

Listing 1.123: Optimizing MSVC 2008

```
$SG792 DB      'it is ten', 00H
$SG793 DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f    PROC
; compare input value with 10
    cmp     DWORD PTR _a$[esp-4], 10
    mov     eax, OFFSET $SG792 ; 'it is ten'
; jump to $LN4@f if equal
    je     SHORT $LN4@f
    mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
    ret    0
_f    ENDP
```

Newer compilers are more concise:

Listing 1.124: Optimizing MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
_f    PROC
; load pointers to the both strings
    lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
    lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; compare input value with 10
    cmp     ecx, 10
; if equal, copy value from RDX ("it is ten")
; if not, do nothing. pointer to the string "it is not ten" is still in RAX as for now.
    cmove   rax, rdx
    ret    0
_f    ENDP
```

Optimizing GCC 4.8 for x86 also uses the `CMOVcc` instruction, while the non-optimizing GCC 4.8 uses conditional jumps.

#### ARM

Optimizing Keil for ARM mode also uses the conditional instructions `ADRcc`:

Listing 1.125: Optimizing Keil 6/2013 (ARM mode)

```
f PROC
; compare input value with 10
    CMP    r0,#0xa
; if comparison result is Equal, copy pointer to the "it is ten" string into R0
    ADREQ  r0,|L0.16| ; "it is ten"
; if comparison result is Not Equal, copy pointer to the "it is not ten" string into R0
    ADRNE  r0,|L0.28| ; "it is not ten"
    BX    lr
    ENDP

|L0.16|
    DCB    "it is ten",0
|L0.28|
    DCB    "it is not ten",0
```

#### 1.14. CONDITIONAL JUMPS

Without manual intervention, the two instructions `ADREQ` and `ADRNE` cannot be executed in the same run.

Optimizing Keil for Thumb mode needs to use conditional jump instructions, since there are no load instructions that support conditional flags:

Listing 1.126: Optimizing Keil 6/2013 (Thumb mode)

```
f PROC
; compare input value with 10
    CMP      r0,#0xa
; jump to |L0.8| if Equal
    BEQ      |L0.8|
    ADR      r0,|L0.12| ; "it is not ten"
    BX       lr
|L0.8|
    ADR      r0,|L0.28| ; "it is ten"
    BX       lr
ENDP

|L0.12|
    DCB      "it is not ten",0
|L0.28|
    DCB      "it is ten",0
```

#### ARM64

Optimizing GCC (Linaro) 4.9 for ARM64 also uses conditional jumps:

Listing 1.127: Optimizing GCC (Linaro) 4.9

```
f:
    cmp      x0, 10
    beq      .L3          ; branch if equal
    adrp    x0, .LC1        ; "it is ten"
    add     x0, x0, :lo12:.LC1
    ret

.L3:
    adrp    x0, .LC0        ; "it is not ten"
    add     x0, x0, :lo12:.LC0
    ret

.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
```

That is because ARM64 does not have a simple load instruction with conditional flags, like `ADRcc` in 32-bit ARM mode or `CMOVcc` in x86.

It has, however, “Conditional SELect” instruction (`CSEL`) [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)p390, C5.5], but GCC 4.9 does not seem to be smart enough to use it in such piece of code.

#### MIPS

Unfortunately, GCC 4.4.5 for MIPS is not very smart, either:

Listing 1.128: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
    .ascii  "it is not ten\000"
$LC1:
    .ascii  "it is ten\000"
f:
    li      $2,10           # 0xa
; compare $a0 and 10, jump if equal:
    beq    $4,$2,$L2
```

## 1.14. CONDITIONAL JUMPS

```
nop ; branch delay slot

; leave address of "it is not ten" string in $v0 and return:
    lui      $2,%hi($LC0)
    j       $31
    addiu   $2,$2,%lo($LC0)

$L2:
; leave address of "it is ten" string in $v0 and return:
    lui      $2,%hi($LC1)
    j       $31
    addiu   $2,$2,%lo($LC1)
```

Let's rewrite it in an **if/else** way

```
const char* f (int a)
{
    if (a==10)
        return "it is ten";
    else
        return "it is not ten";
};
```

Interestingly, optimizing GCC 4.8 for x86 was also able to use **CMOVcc** in this case:

Listing 1.129: Optimizing GCC 4.8

```
.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
f:
.LFB0:
; compare input value with 10
    cmp     DWORD PTR [esp+4], 10
    mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
    mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; if comparison result is Not Equal, copy EDX value to EAX
; if not, do nothing
    cmovne eax, edx
    ret
```

Optimizing Keil in ARM mode generates code identical to listing [1.125](#).

But the optimizing MSVC 2012 is not that good (yet).

## Conclusion

Why optimizing compilers try to get rid of conditional jumps? Read here about it: [2.9.1 on page 465](#).

### 1.14.4 Getting minimal and maximal values

#### 32-bit

```
int my_max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
};

int my_min(int a, int b)
```

## 1.14. CONDITIONAL JUMPS

```
{  
    if (a<b)  
        return a;  
    else  
        return b;  
};
```

Listing 1.130: Non-optimizing MSVC 2013

```
_a$ = 8  
_b$ = 12  
_my_min PROC  
    push    ebp  
    mov     ebp, esp  
    mov     eax, DWORD PTR _a$[ebp]  
; compare A and B:  
    cmp     eax, DWORD PTR _b$[ebp]  
; jump, if A is greater or equal to B:  
    jge     SHORT $LN2@my_min  
; reload A to EAX if otherwise and jump to exit  
    mov     eax, DWORD PTR _a$[ebp]  
    jmp     SHORT $LN3@my_min  
    jmp     SHORT $LN3@my_min ; this is redundant JMP  
$LN2@my_min:  
; return B  
    mov     eax, DWORD PTR _b$[ebp]  
$LN3@my_min:  
    pop    ebp  
    ret    0  
_my_min ENDP  
  
_a$ = 8  
_b$ = 12  
_my_max PROC  
    push    ebp  
    mov     ebp, esp  
    mov     eax, DWORD PTR _a$[ebp]  
; compare A and B:  
    cmp     eax, DWORD PTR _b$[ebp]  
; jump if A is less or equal to B:  
    jle     SHORT $LN2@my_max  
; reload A to EAX if otherwise and jump to exit  
    mov     eax, DWORD PTR _a$[ebp]  
    jmp     SHORT $LN3@my_max  
    jmp     SHORT $LN3@my_max ; this is redundant JMP  
$LN2@my_max:  
; return B  
    mov     eax, DWORD PTR _b$[ebp]  
$LN3@my_max:  
    pop    ebp  
    ret    0  
_my_max ENDP
```

These two functions differ only in the conditional jump instruction: **JGE** ("Jump if Greater or Equal") is used in the first one and **JLE** ("Jump if Less or Equal") in the second.

There is one unneeded **JMP** instruction in each function, which MSVC presumably left by mistake.

### Branchless

ARM for Thumb mode reminds us of x86 code:

Listing 1.131: Optimizing Keil 6/2013 (Thumb mode)

```
my_max PROC  
; R0=A  
; R1=B  
; compare A and B:
```

#### 1.14. CONDITIONAL JUMPS

```

        CMP      r0,r1
; branch if A is greater than B:
        BGT      |L0.6|
; otherwise (A<=B) return R1 (B):
        MOVS    r0,r1
|L0.6|
; return
        BX      lr
        ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
        CMP      r0,r1
; branch if A is less than B:
        BLT      |L0.14|
; otherwise (A>=B) return R1 (B):
        MOVS    r0,r1
|L0.14|
; return
        BX      lr
        ENDP

```

The functions differ in the branching instruction: `BGT` and `BLT`. It's possible to use conditional suffixes in ARM mode, so the code is shorter.

`MOVcc` is to be executed only if the condition is met:

Listing 1.132: Optimizing Keil 6/2013 (ARM mode)

```

my_max PROC
; R0=A
; R1=B
; compare A and B:
        CMP      r0,r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A<=B (hence, LE - Less or Equal)
; if instruction is not triggered (in case of A>B), A is still in R0 register
        MOVLE    r0,r1
        BX      lr
        ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
        CMP      r0,r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A>=B (hence, GE - Greater or Equal)
; if instruction is not triggered (in case of A<B), A value is still in R0 register
        MOVGE    r0,r1
        BX      lr
        ENDP

```

Optimizing GCC 4.8.1 and optimizing MSVC 2013 can use `CMOVcc` instruction, which is analogous to `MOVcc` in ARM:

Listing 1.133: Optimizing MSVC 2013

```

my_max:
        mov      edx, DWORD PTR [esp+4]
        mov      eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
        cmp      edx, eax
; if A>=B, load A value into EAX
; the instruction idle if otherwise (if A<B)
        cmovge  eax, edx

```

## 1.14. CONDITIONAL JUMPS

```
    ret

my_min:
    mov     edx, DWORD PTR [esp+4]
    mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
    cmp     edx, eax
; if A<=B, load A value into EAX
; the instruction idle if otherwise (if A>B)
    cmovle eax, edx
    ret
```

### 64-bit

```
#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
    if (a>b)
        return a;
    else
        return b;
};

int64_t my_min(int64_t a, int64_t b)
{
    if (a<b)
        return a;
    else
        return b;
};
```

There is some unneeded value shuffling, but the code is comprehensible:

Listing 1.134: Non-optimizing GCC 4.9.1 ARM64

```
my_max:
    sub    sp, sp, #16
    str    x0, [sp,8]
    str    x1, [sp]
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    cmp    x1, x0
    ble   .L2
    ldr    x0, [sp,8]
    b     .L3
.L2:
    ldr    x0, [sp]
.L3:
    add    sp, sp, 16
    ret

my_min:
    sub    sp, sp, #16
    str    x0, [sp,8]
    str    x1, [sp]
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    cmp    x1, x0
    bge   .L5
    ldr    x0, [sp,8]
    b     .L6
.L5:
    ldr    x0, [sp]
.L6:
    add    sp, sp, 16
```

## 1.14. CONDITIONAL JUMPS

```
ret
```

### Branchless

No need to load function arguments from the stack, as they are already in the registers:

Listing 1.135: Optimizing GCC 4.9.1 x64

```
my_max:  
; RDI=A  
; RSI=B  
; compare A and B:  
    cmp    rdi, rsi  
; prepare B in RAX for return:  
    mov    rax, rsi  
; if A>=B, put A (RDI) in RAX for return.  
; this instruction is idle if otherwise (if A<B)  
    cmovge rax, rdi  
    ret  
  
my_min:  
; RDI=A  
; RSI=B  
; compare A and B:  
    cmp    rdi, rsi  
; prepare B in RAX for return:  
    mov    rax, rsi  
; if A<=B, put A (RDI) in RAX for return.  
; this instruction is idle if otherwise (if A>B)  
    cmovle rax, rdi  
    ret
```

MSVC 2013 does almost the same.

ARM64 has the `CSEL` instruction, which works just as `M0Vcc` in ARM or `CM0Vcc` in x86, just the name is different: “Conditional SElect”.

Listing 1.136: Optimizing GCC 4.9.1 ARM64

```
my_max:  
; X0=A  
; X1=B  
; compare A and B:  
    cmp    x0, x1  
; select X0 (A) to X0 if X0>=X1 or A>=B (Greater or Equal)  
; select X1 (B) to X0 if A<B  
    csel   x0, x0, x1, ge  
    ret  
  
my_min:  
; X0=A  
; X1=B  
; compare A and B:  
    cmp    x0, x1  
; select X0 (A) to X0 if X0<=X1 or A<=B (Less or Equal)  
; select X1 (B) to X0 if A>B  
    csel   x0, x0, x1, le  
    ret
```

### MIPS

Unfortunately, GCC 4.4.5 for MIPS is not that good:

Listing 1.137: Optimizing GCC 4.4.5 (IDA)

```
my_max:  
; set $v1 to 1 if $a1<$a0, or clear otherwise (if $a1>$a0):
```

#### 1.14. CONDITIONAL JUMPS

```
        slt      $v1, $a1, $a0
; jump, if $v1 is 0 (or $a1>$a0):
        beqz    $v1, locret_10
; this is branch delay slot
; prepare $a1 in $v0 in case of branch triggered:
        move    $v0, $a1
; no branch triggered, prepare $a0 in $v0:
        move    $v0, $a0

locret_10:
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP

; the min() function is same, but input operands in SLT instruction are swapped:
my_min:
        slt      $v1, $a0, $a1
        beqz    $v1, locret_28
        move    $v0, $a1
        move    $v0, $a0

locret_28:
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP
```

Do not forget about the *branch delay slots*: the first `MOVE` is executed *before* `BEQZ`, the second `MOVE` is executed only if the branch hasn't been taken.

#### 1.14.5 Conclusion

##### x86

Here's the rough skeleton of a conditional jump:

Listing 1.138: x86

```
CMP register, register/value
Jcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

##### ARM

Listing 1.139: ARM

```
CMP register, register/value
Bcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

##### MIPS

Listing 1.140: Check for zero

```
BEQZ REG, label
...
```

## 1.14. CONDITIONAL JUMPS

Listing 1.141: Check for less than zero using pseudoinstruction

```
BLTZ REG, label  
...
```

Listing 1.142: Check for equal values

```
BEQ REG1, REG2, label  
...
```

Listing 1.143: Check for non-equal values

```
BNE REG1, REG2, label  
...
```

Listing 1.144: Check for less than (signed)

```
SLT REG1, REG2, REG3  
BEQ REG1, label  
...
```

Listing 1.145: Check for less than (unsigned)

```
SLTU REG1, REG2, REG3  
BEQ REG1, label  
...
```

## Branchless

If the body of a condition statement is very short, the conditional move instruction can be used: `MOVcc` in ARM (in ARM mode), `CSEL` in ARM64, `CMOVcc` in x86.

## ARM

It's possible to use conditional suffixes in ARM mode for some instructions:

Listing 1.146: ARM (ARM mode)

```
CMP register, register/value  
instr1_cc ; some instruction will be executed if condition code is true  
instr2_cc ; some other instruction will be executed if other condition code is true  
... etc...
```

Of course, there is no limit for the number of instructions with conditional code suffixes, as long as the CPU flags are not modified by any of them.

Thumb mode has the `IT` instruction, allowing to add conditional suffixes to the next four instructions. Read more about it: [1.19.7 on page 263](#).

Listing 1.147: ARM (Thumb mode)

```
CMP register, register/value  
ITEEE EQ ; set these suffixes: if-then-else-else-else  
instr1 ; instruction will be executed if condition is true  
instr2 ; instruction will be executed if condition is false  
instr3 ; instruction will be executed if condition is false  
instr4 ; instruction will be executed if condition is false
```

## 1.14.6 Exercise

(ARM64) Try rewriting the code in listing [1.127](#) by removing all conditional jump instructions and using the `CSEL` instruction.

## 1.15 switch()/case/default

### 1.15.1 Small number of cases

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        default: printf ("something unknown\n"); break;
    };
}

int main()
{
    f (2); // test
}
```

### x86

#### Non-optimizing MSVC

Result (MSVC 2010):

Listing 1.148: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 0
    je      SHORT $LN4@f
    cmp     DWORD PTR tv64[ebp], 1
    je      SHORT $LN3@f
    cmp     DWORD PTR tv64[ebp], 2
    je      SHORT $LN2@f
    jmp     SHORT $LN1@f
$LN4@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN7@f
$LN3@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN7@f
$LN2@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN7@f
$LN1@f:
    push    OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call    _printf
    add    esp, 4
$LN7@f:
    mov     esp, ebp
```

## 1.15. SWITCH()/CASE/DEFAULT

```
pop    ebp  
ret    0  
_f    ENDP
```

Our function with a few cases in switch() is in fact analogous to this construction:

```
void f (int a)  
{  
    if (a==0)  
        printf ("zero\n");  
    else if (a==1)  
        printf ("one\n");  
    else if (a==2)  
        printf ("two\n");  
    else  
        printf ("something unknown\n");  
};
```

If we work with switch() with a few cases it is impossible to be sure if it was a real switch() in the source code, or just a pack of if() statements.

This implies that switch() is like syntactic sugar for a large number of nested if()s.

There is nothing especially new to us in the generated code, with the exception of the compiler moving input variable *a* to a temporary local variable `tv64` <sup>93</sup>.

If we compile this in GCC 4.4.1, we'll get almost the same result, even with maximal optimization turned on (`-O3` option).

## Optimizing MSVC

Now let's turn on optimization in MSVC (`/Ox`): `cl 1.c /Fa1.asm /Ox`

Listing 1.149: MSVC

```
_a$ = 8 ; size = 4  
_f    PROC  
    mov    eax, DWORD PTR _a$[esp-4]  
    sub    eax, 0  
    je     SHORT $LN4@f  
    sub    eax, 1  
    je     SHORT $LN3@f  
    sub    eax, 1  
    je     SHORT $LN2@f  
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H  
    jmp    _printf  
$LN2@f:  
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H  
    jmp    _printf  
$LN3@f:  
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H  
    jmp    _printf  
$LN4@f:  
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H  
    jmp    _printf  
_f    ENDP
```

Here we can see some dirty hacks.

First: the value of *a* is placed in `EAX` and 0 is subtracted from it. Sounds absurd, but it is done to check if the value in `EAX` is 0. If yes, the `ZF` flag is to be set (e.g. subtracting from 0 is 0) and the first conditional jump `JE` (*Jump if Equal* or synonym `JZ` —*Jump if Zero*) is to be triggered and control flow is to be passed to the `$LN4@f` label, where the `'zero'` message is being printed. If the first jump doesn't get triggered, 1 is subtracted from the input value and if at some stage the result is 0, the corresponding jump is to be triggered.

<sup>93</sup>Local variables in stack are prefixed with `tv` —that's how MSVC names internal variables for its needs

## 1.15. SWITCH()/CASE/DEFAULT

And if no jump gets triggered at all, the control flow passes to `printf()` with string argument '`'something unknown'`'.

Second: we see something unusual for us: a string pointer is placed into the `a` variable, and then `printf()` is called not via `CALL`, but via `JMP`. There is a simple explanation for that: the `caller` pushes a value to the stack and calls our function via `CALL`. `CALL` itself pushes the return address (`RA`) to the stack and does an unconditional jump to our function address. Our function at any point of execution (since it does not contain any instruction that moves the stack pointer) has the following stack layout:

- `ESP`—points to `RA`
- `ESP+4`—points to the `a` variable

On the other side, when we have to call `printf()` here we need exactly the same stack layout, except for the first `printf()` argument, which needs to point to the string. And that is what our code does.

It replaces the function's first argument with the address of the string and jumps to `printf()`, as if we didn't call our function `f()`, but directly `printf()`. `printf()` prints a string to `stdout` and then executes the `RET` instruction, which `POPs RA` from the stack and control flow is returned not to `f()` but rather to `f()`'s `caller`, bypassing the end of the `f()` function.

All this is possible because `printf()` is called right at the end of the `f()` function in all cases. In some way, it is similar to the `longjmp()`<sup>94</sup> function. And of course, it is all done for the sake of speed.

A similar case with the ARM compiler is described in “`printf()` with several arguments” section, here ([1.8.2 on page 54](#)).

---

<sup>94</sup>[wikipedia](#)

## 1.15. SWITCH()/CASE/DEFAULT

### OllyDbg

Since this example is tricky, let's trace it in OllyDbg.

OllyDbg can detect such switch() constructs, and it can add some useful comments. **EAX** is 2 at the beginning, that's the function's input value:

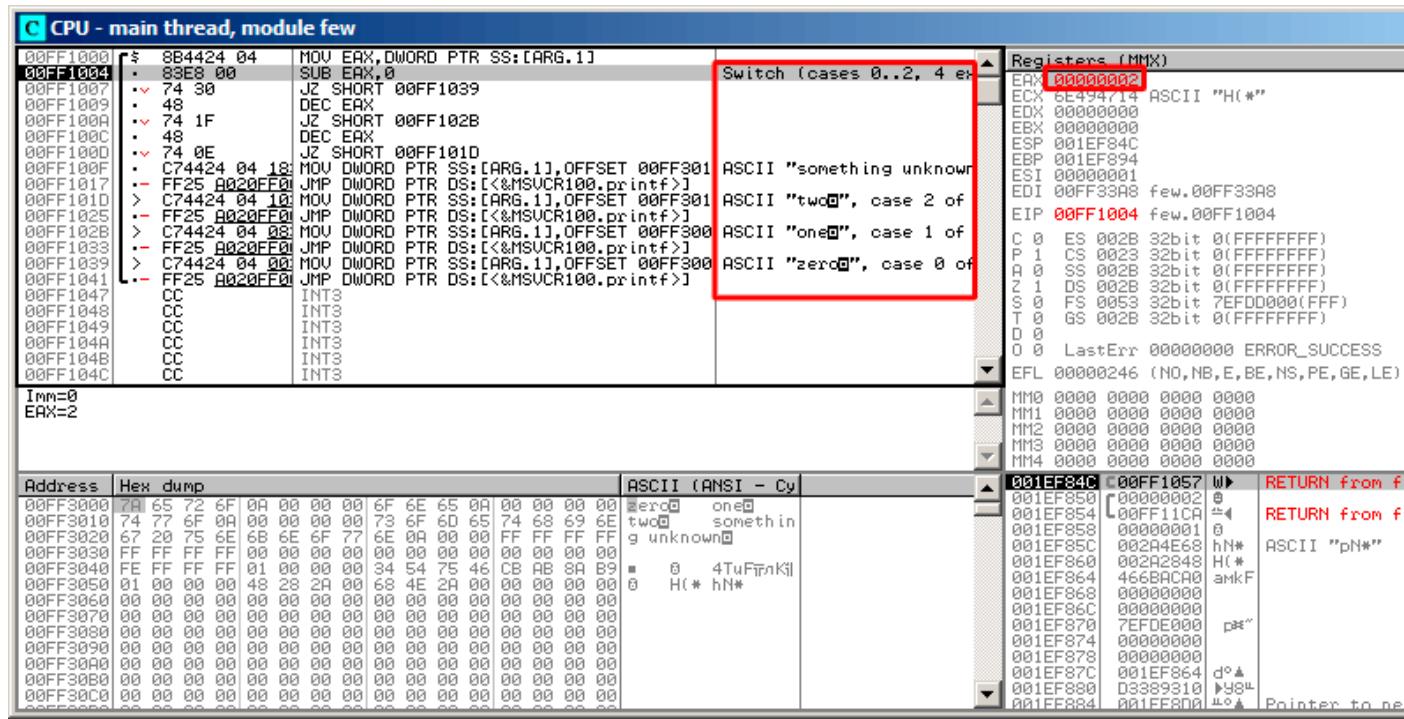


Figure 1.42: OllyDbg: **EAX** now contain the first (and only) function argument

## 1.15. SWITCH()/CASE/DEFAULT

0 is subtracted from 2 in **EAX**. Of course, **EAX** still contains 2. But the **ZF** flag is now 0, indicating that the resulting value is non-zero:

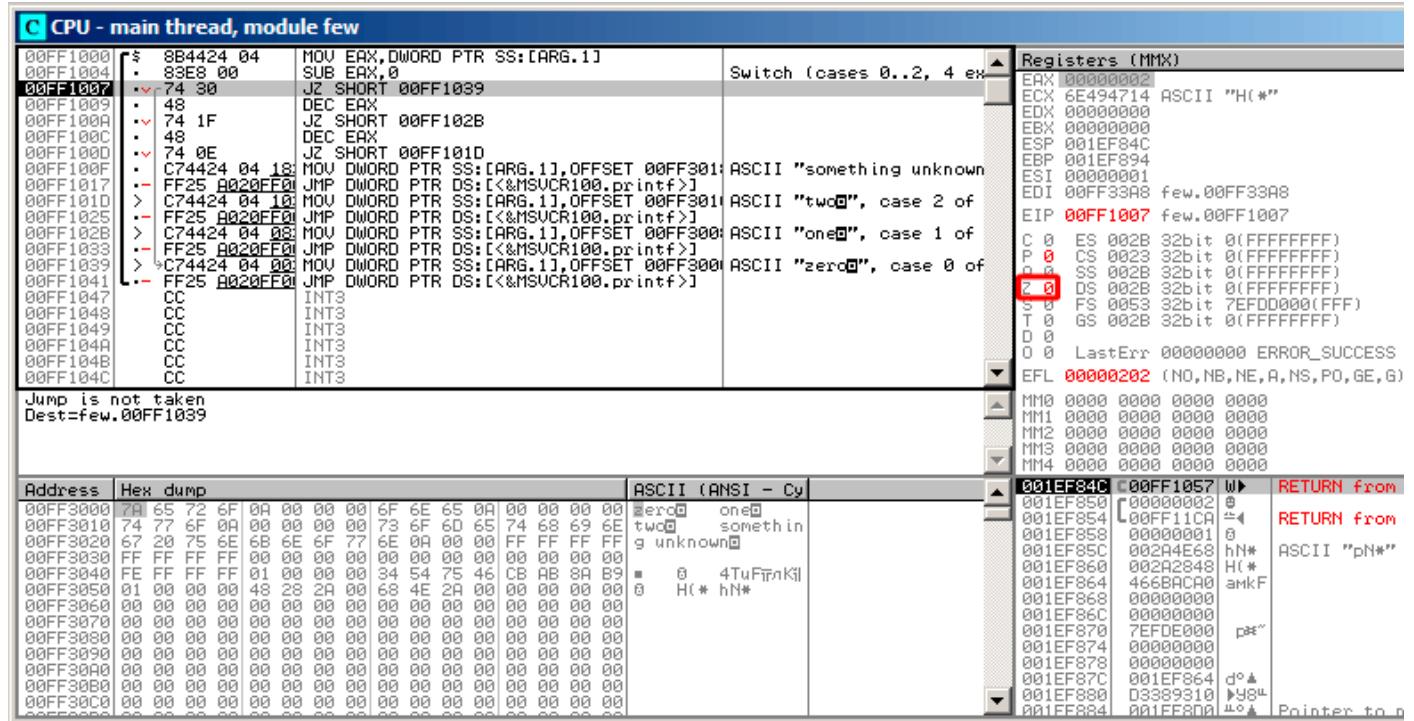


Figure 1.43: OllyDbg: **SUB** executed

## 1.15. SWITCH()/CASE/DEFAULT

DEC is executed and EAX now contains 1. But 1 is non-zero, so the ZF flag is still 0:

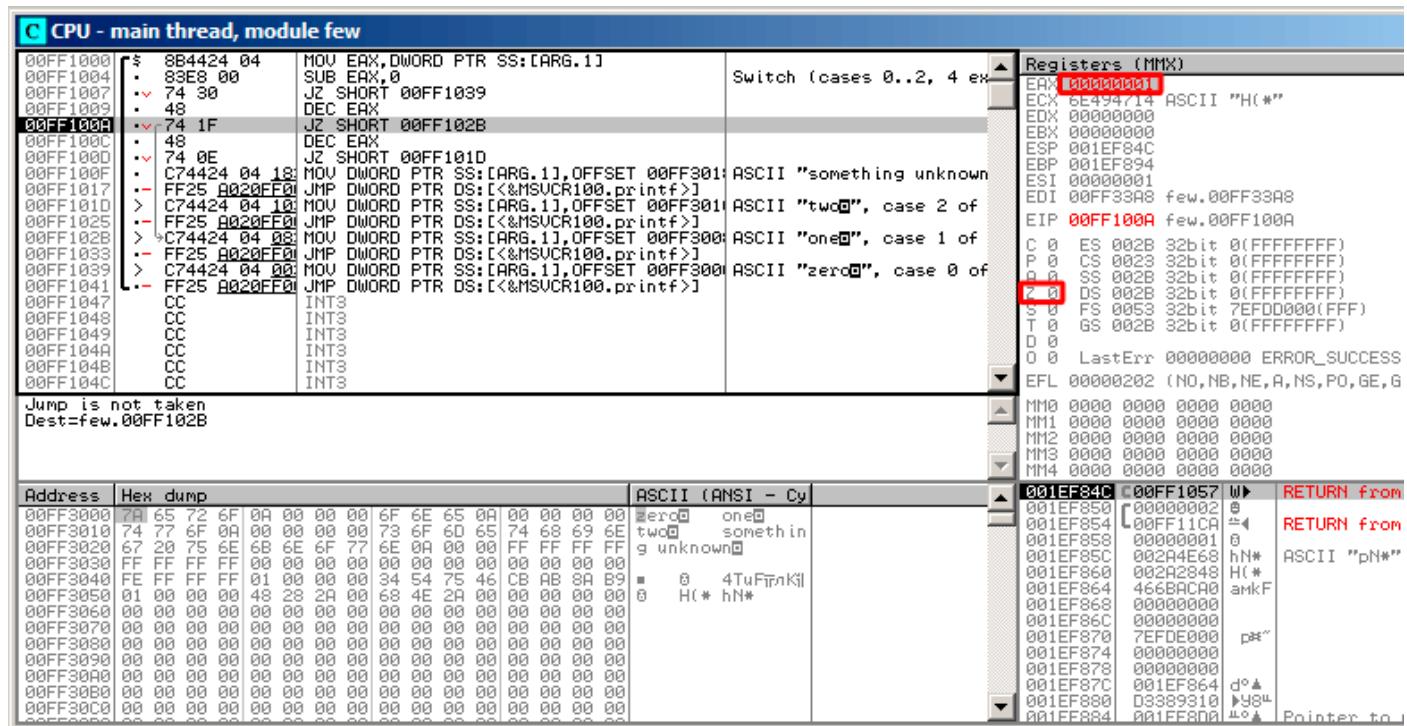


Figure 1.44: OllyDbg: first DEC executed

## 1.15. SWITCH()/CASE/DEFAULT

Next DEC is executed. EAX is finally 0 and the ZF flag gets set, because the result is zero:

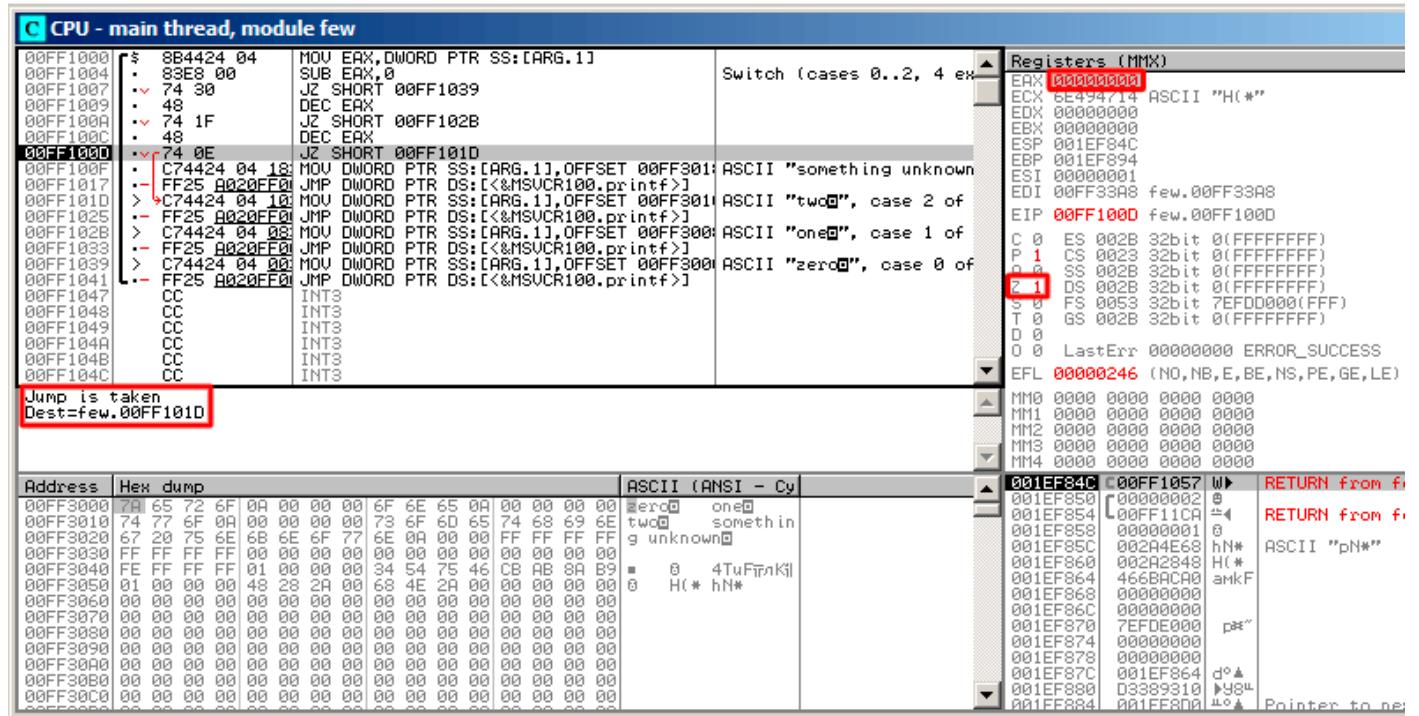


Figure 1.45: OllyDbg: second DEC executed

OllyDbg shows that this jump is to be taken now.

## 1.15. SWITCH()/CASE/DEFAULT

A pointer to the string “two” is to be written into the stack now:

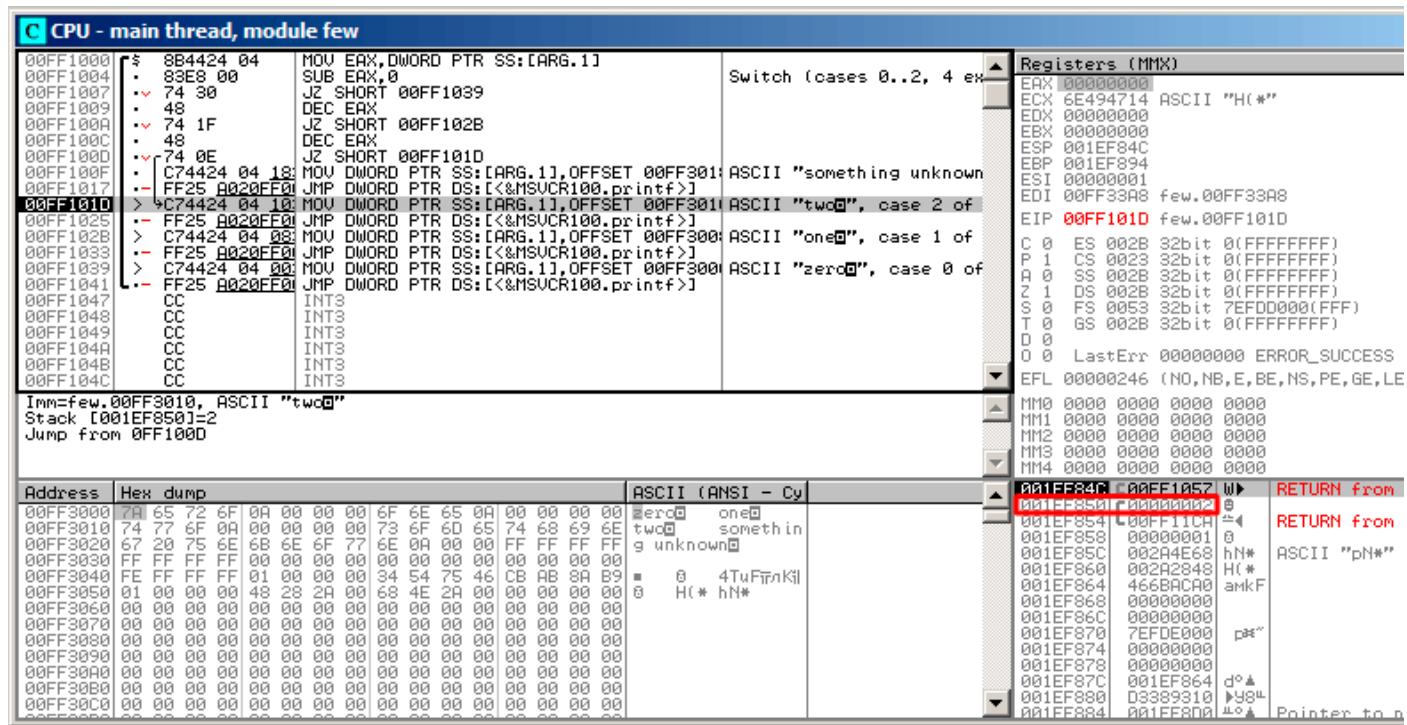


Figure 1.46: OllyDbg: pointer to the string is to be written at the place of the first argument

Please note: the current argument of the function is 2 and 2 is now in the stack at the address `0x001EF850`.

## 1.15. SWITCH()/CASE/DEFAULT

`MOV` writes the pointer to the string at address `0x001EF850` (see the stack window). Then, jump happens. This is the first instruction of the `printf()` function in MSVCR100.DLL (This example was compiled with /MD switch):

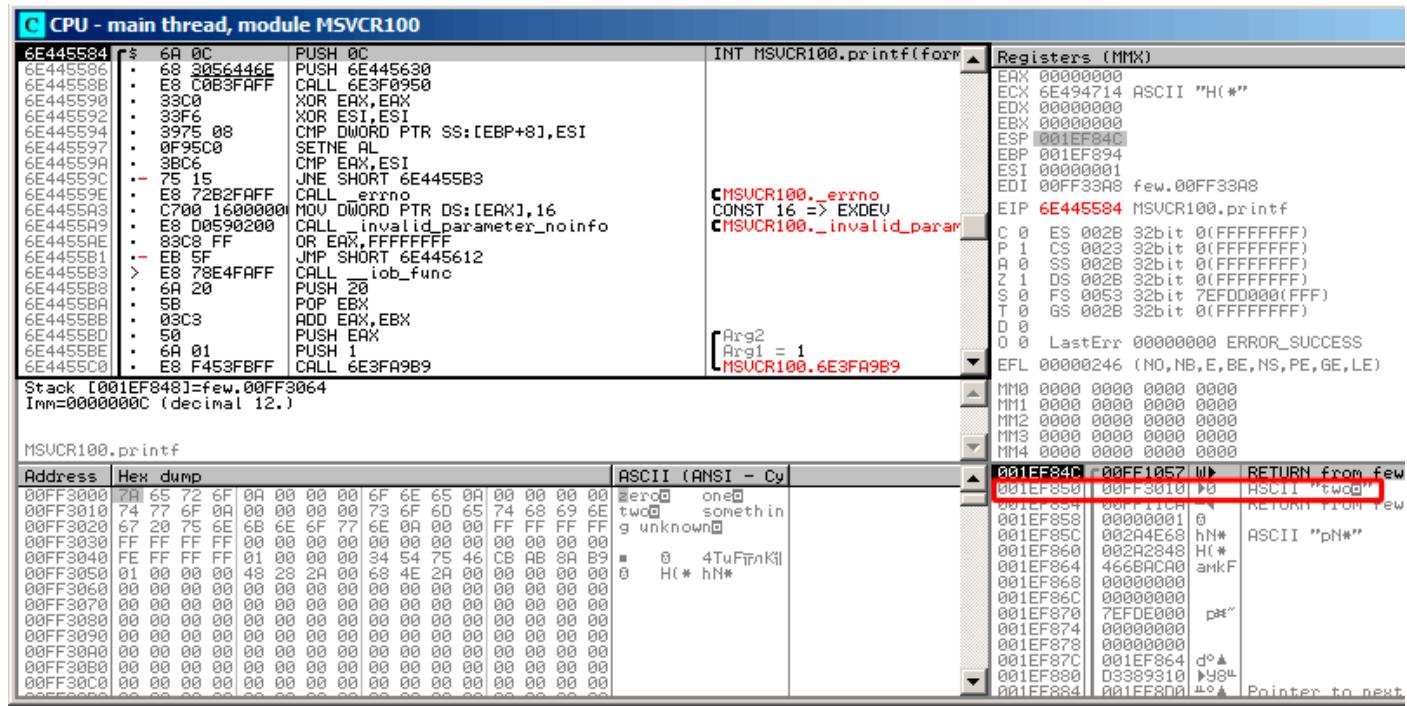


Figure 1.47: OllyDbg: first instruction of `printf()` in MSVCR100.DLL

Now `printf()` treats the string at `0x00FF3010` as its only argument and prints the string.

## 1.15. SWITCH()/CASE/DEFAULT

This is the last instruction of `printf()`:

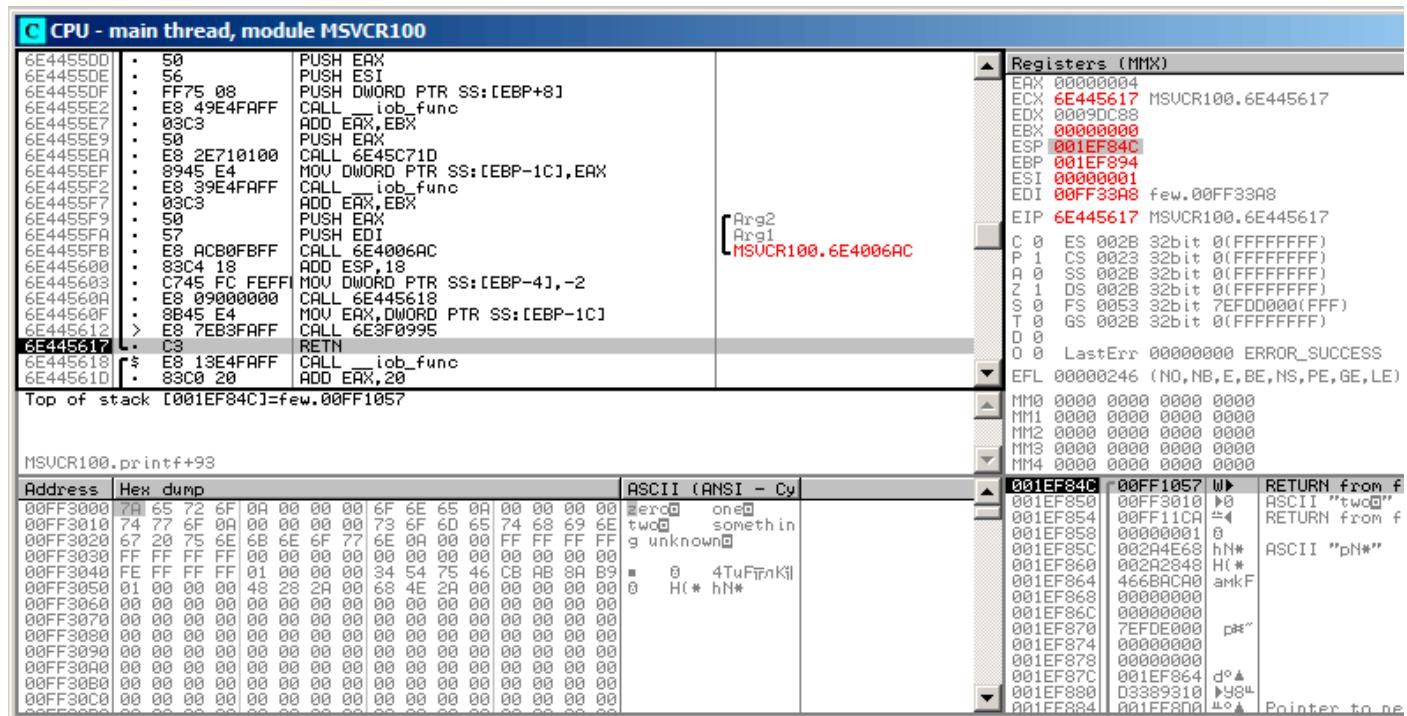


Figure 1.48: OllyDbg: last instruction of `printf()` in MSVCR100.DLL

The string “two” has just been printed to the console window.

## 1.15. SWITCH()/CASE/DEFAULT

Now let's press F7 or F8 (step over) and return...not to `f()`, but rather to `main()`:

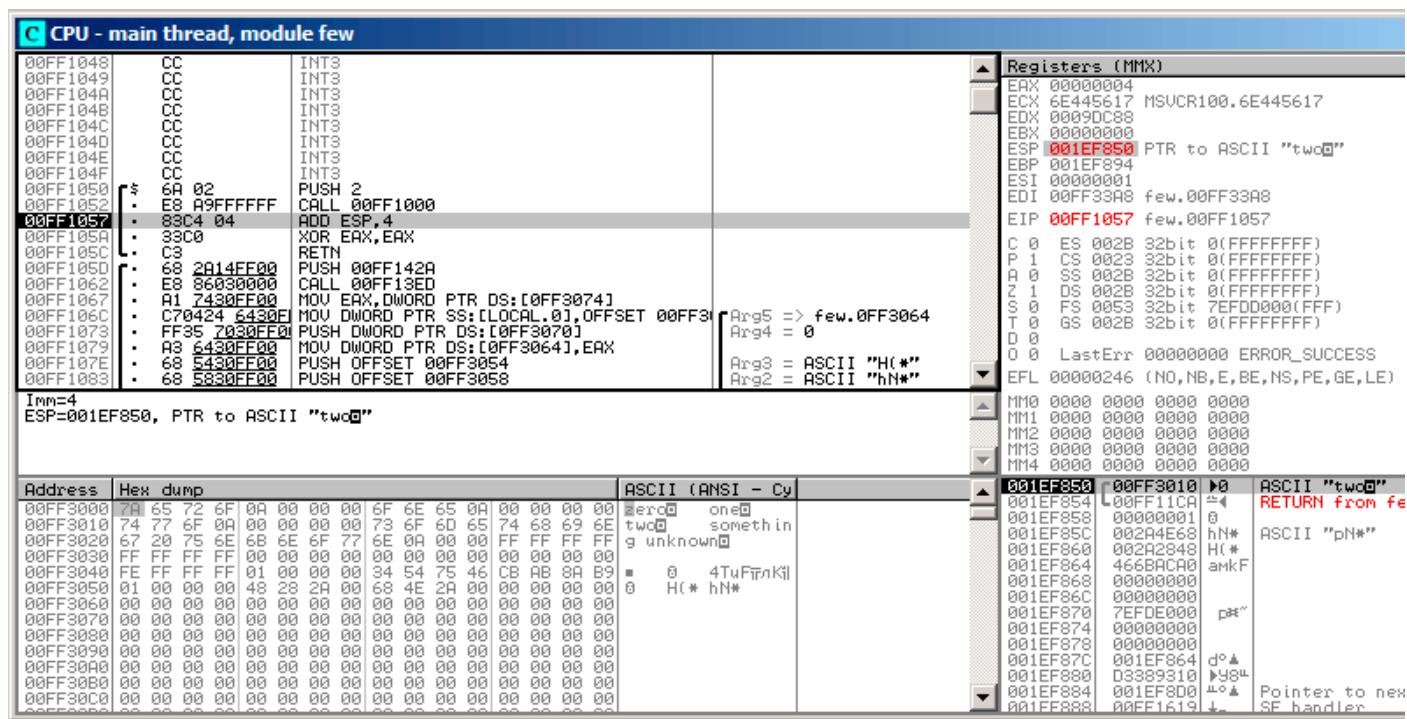


Figure 1.49: OllyDbg: return to `main()`

Yes, the jump has been direct, from the guts of `printf()` to `main()`. Because RA in the stack points not to some place in `f()`, but rather to `main()`. And `CALL 0x00FF1000` has been the actual instruction which called `f()`.

## ARM: Optimizing Keil 6/2013 (ARM mode)

```
.text:0000014C          f1:
.text:0000014C 00 00 50 E3  CMP    R0, #0
.text:00000150 13 0E 8F 02  ADREQ R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A  BEQ    loc_170
.text:00000158 01 00 50 E3  CMP    R0, #1
.text:0000015C 4B 0F 8F 02  ADREQ R0, aOne ; "one\n"
.text:00000160 02 00 00 0A  BEQ    loc_170
.text:00000164 02 00 50 E3  CMP    R0, #2
.text:00000168 4A 0F 8F 12  ADRNE  R0, aSomethingUnkn ; "something unknown\n"
.text:0000016C 4E 0F 8F 02  ADREQ R0, aTwo ; "two\n"
.text:00000170
.loc_170: ; CODE XREF: f1+8
.loc_170: ; f1+14
.text:00000170 78 18 00 EA  B     _2printf
```

Again, by investigating this code we cannot say if it was a `switch()` in the original source code, or just a pack of `if()` statements.

Anyway, we see here predicated instructions again (like `ADREQ (Equal)`) which is triggered only in case  $R0 = 0$ , and then loads the address of the string «zero\n» into `R0`. The next instruction `BEQ` redirects control flow to `loc_170`, if  $R0 = 0$ .

An astute reader may ask, will `BEQ` trigger correctly since `ADREQ` it has already filled the `R0` register before with another value?

Yes, it will since `BEQ` checks the flags set by the `CMP` instruction, and `ADREQ` does not modify any flags at all.

## 1.15. SWITCH()/CASE/DEFAULT

The rest of the instructions are already familiar to us. There is only one call to `printf()`, at the end, and we have already examined this trick here ([1.8.2 on page 54](#)). At the end, there are three paths to `printf()`.

The last instruction, `CMP R0, #2`, is needed to check if  $a = 2$ .

If it is not true, then `ADRNE` loads a pointer to the string «*something unknown \n*» into `R0`, since  $a$  has already been checked to be equal to 0 or 1, and we can sure that the  $a$  variable is not equal to these numbers at this point. And if  $R0 = 2$ , a pointer to the string «*two\n*» will be loaded by `ADREQ` into `R0`.

### ARM: Optimizing Keil 6/2013 (Thumb mode)

```
.text:000000D4          f1:
.text:000000D4 10 B5    PUSH   {R4,LR}
.text:000000D6 00 28    CMP    R0, #0
.text:000000D8 05 D0    BEQ    zero_case
.text:000000DA 01 28    CMP    R0, #1
.text:000000DC 05 D0    BEQ    one_case
.text:000000DE 02 28    CMP    R0, #2
.text:000000E0 05 D0    BEQ    two_case
.text:000000E2 91 A0    ADR    R0, aSomethingUnkno ; "something unknown\n"
.text:000000E4 04 E0    B      default_case

.text:000000E6          zero_case: ; CODE XREF: f1+4
.text:000000E6 95 A0    ADR    R0, aZero ; "zero\n"
.text:000000E8 02 E0    B      default_case

.text:000000EA          one_case: ; CODE XREF: f1+8
.text:000000EA 96 A0    ADR    R0, aOne ; "one\n"
.text:000000EC 00 E0    B      default_case

.text:000000EE          two_case: ; CODE XREF: f1+C
.text:000000EE 97 A0    ADR    R0, aTwo ; "two\n"
.text:000000F0          default_case ; CODE XREF: f1+10
.text:000000F0           ; f1+14
.text:000000F0 06 F0 7E F8 BL     __2printf
.text:000000F4 10 BD    POP   {R4,PC}
```

As was already mentioned, it is not possible to add conditional predicates to most instructions in Thumb mode, so the Thumb-code here is somewhat similar to the easily understandable x86 **CISC**-style code.

### ARM64: Non-optimizing GCC (Linaro) 4.9

```
.LC12:
    .string "zero"
.LC13:
    .string "one"
.LC14:
    .string "two"
.LC15:
    .string "something unknown"
f12:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    str    w0, [x29,28]
    ldr    w0, [x29,28]
    cmp    w0, 1
    beq    .L34
    cmp    w0, 2
    beq    .L35
    cmp    w0, wzr
    bne    .L38          ; jump to default label
    adrp   x0, .LC12      ; "zero"
    add    x0, x0, :lo12:.LC12
    bl     puts
```

## 1.15. SWITCH()/CASE/DEFAULT

```
b      .L32
.L34:
    adrp  x0, .LC13      ; "one"
    add   x0, x0, :lo12:.LC13
    bl    puts
    b     .L32
.L35:
    adrp  x0, .LC14      ; "two"
    add   x0, x0, :lo12:.LC14
    bl    puts
    b     .L32
.L38:
    adrp  x0, .LC15      ; "something unknown"
    add   x0, x0, :lo12:.LC15
    bl    puts
    nop
.L32:
    ldp   x29, x30, [sp], 32
    ret
```

The type of the input value is *int*, hence register `W0` is used to hold it instead of the whole `X0` register.

The string pointers are passed to `puts()` using an `ADRP / ADD` instructions pair just like it was demonstrated in the “Hello, world!” example: [1.5.4 on page 24](#).

## ARM64: Optimizing GCC (Linaro) 4.9

```
f12:
    cmp   w0, 1
    beq  .L31
    cmp   w0, 2
    beq  .L32
    cbz   w0, .L35
; default case
    adrp  x0, .LC15      ; "something unknown"
    add   x0, x0, :lo12:.LC15
    b     puts
.L35:
    adrp  x0, .LC12      ; "zero"
    add   x0, x0, :lo12:.LC12
    b     puts
.L32:
    adrp  x0, .LC14      ; "two"
    add   x0, x0, :lo12:.LC14
    b     puts
.L31:
    adrp  x0, .LC13      ; "one"
    add   x0, x0, :lo12:.LC13
    b     puts
```

Better optimized piece of code. `CBZ` (*Compare and Branch on Zero*) instruction does jump if `W0` is zero. There is also a direct jump to `puts()` instead of calling it, like it was explained before: [1.15.1 on page 155](#).

## MIPS

Listing 1.150: Optimizing GCC 4.4.5 (IDA)

```
f:
; is it 1?
    lui   $gp, (__gnu_local_gp >> 16)
    li    $v0, 1
    beq  $a0, $v0, loc_60
    la    $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; is it 2?
    li    $v0, 2
```

## 1.15. SWITCH()/CASE/DEFAULT

```
        beq    $a0, $v0, loc_4C
        or     $at, $zero ; branch delay slot, NOP
; jump, if not equal to 0:
        bnez   $a0, loc_38
        or     $at, $zero ; branch delay slot, NOP
; zero case:
        lui    $a0, ($LC0 >> 16) # "zero"
        lw    $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jr    $t9 ; branch delay slot, NOP
        la    $a0, ($LC0 & 0xFFFF) # "zero" ; branch delay slot

loc_38:
        lui    $a0, ($LC3 >> 16) # "something unknown"
        lw    $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jr    $t9
        la    $a0, ($LC3 & 0xFFFF) # "something unknown" ; branch delay slot

loc_4C:
        lui    $a0, ($LC2 >> 16) # "two"
        lw    $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jr    $t9
        la    $a0, ($LC2 & 0xFFFF) # "two" ; branch delay slot

loc_60:
        lui    $a0, ($LC1 >> 16) # "one"
        lw    $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jr    $t9
        la    $a0, ($LC1 & 0xFFFF) # "one" ; branch delay slot
```

The function always ends with calling `puts()`, so here we see a jump to `puts()` (`JR`: “Jump Register”) instead of “jump and link”. We talked about this earlier: [1.15.1 on page 155](#).

We also often see `NOP` instructions after `LW` ones. This is “load delay slot”: another *delay slot* in MIPS.

An instruction next to `LW` may execute at the moment while `LW` loads value from memory.

However, the next instruction must not use the result of `LW`.

Modern MIPS CPUs have a feature to wait if the next instruction uses result of `LW`, so this is somewhat outdated, but GCC still adds NOPs for older MIPS CPUs. In general, it can be ignored.

## Conclusion

A `switch()` with few cases is indistinguishable from an `if/else` construction, for example: [listing.1.15.1](#).

### 1.15.2 A lot of cases

If a `switch()` statement contains a lot of cases, it is not very convenient for the compiler to emit too large code with a lot `JE / JNE` instructions.

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        case 3: printf ("three\n"); break;
        case 4: printf ("four\n"); break;
        default: printf ("something unknown\n"); break;
```

## 1.15. SWITCH()/CASE/DEFAULT

```
    };
}

int main()
{
    f(2); // test
}
```

### x86

#### Non-optimizing MSVC

We get (MSVC 2010):

Listing 1.151: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 4
    ja      SHORT $LN1@f
    mov     ecx, DWORD PTR tv64[ebp]
    jmp     DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN9@f
$LN5@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN9@f
$LN4@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN9@f
$LN3@f:
    push    OFFSET $SG745 ; 'three', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN9@f
$LN2@f:
    push    OFFSET $SG747 ; 'four', 0aH, 00H
    call    _printf
    add    esp, 4
    jmp     SHORT $LN9@f
$LN1@f:
    push    OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call    _printf
    add    esp, 4
$LN9@f:
    mov     esp, ebp
    pop    ebp
    ret    0
    npad   2 ; align next label
$LN11@f:
    DD    $LN6@f ; 0
    DD    $LN5@f ; 1
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
```

## 1.15. SWITCH()/CASE/DEFAULT

```
_f      ENDP
```

What we see here is a set of `printf()` calls with various arguments. All they have not only addresses in the memory of the process, but also internal symbolic labels assigned by the compiler. All these labels are also mentioned in the `$LN11@f` internal table.

At the function start, if `a` is greater than 4, control flow is passed to label `$LN1@f`, where `printf()` with argument `'something unknown'` is called.

But if the value of `a` is less or equals to 4, then it gets multiplied by 4 and added with the `$LN11@f` table address. That is how an address inside the table is constructed, pointing exactly to the element we need. For example, let's say `a` is equal to 2.  $2 * 4 = 8$  (all table elements are addresses in a 32-bit process and that is why all elements are 4 bytes wide). The address of the `$LN11@f` table + 8 is the table element where the `$LN4@f` label is stored. `JMP` fetches the `$LN4@f` address from the table and jumps to it.

This table is sometimes called *jump table* or *branch table*<sup>95</sup>.

Then the corresponding `printf()` is called with argument `'two'`.

Literally, the `jmp DWORD PTR $LN11@f[ecx*4]` instruction implies *jump to the DWORD that is stored at address `$LN11@f + ecx * 4`*.

`npad` ([.1.7 on page 1021](#)) is an assembly language macro that align the next label so that it will be stored at an address aligned on a 4 byte (or 16 byte) boundary. This is very suitable for the processor since it is able to fetch 32-bit values from memory through the memory bus, cache memory, etc., in a more effective way if it is aligned.

<sup>95</sup>The whole method was once called *computed GOTO* in early versions of Fortran: [wikipedia](#). Not quite relevant these days, but what a term!

### 1.15. SWITCH() / CASE / DEFAULT

# OllyDbg

Let's try this example in OllyDbg. The input value of the function (2) is loaded into EAX:

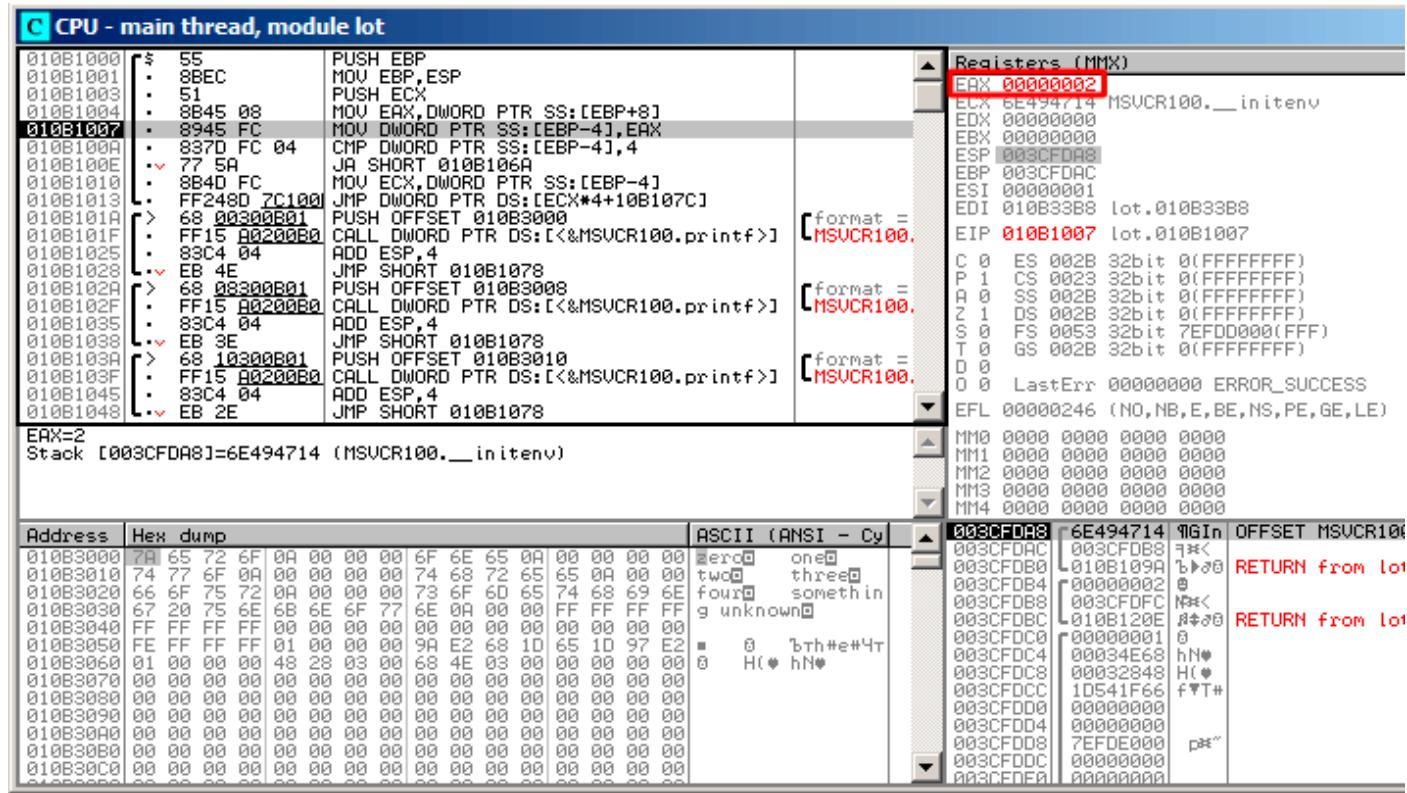


Figure 1.50: OllyDbg: function's input value is loaded in EAX

## 1.15. SWITCH()/CASE/DEFAULT

The input value is checked, is it bigger than 4? If not, the “default” jump is not taken:

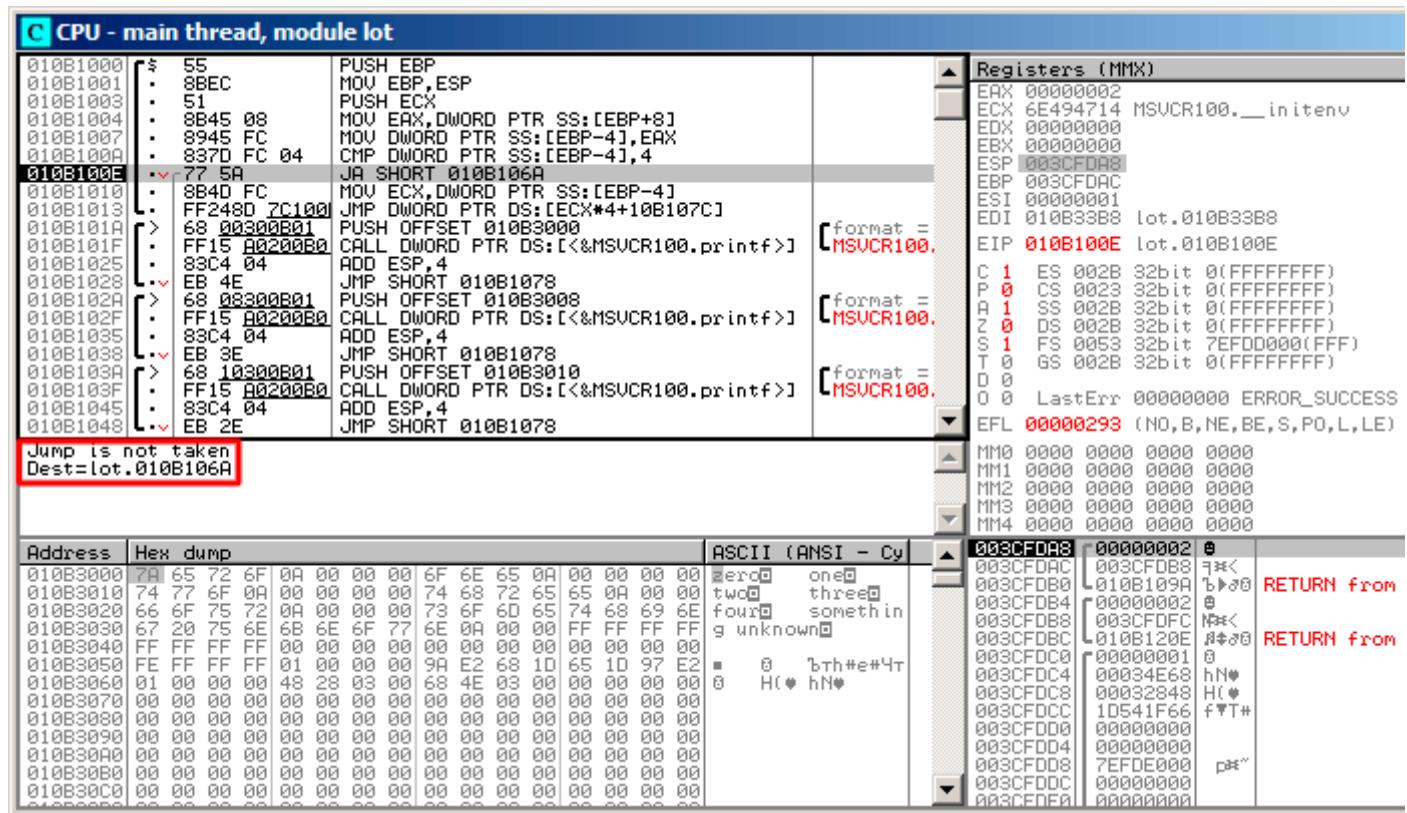


Figure 1.51: OllyDbg: 2 is no bigger than 4: no jump is taken

## 1.15. SWITCH()/CASE/DEFAULT

Here we see a jumptable:

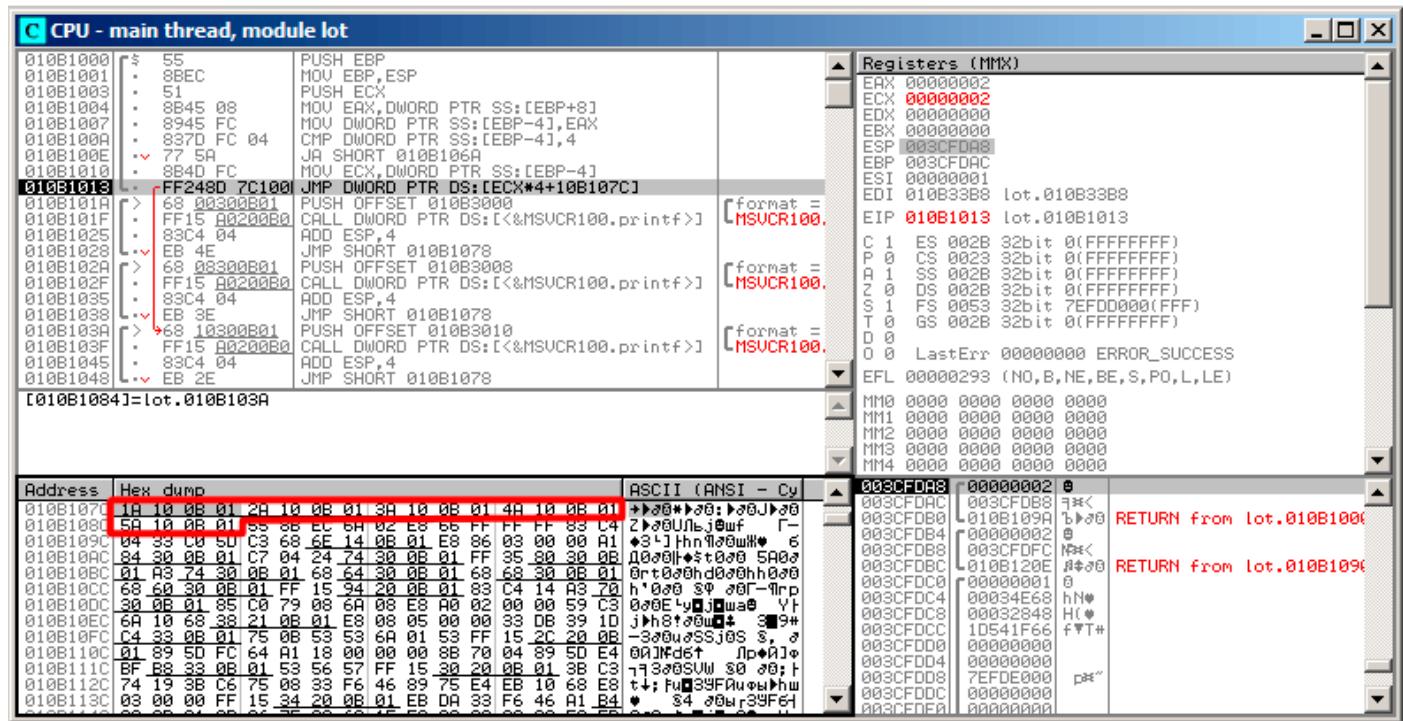


Figure 1.52: OllyDbg: calculating destination address using jumptable

Here we've clicked "Follow in Dump" → "Address constant", so now we see the *jumptable* in the data window. These are 5 32-bit values<sup>96</sup>. ECX is now 2, so the third element (can be indexed as  $2^{97}$ ) of the table is to be used. It's also possible to click "Follow in Dump" → "Memory address" and OllyDbg will show the element addressed by the `JMP` instruction. That's `0x010B103A`.

<sup>96</sup>They are underlined by OllyDbg because these are also FIXUPS: [6.5.2 on page 759](#), we are going to come back to them later

<sup>97</sup>About indexing, see also: [3.20.3](#)

## 1.15. SWITCH()/CASE/DEFAULT

After the jump we are at `0x010B103A`: the code printing "two" will now be executed:

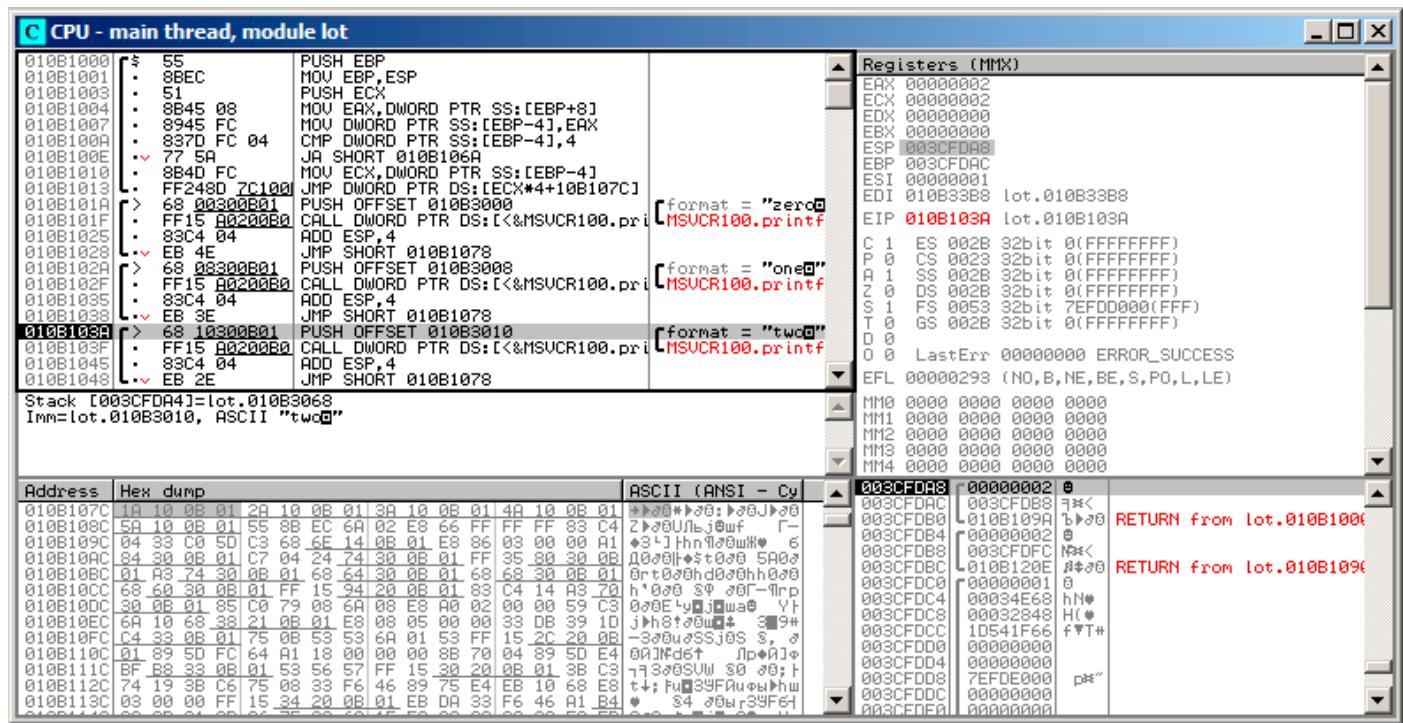


Figure 1.53: OllyDbg: now we at the case: label

## Non-optimizing GCC

Let's see what GCC 4.4.1 generates:

Listing 1.152: GCC 4.4.1

```

public f
f proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0 = dword ptr 8

    push    ebp
    mov     ebp, esp
    sub     esp, 18h
    cmp     [ebp+arg_0], 4
    ja      short loc_8048444
    mov     eax, [ebp+arg_0]
    shl     eax, 2
    mov     eax, ds:off_804855C[eax]
    jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
    mov     [esp+18h+var_18], offset aZero ; "zero"
    call    _puts
    jmp     short locret_8048450

loc_804840C: ; DATA XREF: .rodata:08048560
    mov     [esp+18h+var_18], offset aOne ; "one"
    call    _puts
    jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
    mov     [esp+18h+var_18], offset aTwo ; "two"
    call    _puts
    jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568

```

## 1.15. SWITCH()/CASE/DEFAULT

```
    mov    [esp+18h+var_18], offset aThree ; "three"
    call   _puts
    jmp    short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
    mov    [esp+18h+var_18], offset aFour ; "four"
    call   _puts
    jmp    short locret_8048450

loc_8048444: ; CODE XREF: f+A
    mov    [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
    call   _puts

locret_8048450: ; CODE XREF: f+26
    ; f+34...
    leave
    retn
f    endp

off_804855C dd offset loc_80483FE ; DATA XREF: f+12
    dd offset loc_804840C
    dd offset loc_804841A
    dd offset loc_8048428
    dd offset loc_8048436
```

It is almost the same, with a little nuance: argument `arg_0` is multiplied by 4 by shifting it to left by 2 bits (it is almost the same as multiplication by 4) ([1.18.2 on page 217](#)). Then the address of the label is taken from the `off_804855C` array, stored in `EAX`, and then `JMP EAX` does the actual jump.

## ARM: Optimizing Keil 6/2013 (ARM mode)

Listing 1.153: Optimizing Keil 6/2013 (ARM mode)

```
00000174      f2
00000174 05 00 50 E3      CMP    R0, #5           ; switch 5 cases
00000178 00 F1 8F 30      ADDCC  PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA      B      default_case     ; jumptable 00000178 default case

00000180
00000180          loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA      B      zero_case       ; jumptable 00000178 case 0

00000184
00000184          loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA      B      one_case        ; jumptable 00000178 case 1

00000188
00000188          loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA      B      two_case        ; jumptable 00000178 case 2

0000018C
0000018C          loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA      B      three_case      ; jumptable 00000178 case 3

00000190
00000190          loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA      B      four_case       ; jumptable 00000178 case 4

00000194
00000194          zero_case ; CODE XREF: f2+4
00000194          ; f2:loc_180
00000194 EC 00 8F E2      ADR    R0, aZero       ; jumptable 00000178 case 0
00000198 06 00 00 EA      B      loc_1B8

0000019C
0000019C          one_case ; CODE XREF: f2+4
0000019C          ; f2:loc_184
```

## 1.15. SWITCH()/CASE/DEFAULT

```

0000019C EC 00 8F E2      ADR    R0, aOne           ; jumptable 00000178 case 1
000001A0 04 00 00 EA      B      loc_1B8

000001A4
000001A4      two_case ; CODE XREF: f2+4
000001A4          ; f2:loc_188
000001A4 01 0C 8F E2      ADR    R0, aTwo           ; jumptable 00000178 case 2
000001A8 02 00 00 EA      B      loc_1B8

000001AC
000001AC      three_case ; CODE XREF: f2+4
000001AC          ; f2:loc_18C
000001AC 01 0C 8F E2      ADR    R0, aThree         ; jumptable 00000178 case 3
000001B0 00 00 00 EA      B      loc_1B8

000001B4
000001B4      four_case ; CODE XREF: f2+4
000001B4          ; f2:loc_190
000001B4 01 0C 8F E2      ADR    R0, aFour          ; jumptable 00000178 case 4
000001B8
000001B8      loc_1B8   ; CODE XREF: f2+24
000001B8          ; f2+2C
000001B8 66 18 00 EA      B      __2printf

000001BC
000001BC      default_case ; CODE XREF: f2+4
000001BC          ; f2+8
000001BC D4 00 8F E2      ADR    R0, aSomethingUnkno ; jumptable 00000178 default case
000001C0 FC FF FF EA      B      loc_1B8

```

This code makes use of the ARM mode feature in which all instructions have a fixed size of 4 bytes.

Let's keep in mind that the maximum value for  $a$  is 4 and any greater value will cause «*something unknown\n*» string to be printed.

The first `CMP R0, #5` instruction compares the input value of  $a$  with 5.

<sup>98</sup> The next `ADCC PC, PC, R0, LSL#2` instruction is being executed only if  $R0 < 5$  (*CC=Carry clear / Less than*). Consequently, if `ADCC` does not trigger (it is a  $R0 \geq 5$  case), a jump to `default_case` label will occur.

But if  $R0 < 5$  and `ADCC` triggers, the following is to be happen:

The value in `R0` is multiplied by 4. In fact, `LSL#2` at the instruction's suffix stands for “shift left by 2 bits”. But as we will see later ([1.18.2 on page 217](#)) in section “Shifts”, shift left by 2 bits is equivalent to multiplying by 4.

Then we add  $R0 * 4$  to the current value in `PC`, thus jumping to one of the `B` (*Branch*) instructions located below.

At the moment of the execution of `ADCC`, the value in `PC` is 8 bytes ahead (0x180) than the address at which the `ADCC` instruction is located (0x178), or, in other words, 2 instructions ahead.

This is how the pipeline in ARM processors works: when `ADCC` is executed, the processor at the moment is beginning to process the instruction after the next one, so that is why `PC` points there. This has to be memorized.

If  $a = 0$ , then is to be added to the value in `PC`, and the actual value of the `PC` will be written into `PC` (which is 8 bytes ahead) and a jump to the label `loc_180` will happen, which is 8 bytes ahead of the point where the `ADCC` instruction is.

If  $a = 1$ , then  $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$  will be written to `PC`, which is the address of the `loc_184` label.

With every 1 added to  $a$ , the resulting `PC` is increased by 4.

4 is the instruction length in ARM mode and also, the length of each `B` instruction, of which there are 5 in row.

Each of these five `B` instructions passes control further, to what was programmed in the `switch()`.

<sup>98</sup>ADD—addition

## 1.15. SWITCH()/CASE/DEFAULT

Pointer loading of the corresponding string occurs there, etc.

### ARM: Optimizing Keil 6/2013 (Thumb mode)

Listing 1.154: Optimizing Keil 6/2013 (Thumb mode)

```

000000F6          EXPORT f2
000000F6          f2
000000F6 10 B5    PUSH   {R4,LR}
000000F8 03 00    MOVS   R3, R0
000000FA 06 F0 69 F8  BL     __ARM_common_switch8_thumb ; switch 6 cases

000000FE 05      DCB 5
000000FF 04 06 08 0A 0C 10  DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for switch statement
00000105 00      ALIGN 2

00000106
00000106 zero_case ; CODE XREF: f2+4
00000106 8D A0    ADR    R0, aZero ; jumptable 000000FA case 0
00000108 06 E0    B      loc_118

0000010A
0000010A one_case ; CODE XREF: f2+4
0000010A 8E A0    ADR    R0, aOne ; jumptable 000000FA case 1
0000010C 04 E0    B      loc_118

0000010E
0000010E two_case ; CODE XREF: f2+4
0000010E 8F A0    ADR    R0, aTwo ; jumptable 000000FA case 2
00000110 02 E0    B      loc_118

00000112
00000112 three_case ; CODE XREF: f2+4
00000112 90 A0    ADR    R0, aThree ; jumptable 000000FA case 3
00000114 00 E0    B      loc_118

00000116
00000116 four_case ; CODE XREF: f2+4
00000116 91 A0    ADR    R0, aFour ; jumptable 000000FA case 4
00000118
00000118 loc_118 ; CODE XREF: f2+12
00000118      ; f2+16
00000118 06 F0 6A F8  BL     __2printf
0000011C 10 BD    POP    {R4,PC}

0000011E
0000011E default_case ; CODE XREF: f2+4
0000011E 82 A0    ADR    R0, aSomethingUnkno ; jumptable 000000FA default case
00000120 FA E7    B      loc_118

000061D0          EXPORT __ARM_common_switch8_thumb
000061D0          __ARM_common_switch8_thumb ; CODE XREF: example6_f2+4
000061D0 78 47    BX     PC

000061D2 00 00    ALIGN 4
000061D2      ; End of function __ARM_common_switch8_thumb
000061D2
000061D4      _32__ARM_common_switch8_thumb ; CODE XREF: ↵
000061D4      ↴ __ARM_common_switch8_thumb
000061D4 01 C0 5E E5  LDRB   R12, [LR,#-1]
000061D8 0C 00 53 E1  CMP    R3, R12
000061DC 0C 30 DE 27  LDRCSB R3, [LR,R12]
000061E0 03 30 DE 37  LDRCCB R3, [LR,R3]
000061E4 83 C0 8E E0  ADD    R12, LR, R3,LSL#1
000061E8 1C FF 2F E1  BX     R12
000061E8      ; End of function _32__ARM_common_switch8_thumb

```

One cannot be sure that all instructions in Thumb and Thumb-2 modes has the same size. It can even be said that in these modes the instructions have variable lengths, just like in x86.

### 1.15. SWITCH()/CASE/DEFAULT

So there is a special table added that contains information about how much cases are there (not including default-case), and an offset for each with a label to which control must be passed in the corresponding case.

A special function is present here in order to deal with the table and pass control, named `__ARM_common_switch8_thumb`. It starts with `BX PC`, whose function is to switch the processor to ARM-mode. Then you see the function for table processing.

It is too advanced to describe it here now, so let's omit it.

It is interesting to note that the function uses the `LR` register as a pointer to the table.

Indeed, after calling of this function, `LR` contains the address after

`BL __ARM_common_switch8_thumb` instruction, where the table starts.

It is also worth noting that the code is generated as a separate function in order to reuse it, so the compiler doesn't generate the same code for every `switch()` statement.

IDA successfully perceived it as a service function and a table, and added comments to the labels like `jumptable 000000FA case 0`.

## MIPS

Listing 1.155: Optimizing GCC 4.4.5 (IDA)

```
f:
    lui      $gp, (_gnu_local_gp >> 16)
; jump to loc_24 if input value is lesser than 5:
    sltiu   $v0, $a0, 5
    bnez   $v0, loc_24
    la      $gp, (_gnu_local_gp & 0xFFFF) ; branch delay slot
; input value is greater or equal to 5.
; print "something unknown" and finish:
    lui      $a0, ($LC5 >> 16) # "something unknown"
    lw       $t9, (puts & 0xFFFF)($gp)
    or      $at, $zero ; NOP
    jr      $t9
    la      $a0, ($LC5 & 0xFFFF) # "something unknown" ; branch delay slot

loc_24:                      # CODE XREF: f+8
; load address of jumptable
; LA is pseudoinstruction, LUI and ADDIU pair are there in fact:
    la      $v0, off_120
; multiply input value by 4:
    sll     $a0, 2
; sum up multiplied value and jumptable address:
    addu   $a0, $v0, $a0
; load element from jumptable:
    lw       $v0, 0($a0)
    or      $at, $zero ; NOP
; jump to the address we got in jumptable:
    jr      $v0
    or      $at, $zero ; branch delay slot, NOP

sub_44:                      # DATA XREF: .rodata:0000012C
; print "three" and finish
    lui      $a0, ($LC3 >> 16) # "three"
    lw       $t9, (puts & 0xFFFF)($gp)
    or      $at, $zero ; NOP
    jr      $t9
    la      $a0, ($LC3 & 0xFFFF) # "three" ; branch delay slot

sub_58:                      # DATA XREF: .rodata:00000130
; print "four" and finish
    lui      $a0, ($LC4 >> 16) # "four"
    lw       $t9, (puts & 0xFFFF)($gp)
    or      $at, $zero ; NOP
    jr      $t9
    la      $a0, ($LC4 & 0xFFFF) # "four" ; branch delay slot
```

## 1.15. SWITCH()/CASE/DEFAULT

```
sub_6C:                                # DATA XREF: .rodata:off_120
; print "zero" and finish
    lui      $a0, ($LC0 >> 16)  # "zero"
    lw       $t9, (puts & 0xFFFF)($gp)
    or       $at, $zero ; NOP
    jr       $t9
    la       $a0, ($LC0 & 0xFFFF)  # "zero" ; branch delay slot

sub_80:                                # DATA XREF: .rodata:00000124
; print "one" and finish
    lui      $a0, ($LC1 >> 16)  # "one"
    lw       $t9, (puts & 0xFFFF)($gp)
    or       $at, $zero ; NOP
    jr       $t9
    la       $a0, ($LC1 & 0xFFFF)  # "one" ; branch delay slot

sub_94:                                # DATA XREF: .rodata:00000128
; print "two" and finish
    lui      $a0, ($LC2 >> 16)  # "two"
    lw       $t9, (puts & 0xFFFF)($gp)
    or       $at, $zero ; NOP
    jr       $t9
    la       $a0, ($LC2 & 0xFFFF)  # "two" ; branch delay slot

; may be placed in .rodata section:
off_120:     .word sub_6C
              .word sub_80
              .word sub_94
              .word sub_44
              .word sub_58
```

The new instruction for us is **SLTIU** ("Set on Less Than Immediate Unsigned").

This is the same as **SLTU** ("Set on Less Than Unsigned"), but "I" stands for "immediate", i.e., a number has to be specified in the instruction itself.

**BNEZ** is "Branch if Not Equal to Zero".

Code is very close to the other **ISAs**. **SLL** ("Shift Word Left Logical") does multiplication by 4.

MIPS is a 32-bit CPU after all, so all addresses in the *jumptable* are 32-bit ones.

## Conclusion

Rough skeleton of *switch()*:

Listing 1.156: x86

```
MOV REG, input
CMP REG, 4 ; maximal number of cases
JA default
SHL REG, 2 ; find element in table. shift for 3 bits in x64.
MOV REG, jump_table[REG]
JMP REG

case1:
; do something
JMP exit
case2:
; do something
JMP exit
case3:
; do something
JMP exit
case4:
; do something
JMP exit
case5:
; do something
```

## 1.15. SWITCH()/CASE/DEFAULT

```
JMP exit

default:
    ...
exit:
    ...
jump_table dd case1
    dd case2
    dd case3
    dd case4
    dd case5
```

The jump to the address in the jump table may also be implemented using this instruction:

`JMP jump_table[REG*4]`. Or `JMP jump_table[REG*8]` in x64.

A *jumptable* is just array of pointers, like the one described later: [1.20.5 on page 287](#).

### 1.15.3 When there are several case statements in one block

Here is a very widespread construction: several case statements for a single block:

```
#include <stdio.h>

void f(int a)
{
    switch (a)
    {
        case 1:
        case 2:
        case 7:
        case 10:
            printf ("1, 2, 7, 10\n");
            break;
        case 3:
        case 4:
        case 5:
        case 6:
            printf ("3, 4, 5\n");
            break;
        case 8:
        case 9:
        case 20:
        case 21:
            printf ("8, 9, 21\n");
            break;
        case 22:
            printf ("22\n");
            break;
        default:
            printf ("default\n");
            break;
    };
}

int main()
{
    f(4);
}
```

It's too wasteful to generate a block for each possible case, so what is usually done is to generate each block plus some kind of dispatcher.

Listing 1.157: Optimizing MSVC 2010

```

1 $SG2798 DB      '1, 2, 7, 10', 0aH, 00H
2 $SG2800 DB      '3, 4, 5', 0aH, 00H
3 $SG2802 DB      '8, 9, 21', 0aH, 00H
4 $SG2804 DB      '22', 0aH, 00H
5 $SG2806 DB      'default', 0aH, 00H
6
7 _a$ = 8
8 _f     PROC
9      mov    eax, DWORD PTR _a$[esp-4]
10     dec   eax
11     cmp   eax, 21
12     ja    SHORT $LN1@f
13     movzx eax, BYTE PTR $LN10@f[eax]
14     jmp   DWORD PTR $LN11@f[eax*4]
15 $LN5@f:
16     mov    DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
17     jmp   DWORD PTR __imp__printf
18 $LN4@f:
19     mov    DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
20     jmp   DWORD PTR __imp__printf
21 $LN3@f:
22     mov    DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
23     jmp   DWORD PTR __imp__printf
24 $LN2@f:
25     mov    DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
26     jmp   DWORD PTR __imp__printf
27 $LN1@f:
28     mov    DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
29     jmp   DWORD PTR __imp__printf
30     npad  2 ; align $LN11@f table on 16-byte boundary
31 $LN11@f:
32     DD    $LN5@f ; print '1, 2, 7, 10'
33     DD    $LN4@f ; print '3, 4, 5'
34     DD    $LN3@f ; print '8, 9, 21'
35     DD    $LN2@f ; print '22'
36     DD    $LN1@f ; print 'default'
37 $LN10@f:
38     DB    0 ; a=1
39     DB    0 ; a=2
40     DB    1 ; a=3
41     DB    1 ; a=4
42     DB    1 ; a=5
43     DB    1 ; a=6
44     DB    0 ; a=7
45     DB    2 ; a=8
46     DB    2 ; a=9
47     DB    0 ; a=10
48     DB    4 ; a=11
49     DB    4 ; a=12
50     DB    4 ; a=13
51     DB    4 ; a=14
52     DB    4 ; a=15
53     DB    4 ; a=16
54     DB    4 ; a=17
55     DB    4 ; a=18
56     DB    4 ; a=19
57     DB    2 ; a=20
58     DB    2 ; a=21
59     DB    3 ; a=22
60 _f     ENDP

```

We see two tables here: the first table (`$LN10@f`) is an index table, and the second one (`$LN11@f`) is an array of pointers to blocks.

First, the input value is used as an index in the index table (line 13).

## 1.15. SWITCH()/CASE/DEFAULT

Here is a short legend for the values in the table: 0 is the first case block (for values 1, 2, 7, 10), 1 is the second one (for values 3, 4, 5), 2 is the third one (for values 8, 9, 21), 3 is the fourth one (for value 22), 4 is for the default block.

There we get an index for the second table of code pointers and we jump to it (line 14).

What is also worth noting is that there is no case for input value 0.

That's why we see the `DEC` instruction at line 10, and the table starts at  $a = 1$ , because there is no need to allocate a table element for  $a = 0$ .

This is a very widespread pattern.

So why is this economical? Why isn't it possible to make it as before ([1.15.2 on page 173](#)), just with one table consisting of block pointers? The reason is that the elements in index table are 8-bit, hence it's all more compact.

## GCC

GCC does the job in the way we already discussed ([1.15.2 on page 173](#)), using just one table of pointers.

### ARM64: Optimizing GCC 4.9.1

There is no code to be triggered if the input value is 0, so GCC tries to make the jump table more compact and so it starts at 1 as an input value.

GCC 4.9.1 for ARM64 uses an even cleverer trick. It's able to encode all offsets as 8-bit bytes.

Let's recall that all ARM64 instructions have a size of 4 bytes.

GCC is uses the fact that all offsets in my tiny example are in close proximity to each other. So the jump table consisting of single bytes.

Listing 1.158: Optimizing GCC 4.9.1 ARM64

```
f14:  
; input value in W0  
    sub    w0, w0, #1  
    cmp    w0, 21  
; branch if less or equal (unsigned):  
    bls    .L9  
.L2:  
; print "default":  
    adrp   x0, .LC4  
    add    x0, x0, :lo12:.LC4  
    b      puts  
.L9:  
; load jumptable address to X1:  
    adrp   x1, .L4  
    add    x1, x1, :lo12:.L4  
; W0=input_value-1  
; load byte from the table:  
    ldrb   w0, [x1,w0,uxtw]  
; load address of the Lrtx label:  
    adr    x1, .Lrtx4  
; multiply table element by 4 (by shifting 2 bits left) and add (or subtract) to the address of ↴  
    ↴ Lrtx:  
    add    x0, x1, w0, sxtb #2  
; jump to the calculated address:  
    br    x0  
; this label is pointing in code (text) segment:  
.Lrtx4:  
    .section    .rodata  
; everything after ".section" statement is allocated in the read-only data (rodata) segment:  
.L4:  
    .byte   (.L3 - .Lrtx4) / 4      ; case 1  
    .byte   (.L3 - .Lrtx4) / 4      ; case 2  
    .byte   (.L5 - .Lrtx4) / 4      ; case 3  
    .byte   (.L5 - .Lrtx4) / 4      ; case 4  
    .byte   (.L5 - .Lrtx4) / 4      ; case 5
```

### 1.15. SWITCH()/CASE/DEFAULT

```

.byte  (.L5 - .Lrtx4) / 4 ; case 6
.byte  (.L3 - .Lrtx4) / 4 ; case 7
.byte  (.L6 - .Lrtx4) / 4 ; case 8
.byte  (.L6 - .Lrtx4) / 4 ; case 9
.byte  (.L3 - .Lrtx4) / 4 ; case 10
.byte  (.L2 - .Lrtx4) / 4 ; case 11
.byte  (.L2 - .Lrtx4) / 4 ; case 12
.byte  (.L2 - .Lrtx4) / 4 ; case 13
.byte  (.L2 - .Lrtx4) / 4 ; case 14
.byte  (.L2 - .Lrtx4) / 4 ; case 15
.byte  (.L2 - .Lrtx4) / 4 ; case 16
.byte  (.L2 - .Lrtx4) / 4 ; case 17
.byte  (.L2 - .Lrtx4) / 4 ; case 18
.byte  (.L2 - .Lrtx4) / 4 ; case 19
.byte  (.L6 - .Lrtx4) / 4 ; case 20
.byte  (.L6 - .Lrtx4) / 4 ; case 21
.byte  (.L7 - .Lrtx4) / 4 ; case 22
.text
; everything after ".text" statement is allocated in the code (text) segment:
.L7:
; print "22"
    adrp    x0, .LC3
    add     x0, x0, :lo12:.LC3
    b      puts
.L6:
; print "8, 9, 21"
    adrp    x0, .LC2
    add     x0, x0, :lo12:.LC2
    b      puts
.L5:
; print "3, 4, 5"
    adrp    x0, .LC1
    add     x0, x0, :lo12:.LC1
    b      puts
.L3:
; print "1, 2, 7, 10"
    adrp    x0, .LC0
    add     x0, x0, :lo12:.LC0
    b      puts
.LC0:
    .string "1, 2, 7, 10"
.LC1:
    .string "3, 4, 5"
.LC2:
    .string "8, 9, 21"
.LC3:
    .string "22"
.LC4:
    .string "default"

```

Let's compile this example to object file and open it in [IDA](#). Here is the jump table:

Listing 1.159: jumptable in IDA

.rodata:00000000000000064	AREA .rodata, DATA, READONLY
.rodata:00000000000000064	; ORG 0x64
.rodata:00000000000000064 \$d	DCB 9 ; case 1
.rodata:00000000000000065	DCB 9 ; case 2
.rodata:00000000000000066	DCB 6 ; case 3
.rodata:00000000000000067	DCB 6 ; case 4
.rodata:00000000000000068	DCB 6 ; case 5
.rodata:00000000000000069	DCB 6 ; case 6
.rodata:0000000000000006A	DCB 9 ; case 7
.rodata:0000000000000006B	DCB 3 ; case 8
.rodata:0000000000000006C	DCB 3 ; case 9
.rodata:0000000000000006D	DCB 9 ; case 10
.rodata:0000000000000006E	DCB 0xF7 ; case 11
.rodata:0000000000000006F	DCB 0xF7 ; case 12
.rodata:00000000000000070	DCB 0xF7 ; case 13
.rodata:00000000000000071	DCB 0xF7 ; case 14
.rodata:00000000000000072	DCB 0xF7 ; case 15

## 1.15. SWITCH()/CASE/DEFAULT

```
.rodata:00000000000000073          DCB 0xF7 ; case 16
.rodata:00000000000000074          DCB 0xF7 ; case 17
.rodata:00000000000000075          DCB 0xF7 ; case 18
.rodata:00000000000000076          DCB 0xF7 ; case 19
.rodata:00000000000000077          DCB    3 ; case 20
.rodata:00000000000000078          DCB    3 ; case 21
.rodata:00000000000000079          DCB    0 ; case 22
.rodata:0000000000000007B ; .rodata ends
```

So in case of 1, 9 is to be multiplied by 4 and added to the address of `Lrtx4` label.

In case of 22, 0 is to be multiplied by 4, resulting in 0.

Right after the `Lrtx4` label is the `L7` label, where you can find the code that prints “22”.

There is no jump table in the code segment, it's allocated in a separate `.rodata` section (there is no special necessity to place it in the code section).

There are also negative bytes (`0xF7`), they are used for jumping back to the code that prints the “default” string (at `.L2`).

### 1.15.4 Fall-through

Another popular usage of `switch()` operator is so-called “fallthrough”. Here is simple example<sup>99</sup>:

```
1 bool is_whitespace(char c) {
2     switch (c) {
3         case ' ': // fallthrough
4         case '\t': // fallthrough
5         case '\r': // fallthrough
6         case '\n':
7             return true;
8         default: // not whitespace
9             return false;
10    }
11 }
```

Slightly harder, from Linux kernel<sup>100</sup>:

```
1 char nco1, nco2;
2
3 void f(int if_freq_khz)
4 {
5
6     switch (if_freq_khz) {
7         default:
8             printf("IF=%d KHz is not supported, 3250 assumed\n", if_freq_khz);
9             /* fallthrough */
10            case 3250: /* 3.25Mhz */
11                nco1 = 0x34;
12                nco2 = 0x00;
13                break;
14            case 3500: /* 3.50Mhz */
15                nco1 = 0x38;
16                nco2 = 0x00;
17                break;
18            case 4000: /* 4.00Mhz */
19                nco1 = 0x40;
20                nco2 = 0x00;
21                break;
22            case 5000: /* 5.00Mhz */
23                nco1 = 0x50;
24                nco2 = 0x00;
25                break;
26            case 5380: /* 5.38Mhz */
27                nco1 = 0x56;
```

<sup>99</sup>Copypasted from [https://github.com/azonalon/prgraas/blob/master/progllib/lecture\\_examples/is\\_whitespace.c](https://github.com/azonalon/prgraas/blob/master/progllib/lecture_examples/is_whitespace.c)

<sup>100</sup>Copypasted from <https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c>

### 1.15. SWITCH()/CASE/DEFAULT

```
28             nco2 = 0x14;
29             break;
30     }
31 }
```

Listing 1.160: Optimizing GCC 5.4.0 x86

```
1 .LC0:
2     .string "IF=%d KHz is not supported, 3250 assumed\n"
3 f:
4     sub    esp, 12
5     mov     eax, DWORD PTR [esp+16]
6     cmp     eax, 4000
7     je      .L3
8     jg      .L4
9     cmp     eax, 3250
10    je     .L5
11    cmp     eax, 3500
12    jne     .L2
13    mov     BYTE PTR nco1, 56
14    mov     BYTE PTR nco2, 0
15    add     esp, 12
16    ret
17 .L4:
18    cmp     eax, 5000
19    je      .L7
20    cmp     eax, 5380
21    jne     .L2
22    mov     BYTE PTR nco1, 86
23    mov     BYTE PTR nco2, 20
24    add     esp, 12
25    ret
26 .L2:
27    sub    esp, 8
28    push   eax
29    push   OFFSET FLAT:.LC0
30    call   printf
31    add    esp, 16
32 .L5:
33    mov     BYTE PTR nco1, 52
34    mov     BYTE PTR nco2, 0
35    add     esp, 12
36    ret
37 .L3:
38    mov     BYTE PTR nco1, 64
39    mov     BYTE PTR nco2, 0
40    add     esp, 12
41    ret
42 .L7:
43    mov     BYTE PTR nco1, 80
44    mov     BYTE PTR nco2, 0
45    add     esp, 12
46    ret
```

We can get to `.L5` label if there is number 3250 at function's input. But we can get to this label from the other side: we see that there are no jumps between `printf()` call and `.L5` label.

Now we can understand why `switch()` statement is sometimes a source of bugs: one forgotten `break` will transform your `switch()` statement into *fallthrough* one, and several blocks will be executed instead of single one.

### 1.15.5 Exercises

#### Exercise #1

It's possible to rework the C example in [1.15.2 on page 167](#) in such way that the compiler can produce even smaller code, but will work just the same. Try to achieve it.

## 1.16 Loops

### 1.16.1 Simple example

#### x86

There is a special `LOOP` instruction in x86 instruction set for checking the value in register `ECX` and if it is not 0, to [decrement](#) `ECX` and pass control flow to the label in the `LOOP` operand. Probably this instruction is not very convenient, and there are no any modern compilers which emit it automatically. So, if you see this instruction somewhere in code, it is most likely that this is a manually written piece of assembly code.

In C/C++ loops are usually constructed using `for()`, `while()` or `do/while()` statements.

Let's start with `for()`.

This statement defines loop initialization (set loop counter to initial value), loop condition (is the counter bigger than a limit?), what is performed at each iteration ([increment/decrement](#)) and of course loop body.

```
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

The generated code is consisting of four parts as well.

Let's start with a simple example:

```
#include <stdio.h>

void printing_function(int i)
{
    printf ("f(%d)\n", i);
}

int main()
{
    int i;

    for (i=2; i<10; i++)
        printing_function(i);

    return 0;
}
```

Result (MSVC 2010):

Listing 1.161: MSVC 2010

```
i$ = -4
_main    PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     DWORD PTR _i$[ebp], 2      ; loop initialization
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp] ; here is what we do after each iteration:
    add     eax, 1                 ; add 1 to (i) value
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 10   ; this condition is checked before each iteration
    jge     SHORT $LN1@main         ; if (i) is biggest or equals to 10, lets finish loop
    mov     ecx, DWORD PTR _i$[ebp] ; loop body: call printing_function(i)
    push    ecx
    call    _printing_function
    add     esp, 4
    jmp     SHORT $LN2@main         ; jump to loop begin
$LN1@main:                           ; loop end
    xor     eax, eax
```

## 1.16. LOOPS

```
    mov    esp, ebp
    pop    ebp
    ret    0
_main  ENDP
```

As we see, nothing special.

GCC 4.4.1 emits almost the same code, with one subtle difference:

Listing 1.162: GCC 4.4.1

```
main      proc near
var_20     = dword ptr -20h
var_4      = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 20h
        mov     [esp+20h+var_4], 2 ; (i) initializing
        jmp     short loc_8048476

loc_8048465:
        mov     eax, [esp+20h+var_4]
        mov     [esp+20h+var_20], eax
        call    printing_function
        add     [esp+20h+var_4], 1 ; (i) increment

loc_8048476:
        cmp     [esp+20h+var_4], 9
        jle     short loc_8048465 ; if i<=9, continue loop
        mov     eax, 0
        leave
        retn
main    endp
```

Now let's see what we get with optimization turned on (`/Ox`):

Listing 1.163: Optimizing MSVC

```
_main  PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10    ; 0000000aH
    jl    SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main  ENDP
```

What happens here is that space for the *i* variable is not allocated in the local stack anymore, but uses an individual register for it, `ESI`. This is possible in such small functions where there aren't many local variables.

One very important thing is that the `f()` function must not change the value in `ESI`. Our compiler is sure here. And if the compiler decides to use the `ESI` register in `f()` too, its value would have to be saved at the function's prologue and restored at the function's epilogue, almost like in our listing: please note `PUSH ESI/POP ESI` at the function start and end.

Let's try GCC 4.4.1 with maximal optimization turned on (`-O3` option):

Listing 1.164: Optimizing GCC 4.4.1

```
main      proc near
```

```

var_10      = dword ptr -10h

push    ebp
mov     ebp, esp
and    esp, 0FFFFFFF0h
sub    esp, 10h
mov     [esp+10h+var_10], 2
call    printing_function
mov     [esp+10h+var_10], 3
call    printing_function
mov     [esp+10h+var_10], 4
call    printing_function
mov     [esp+10h+var_10], 5
call    printing_function
mov     [esp+10h+var_10], 6
call    printing_function
mov     [esp+10h+var_10], 7
call    printing_function
mov     [esp+10h+var_10], 8
call    printing_function
mov     [esp+10h+var_10], 9
call    printing_function
xor    eax, eax
leave
retn
main
endp

```

Huh, GCC just unwound our loop.

[Loop unwinding](#) has an advantage in the cases when there aren't much iterations and we could cut some execution time by removing all loop support instructions. On the other side, the resulting code is obviously larger.

Big unrolled loops are not recommended in modern times, because bigger functions may require bigger cache footprint<sup>101</sup>.

OK, let's increase the maximum value of the *i* variable to 100 and try again. GCC does:

Listing 1.165: GCC

```

public main
main      proc near

var_20      = dword ptr -20h

push    ebp
mov     ebp, esp
and    esp, 0FFFFFFF0h
push    ebx
mov     ebx, 2      ; i=2
sub    esp, 1Ch

; aligning label loc_80484D0 (loop body begin) by 16-byte border:
nop

loc_80484D0:
; pass (i) as first argument to printing_function():
    mov     [esp+20h+var_20], ebx
    add    ebx, 1      ; i++
    call   printing_function
    cmp    ebx, 64h    ; i==100?
    jnz    short loc_80484D0 ; if not, continue
    add    esp, 1Ch
    xor    eax, eax    ; return 0
    pop    ebx
    mov    esp, ebp
    pop    ebp

```

<sup>101</sup>A very good article about it: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)]<sup>102</sup>. Another recommendations about loop unrolling from Intel are here: [[Intel® 64 and IA-32 Architectures Optimization Reference Manual, (2014)]3.4.1.7].

## 1.16. LOOPS

```
main          retn  
             endp
```

It is quite similar to what MSVC 2010 with optimization ( /Ox ) produce, with the exception that the EBX register is allocated for the *i* variable.

GCC is sure this register will not be modified inside of the `f()` function, and if it will, it will be saved at the function prologue and restored at epilogue, just like here in the `main()` function.

## 1.16. LOOPS

### x86: OllyDbg

Let's compile our example in MSVC 2010 with `/Ox` and `/Ob0` options and load it into OllyDbg.

It seems that OllyDbg is able to detect simple loops and show them in square brackets, for convenience:

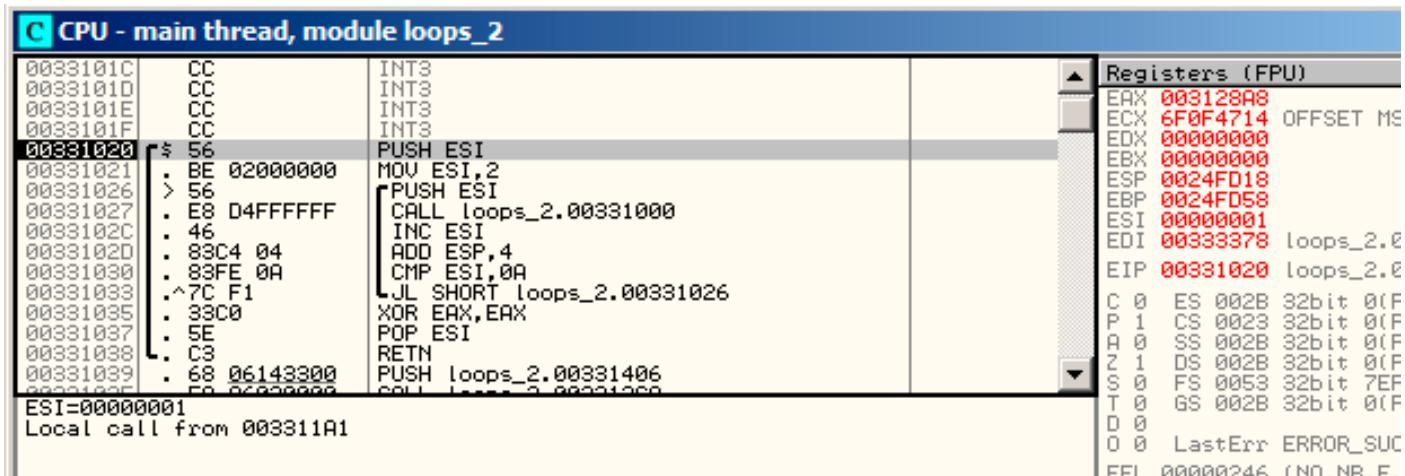


Figure 1.54: OllyDbg: `main()` begin

By tracing (F8 — step over) we see `ESI` incrementing. Here, for instance,  $ESI = i = 6$ :

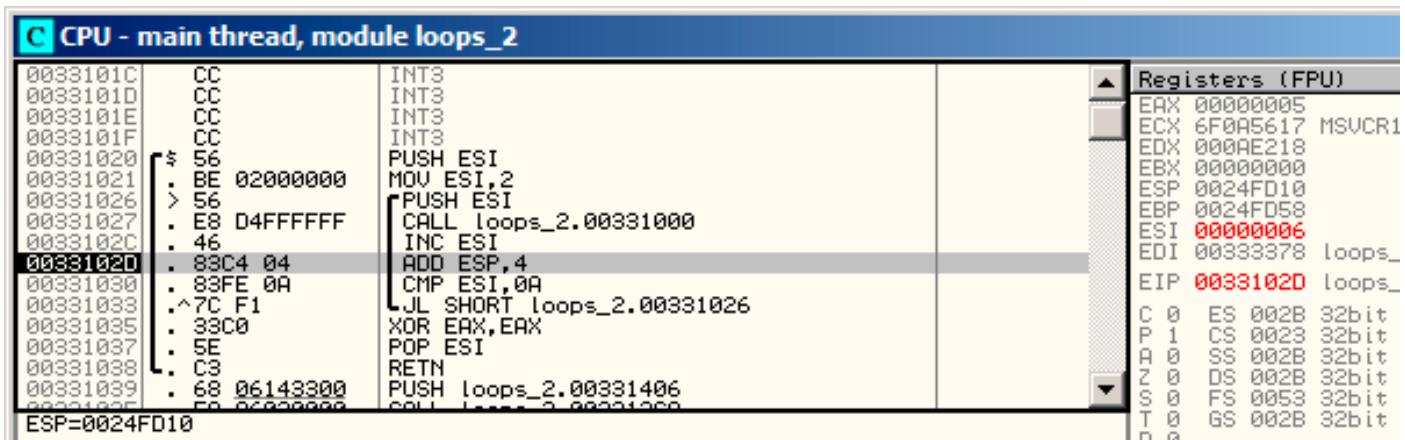


Figure 1.55: OllyDbg: loop body just executed with  $i = 6$

9 is the last loop value. That's why `JL` is not triggering after the increment, and the function will finish:

## 1.16. LOOPS

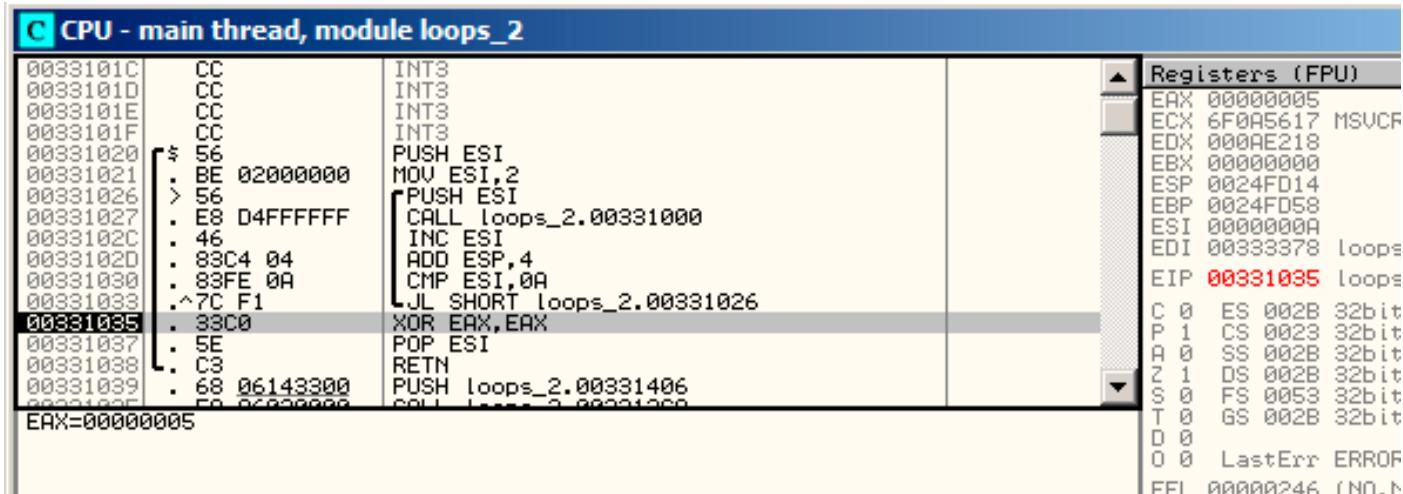


Figure 1.56: OllyDbg:  $ESI = 10$ , loop end

### x86: tracer

As we might see, it is not very convenient to trace manually in the debugger. That's a reason we will try [tracer](#).

We open compiled example in [IDA](#), find the address of the instruction `PUSH ESI` (passing the sole argument to `f()`), which is `0x401026` for this case and we run the [tracer](#):

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

`BPX` just sets a breakpoint at the address and [tracer](#) will then print the state of the registers.

In the `tracer.log`, this is what we see:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
```

## 1.16. LOOPS

```
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

We see how the value of `ESI` register changes from 2 to 9.

Even more than that, the `tracer` can collect register values for all addresses within the function. This is called *trace* there. Every instruction gets traced, all interesting register values are recorded.

Then, an `IDA.idc-script` is generated, that adds comments. So, in the `IDA` we've learned that the `main()` function address is `0x00401020` and we run:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

`BPF` stands for set breakpoint on function.

As a result, we get the `loops_2.exe.idc` and `loops_2.exe_clear.idc` scripts.

## 1.16. LOOPS

We load `loops_2.exe.idc` into [IDA](#) and see:

```
.text:00401020 ; ====== S U B R O U T I N E ======
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main proc near ; CODE XREF: __tmainCRTStartup+11D↓p
.text:00401020
.text:00401020     argc      = dword ptr  4
.text:00401020     argv      = dword ptr  8
.text:00401020     envp      = dword ptr  0Ch
.text:00401020
.text:00401020     push    esi          ; ESI=1
.text:00401021     mov     esi, 2
.text:00401026 loc_401026:           ; CODE XREF: _main+13↓j
.text:00401026     push    esi          ; ESI=2..9
.text:00401027     call    sub_401000 ; tracing nested maximum level (1) reached,
.text:0040102C     inc    esi          ; ESI=2..9
.text:0040102D     add    esp, 4          ; ESP=0x38FcBC
.text:00401030     cmp    esi, 0Ah        ; ESI=3..0xA
.text:00401033     jl    short loc_401026 ; SF=False,true OF=False
.text:00401035     xor    eax, eax
.text:00401037     pop    esi
.text:00401038     retn          ; EAX=0
.text:00401038 _main    endp
```

Figure 1.57: [IDA](#) with .idc-script loaded

We see that `ESI` can be from 2 to 9 at the start of the loop body, but from 3 to `0xA` (10) after the increment. We can also see that `main()` is finishing with 0 in `EAX`.

[tracer](#) also generates `loops_2.exe.txt`, that contains information about how many times each instruction has been executed and register values:

Listing 1.166: `loops_2.exe.txt`

0x401020 (.text+0x20), e=	1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=	1 [MOV ESI, 2]
0x401026 (.text+0x26), e=	8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=	8 [CALL 8D1000h] tracing nested maximum level (1) reached, ↴ ↳ skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=	8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=	8 [ADD ESP, 4] ESP=0x38fcBC
0x401030 (.text+0x30), e=	8 [CMP ESI, 0Ah] ESI=3..0xA
0x401033 (.text+0x33), e=	8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=	1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=	1 [POP ESI]
0x401038 (.text+0x38), e=	1 [RETN] EAX=0

We can use grep here.

## ARM

### Non-optimizing Keil 6/2013 (ARM mode)

```
main
    STMFD   SP!, {R4,LR}
    MOV     R4, #2
    B      loc_368
loc_35C ; CODE XREF: main+1C
    MOV     R0, R4
    BL     printing_function
    ADD     R4, R4, #1
```

## 1.16. LOOPS

```
loc_368 ; CODE XREF: main+8
    CMP    R4, #0xA
    BLT    loc_35C
    MOV    R0, #0
    LDMFD  SP!, {R4,PC}
```

Iteration counter  $i$  is to be stored in the `R4` register. The `MOV R4, #2` instruction just initializes  $i$ . The `MOV R0, R4` and `BL printing_function` instructions compose the body of the loop, the first instruction preparing the argument for `f()` function and the second calling the function. The `ADD R4, R4, #1` instruction just adds 1 to the  $i$  variable at each iteration. `CMP R4, #0xA` compares  $i$  with `0xA` (10). The next instruction `BLT (Branch Less Than)` jumps if  $i$  is less than 10. Otherwise, 0 is to be written into `R0` (since our function returns 0) and function execution finishes.

### Optimizing Keil 6/2013 (Thumb mode)

```
_main
    PUSH   {R4,LR}
    MOVS   R4, #2

loc_132
    MOVS   R0, R4           ; CODE XREF: _main+E
    BL     printing_function
    ADDS   R4, R4, #1
    CMP    R4, #0xA
    BLT   loc_132
    MOVS   R0, #0
    POP    {R4,PC}
```

Practically the same.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
_main
    PUSH   {R4,R7,LR}
    MOVW   R4, #0x1124 ; "%d\n"
    MOVS   R1, #2
    MOVT.W R4, #0
    ADD    R7, SP, #4
    ADD    R4, PC
    MOV    R0, R4
    BLX   _printf
    MOV    R0, R4
    MOVS   R1, #3
    BLX   _printf
    MOV    R0, R4
    MOVS   R1, #4
    BLX   _printf
    MOV    R0, R4
    MOVS   R1, #5
    BLX   _printf
    MOV    R0, R4
    MOVS   R1, #6
    BLX   _printf
    MOV    R0, R4
    MOVS   R1, #7
    BLX   _printf
    MOV    R0, R4
    MOVS   R1, #8
    BLX   _printf
    MOV    R0, R4
    MOVS   R1, #9
    BLX   _printf
    MOVS   R0, #0
```

## 1.16. LOOPS

```
POP          {R4,R7,PC}
```

In fact, this was in my `f()` function:

```
void printing_function(int i)
{
    printf ("%d\n", i);
}
```

So, LLVM not just *unrolled* the loop, but also *inlined* my very simple function `f()`, and inserted its body 8 times instead of calling it.

This is possible when the function is so simple (like mine) and when it is not called too much (like here).

### ARM64: Optimizing GCC 4.9.1

Listing 1.167: Optimizing GCC 4.9.1

```
printing_function:
; prepare second argument of printf():
    mov    w1, w0
; load address of the "f(%d)\n" string
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; just branch here instead of branch with link and return:
    b      printf
main:
; save FP and LR in the local stack:
    stp    x29, x30, [sp, -32]!
; set up stack frame:
    add    x29, sp, 0
; save contents of X19 register in the local stack:
    str    x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
    mov    w19, 2
.L3:
; prepare first argument of printing_function():
    mov    w0, w19
; increment counter register.
    add    w19, w19, 1
; W0 here still holds value of counter value before increment.
    bl     printing_function
; is it end?
    cmp    w19, 10
; no, jump to the loop body begin:
    bne   .L3
; return 0
    mov    w0, 0
; restore contents of X19 register:
    ldr    x19, [sp,16]
; restore FP and LR values:
    ldp    x29, x30, [sp], 32
    ret
.LC0:
.string "f(%d)\n"
```

### ARM64: Non-optimizing GCC 4.9.1

Listing 1.168: Non-optimizing GCC 4.9.1 -fno-inline

```
printing_function:
; prepare second argument of printf():
    mov    w1, w0
; load address of the "f(%d)\n" string
```

## 1.16. LOOPS

```
    adrp    x0, .LC0
    add     x0, x0, :lo12:.LC0
; just branch here instead of branch with link and return:
    b      printf
main:
; save FP and LR in the local stack:
    stp    x29, x30, [sp, -32]!
; set up stack frame:
    add    x29, sp, 0
; save contents of X19 register in the local stack:
    str    x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
    mov    w19, 2
.L3:
; prepare first argument of printing_function():
    mov    w0, w19
; increment counter register.
    add    w19, w19, 1
; W0 here still holds value of counter value before increment.
    bl    printing_function
; is it end?
    cmp    w19, 10
; no, jump to the loop body begin:
    bne    .L3
; return 0
    mov    w0, 0
; restore contents of X19 register:
    ldr    x19, [sp,16]
; restore FP and LR values:
    ldp    x29, x30, [sp], 32
    ret
.LC0:
.string "f(%d)\n"
```

## MIPS

Listing 1.169: Non-optimizing GCC 4.4.5 (IDA)

```
main:
; IDA is not aware of variable names in local stack
; We gave them names manually:
i          = -0x10
saved_FP   = -8
saved_RA   = -4

; function prologue:
    addiu $sp, -0x28
    sw    $ra, 0x28+saved_RA($sp)
    sw    $fp, 0x28+saved_FP($sp)
    move $fp, $sp
; initialize counter at 2 and store this value in local stack
    li    $v0, 2
    sw    $v0, 0x28+i($fp)
; pseudoinstruction. "BEQ $ZERO, $ZERO, loc_9C" there in fact:
    b    loc_9C
    or    $at, $zero ; branch delay slot, NOP

loc_80:           # CODE XREF: main+48
; load counter value from local stack and call printing_function():
    lw    $a0, 0x28+i($fp)
    jal   printing_function
    or    $at, $zero ; branch delay slot, NOP
; load counter, increment it, store it back:
    lw    $v0, 0x28+i($fp)
    or    $at, $zero ; NOP
    addiu $v0, 1
```

## 1.16. LOOPS

```
sw      $v0, 0x28+i($fp)

loc_9C:                      # CODE XREF: main+18
; check counter, is it 10?
    lw      $v0, 0x28+i($fp)
    or      $at, $zero ; NOP
    slti   $v0, 0xA
; if it is less than 10, jump to loc_80 (loop body begin):
    bnez   $v0, loc_80
    or      $at, $zero ; branch delay slot, NOP
; finishing, return 0:
    move   $v0, $zero
; function epilogue:
    move   $sp, $fp
    lw      $ra, 0x28+saved_RA($sp)
    lw      $fp, 0x28+saved_FP($sp)
    addiu $sp, 0x28
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP
```

The instruction that's new to us is **B**. It is actually the pseudo instruction (**BEQ**).

### One more thing

In the generated code we can see: after initializing *i*, the body of the loop is not to be executed, as the condition for *i* is checked first, and only after that loop body can be executed. And that is correct.

Because, if the loop condition is not met at the beginning, the body of the loop must not be executed. This is possible in the following case:

```
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

If *total\_entries\_to\_process* is 0, the body of the loop must not be executed at all.

This is why the condition checked before the execution.

However, an optimizing compiler may swap the condition check and loop body, if it sure that the situation described here is not possible (like in the case of our very simple example and using compilers like Keil, Xcode (LLVM), MSVC in optimization mode).

### 1.16.2 Memory blocks copying routine

Real-world memory copy routines may copy 4 or 8 bytes at each iteration, use SIMD<sup>103</sup>, vectorization, etc. But for the sake of simplicity, this example is the simplest possible.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
}
```

### Straight-forward implementation

Listing 1.170: GCC 4.9 x64 optimized for size (-Os)

```
my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block
```

<sup>103</sup>Single instruction, multiple data

## 1.16. LOOPS

```
; initialize counter (i) at 0
    xor    eax, eax
.L2:
; all bytes copied? exit then:
    cmp    rax, rdx
    je     .L5
; load byte at RSI+i:
    mov    cl, BYTE PTR [rsi+rax]
; store byte at RDI+i:
    mov    BYTE PTR [rdi+rax], cl
    inc    rax ; i++
    jmp    .L2
.L5:
    ret
```

Listing 1.171: GCC 4.9 ARM64 optimized for size (-Os)

```
my_memcpy:
; X0 = destination address
; X1 = source address
; X2 = size of block

; initialize counter (i) at 0
    mov    x3, 0
.L2:
; all bytes copied? exit then:
    cmp    x3, x2
    beq    .L5
; load byte at X1+i:
    ldrb   w4, [x1,x3]
; store byte at X0+i:
    strb   w4, [x0,x3]
    add    x3, x3, 1 ; i++
    b     .L2
.L5:
    ret
```

Listing 1.172: Optimizing Keil 6/2013 (Thumb mode)

```
my_memcpy PROC
; R0 = destination address
; R1 = source address
; R2 = size of block

    PUSH    {r4,lr}
; initialize counter (i) at 0
    MOVS    r3,#0
; condition checked at the end of function, so jump there:
    B       |L0.12|
|L0.6|
; load byte at R1+i:
    LDRB    r4,[r1,r3]
; store byte at R0+i:
    STRB    r4,[r0,r3]
; i++
    ADDS    r3,r3,#1
|L0.12|
; i<size?
    CMP    r3,r2
; jump to the loop begin if its so:
    BCC    |L0.6|
    POP    {r4,pc}
ENDP
```

## ARM in ARM mode

Keil in ARM mode takes full advantage of conditional suffixes:

## 1.16. LOOPS

Listing 1.173: Optimizing Keil 6/2013 (ARM mode)

```
my_memcpy PROC
; R0 = destination address
; R1 = source address
; R2 = size of block

; initialize counter (i) at 0
    MOV      r3,#0
|L0.4|
; all bytes copied?
    CMP      r3,r2
; the following block is executed only if less than condition,
; i.e., if R2<R3 or i<size.
; load byte at R1+i:
    LDRBCC  r12,[r1,r3]
; store byte at R0+i:
    STRBCC  r12,[r0,r3]
; i++
    ADDCC   r3,r3,#1
; the last instruction of the conditional block.
; jump to loop begin if i<size
; do nothing otherwise (i.e., if i>=size)
    BCC     |L0.4|
; return
    BX      lr
ENDP
```

That's why there is only one branch instruction instead of 2.

## MIPS

Listing 1.174: GCC 4.4.5 optimized for size (-Os) (IDA)

```
my_memcpy:
; jump to loop check part:
    b      loc_14
; initialize counter (i) at 0
; it will always reside in $v0:
    move   $v0, $zero ; branch delay slot

loc_8:                      # CODE XREF: my_memcpy+1C
; load byte as unsigned at address in $t0 to $v1:
    lbu   $v1, 0($t0)
; increment counter (i):
    addiu $v0, 1
; store byte at $a3
    sb    $v1, 0($a3)

loc_14:                     # CODE XREF: my_memcpy
; check if counter (i) in $v0 is still less then 3rd function argument ("cnt" in $a2):
    sltu  $v1, $v0, $a2
; form address of byte in source block:
    addu  $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; jump to loop body if counter sill less then "cnt":
    bnez  $v1, loc_8
; form address of byte in destination block ($a3 = $a0+$v0 = dst+i):
    addu  $a3, $a0, $v0 ; branch delay slot
; finish if BNEZ wasnt triggered:
    jr    $ra
    or    $at, $zero ; branch delay slot, NOP
```

Here we have two new instructions: LBU (“Load Byte Unsigned”) and SB (“Store Byte”).

Just like in ARM, all MIPS registers are 32-bit wide, there are no byte-wide parts like in x86.

So when dealing with single bytes, we have to allocate whole 32-bit registers for them.

## 1.16. LOOPS

LBU loads a byte and clears all other bits ("Unsigned").

On the other hand, LB ("Load Byte") instruction sign-extends the loaded byte to a 32-bit value.

SB just writes a byte from lowest 8 bits of register to memory.

### Vectorization

Optimizing GCC can do much more on this example: [1.29.1 on page 414](#).

## 1.16.3 Conclusion

Rough skeleton of loop from 2 to 9 inclusive:

Listing 1.175: x86

```
mov [counter], 2 ; initialization
jmp check
body:
; loop body
; do something here
; use counter variable in local stack
add [counter], 1 ; increment
check:
cmp [counter], 9
jle body
```

The increment operation may be represented as 3 instructions in non-optimized code:

Listing 1.176: x86

```
MOV [counter], 2 ; initialization
JMP check
body:
; loop body
; do something here
; use counter variable in local stack
MOV REG, [counter] ; increment
INC REG
MOV [counter], REG
check:
CMP [counter], 9
JLE body
```

If the body of the loop is short, a whole register can be dedicated to the counter variable:

Listing 1.177: x86

```
MOV EBX, 2 ; initialization
JMP check
body:
; loop body
; do something here
; use counter in EBX, but do not modify it!
INC EBX ; increment
check:
CMP EBX, 9
JLE body
```

Some parts of the loop may be generated by compiler in different order:

Listing 1.178: x86

```
MOV [counter], 2 ; initialization
JMP label_check
label_increment:
ADD [counter], 1 ; increment
label_check:
```

## 1.16. LOOPS

```
CMP [counter], 10
JGE exit
; loop body
; do something here
; use counter variable in local stack
JMP label_increment
exit:
```

Usually the condition is checked *before* loop body, but the compiler may rearrange it in a way that the condition is checked *after* loop body.

This is done when the compiler is sure that the condition is always *true* on the first iteration, so the body of the loop is to be executed at least once:

Listing 1.179: x86

```
MOV REG, 2 ; initialization
body:
; loop body
; do something here
; use counter in REG, but do not modify it!
INC REG ; increment
CMP REG, 10
JL body
```

Using the `LOOP` instruction. This is rare, compilers are not using it. When you see it, it's a sign that this piece of code is hand-written:

Listing 1.180: x86

```
; count from 10 to 1
MOV ECX, 10
body:
; loop body
; do something here
; use counter in ECX, but do not modify it!
LOOP body
```

ARM.

The `R4` register is dedicated to counter variable in this example:

Listing 1.181: ARM

```
MOV R4, 2 ; initialization
B check
body:
; loop body
; do something here
; use counter in R4, but do not modify it!
ADD R4,R4, #1 ; increment
check:
CMP R4, #10
BLT body
```

### 1.16.4 Exercises

- <http://challenges.re/54>
- <http://challenges.re/55>
- <http://challenges.re/56>
- <http://challenges.re/57>

## 1.17 More about strings

### 1.17.1 strlen()

Let's talk about loops one more time. Often, the `strlen()` function <sup>104</sup> is implemented using a `while()` statement. Here is how it is done in the MSVC standard libraries:

```
int my_strlen (const char * str)
{
    const char *eos = str;

    while( *eos++ ) ;

    return( eos - str - 1 );
}

int main()
{
    // test
    return my_strlen("hello!");
};
```

## x86

### Non-optimizing MSVC

Let's compile:

```
_eos$ = -4                      ; size = 4
_str$ = 8                       ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp] ; place pointer to string from "str"
    mov     DWORD PTR _eos$[ebp], eax ; place it to local variable "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp] ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit value to EDX with sign ↴
    ; extension

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add     eax, 1                  ; increment EAX
    mov     DWORD PTR _eos$[ebp], eax ; place EAX back to "eos"
    test    edx, edx              ; EDX is zero?
    je      SHORT $LN1@strlen_    ; yes, then finish loop
    jmp     SHORT $LN2@strlen_    ; continue loop
$LN1@strlen_:
    ; here we calculate the difference between two pointers

    mov     eax, DWORD PTR _eos$[ebp]
    sub     eax, DWORD PTR _str$[ebp]
    sub     eax, 1                  ; subtract 1 and return result
    mov     esp, ebp
    pop     ebp
    ret     0
_strlen ENDP
```

We get two new instructions here: `MOVsx` and `TEST`.

<sup>104</sup>counting the characters in a string in the C language

## 1.17. MORE ABOUT STRINGS

The first one—`MOVsx`—takes a byte from an address in memory and stores the value in a 32-bit register. `MOVsx` stands for *MOV with Sign-Extend*. `MOVsx` sets the rest of the bits, from the 8th to the 31th, to 1 if the source byte is *negative* or to 0 if it is *positive*.

And here is why.

By default, the *char* type is signed in MSVC and GCC. If we have two values of which one is *char* and the other is *int*, (*int* is signed too), and if the first value contain -2 (coded as `0xFE`) and we just copy this byte into the *int* container, it makes `0x000000FE`, and this from the point of signed *int* view is 254, but not -2. In signed *int*, -2 is coded as `0xFFFFFFF`. So if we have to transfer `0xFE` from a variable of *char* type to *int*, we have to identify its sign and extend it. That is what `MOVsx` does.

You can also read about it in “*Signed number representations*” section ( [2.2 on page 454](#)).

It's hard to say if the compiler needs to store a *char* variable in `EDX`, it could just take a 8-bit register part (for example `DL`). Apparently, the compiler's [register allocator](#) works like that.

Then we see `TEST EDX, EDX`. You can read more about the `TEST` instruction in the section about bit fields ( [1.22 on page 304](#)). Here this instruction just checks if the value in `EDX` equals to 0.

## Non-optimizing GCC

Let's try GCC 4.4.1:

```
public strlen
strlen proc near

eos      = dword ptr -4
arg_0    = dword ptr  8

        push   ebp
        mov    ebp, esp
        sub    esp, 10h
        mov    eax, [ebp+arg_0]
        mov    [ebp+eos], eax

loc_80483F0:
        mov    eax, [ebp+eos]
        movzx eax, byte ptr [eax]
        test   al, al
        setnz al
        add    [ebp+eos], 1
        test   al, al
        jnz    short loc_80483F0
        mov    edx, [ebp+eos]
        mov    eax, [ebp+arg_0]
        mov    ecx, edx
        sub    ecx, eax
        mov    eax, ecx
        sub    eax, 1
        leave
        retn
strlen endp
```

The result is almost the same as in MSVC, but here we see `MOVZX` instead of `MOVsx`. `MOVZX` stands for *MOV with Zero-Extend*. This instruction copies a 8-bit or 16-bit value into a 32-bit register and sets the rest of the bits to 0. In fact, this instruction is convenient only because it enable us to replace this instruction pair:

```
xor eax, eax / mov al, [...].
```

On the other hand, it is obvious that the compiler could produce this code:

`mov al, byte ptr [eax] / test al, al`—it is almost the same, however, the highest bits of the `EAX` register will contain random noise. But let's think it is compiler's drawback—it cannot produce more understandable code. Strictly speaking, the compiler is not obliged to emit understandable (to humans) code at all.

## 1.17. MORE ABOUT STRINGS

The next new instruction for us is `SETNZ`. Here, if `AL` doesn't contain zero, `test al, al` sets the `ZF` flag to 0, but `SETNZ`, if `ZF==0` (`NZ` stands for *not zero*) sets `AL` to 1. Speaking in natural language, *if AL is not zero, let's jump to loc\_80483F0*. The compiler emits some redundant code, but let's not forget that the optimizations are turned off.

### Optimizing MSVC

Now let's compile all this in MSVC 2012, with optimizations turned on (`/Ox`):

Listing 1.182: Optimizing MSVC 2012 /Ob0

```
_str$ = 8           ; size = 4
_strlen PROC
    mov     edx, DWORD PTR _str$[esp-4] ; EDX -> pointer to the string
    mov     eax, edx                   ; move to EAX
$LL2@strlen:
    mov     cl, BYTE PTR [eax]        ; CL = *EAX
    inc     eax                     ; EAX++
    test    cl, cl                 ; CL==0?
    jne    SHORT $LL2@strlen       ; no, continue loop
    sub     eax, edx               ; calculate pointers difference
    dec     eax                     ; decrement EAX
    ret     0
_strlen ENDP
```

Now it is all simpler. Needless to say, the compiler could use registers with such efficiency only in small functions with a few local variables.

`INC / DEC` —are [increment/decrement](#) instructions, in other words: add or subtract 1 to/from a variable.

**Optimizing MSVC + OllyDbg**

We can try this (optimized) example in OllyDbg. Here is the first iteration:

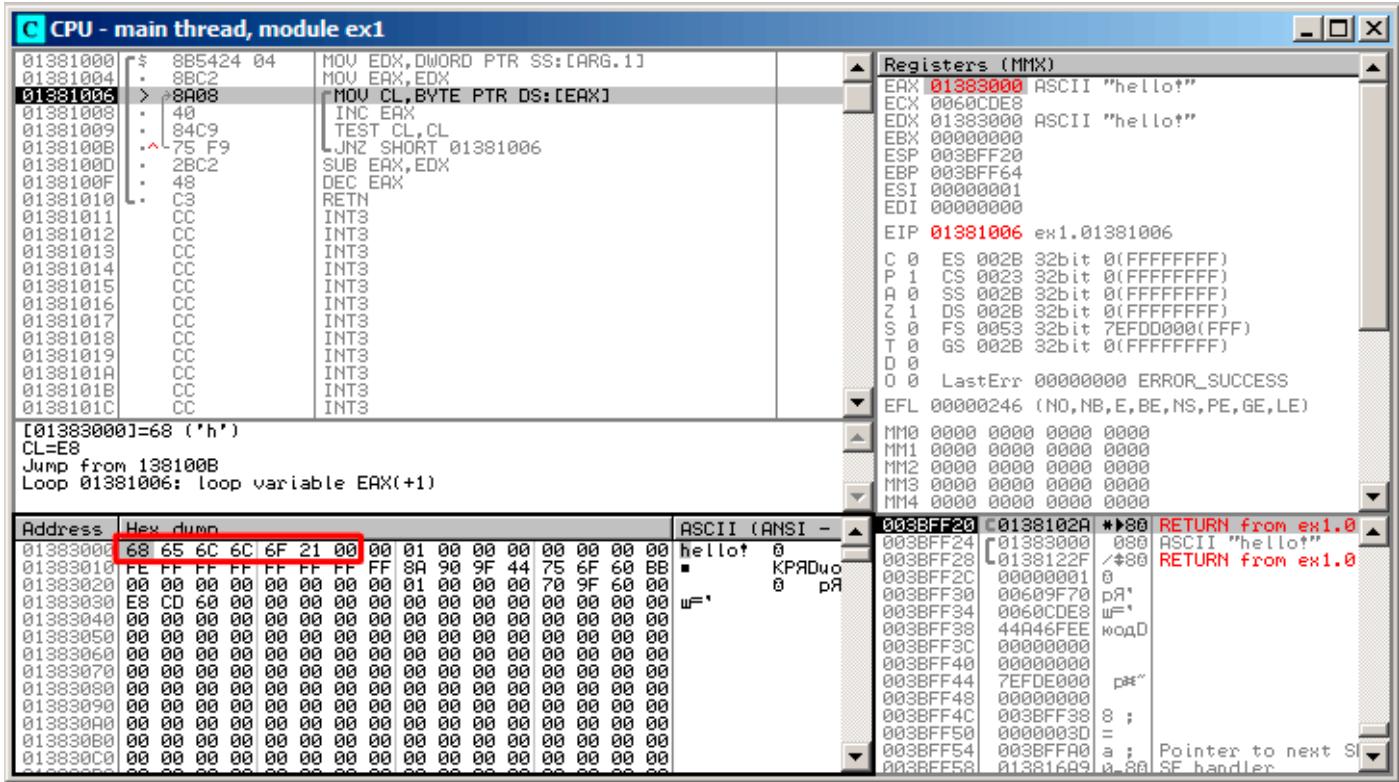


Figure 1.58: OllyDbg: first iteration start

We see that OllyDbg found a loop and, for convenience, wrapped its instructions in brackets. By clicking the right button on `EAX`, we can choose “Follow in Dump” and the memory window scrolls to the right place. Here we can see the string “hello!” in memory. There is at least one zero byte after it and then random garbage.

If OllyDbg sees a register with a valid address in it, that points to some string, it is shown as a string.

## 1.17. MORE ABOUT STRINGS

Let's press F8 (step over) a few times, to get to the start of the body of the loop:

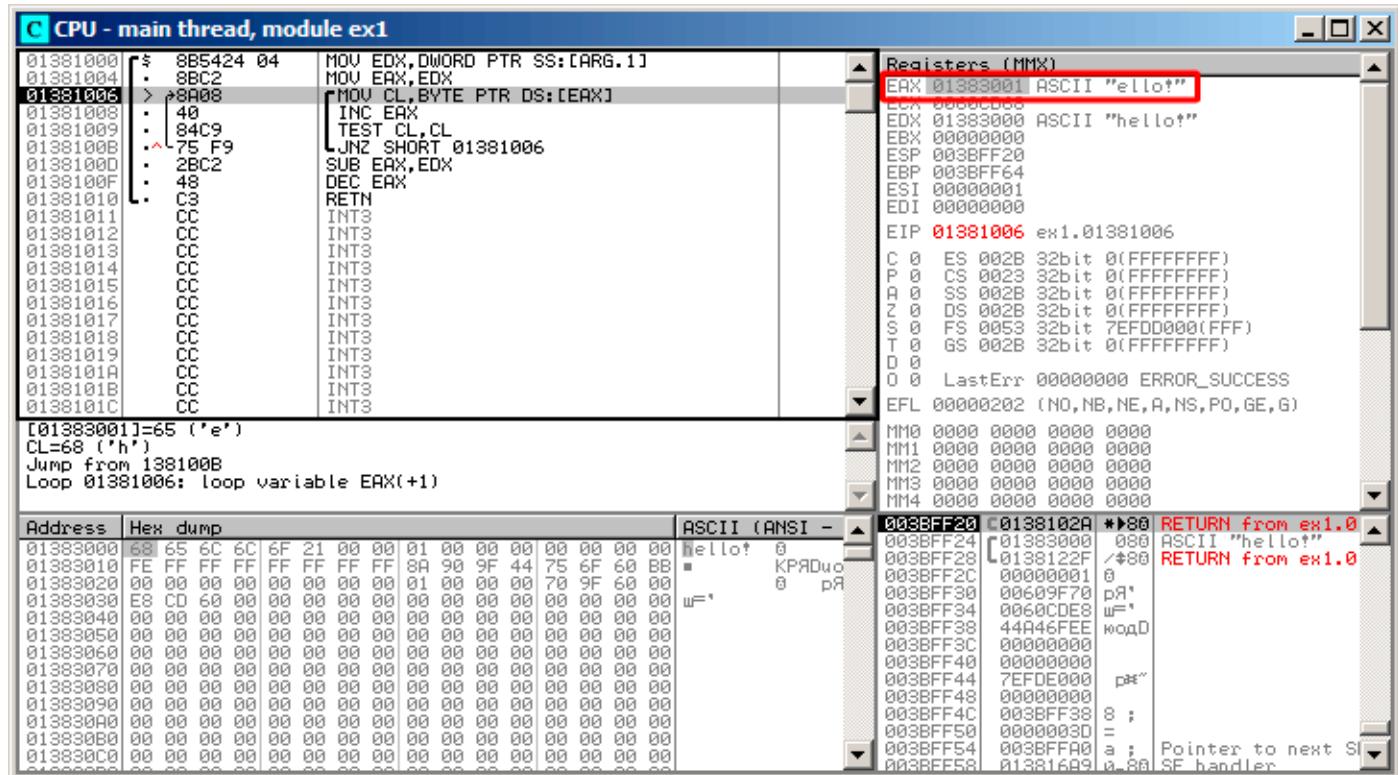


Figure 1.59: OllyDbg: second iteration start

We see that **EAX** contains the address of the second character in the string.

## 1.17. MORE ABOUT STRINGS

We have to press F8 enough number of times in order to escape from the loop:

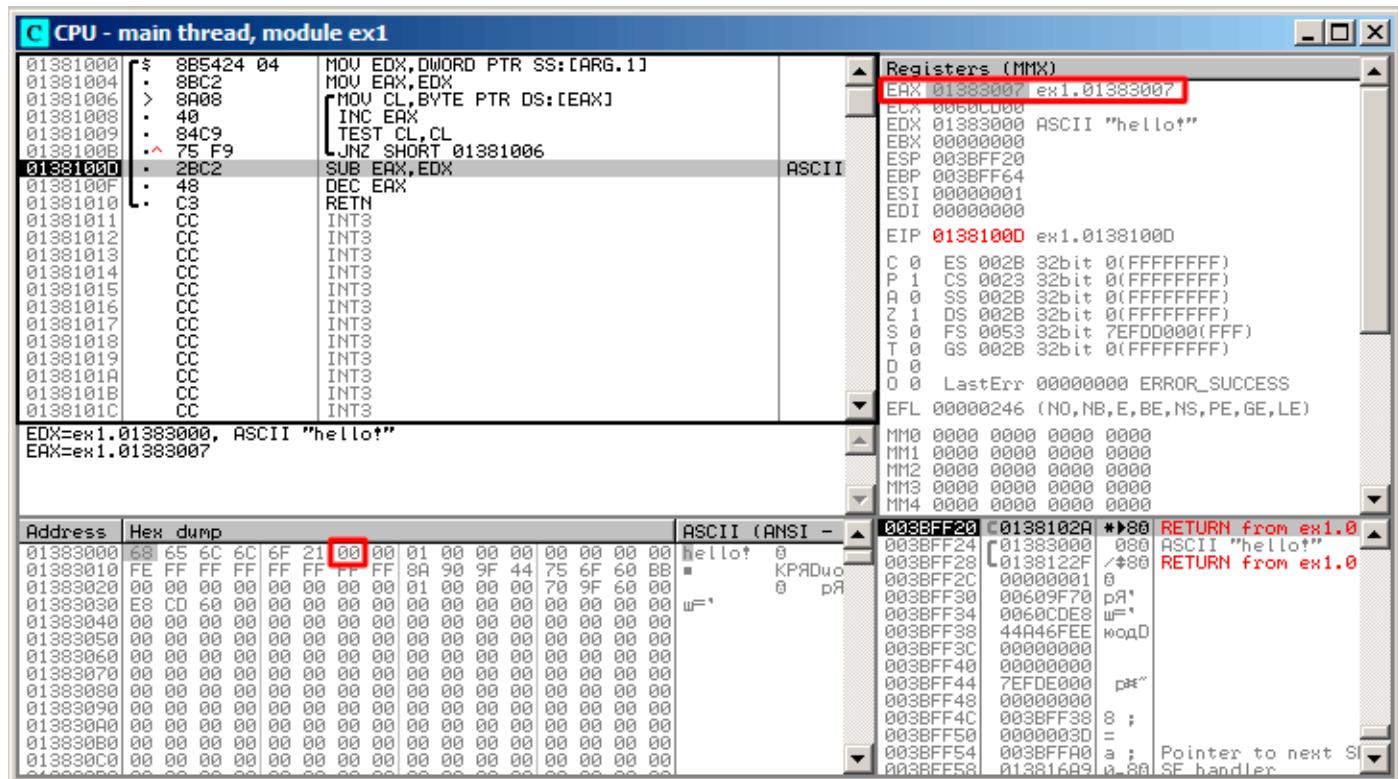


Figure 1.60: OllyDbg: pointers difference to be calculated now

We see that `EAX` now contains the address of zero byte that's right after the string. Meanwhile, `EDX` hasn't changed, so it still pointing to the start of the string.

The difference between these two addresses is being calculated now.

## 1.17. MORE ABOUT STRINGS

The `SUB` instruction just got executed:

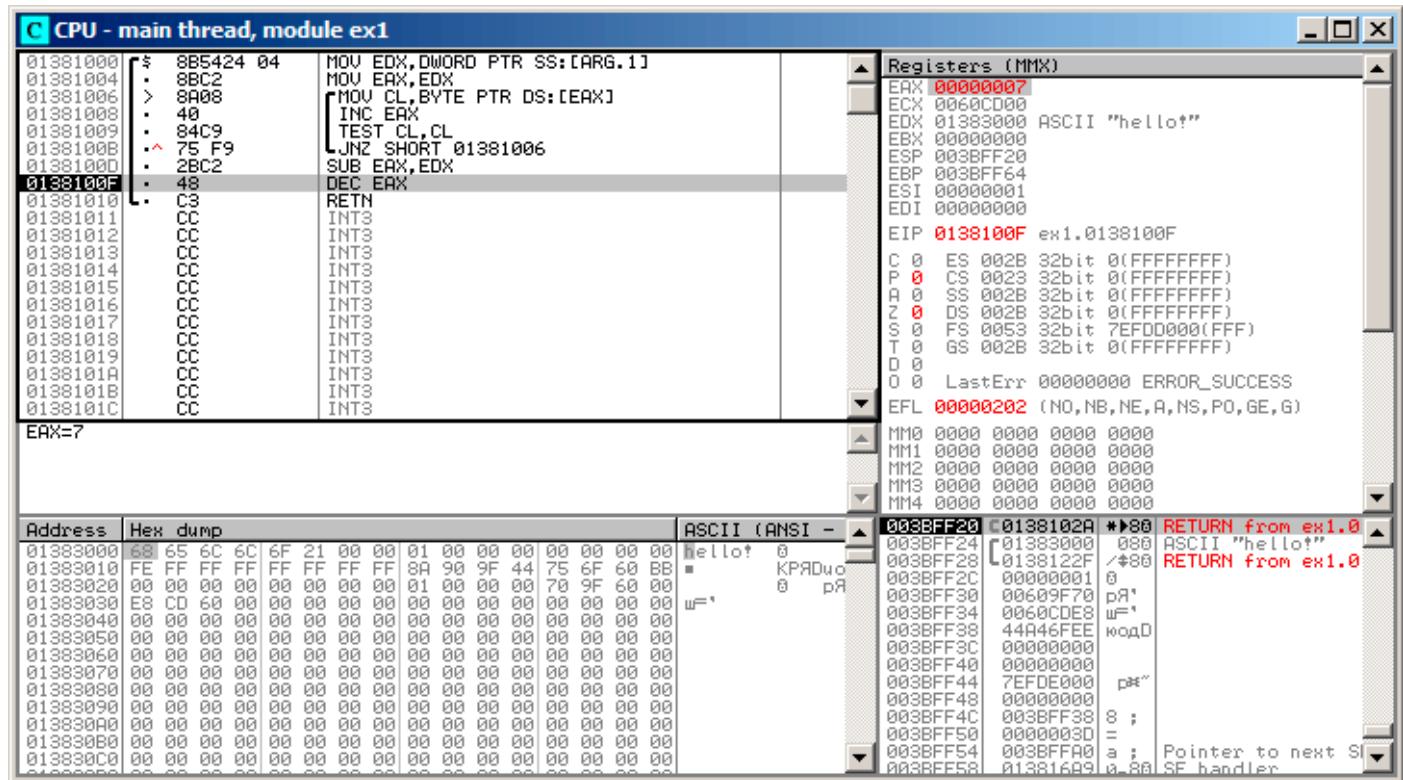


Figure 1.61: OllyDbg: `EAX` to be decremented now

The difference of pointers is in the `EAX` register now—7. Indeed, the length of the “hello!” string is 6, but with the zero byte included—7. But `strlen()` must return the number of non-zero characters in the string. So the decrement executes and then the function returns.

## Optimizing GCC

Let's check GCC 4.4.1 with optimizations turned on (-O3 key):

```

public strlen
strlen proc near

arg_0      = dword ptr 8

        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp+arg_0]
        mov     eax, ecx

loc_8048418:
        movzx  edx, byte ptr [eax]
        add    eax, 1
        test   dl, dl
        jnz    short loc_8048418
        not    ecx
        add    eax, ecx
        pop    ebp
        retn
strlen endp

```

Here GCC is almost the same as MSVC, except for the presence of `Movzx`. However, here `Movzx` could be replaced with

`mov dl, byte ptr [eax]`.

## 1.17. MORE ABOUT STRINGS

Perhaps it is simpler for GCC's code generator to *remember* the whole 32-bit `EDX` register is allocated for a `char` variable and it then can be sure that the highest bits has no any noise at any point.

After that we also see a new instruction—`NOT`. This instruction inverts all bits in the operand.

You can say that it is a synonym to the `XOR ECX, 0xffffffff` instruction. `NOT` and the following `ADD` calculate the pointer difference and subtract 1, just in a different way. At the start `ECX`, where the pointer to `str` is stored, gets inverted and 1 is subtracted from it.

See also: "Signed number representations" ( [2.2 on page 454](#)).

In other words, at the end of the function just after loop body, these operations are executed:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... and this is effectively equivalent to:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Why did GCC decide it would be better? Hard to guess. But perhaps the both variants are equivalent in efficiency.

## ARM

### 32-bit ARM

#### Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 1.183: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
_strlen

eos  = -8
str  = -4

    SUB   SP, SP, #8 ; allocate 8 bytes for local variables
    STR   R0, [SP,#8+str]
    LDR   R0, [SP,#8+str]
    STR   R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
    LDR   R0, [SP,#8+eos]
    ADD   R1, R0, #1
    STR   R1, [SP,#8+eos]
    LDRSB R0, [R0]
    CMP   R0, #0
    BEQ   loc_2CD4
    B     loc_2CB8

loc_2CD4 ; CODE XREF: _strlen+24
    LDR   R0, [SP,#8+eos]
    LDR   R1, [SP,#8+str]
    SUB   R0, R0, R1 ; R0=eos-str
    SUB   R0, R0, #1 ; R0=R0-1
    ADD   SP, SP, #8 ; free allocated 8 bytes
    BX    LR
```

## 1.17. MORE ABOUT STRINGS

Non-optimizing LLVM generates too much code, however, here we can see how the function works with local variables in the stack. There are only two local variables in our function: *eos* and *str*. In this listing, generated by [IDA](#), we have manually renamed *var\_8* and *var\_4* to *eos* and *str*.

The first instructions just saves the input values into both *str* and *eos*.

The body of the loop starts at label *loc\_2CB8*.

The first three instruction in the loop body (`LDR`, `ADD`, `STR`) load the value of *eos* into `R0`. Then the value is [incremented](#) and saved back into *eos*, which is located in the stack.

The next instruction, `LDRSB R0, [R0]` (“Load Register Signed Byte”), loads a byte from memory at the address stored in `R0` and sign-extends it to 32-bit <sup>105</sup>. This is similar to the `MOVVSX` instruction in x86.

The compiler treats this byte as signed since the *char* type is signed according to the C standard. It was already written about it ([1.17.1 on page 201](#)) in this section, in relation to x86.

It has to be noted that it is impossible to use 8- or 16-bit part of a 32-bit register in ARM separately of the whole register, as it is in x86.

Apparently, it is because x86 has a huge history of backwards compatibility with its ancestors up to the 16-bit 8086 and even 8-bit 8080, but ARM was developed from scratch as a 32-bit RISC-processor.

Consequently, in order to process separate bytes in ARM, one has to use 32-bit registers anyway.

So, `LDRSB` loads bytes from the string into `R0`, one by one. The following `CMP` and `BEQ` instructions check if the loaded byte is 0. If it's not 0, control passes to the start of the body of the loop. And if it's 0, the loop ends.

At the end of the function, the difference between *eos* and *str* is calculated, 1 is subtracted from it, and resulting value is returned via `R0`.

N.B. Registers were not saved in this function.

That's because in the ARM calling convention registers `R0 - R3` are “scratch registers”, intended for arguments passing, and we're not required to restore their value when the function exits, since the calling function will not use them anymore. Consequently, they may be used for anything we want.

No other registers are used here, so that is why we have nothing to save on the stack.

Thus, control may be returned back to calling function by a simple jump (`BX`), to the address in the `LR` register.

## Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

Listing 1.184: Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```
_strlen
        MOV      R1, R0
loc_2DF6
        LDRB.W  R2, [R1],#1
        CMP      R2, #0
        BNE      loc_2DF6
        MVNS    R0, R0
        ADD      R0, R1
        BX      LR
```

As optimizing LLVM concludes, *eos* and *str* do not need space on the stack, and can always be stored in registers.

Before the start of the loop body, *str* is always in `R0`, and *eos*—in `R1`.

The `LDRB.W R2, [R1],#1` instruction loads a byte from the memory at the address stored in `R1`, to `R2`, sign-extending it to a 32-bit value, but not just that. `#1` at the instruction's end is implies “Post-indexed addressing”, which means that 1 is to be added to `R1` after the byte is loaded. Read more about it: [1.3.2 on page 441](#).

Then you can see `CMP` and `BNE106` in the body of the loop, these instructions continue looping until 0 is

<sup>105</sup>The Keil compiler treats the *char* type as signed, just like MSVC and GCC.

<sup>106</sup>(PowerPC, ARM) Branch if Not Equal

## 1.17. MORE ABOUT STRINGS

found in the string.

`MVNS`<sup>107</sup> (inverts all bits, like `NOT` in x86) and `ADD` instructions compute  $eos - str - 1$ . In fact, these two instructions compute  $R0 = str + eos$ , which is effectively equivalent to what was in the source code, and why it is so, was already explained here ([1.17.1 on page 208](#)).

Apparently, LLVM, just like GCC, concludes that this code can be shorter (or faster).

### Optimizing Keil 6/2013 (ARM mode)

Listing 1.185: Optimizing Keil 6/2013 (ARM mode)

```
_strlen
      MOV     R1, R0
loc_2C8
      LDRB   R2, [R1],#1
      CMP    R2, #0
      SUBEQ R0, R1, R0
      SUBEQ R0, R0, #1
      BNE    loc_2C8
      BX     LR
```

Almost the same as what we saw before, with the exception that the  $str - eos - 1$  expression can be computed not at the function's end, but right in the body of the loop. The `-EQ` suffix, as we may recall, implies that the instruction executes only if the operands in the `CMP` that has been executed before were equal to each other. Thus, if `R0` contains 0, both `SUBEQ` instructions execute and result is left in the `R0` register.

## ARM64

### Optimizing GCC (Linaro) 4.9

```
my_strlen:
      mov     x1, x0
      ; X1 is now temporary pointer (eos), acting like cursor
.L58:
      ; load byte from X1 to W2, increment X1 (post-index)
      ldrb   w2, [x1],1
      ; Compare and Branch if NonZero: compare W2 with 0, jump to .L58 if it is not
      cbnz   w2, .L58
      ; calculate difference between initial pointer in X0 and current address in X1
      sub    x0, x1, x0
      ; decrement lowest 32-bit of result
      sub    w0, w0, #1
      ret
```

The algorithm is the same as in [1.17.1 on page 203](#): find a zero byte, calculate the difference between the pointers and decrement the result by 1. Some comments were added by the author of this book.

The only thing worth noting is that our example is somewhat wrong:

`my_strlen()` returns 32-bit `int`, while it has to return `size_t` or another 64-bit type.

The reason is that, theoretically, `strlen()` can be called for a huge blocks in memory that exceeds 4GB, so it must be able to return a 64-bit value on 64-bit platforms.

Because of my mistake, the last `SUB` instruction operates on a 32-bit part of register, while the penultimate `SUB` instruction works on full the 64-bit register (it calculates the difference between the pointers).

It's my mistake, it is better to leave it as is, as an example of how the code could look like in such case.

<sup>107</sup>MoVe Not

```

my_strlen:
; function prologue
    sub    sp, sp, #32
; first argument (str) will be stored in [sp,8]
    str    x0, [sp,8]
    ldr    x0, [sp,8]
; copy "str" to "eos" variable
    str    x0, [sp,24]
    nop
.L62:
; eos++
    ldr    x0, [sp,24] ; load "eos" to X0
    add    x1, x0, 1    ; increment X0
    str    x1, [sp,24] ; save X0 to "eos"
; load byte from memory at address in X0 to W0
    ldrb   w0, [x0]
; is it zero? (WZR is the 32-bit register always contain zero)
    cmp    w0, wzr
; jump if not zero (Branch Not Equal)
    bne    .L62
; zero byte found. now calculate difference.
; load "eos" to X1
    ldr    x1, [sp,24]
; load "str" to X0
    ldr    x0, [sp,8]
; calculate difference
    sub    x0, x1, x0
; decrement result
    sub    w0, w0, #1
; function epilogue
    add    sp, sp, 32
    ret

```

It's more verbose. The variables are often tossed here to and from memory (local stack). The same mistake here: the decrement operation happens on a 32-bit register part.

## MIPS

Listing 1.186: Optimizing GCC 4.4.5 (IDA)

```

my_strlen:
; "eos" variable will always reside in $v1:
    move   $v1, $a0

loc_4:
; load byte at address in "eos" into $a1:
    lb    $a1, 0($v1)
    or    $at, $zero ; load delay slot, NOP
; if loaded byte is not zero, jump to loc_4:
    bnez $a1, loc_4
; increment "eos" anyway:
    addiu $v1, 1 ; branch delay slot
; loop finished. invert "str" variable:
    nor   $v0, $zero, $a0
; $v0=-str-1
    jr    $ra
; return value = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
    addu $v0, $v1, $v0 ; branch delay slot

```

MIPS lacks a **NOT** instruction, but has **NOR** which is **OR + NOT** operation.

This operation is widely used in digital electronics<sup>108</sup>. For example, the Apollo Guidance Computer used in the Apollo program, was built by only using 5600 NOR gates: [Jens Eickhoff, *Onboard Computers*,

<sup>108</sup>NOR is called “universal gate”

## 1.18. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

*Onboard Software and Satellite Operations: An Introduction, (2011)*. But NOR element isn't very popular in computer programming.

So, the NOT operation is implemented here as `NOR DST, $ZERO, SRC`.

From fundamentals [2.2 on page 454](#) we know that bitwise inverting a signed number is the same as changing its sign and subtracting 1 from the result.

So what `NOT` does here is to take the value of `str` and transform it into  $-str - 1$ . The addition operation that follows prepares result.

### 1.17.2 Boundaries of strings

It's interesting to note, how parameters are passed into `win32 GetOpenFileName()` function. In order to call it, one must set list of allowed file extensions:

```
OPENFILENAME *LPOPENFILENAME;
...
char * filter = "Text files (*.txt)\0*.txt\0MS Word files (*.doc)\0*.doc\0\0";
...
LPOPENFILENAME = (OPENFILENAME *)malloc(sizeof(OPENFILENAME));
...
LPOPENFILENAME->lpstrFilter = filter;
...
if(GetOpenFileName(LPOPENFILENAME))
{
    ...
}
```

What happens here is that list of strings are passed into `GetOpenFileName()`. It is not a problem to parse it: whenever you encounter single zero byte, this is an item. Whenever you encounter two zero bytes, this is end of the list. If you will pass this string into `printf()`, it will treat first item as a single string.

So this is string, or...? It's better say this is buffer containing several zero-terminated C-strings, which can be stored and processed as a whole.

Another example is `strtok()` function. It takes a string and write zero bytes in the middle of it. It thus transforms input string into some kind of buffer, which has several zero-terminated C-strings.

## 1.18 Replacing arithmetic instructions to other ones

In the pursuit of optimization, one instruction may be replaced by another, or even with a group of instructions. For example, `ADD` and `SUB` can replace each other: line 18 in listing [3.123](#).

For example, the `LEA` instruction is often used for simple arithmetic calculations: [1.6 on page 1011](#).

### 1.18.1 Multiplication

#### Multiplication using addition

Here is a simple example:

```
unsigned int f(unsigned int a)
{
    return a*8;
};
```

Multiplication by 8 is replaced by 3 addition instructions, which do the same. Apparently, MSVC's optimizer decided that this code can be faster.

Listing 1.187: Optimizing MSVC 2010

```
_TEXT SEGMENT
a$ = 8      ; size = 4
```

## 1.18. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

```
_f      PROC
; File c:\polygon\c\2.c
    mov     eax, DWORD PTR _a$[esp-4]
    add     eax, eax
    add     eax, eax
    add     eax, eax
    ret     0
_f      ENDP
_TEXT   ENDS
END
```

### Multiplication using shifting

Multiplication and division instructions by a numbers that's a power of 2 are often replaced by shift instructions.

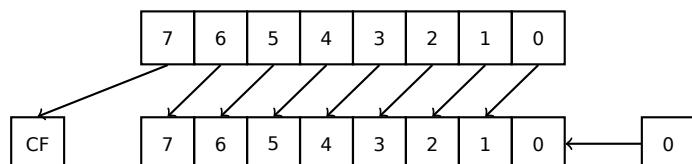
```
unsigned int f(unsigned int a)
{
    return a*4;
};
```

Listing 1.188: Non-optimizing MSVC 2010

```
_a$ = 8          ; size = 4
_f      PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    shl     eax, 2
    pop     ebp
    ret     0
_f      ENDP
```

Multiplication by 4 is just shifting the number to the left by 2 bits and inserting 2 zero bits at the right (as the last two bits). It is just like multiplying 3 by 100 —we just have to add two zeros at the right.

That's how the shift left instruction works:



The added bits at right are always zeros.

Multiplication by 4 in ARM:

Listing 1.189: Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
    LSL      r0, r0, #2
    BX       lr
    ENDP
```

Multiplication by 4 in MIPS:

Listing 1.190: Optimizing GCC 4.4.5 (IDA)

```
jr      $ra
sll     $v0, $a0, 2 ; branch delay slot
```

SLL is “Shift Left Logical”.

### Multiplication using shifting, subtracting, and adding

It's still possible to get rid of the multiplication operation when you multiply by numbers like 7 or 17 again by using shifting. The mathematics used here is relatively easy.

```
#include <stdint.h>

int f1(int a)
{
    return a*7;
}

int f2(int a)
{
    return a*28;
}

int f3(int a)
{
    return a*17;
}
```

**x86**

Listing 1.191: Optimizing MSVC 2012

```
; a*7
_a$ = 8
_f1    PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1    ENDP

; a*28
_a$ = 8
_f2    PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl    eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2    ENDP

; a*17
_a$ = 8
_f3    PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl    eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3    ENDP
```

**ARM**

Keil for ARM mode takes advantage of the second operand's shift modifiers:

## 1.18. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

Listing 1.192: Optimizing Keil 6/2013 (ARM mode)

```
; a*7
||f1|| PROC
    RSB      r0, r0, r0, LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    BX       lr
    ENDP

; a*28
||f2|| PROC
    RSB      r0, r0, r0, LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    LSL      r0, r0, #2
; R0=R0<<2=R0*4=a*7*4=a*28
    BX       lr
    ENDP

; a*17
||f3|| PROC
    ADD      r0, r0, r0, LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
    BX       lr
    ENDP
```

But there are no such modifiers in Thumb mode. It also can't optimize `f2()`:

Listing 1.193: Optimizing Keil 6/2013 (Thumb mode)

```
; a*7
||f1|| PROC
    LSLS      r1, r0, #3
; R1=R0<<3=a<<3=a*8
    SUBS      r0, r1, r0
; R0=R1-R0=a*8-a=a*7
    BX       lr
    ENDP

; a*28
||f2|| PROC
    MOVS      r1, #0x1c ; 28
; R1=28
    MULS      r0, r1, r0
; R0=R1*R0=28*a
    BX       lr
    ENDP

; a*17
||f3|| PROC
    LSLS      r1, r0, #4
; R1=R0<<4=R0*16=a*16
    ADDS      r0, r0, r1
; R0=R0+R1=a+a*16=a*17
    BX       lr
    ENDP
```

## MIPS

Listing 1.194: Optimizing GCC 4.4.5 (IDA)

```
_f1:
    sll      $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
    jr      $ra
    subu   $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
```

## 1.18. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

```
sll      $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
    sll      $a0, 2
; $a0 = $a0<<2 = $a0*4
        jr      $ra
        subu   $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
    sll      $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
    jr      $ra
    addu   $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

## 64-bit

```
#include <stdint.h>

int64_t f1(int64_t a)
{
    return a*7;
};

int64_t f2(int64_t a)
{
    return a*28;
};

int64_t f3(int64_t a)
{
    return a*17;
};
```

## x64

Listing 1.195: Optimizing MSVC 2012

```
; a*7
f1:
    lea      rax, [0+rdi*8]
; RAX=RDI*8=a*8
    sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
    ret

; a*28
f2:
    lea      rax, [0+rdi*4]
; RAX=RDI*4=a*4
    sal      rdi, 5
; RDI=RDI<<5=RDI*32=a*32
    sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
    mov      rax, rdi
    ret

; a*17
f3:
    mov      rax, rdi
    sal      rax, 4
; RAX=RAX<<4=a*16
    add      rax, rdi
; RAX=a*16+a=a*17
    ret
```

GCC 4.9 for ARM64 is also terse, thanks to the shift modifiers:

Listing 1.196: Optimizing GCC (Linaro) 4.9 ARM64

```
; a*7
f1:
    lsl      x1, x0, 3
; X1=X0<<3=X0*8=a*8
    sub     x0, x1, x0
; X0=X1-X0=a*8-a=a*7
    ret

; a*28
f2:
    lsl      x1, x0, 5
; X1=X0<<5=a*32
    sub     x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
    ret

; a*17
f3:
    add     x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
    ret
```

## Booth's multiplication algorithm

There was a time when computers were big and that expensive, that some of them lacked hardware support of multiplication operation in [CPU](#), like Data General Nova. And when one need multiplication operation, it can be provided at software level, for example, using Booth's multiplication algorithm. This is a multiplication algorithm which uses only addition operation and shifts.

What modern optimizing compilers do, isn't the same, but the goal (multiplication) and resources (faster operations) are the same.

## 1.18.2 Division

### Division using shifts

Example of division by 4:

```
unsigned int f(unsigned int a)
{
    return a/4;
};
```

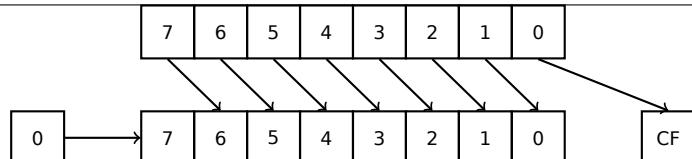
We get (MSVC 2010):

Listing 1.197: MSVC 2010

```
_a$ = 8          ; size = 4
_f      PROC
    mov     eax, DWORD PTR _a$[esp-4]
    shr     eax, 2
    ret     0
_f      ENDP
```

The **SHR** (*SHift Right*) instruction in this example is shifting a number by 2 bits to the right. The two freed bits at left (e.g., two most significant bits) are set to zero. The two least significant bits are dropped. In fact, these two dropped bits are the division operation remainder.

The **SHR** instruction works just like **SHL**, but in the other direction.



It is easy to understand if you imagine the number 23 in the decimal numeral system. 23 can be easily divided by 10 just by dropping last digit (3—division remainder). 2 is left after the operation as a **quotient**.

So the remainder is dropped, but that's OK, we work on integer values anyway, these are not **real numbers!**

Division by 4 in ARM:

Listing 1.198: Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
    LSR      r0, r0, #2
    BX       lr
ENDP
```

Division by 4 in MIPS:

Listing 1.199: Optimizing GCC 4.4.5 (IDA)

```
jr      $ra
srl     $v0, $a0, 2 ; branch delay slot
```

The SRL instruction is “Shift Right Logical”.

### 1.18.3 Exercise

- <http://challenges.re/59>

## 1.19 Floating-point unit

The **FPU** is a device within the main **CPU**, specially designed to deal with floating point numbers.

It was called “coprocessor” in the past and it stays somewhat aside of the main **CPU**.

### 1.19.1 IEEE 754

A number in the IEEE 754 format consists of a *sign*, a *significand* (also called *fraction*) and an *exponent*.

### 1.19.2 x86

It is worth looking into stack machines<sup>109</sup> or learning the basics of the Forth language<sup>110</sup>, before studying the **FPU** in x86.

It is interesting to know that in the past (before the 80486 CPU) the coprocessor was a separate chip and it was not always pre-installed on the motherboard. It was possible to buy it separately and install it<sup>111</sup>.

Starting with the 80486 DX CPU, the **FPU** is integrated in the **CPU**.

The **FWAIT** instruction reminds us of that fact—it switches the **CPU** to a waiting state, so it can wait until the **FPU** has finished with its work.

Another rudiment is the fact that the **FPU** instruction opcodes start with the so called “escape”-opcodes (**D8..DF**), i.e., opcodes passed to a separate coprocessor.

<sup>109</sup>[wikipedia.org/wiki/Stack\\_machine](https://en.wikipedia.org/wiki/Stack_machine)

<sup>110</sup>[wikipedia.org/wiki/Forth\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Forth_(programming_language))

<sup>111</sup>For example, John Carmack used fixed-point arithmetic ([wikipedia.org/wiki/Fixed-point\\_arithmetic](https://en.wikipedia.org/wiki/Fixed-point_arithmetic)) values in his Doom video game, stored in 32-bit **GPR** registers (16 bit for integral part and another 16 bit for fractional part), so Doom could work on 32-bit computers without FPU, i.e., 80386 and 80486 SX.

## 1.19. FLOATING-POINT UNIT

The FPU has a stack capable to holding 8 80-bit registers, and each register can hold a number in the IEEE 754<sup>112</sup> format.

They are `ST(0) .. ST(7)`. For brevity, IDA and OllyDbg show `ST(0)` as `ST`, which is represented in some textbooks and manuals as “Stack Top”.

## 1.19.3 ARM, MIPS, x86/x64 SIMD

In ARM and MIPS the FPU is not a stack, but a set of registers.

The same ideology is used in the SIMD extensions of x86/x64 CPUs.

## 1.19.4 C/C++

The standard C/C++ languages offer at least two floating number types, `float` (*single-precision*<sup>113</sup>, 32 bits)<sup>114</sup> and `double` (*double-precision*<sup>115</sup>, 64 bits).

In [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997)246] we can find the *single-precision* means that the floating point value can be placed into a single [32-bit] machine word, *double-precision* means it can be stored in two words (64 bits).

GCC also supports the `long double` type (*extended precision*<sup>116</sup>, 80 bit), which MSVC doesn't.

The `float` type requires the same number of bits as the `int` type in 32-bit environments, but the number representation is completely different.

## 1.19.5 Simple example

Let's consider this simple example:

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
}

int main()
{
    printf ("%f\n", f(1.2, 3.4));
}
```

## x86

### MSVC

Compile it in MSVC 2010:

Listing 1.200: MSVC 2010: `f()`

```
CONST SEGMENT
__real@4010666666666666 DQ 0401066666666666r ; 4.1
CONST ENDS
CONST SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14
CONST ENDS
_TEXT SEGMENT
```

<sup>112</sup>[wikipedia.org/wiki/IEEE\\_floating\\_point](https://en.wikipedia.org/wiki/IEEE_floating_point)

<sup>113</sup>[wikipedia.org/wiki/Single-precision\\_floating-point\\_format](https://en.wikipedia.org/wiki/Single-precision_floating-point_format)

<sup>114</sup>the single precision floating point number format is also addressed in the *Handling float data type as a structure* ( [1.24.6 on page 374](#)) section

<sup>115</sup>[wikipedia.org/wiki/Double-precision\\_floating-point\\_format](https://en.wikipedia.org/wiki/Double-precision_floating-point_format)

<sup>116</sup>[wikipedia.org/wiki/Extended\\_precision](https://en.wikipedia.org/wiki/Extended_precision)

## 1.19. FLOATING-POINT UNIT

```
_a$ = 8           ; size = 8
_b$ = 16          ; size = 8
_f PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _a$[ebp]

; current stack state: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; current stack state: ST(0) = result of _a divided by 3.14

    fld     QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b;
; ST(1) = result of _a divided by 3.14

    fmul   QWORD PTR __real@4010666666666666

; current stack state:
; ST(0) = result of _b * 4.1;
; ST(1) = result of _a divided by 3.14

    faddp  ST(1), ST(0)

; current stack state: ST(0) = result of addition

    pop    ebp
    ret    0
_f ENDP
```

**FLD** takes 8 bytes from stack and loads the number into the **ST(0)** register, automatically converting it into the internal 80-bit format (*extended precision*).

**FDIV** divides the value in **ST(0)** by the number stored at address

**\_\_real@40091eb851eb851f** —the value 3.14 is encoded there. The assembly syntax doesn't support floating point numbers, so what we see here is the hexadecimal representation of 3.14 in 64-bit IEEE 754 format.

After the execution of **FDIV ST(0)** holds the **quotient**.

By the way, there is also the **FDIVP** instruction, which divides **ST(1)** by **ST(0)**, popping both these values from stack and then pushing the result. If you know the Forth language<sup>117</sup>, you can quickly understand that this is a stack machine<sup>118</sup>.

The subsequent **FLD** instruction pushes the value of *b* into the stack.

After that, the quotient is placed in **ST(1)**, and **ST(0)** has the value of *b*.

The next **FMUL** instruction does multiplication: *b* from **ST(0)** is multiplied by value at **\_\_real@4010666666666666** (the number 4.1 is there) and leaves the result in the **ST(0)** register.

The last **FADDP** instruction adds the two values at top of stack, storing the result in **ST(1)** and then popping the value of **ST(0)**, thereby leaving the result at the top of the stack, in **ST(0)**.

The function must return its result in the **ST(0)** register, so there are no any other instructions except the function epilogue after **FADDP**.

<sup>117</sup>[wikipedia.org/wiki/Forth\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Forth_(programming_language))

<sup>118</sup>[wikipedia.org/wiki/Stack\\_machine](https://en.wikipedia.org/wiki/Stack_machine)

## 1.19. FLOATING-POINT UNIT

**MSVC + OllyDbg**

2 pairs of 32-bit words are marked by red in the stack. Each pair is a double-number in IEEE 754 format and is passed from `main()`.

We see how the first `FLD` loads a value (1.2) from the stack and puts it into `ST(0)`:

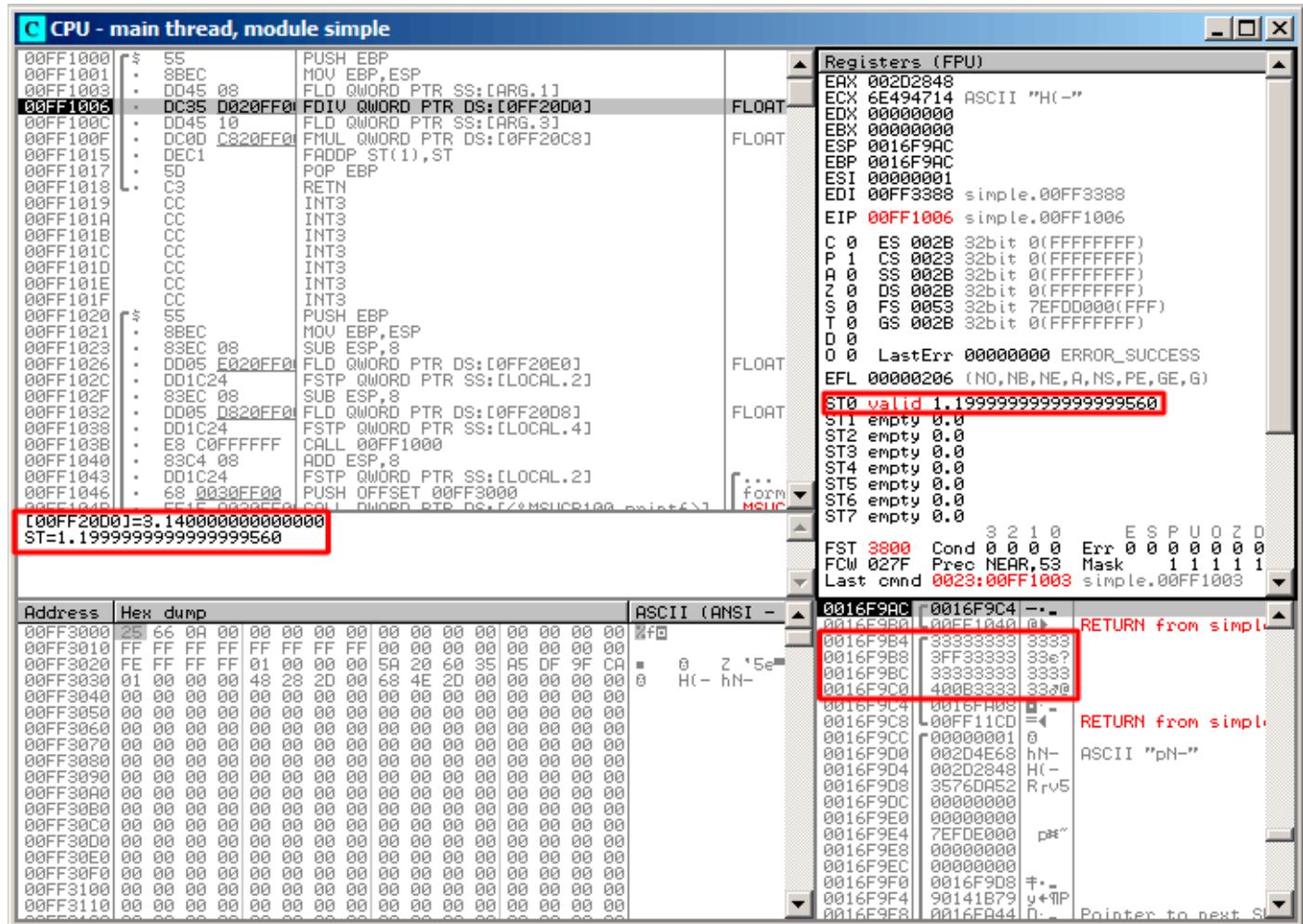


Figure 1.62: OllyDbg: the first `FLD` has been executed

Because of unavoidable conversion errors from 64-bit IEEE 754 floating point to 80-bit (used internally in the FPU), here we see 1.999..., which is close to 1.2.

`EIP` now points to the next instruction (`FDIV`), which loads a double-number (a constant) from memory. For convenience, OllyDbg shows its value: 3.14

## 1.19. FLOATING-POINT UNIT

Let's trace further. FDIV has been executed, now ST(0) contains 0.382...(quotient):

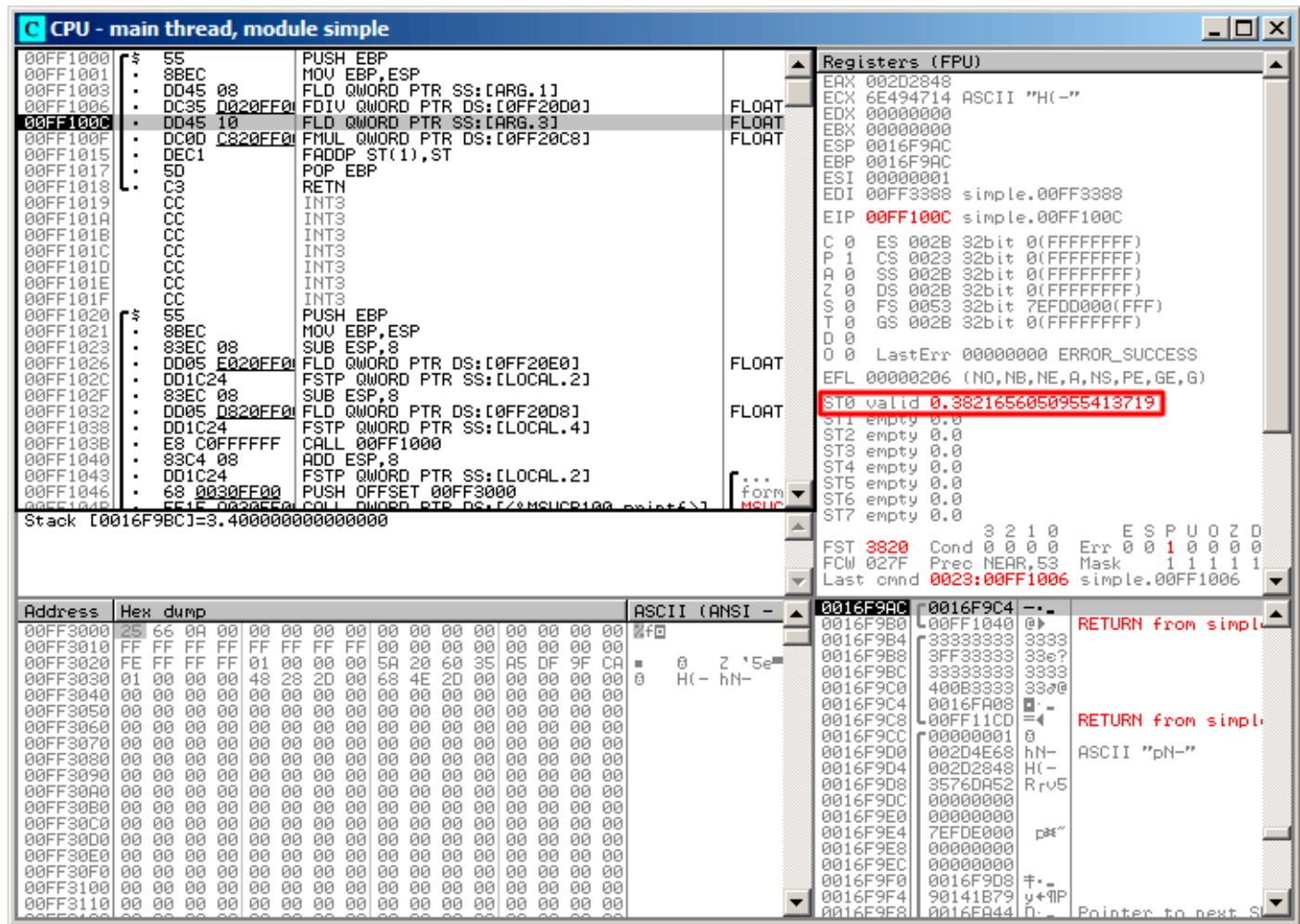


Figure 1.63: OllyDbg: FDIV has been executed

## 1.19. FLOATING-POINT UNIT

Third step: the next `FLD` has been executed, loading 3.4 into `ST(0)` (here we see the approximate value 3.39999...):

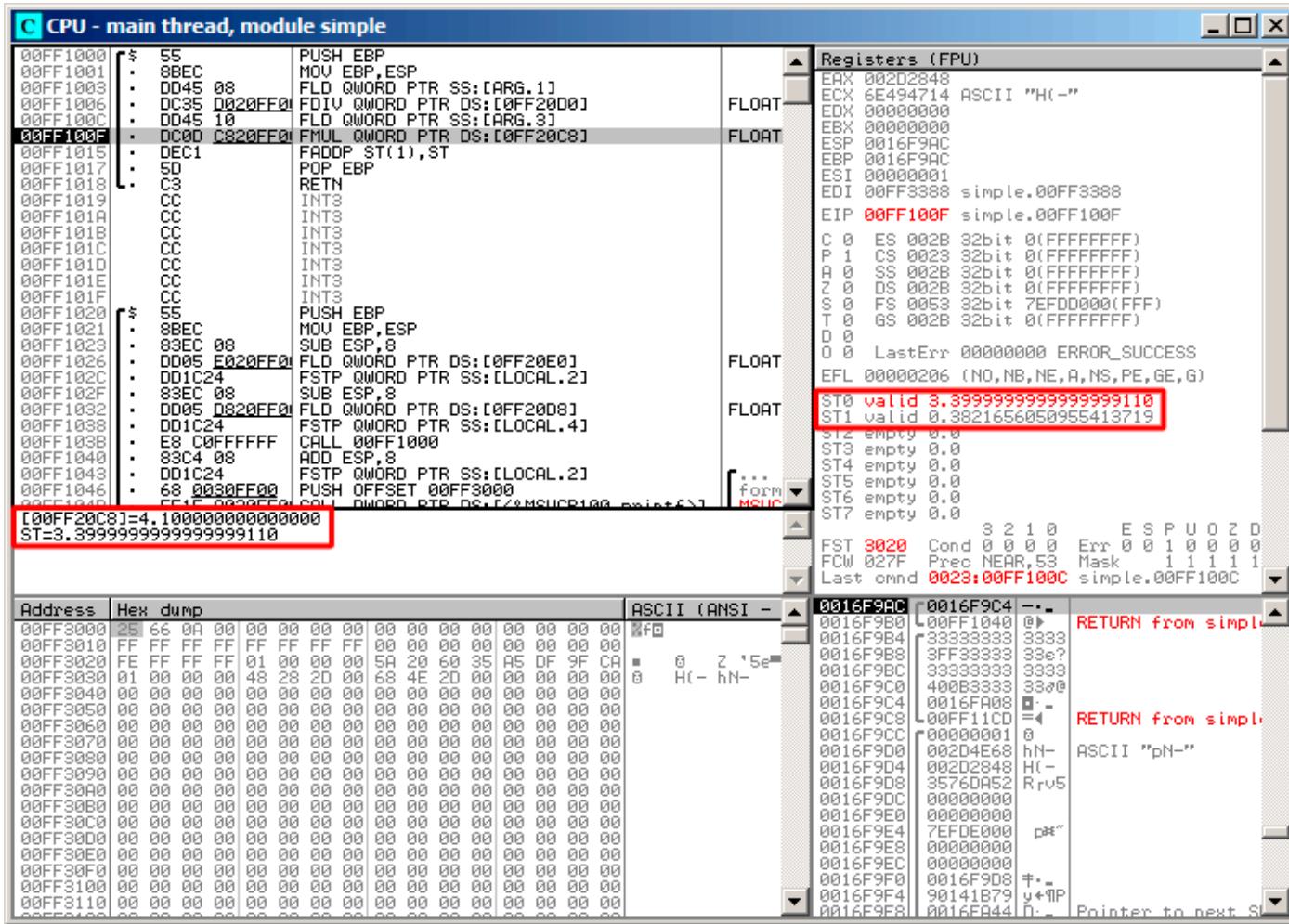


Figure 1.64: OllyDbg: the second `FLD` has been executed

At the same time, `quotient` is pushed into `ST(1)`. Right now, `EIP` points to the next instruction: `FMUL`. It loads the constant 4.1 from memory, which OllyDbg shows.

## 1.19. FLOATING-POINT UNIT

Next: FMUL has been executed, so now the product is in ST(0) :

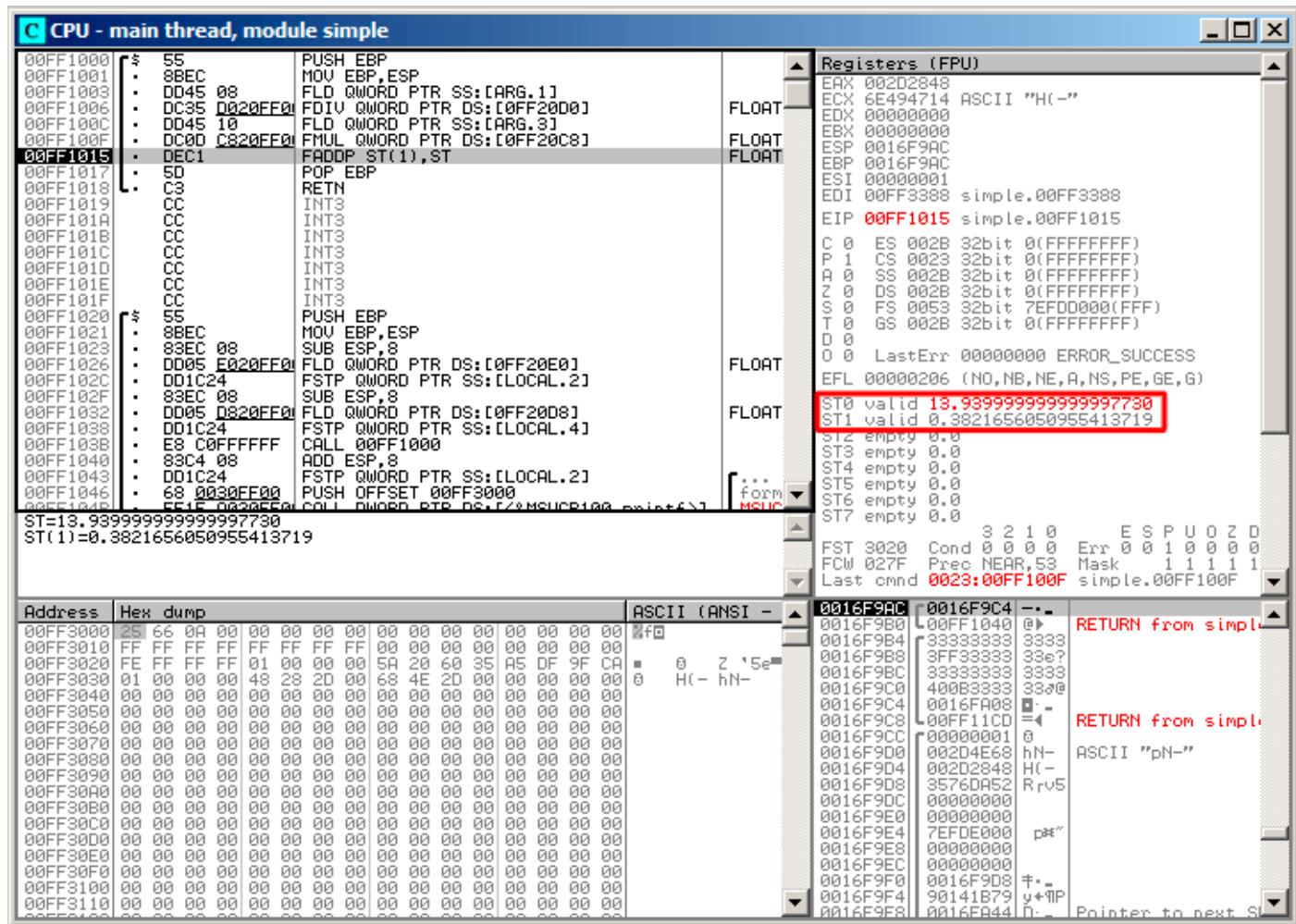


Figure 1.65: OllyDbg: the FMUL has been executed

## **1.19. FLOATING-POINT UNIT**

Next: the FADDP has been executed, now the result of the addition is in ST(0), and ST(1) is cleared:

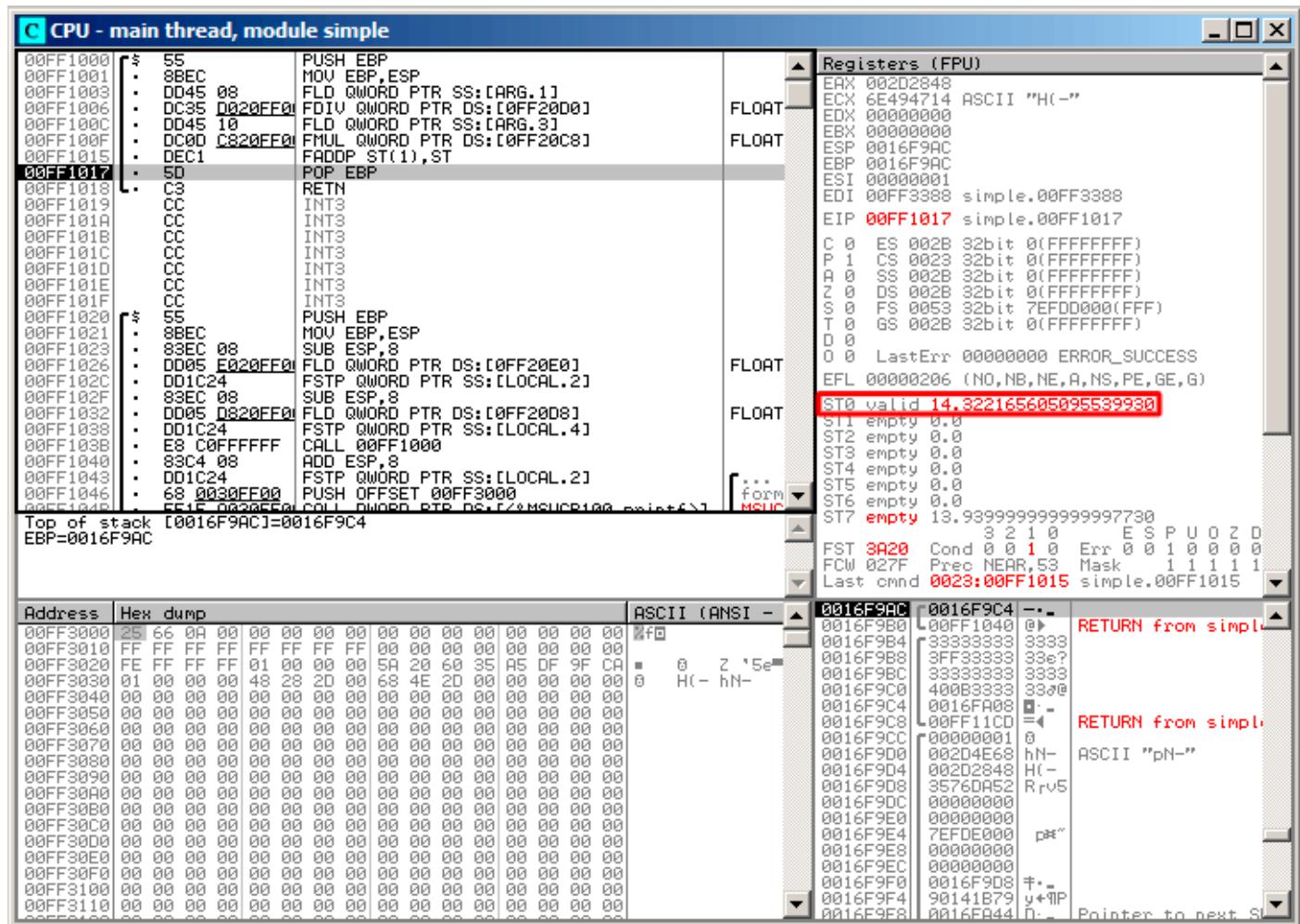


Figure 1.66: OllyDbg: FADDP has been executed

The result is left in `ST(0)`, because the function returns its value in `ST(0)`.

`main()` takes this value from the register later.

We also see something unusual: the 13.93...value is now located in ST(7) . Why?

As we have read some time before in this book, the FPU registers are a stack: [1.19.2 on page 218](#). But this is a simplification.

Imagine if it was implemented *in hardware* as it's described, then all 7 register's contents must be moved (or copied) to adjacent registers during pushing and popping, and that's a lot of work.

In reality, the FPU has just 8 registers and a pointer (called `TOP`) which contains a register number, which is the current "top of stack".

When a value is pushed to the stack, `TOP` is pointed to the next available register, and then a value is written to that register.

The procedure is reversed if a value is popped, however, the register which has been freed is not cleared (it could possibly be cleared, but this is more work which can degrade performance). So that's what we see here.

It can be said that FADDP saved the sum in the stack, and then popped one element.

But in fact, this instruction saved the sum and then shifted `TOP`.

More precisely, the registers of the FPU are a circular buffer.

GCC 4.4.1 (with `-O3` option) emits the same code, just slightly different:

Listing 1.201: Optimizing GCC 4.4.1

```

f          public f
          proc near

arg_0      = qword ptr  8
arg_8      = qword ptr  10h

          push    ebp
          fld     ds:dbl_8048608 ; 3.14

; stack state now: ST(0) = 3.14

          mov     ebp, esp
          fdivr [ebp+arg_0]

; stack state now: ST(0) = result of division

          fld     ds:dbl_8048610 ; 4.1

; stack state now: ST(0) = 4.1, ST(1) = result of division

          fmul   [ebp+arg_8]

; stack state now: ST(0) = result of multiplication, ST(1) = result of division

          pop    ebp
          faddp st(1), st

; stack state now: ST(0) = result of addition

          retn
f          endp

```

The difference is that, first of all, 3.14 is pushed to the stack (into `ST(0)`), and then the value in `arg_0` is divided by the value in `ST(0)`.

`FDIVR` stands for *Reverse Divide* —to divide with divisor and dividend swapped with each other. There is no likewise instruction for multiplication since it is a commutative operation, so we just have `FMUL` without its `-R` counterpart.

`FADDP` adds the two values but also pops one value from the stack. After that operation, `ST(0)` holds the sum.

### ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Until ARM got standardized floating point support, several processor manufacturers added their own instructions extensions. Then, VFP (*Vector Floating Point*) was standardized.

One important difference from x86 is that in ARM, there is no stack, you work just with registers.

Listing 1.202: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```

f
          VLDR      D16, =3.14
          VMOV      D17, R0, R1 ; load "a"
          VMOV      D18, R2, R3 ; load "b"
          VDIV.F64  D16, D17, D16 ; a/3.14
          VLDR      D17, =4.1
          VMUL.F64  D17, D18, D17 ; b*4.1
          VADD.F64  D16, D17, D16 ; +
          VMOV      R0, R1, D16
          BX        LR

```

## 1.19. FLOATING-POINT UNIT

dbl_2C98	DCFD 3.14	; DATA XREF: f
dbl_2CA0	DCFD 4.1	; DATA XREF: f+10

So, we see here new some registers used, with D prefix.

These are 64-bit registers, there are 32 of them, and they can be used both for floating-point numbers (double) but also for SIMD (it is called NEON here in ARM).

There are also 32 32-bit S-registers, intended to be used for single precision floating pointer numbers (float).

It is easy to memorize: D-registers are for double precision numbers, while S-registers—for single precision numbers. More about it: [2.3 on page 1023](#).

Both constants (3.14 and 4.1) are stored in memory in IEEE 754 format.

**VLDR** and **VMOV**, as it can be easily deduced, are analogous to the **LDR** and **MOV** instructions, but they work with D-registers.

It has to be noted that these instructions, just like the D-registers, are intended not only for floating point numbers, but can be also used for SIMD (NEON) operations and this will also be shown soon.

The arguments are passed to the function in a common way, via the R-registers, however each number that has double precision has a size of 64 bits, so two R-registers are needed to pass each one.

**VMOV D17, R0, R1** at the start, composes two 32-bit values from **R0** and **R1** into one 64-bit value and saves it to **D17**.

**VMOV R0, R1, D16** is the inverse operation: what has been in **D16** is split in two registers, **R0** and **R1**, because a double-precision number that needs 64 bits for storage, is returned in **R0** and **R1**.

**VDIV**, **VMUL** and **VADD**, are instruction for processing floating point numbers that compute **quotient**, **product** and sum, respectively.

The code for Thumb-2 is same.

## ARM: Optimizing Keil 6/2013 (Thumb mode)

```
f
    PUSH {R3-R7,LR}
    MOVS R7, R2
    MOVS R4, R3
    MOVS R5, R0
    MOVS R6, R1
    LDR R2, =0x66666666 ; 4.1
    LDR R3, =0x40106666
    MOVS R0, R7
    MOVS R1, R4
    BL __aeabi_dmul
    MOVS R7, R0
    MOVS R4, R1
    LDR R2, =0x51EB851F ; 3.14
    LDR R3, =0x40091EB8
    MOVS R0, R5
    MOVS R1, R6
    BL __aeabi_ddiv
    MOVS R2, R7
    MOVS R3, R4
    BL __aeabi_dadd
    POP {R3-R7,PC}

; 4.1 in IEEE 754 form:
dword_364 DCD 0x66666666 ; DATA XREF: f+A
dword_368 DCD 0x40106666 ; DATA XREF: f+C
; 3.14 in IEEE 754 form:
dword_36C DCD 0x51EB851F ; DATA XREF: f+1A
dword_370 DCD 0x40091EB8 ; DATA XREF: f+1C
```

Keil generated code for a processor without FPU or NEON support.

## 1.19. FLOATING-POINT UNIT

The double-precision floating-point numbers are passed via generic R-registers, and instead of FPU-instructions, service library functions are called

(like `__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd`) which emulate multiplication, division and addition for floating-point numbers.

Of course, that is slower than FPU-coprocessor, but it's still better than nothing.

By the way, similar FPU-emulating libraries were very popular in the x86 world when coprocessors were rare and expensive, and were installed only on expensive computers.

The FPU-coprocessor emulation is called *soft float* or *armel (emulation)* in the ARM world, while using the coprocessor's FPU-instructions is called *hard float* or *armhf*.

## ARM64: Optimizing GCC (Linaro) 4.9

Very compact code:

Listing 1.203: Optimizing GCC (Linaro) 4.9

```
f:  
; D0 = a, D1 = b  
    ldr      d2, .LC25      ; 3.14  
; D2 = 3.14  
    fdiv    d0, d0, d2  
; D0 = D0/D2 = a/3.14  
    ldr      d2, .LC26      ; 4.1  
; D2 = 4.1  
    fmadd   d0, d1, d2, d0  
; D0 = D1*D2+D0 = b*4.1+a/3.14  
    ret  
  
; constants in IEEE 754 format:  
.LC25:  
    .word   1374389535      ; 3.14  
    .word   1074339512  
.LC26:  
    .word   1717986918      ; 4.1  
    .word   1074816614
```

## ARM64: Non-optimizing GCC (Linaro) 4.9

Listing 1.204: Non-optimizing GCC (Linaro) 4.9

```
f:  
    sub    sp, sp, #16  
    str    d0, [sp,8]      ; save "a" in Register Save Area  
    str    d1, [sp]        ; save "b" in Register Save Area  
    ldr    x1, [sp,8]  
; X1 = a  
    ldr    x0, .LC25  
; X0 = 3.14  
    fmov   d0, x1  
    fmov   d1, x0  
; D0 = a, D1 = 3.14  
    fdiv   d0, d0, d1  
; D0 = D0/D1 = a/3.14  
  
    fmov   x1, d0  
; X1 = a/3.14  
    ldr    x2, [sp]  
; X2 = b  
    ldr    x0, .LC26  
; X0 = 4.1  
    fmov   d0, x2  
; D0 = b  
    fmov   d1, x0  
; D1 = 4.1
```

## 1.19. FLOATING-POINT UNIT

```
fmul    d0, d0, d1
; D0 = D0*D1 = b*4.1

        fmov    x0, d0
; X0 = D0 = b*4.1
        fmov    d0, x1
; D0 = a/3.14
        fmov    d1, x0
; D1 = X0 = b*4.1
        fadd    d0, d0, d1
; D0 = D0+D1 = a/3.14 + b*4.1

        fmov    x0, d0 ; \ redundant code
        fmov    d0, x0 ; /
        add    sp, sp, 16
        ret

.LC25:
.word   1374389535      ; 3.14
.word   1074339512

.LC26:
.word   1717986918      ; 4.1
.word   1074816614
```

Non-optimizing GCC is more verbose.

There is a lot of unnecessary value shuffling, including some clearly redundant code (the last two `FMOV` instructions). Probably, GCC 4.9 is not yet good in generating ARM64 code.

What is worth noting is that ARM64 has 64-bit registers, and the D-registers are 64-bit ones as well.

So the compiler is free to save values of type `double` in `GPRs` instead of the local stack. This isn't possible on 32-bit CPUs.

And again, as an exercise, you can try to optimize this function manually, without introducing new instructions like `FMADD`.

## MIPS

MIPS can support several coprocessors (up to 4), the zeroth of which is a special control coprocessor, and first coprocessor is the FPU.

As in ARM, the MIPS coprocessor is not a stack machine, it has 32 32-bit registers (\$F0-\$F31): [.3.1 on page 1025](#).

When one needs to work with 64-bit `double` values, a pair of 32-bit F-registers is used.

Listing 1.205: Optimizing GCC 4.4.5 (IDA)

```
f:
; $f12-$f13=A
; $f14-$f15=B
        lui    $v0, (dword_C4 >> 16) ; ?
; load low 32-bit part of 3.14 constant to $f0:
        lwcl  $f0, dword_BC
        or    $at, $zero           ; load delay slot, NOP
; load high 32-bit part of 3.14 constant to $f1:
        lwcl  $f1, $LC0
        lui    $v0, ($LC1 >> 16) ; ?
; A in $f12-$f13, 3.14 constant in $f0-$f1, do division:
        div.d $f0, $f12, $f0
; $f0-$f1=A/3.14
; load low 32-bit part of 4.1 to $f2:
        lwcl  $f2, dword_C4
        or    $at, $zero           ; load delay slot, NOP
; load high 32-bit part of 4.1 to $f3:
        lwcl  $f3, $LC1
        or    $at, $zero           ; load delay slot, NOP
; B in $f14-$f15, 4.1 constant in $f2-$f3, do multiplication:
        mul.d $f2, $f14, $f2
; $f2-$f3=B*4.1
```

## 1.19. FLOATING-POINT UNIT

```
jr      $ra
; sum 64-bit parts and leave result in $f0-$f1:
add.d  $f0, $f2          ; branch delay slot, NOP

.rodata.cst8:000000B8 $LC0:    .word 0x40091EB8      # DATA XREF: f+C
.rodata.cst8:000000BC dword_BC: .word 0x51EB851F      # DATA XREF: f+4
.rodata.cst8:000000C0 $LC1:    .word 0x40106666      # DATA XREF: f+10
.rodata.cst8:000000C4 dword_C4: .word 0x66666666      # DATA XREF: f
```

The new instructions here are:

- `LWC1` loads a 32-bit word into a register of the first coprocessor (hence “1” in instruction name). A pair of `LWC1` instructions may be combined into a `L.D` pseudo instruction.
- `DIV.D`, `MUL.D`, `ADD.D` do division, multiplication, and addition respectively (“.D” in the suffix stands for double precision, “.S” stands for single precision)

There is also a weird compiler anomaly: the `LUI` instructions that we’ve marked with a question mark. It’s hard for me to understand why load a part of a 64-bit constant of *double* type into the `$V0` register. These instructions has no effect. If someone knows more about it, please drop an email to author<sup>119</sup>.

### 1.19.6 Passing floating point numbers via arguments

```
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}
```

## x86

Let’s see what we get in (MSVC 2010):

Listing 1.206: MSVC 2010

```
CONST SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST ENDS

_main PROC
    push ebp
    mov ebp, esp
    sub esp, 8 ; allocate space for the first variable
    fld QWORD PTR __real@3ff8a3d70a3d70a4
    fstp QWORD PTR [esp]
    sub esp, 8 ; allocate space for the second variable
    fld QWORD PTR __real@40400147ae147ae1
    fstp QWORD PTR [esp]
    call _pow
    add esp, 8 ; return back place of one variable.

; in local stack here 8 bytes still reserved for us.
; result now in ST(0)

    fstp QWORD PTR [esp] ; move result from ST(0) to local stack for printf()
    push OFFSET $SG2651
    call _printf
    add esp, 12
```

<sup>119</sup>dennis(a)yurichev.com

## 1.19. FLOATING-POINT UNIT

```
xor    eax, eax
pop    ebp
ret    0
_main  ENDP
```

`FLD` and `FSTP` move variables between the data segment and the FPU stack. `pow()`<sup>120</sup> takes both values from the stack of the FPU and returns its result in the `ST(0)` register. `printf()` takes 8 bytes from the local stack and interprets them as *double* type variable.

By the way, a pair of `MOV` instructions could be used here for moving values from the memory into the stack, because the values in memory are stored in IEEE 754 format, and `pow()` also takes them in this format, so no conversion is necessary. That's how it's done in the next example, for ARM: [1.19.6](#).

### ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
_main
var_C    = -0xC

PUSH    {R7,LR}
MOV     R7, SP
SUB    SP, SP, #4
VLDR   D16, =32.01
VMOV   R0, R1, D16
VLDR   D16, =1.54
VMOV   R2, R3, D16
BLX    _pow
VMOV   D16, R0, R1
MOV     R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
ADD    R0, PC
VMOV   R1, R2, D16
BLX    _printf
MOVS   R1, 0
STR    R0, [SP,#0xC+var_C]
MOV     R0, R1
ADD    SP, SP, #4
POP    {R7,PC}

dbl_2F90 DCFD 32.01      ; DATA XREF: _main+6
dbl_2F98 DCFD 1.54       ; DATA XREF: _main+E
```

As it was mentioned before, 64-bit floating pointer numbers are passed in R-registers pairs.

This code is a bit redundant (certainly because optimization is turned off), since it is possible to load values into the R-registers directly without touching the D-registers.

So, as we see, the `_pow` function receives its first argument in `R0` and `R1`, and its second one in `R2` and `R3`. The function leaves its result in `R0` and `R1`. The result of `_pow` is moved into `D16`, then in the `R1` and `R2` pair, from where `printf()` takes the resulting number.

### ARM + Non-optimizing Keil 6/2013 (ARM mode)

```
_main
STMFD  SP!, {R4-R6,LR}
LDR    R2, =0xA3D70A4 ; y
LDR    R3, =0x3FF8A3D7
LDR    R0, =0xAE147AE1 ; x
LDR    R1, =0x40400147
BL    pow
MOV    R4, R0
MOV    R2, R4
MOV    R3, R1
```

<sup>120</sup>a standard C function, raises a number to the given power (exponentiation)

### 1.19. FLOATING-POINT UNIT

```

ADR      R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
BL       __2printf
MOV      R0, #0
LDMFD   SP!, {R4-R6,PC}

y          DCD 0xA3D70A4           ; DATA XREF: _main+4
dword_520  DCD 0x3FF8A3D7        ; DATA XREF: _main+8
x          DCD 0xAE147AE1         ; DATA XREF: _main+C
dword_528  DCD 0x40400147        ; DATA XREF: _main+10
a32_011_54Lf DCB "32.01 ^ 1.54 = %lf",0xA,0
                                         ; DATA XREF: _main+24

```

D-registers are not used here, just R-register pairs.

### ARM64 + Optimizing GCC (Linaro) 4.9

Listing 1.207: Optimizing GCC (Linaro) 4.9

```

f:
    stp    x29, x30, [sp, -16]!
    add    x29, sp, 0
    ldr    d1, .LC1 ; load 1.54 into D1
    ldr    d0, .LC0 ; load 32.01 into D0
    bl     pow
; result of pow() in D0
    adrp   x0, .LC2
    add    x0, x0, :lo12:.LC2
    bl     printf
    mov    w0, 0
    ldp    x29, x30, [sp], 16
    ret
.LC0:
; 32.01 in IEEE 754 format
    .word   -1374389535
    .word   1077936455
.LC1:
; 1.54 in IEEE 754 format
    .word   171798692
    .word   1073259479
.LC2:
    .string "32.01 ^ 1.54 = %lf\n"

```

The constants are loaded into `D0` and `D1`: `pow()` takes them from there. The result will be in `D0` after the execution of `pow()`. It is to be passed to `printf()` without any modification and moving, because `printf()` takes arguments of [integral types](#) and pointers from X-registers, and floating point arguments from D-registers.

### MIPS

Listing 1.208: Optimizing GCC 4.4.5 (IDA)

```

main:
var_10      = -0x10
var_4       = -4

; function prologue:
    lui     $gp, (dword_9C >> 16)
    addiu  $sp, -0x20
    la     $gp, (__gnu_local_gp & 0xFFFF)
    sw     $ra, 0x20+var_4($sp)
    sw     $gp, 0x20+var_10($sp)
    lui     $v0, (dword_A4 >> 16) ; ?
; load low 32-bit part of 32.01:
    lwcl   $f12, dword_9C

```

## 1.19. FLOATING-POINT UNIT

```
; load address of pow() function:
    lw      $t9, (pow & 0xFFFF)($gp)
; load high 32-bit part of 32.01:
    lwcl   $f13, $LC0
    lui    $v0, ($LC1 >> 16) ; ?
; load low 32-bit part of 1.54:
    lwcl   $f14, dword_A4
    or     $at, $zero ; load delay slot, NOP
; load high 32-bit part of 1.54:
    lwcl   $f15, $LC1
; call pow():
    jalr   $t9
    or     $at, $zero ; branch delay slot, NOP
    lw     $gp, 0x20+var_10($sp)
; copy result from $f0 and $f1 to $a3 and $a2:
    mfcl   $a3, $f0
    lw     $t9, (printf & 0xFFFF)($gp)
    mfcl   $a2, $f1
; call printf():
    lui    $a0, ($LC2 >> 16) # "32.01 ^ 1.54 = %lf\n"
    jalr   $t9
    la     $a0, ($LC2 & 0xFFFF) # "32.01 ^ 1.54 = %lf\n"
; function epilogue:
    lw     $ra, 0x20+var_4($sp)
; return 0:
    move   $v0, $zero
    jr     $ra
    addiu $sp, 0x20

.rodata.str1.4:00000084 $LC2:          .ascii "32.01 ^ 1.54 = %lf\n<0>

; 32.01:
.rodata.cst8:00000098 $LC0:          .word 0x40400147      # DATA XREF: main+20
.rodata.cst8:0000009C dword_9C:       .word 0xAE147AE1      # DATA XREF: main
.rodata.cst8:0000009C:                 .word 0x00000000      # main+18
; 1.54:
.rodata.cst8:000000A0 $LC1:          .word 0x3FF8A3D7      # DATA XREF: main+24
.rodata.cst8:000000A0:                 .word 0x00000000      # main+30
.rodata.cst8:000000A4 dword_A4:       .word 0xA3D70A4      # DATA XREF: main+14
```

And again, we see here `LUI` loading a 32-bit part of a *double* number into `$V0`. And again, it's hard to comprehend why.

The new instruction for us here is `MFC1` ("Move From Coprocessor 1"). The FPU is coprocessor number 1, hence "1" in the instruction name. This instruction transfers values from the coprocessor's registers to the registers of the CPU (`GPR`). So at the end the result of `pow()` is moved to registers `$A3` and `$A2`, and `printf()` takes a 64-bit double value from this register pair.

## 1.19.7 Comparison example

Let's try this:

```
#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
}
```

## 1.19. FLOATING-POINT UNIT

Despite the simplicity of the function, it will be harder to understand how it works.

### x86

#### Non-optimizing MSVC

MSVC 2010 generates the following:

Listing 1.209: Non-optimizing MSVC 2010

```
PUBLIC      _d_max
_TEXT       SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_d_max      PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b
; compare _b (ST(0)) and _a, and pop register

    fcomp   QWORD PTR _a$[ebp]

; stack is empty here

    fnstsw ax
    test    ah, 5
    jp      SHORT $LN1@d_max

; we are here only if a>b

    fld     QWORD PTR _a$[ebp]
    jmp    SHORT $LN2@d_max
$LN1@d_max:
    fld     QWORD PTR _b$[ebp]
$LN2@d_max:
    pop    ebp
    ret    0
_d_max      ENDP
```

So, `FLD` loads `_b` into `ST(0)`.

`FCOMP` compares the value in `ST(0)` with what is in `_a` and sets `C3 / C2 / C0` bits in FPU status word register, accordingly. This is a 16-bit register that reflects the current state of the FPU.

After the bits are set, the `FCOMP` instruction also pops one variable from the stack. This is what distinguishes it from `FCOM`, which is just compares values, leaving the stack in the same state.

Unfortunately, CPUs before Intel P6<sup>121</sup> don't have any conditional jumps instructions which check the `C3 / C2 / C0` bits. Perhaps, it is a matter of history (recall: FPU was a separate chip in past).

Modern CPU starting at Intel P6 have `FCOMI / FC0MIP / FUCOMI / FUC0MIP` instructions—which do the same, but modify the `ZF / PF / CF` CPU flags.

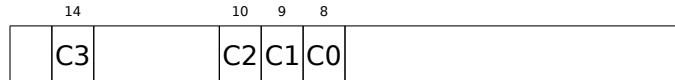
The `FNSTSW` instruction copies FPU the status word register to `AX`. `C3 / C2 / C0` bits are placed at positions 14/10/8, they are at the same positions in the `AX` register and all they are placed in the high part of `AX` — `AH`.

- If  $b > a$  in our example, then `C3 / C2 / C0` bits are to be set as following: 0, 0, 0.
- If  $a > b$ , then the bits are: 0, 0, 1.
- If  $a = b$ , then the bits are: 1, 0, 0.
- If the result is unordered (in case of error), then the set bits are: 1, 1, 1.

<sup>121</sup>Intel P6 is Pentium Pro, Pentium II, etc.

## 1.19. FLOATING-POINT UNIT

This is how C3 / C2 / C0 bits are located in the AX register:



This is how C3 / C2 / C0 bits are located in the AH register:



After the execution of `test ah, 5`<sup>122</sup>, only C0 and C2 bits (on 0 and 2 position) are considered, all other bits are just ignored.

Now let's talk about the *parity flag*, another notable historical rudiment.

This flag is set to 1 if the number of ones in the result of the last calculation is even, and to 0 if it is odd.

Let's look into Wikipedia<sup>123</sup>:

One common reason to test the parity flag actually has nothing to do with parity. The FPU has four condition flags (C0 to C3), but they cannot be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag. The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.

As noted in Wikipedia, the parity flag used sometimes in FPU code, let's see how.

The PF flag is to be set to 1 if both C0 and C2 are set to 0 or both are 1, in which case the subsequent JP (*jump if PF==1*) is triggering. If we recall the values of C3 / C2 / C0 for various cases, we can see that the conditional jump JP is triggering in two cases: if  $b > a$  or  $a = b$  (C3 bit is not considered here, since it has been cleared by the `test ah, 5` instruction).

It is all simple after that. If the conditional jump has been triggered, FLD loads the value of `_b` in ST(0), and if it hasn't been triggered, the value of `_a` is loaded there.

### And what about checking C2?

The C2 flag is set in case of error (NaN, etc.), but our code doesn't check it.

If the programmer cares about FPU errors, he/she must add additional checks.

<sup>122</sup>5=101b

<sup>123</sup>[wikipedia.org/wiki/Parity\\_flag](https://en.wikipedia.org/wiki/Parity_flag)

## 1.19. FLOATING-POINT UNIT

### First OllyDbg example: a=1.2 and b=3.4

Let's load the example into OllyDbg:

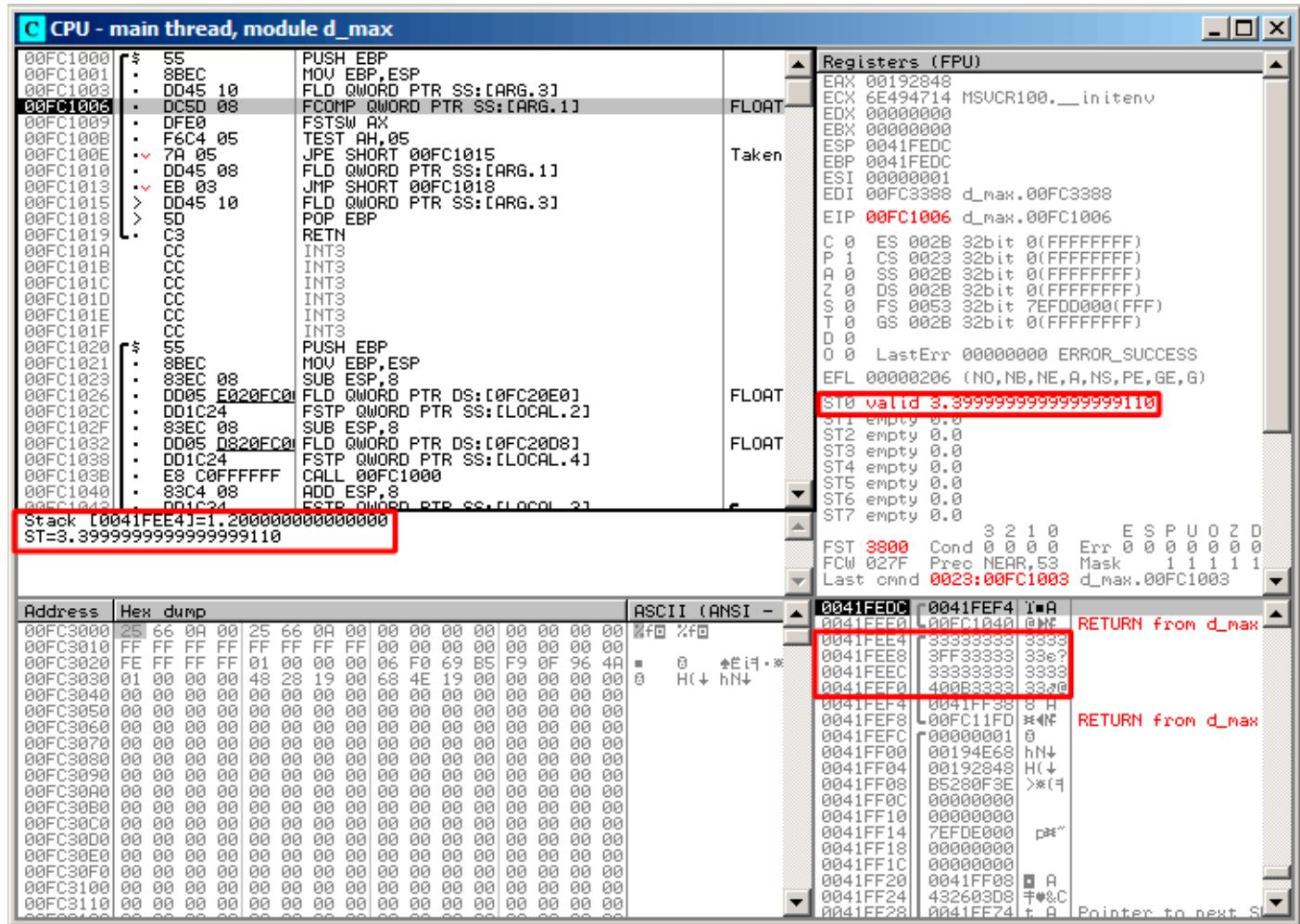


Figure 1.67: OllyDbg: first FLD has been executed

Current arguments of the function:  $a = 1.2$  and  $b = 3.4$  (We can see them in the stack: two pairs of 32-bit values).  $b$  (3.4) is already loaded in **ST(0)**. Now **FCOMP** is being executed. OllyDbg shows the second **FCOMP** argument, which is in stack right now.

## 1.19. FLOATING-POINT UNIT

FCOMP has been executed:

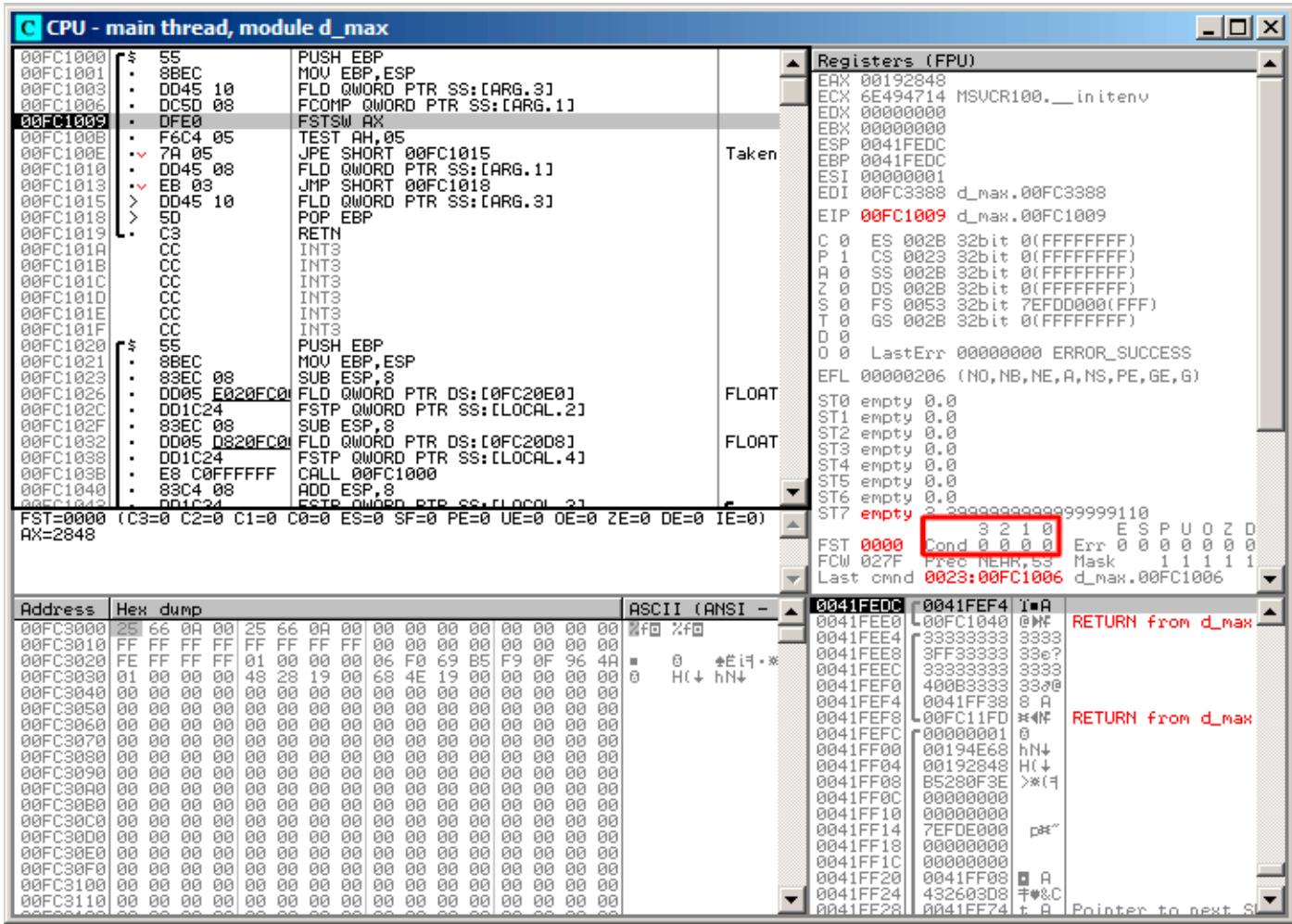


Figure 1.68: OllyDbg: FCOMP has been executed

We see the state of the FPU's condition flags: all zeros. The popped value is reflected as ST(7), it was written earlier about reason for this: [1.19.5 on page 225](#).

## 1.19. FLOATING-POINT UNIT

FNSTSW has been executed:

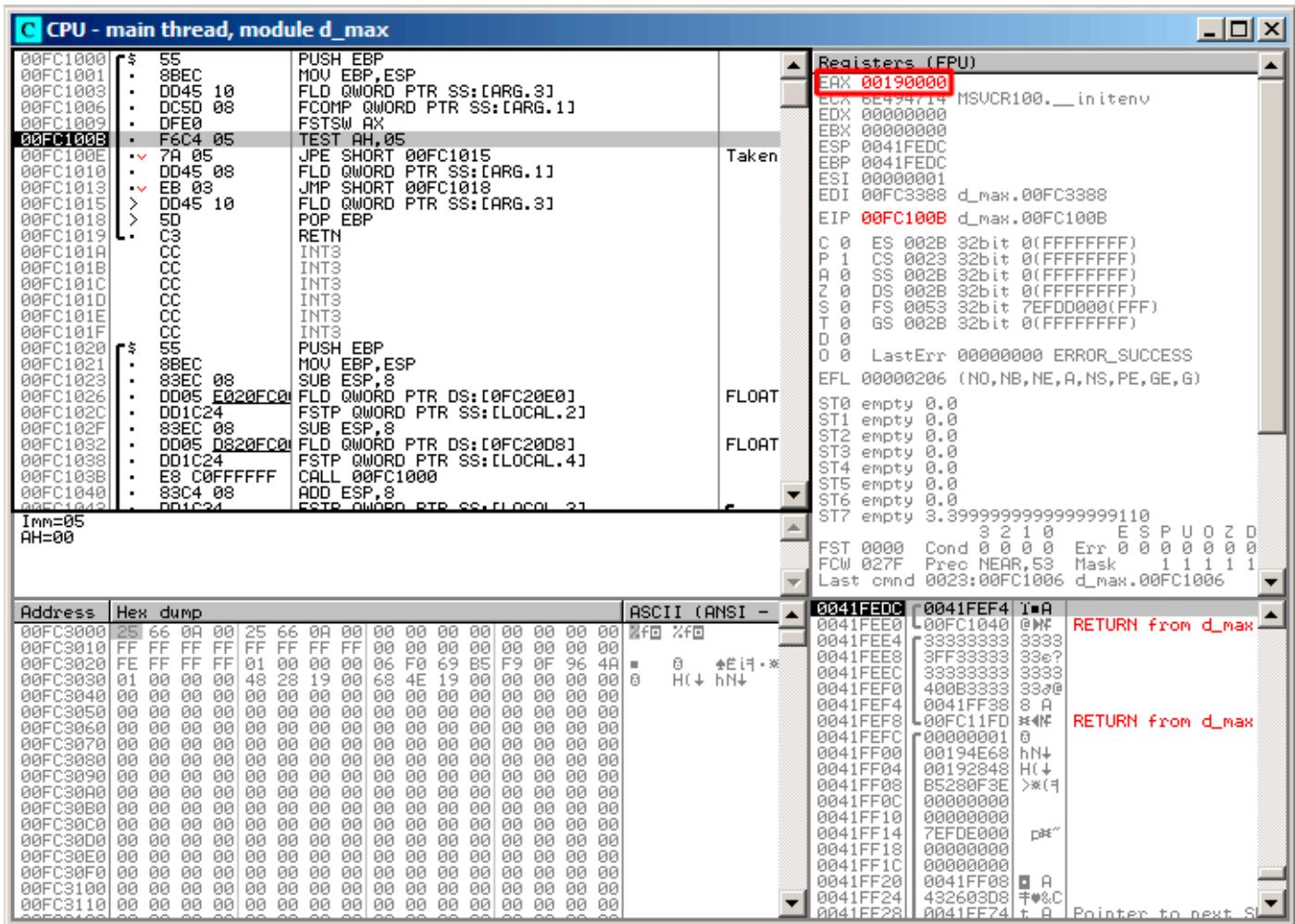


Figure 1.69: OllyDbg: FNSTSW has been executed

We see that the **AX** register contain zeros: indeed, all condition flags are zero. (OllyDbg disassembles the **FNSTSW** instruction as **FSTSW**—they are synonyms).

## 1.19. FLOATING-POINT UNIT

TEST has been executed:

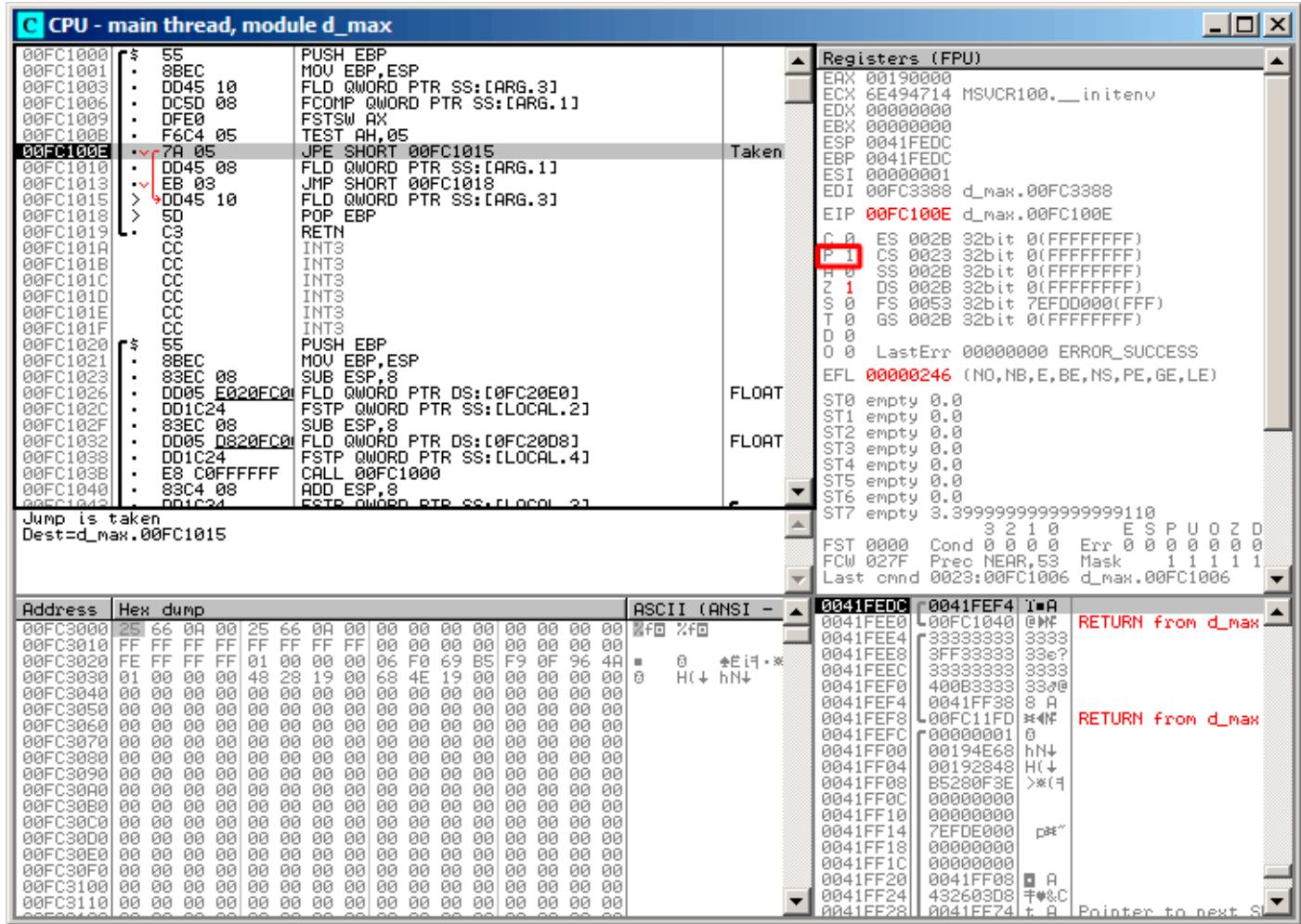


Figure 1.70: OllyDbg: TEST has been executed

The PF flag is set to 1.

Indeed: the number of bits set to 0 is 0 and 0 is an even number. OllyDbg disassembles `JP` as `JPE124`—they are synonyms. And it is about to trigger now.

## <sup>124</sup>Jump Parity Even (x86 instruction)

## 1.19. FLOATING-POINT UNIT

JPE triggered, FLD loads the value of  $b$  (3.4) in ST(0) :

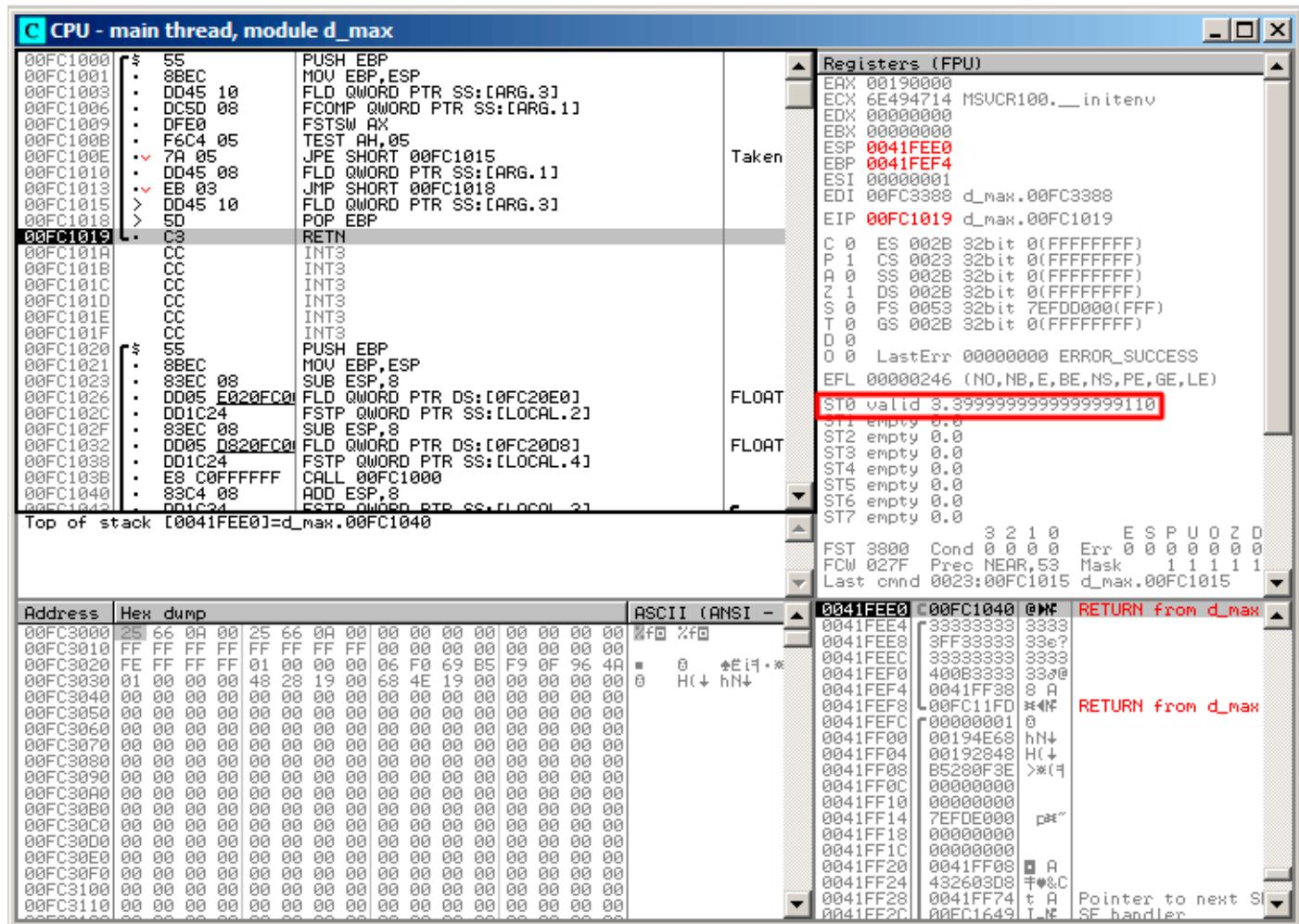


Figure 1.71: OllyDbg: the second FLD has been executed

The function finishes its work.

## 1.19. FLOATING-POINT UNIT

### Second OllyDbg example: a=5.6 and b=-4

Let's load example into OllyDbg:

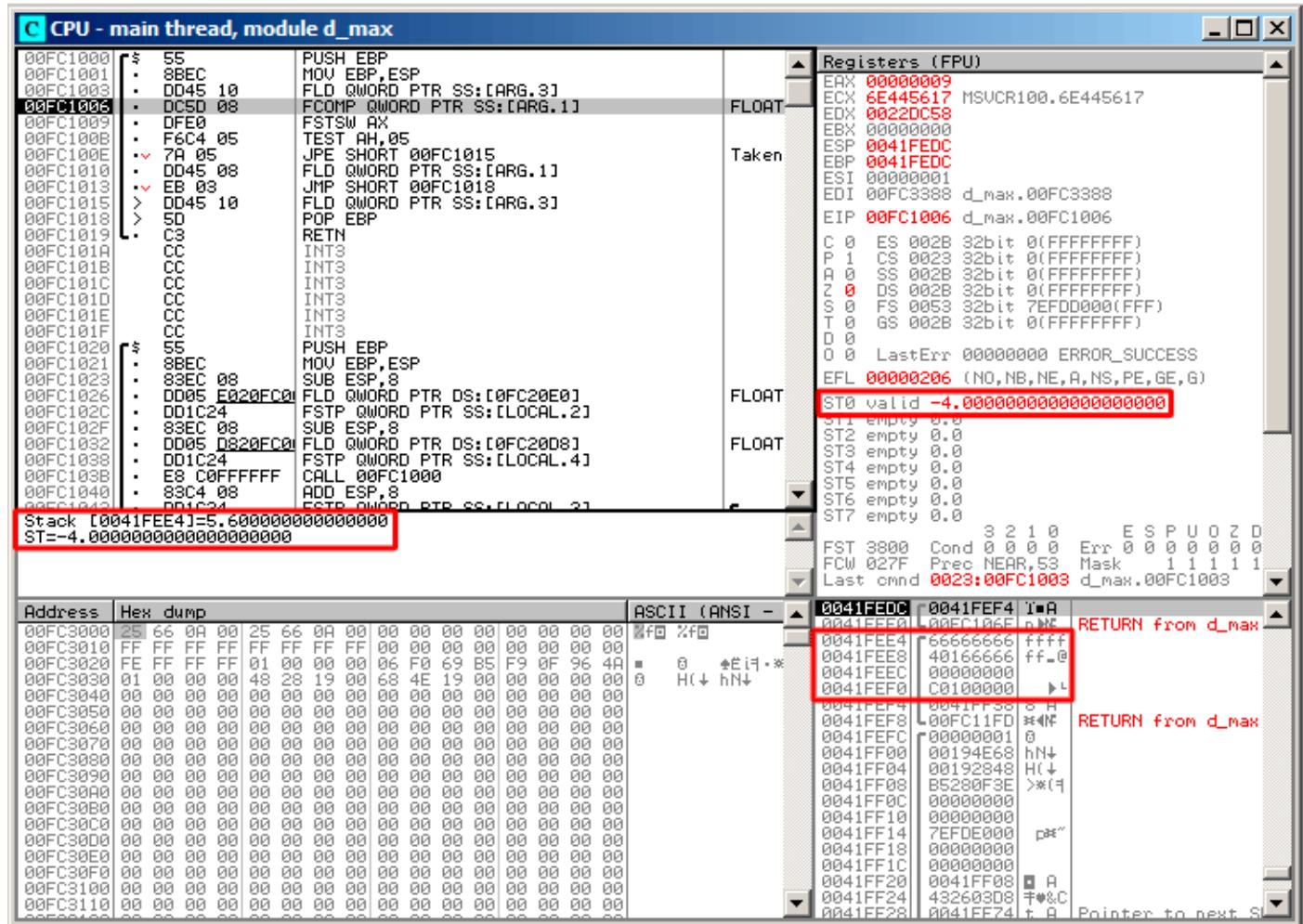


Figure 1.72: OllyDbg: first **FLD** executed

Current function arguments:  $a = 5.6$  and  $b = -4$ .  $b (-4)$  is already loaded in **ST(0)**. **FCOMP** about to execute now. OllyDbg shows the second **FCOMP** argument, which is in stack right now.

## 1.19. FLOATING-POINT UNIT

FCOMP executed:

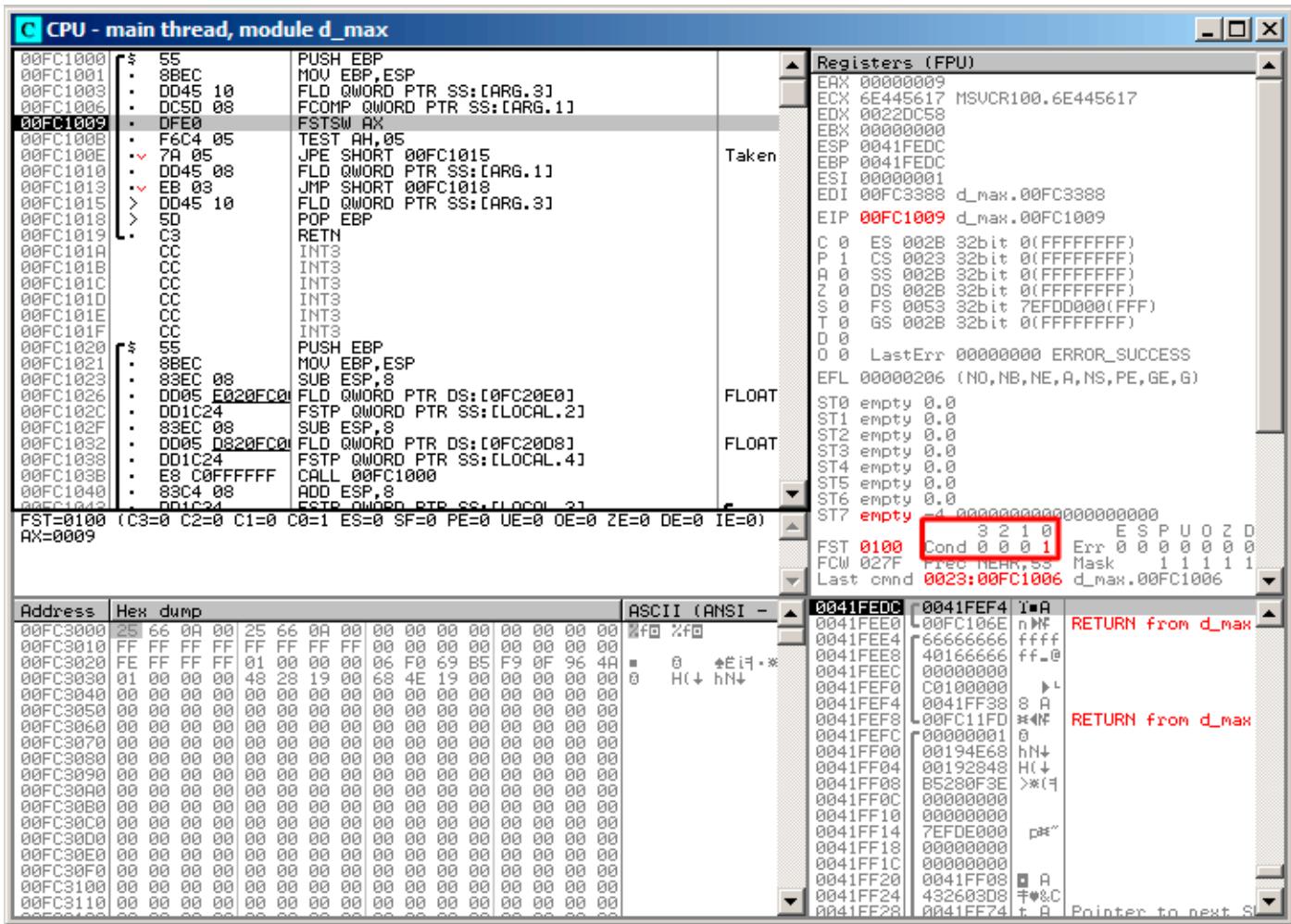


Figure 1.73: OllyDbg: FCOMP executed

We see the state of the FPU's condition flags: all zeros except C0 .

## 1.19. FLOATING-POINT UNIT

FNSTSW executed:

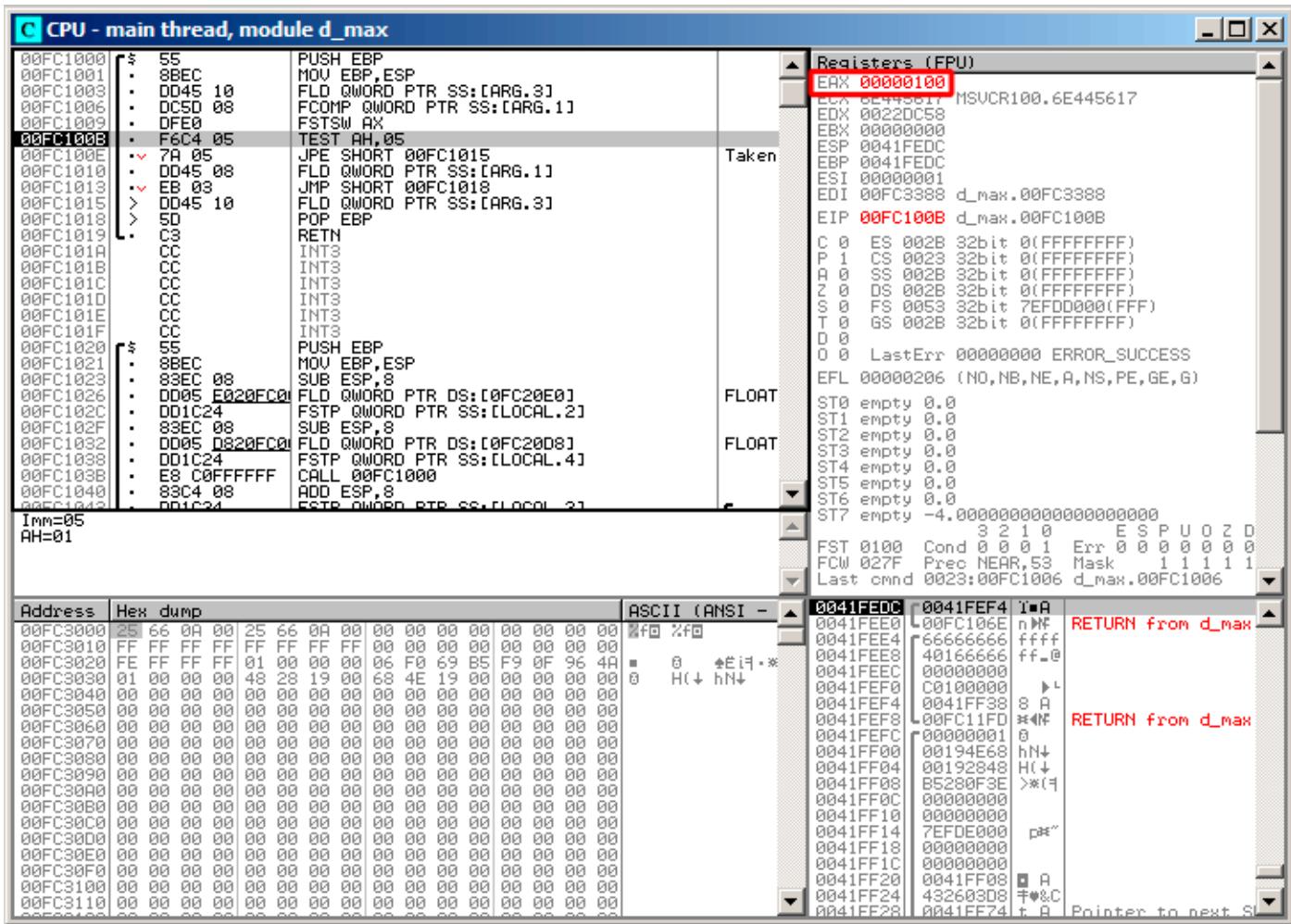


Figure 1.74: OllyDbg: FNSTSW executed

We see that the AX register contains 0x100 : the C0 flag is at the 8th bit.

## 1.19. FLOATING-POINT UNIT

TEST executed:

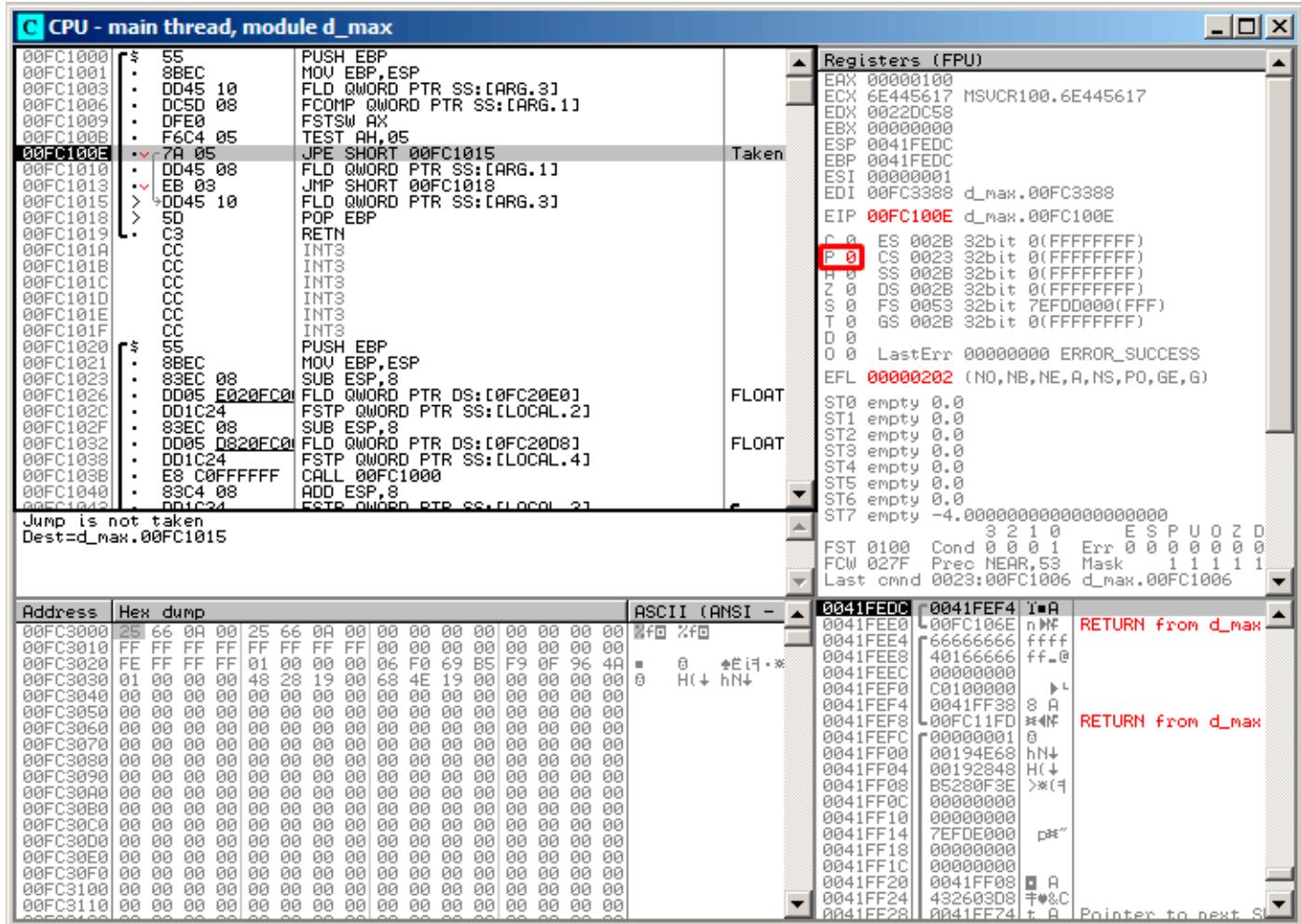


Figure 1.75: OllyDbg: TEST executed

The PF flag is cleared. Indeed:

the count of bits set in `0x100` is 1 and 1 is an odd number. [JPE](#) is being skipped now.

## 1.19. FLOATING-POINT UNIT

JPE hasn't been triggered, so FLD loads the value of *a* (5.6) in ST(0) :

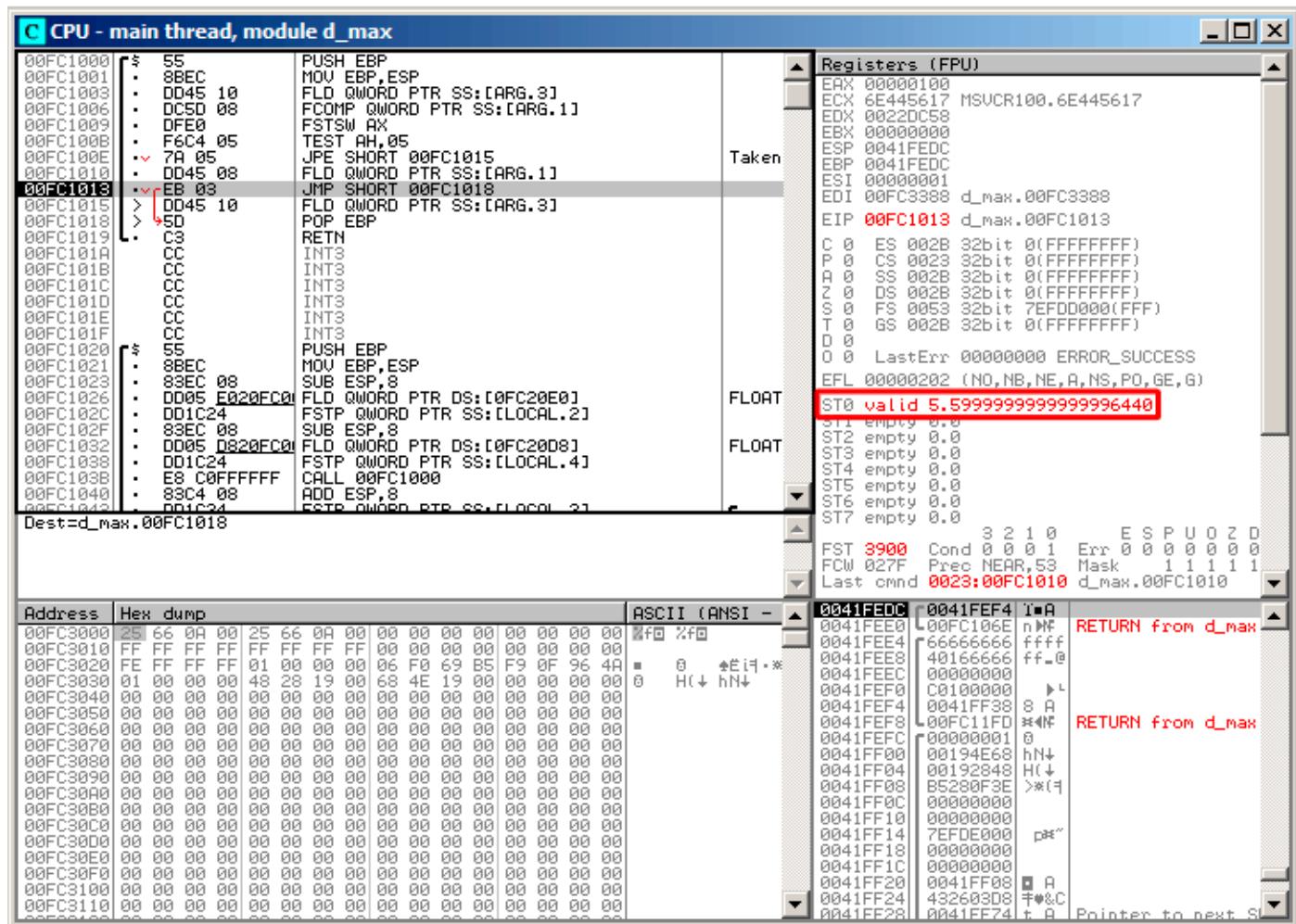


Figure 1.76: OllyDbg: second FLD executed

The function finishes its work.

## Optimizing MSVC 2010

Listing 1.210: Optimizing MSVC 2010

```

a$ = 8           ; size = 8
b$ = 16          ; size = 8
_d_max PROC
    fld  QWORD PTR _b$[esp-4]
    fld  QWORD PTR _a$[esp-4]

; current stack state: ST(0) = _a, ST(1) = _b

    fcom   ST(1) ; compare _a and ST(1) = (_b)
    fnstsw ax
    test   ah, 65 ; 00000041H
    jne    SHORT $LN5@_d_max
; copy ST(0) to ST(1) and pop register,
; leave (_a) on top
    fstp   ST(1)

; current stack state: ST(0) = _a

    ret    0
$LN5@_d_max:
; copy ST(0) to ST(0) and pop register,

```

## 1.19. FLOATING-POINT UNIT

```
; leave (_b) on top
    fstp    ST(0)

; current stack state: ST(0) = _b

    ret     0
_d_max  ENDP
```

**FCOM** differs from **FCOMP** in the sense that it just compares the values and doesn't change the FPU stack. Unlike the previous example, here the operands are in reverse order, which is why the result of the comparison in **C3 / C2 / C0** is different:

- If  $a > b$  in our example, then **C3 / C2 / C0** bits are to be set as: 0, 0, 0.
- If  $b > a$ , then the bits are: 0, 0, 1.
- If  $a = b$ , then the bits are: 1, 0, 0.

The **test ah, 65** instruction leaves just two bits — **C3** and **C0**. Both will be zero if  $a > b$ : in that case the **JNE** jump will not be triggered. Then **FSTP ST(1)** follows —this instruction copies the value from **ST(0)** to the operand and pops one value from the FPU stack. In other words, the instruction copies **ST(0)** (where the value of **\_a** is now) into **ST(1)**. After that, two copies of **\_a** are at the top of the stack. Then, one value is popped. After that, **ST(0)** contains **\_a** and the function is finished.

The conditional jump **JNE** is triggering in two cases: if  $b > a$  or  $a = b$ . **ST(0)** is copied into **ST(0)**, it is just like an idle (**NOP**) operation, then one value is popped from the stack and the top of the stack (**ST(0)**) is contain what has been in **ST(1)** before (that is **\_b**). Then the function finishes. The reason this instruction is used here probably is because the **FPU** has no other instruction to pop a value from the stack and discard it.

## 1.19. FLOATING-POINT UNIT

### First OllyDbg example: a=1.2 and b=3.4

Both **FLD** are executed:

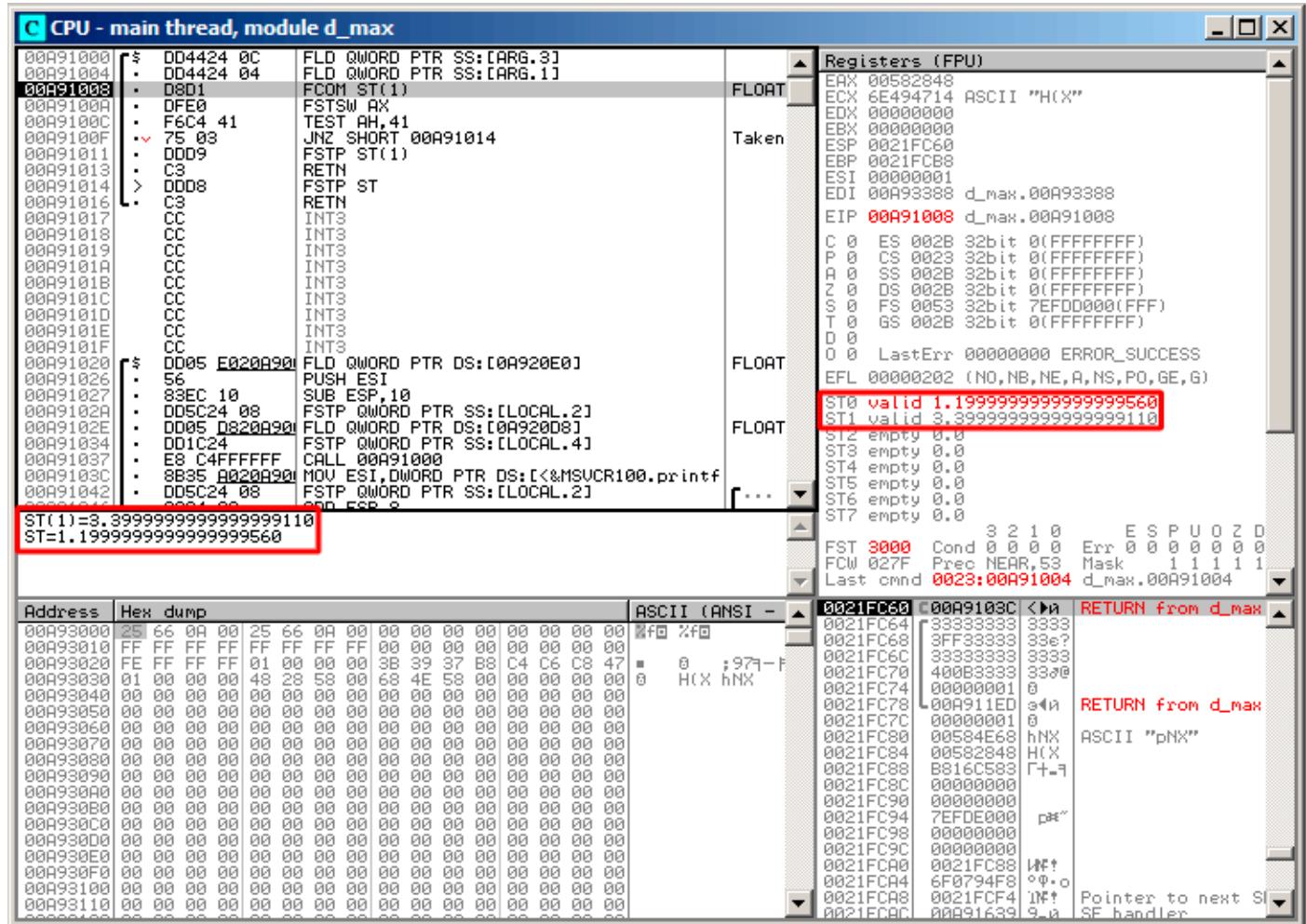


Figure 1.77: OllyDbg: both **FLD** are executed

**FCOM** being executed: OllyDbg shows the contents of **ST(0)** and **ST(1)** for convenience.

## 1.19. FLOATING-POINT UNIT

FCOM has been executed:

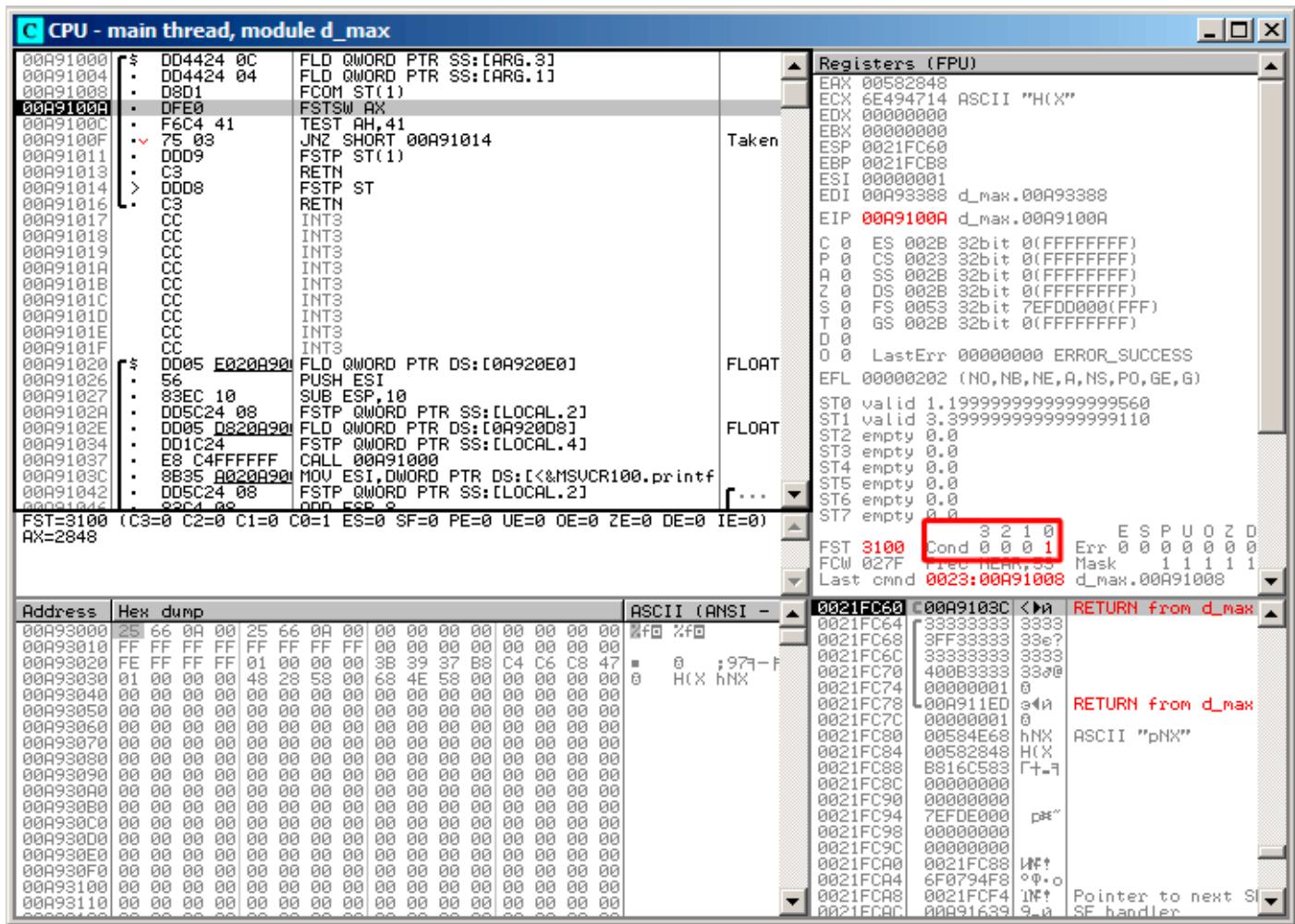


Figure 1.78: OllyDbg: FCOM has been executed

C0 is set, all other condition flags are cleared.

## 1.19. FLOATING-POINT UNIT

FNSTSW has been executed, AX = 0x3100:

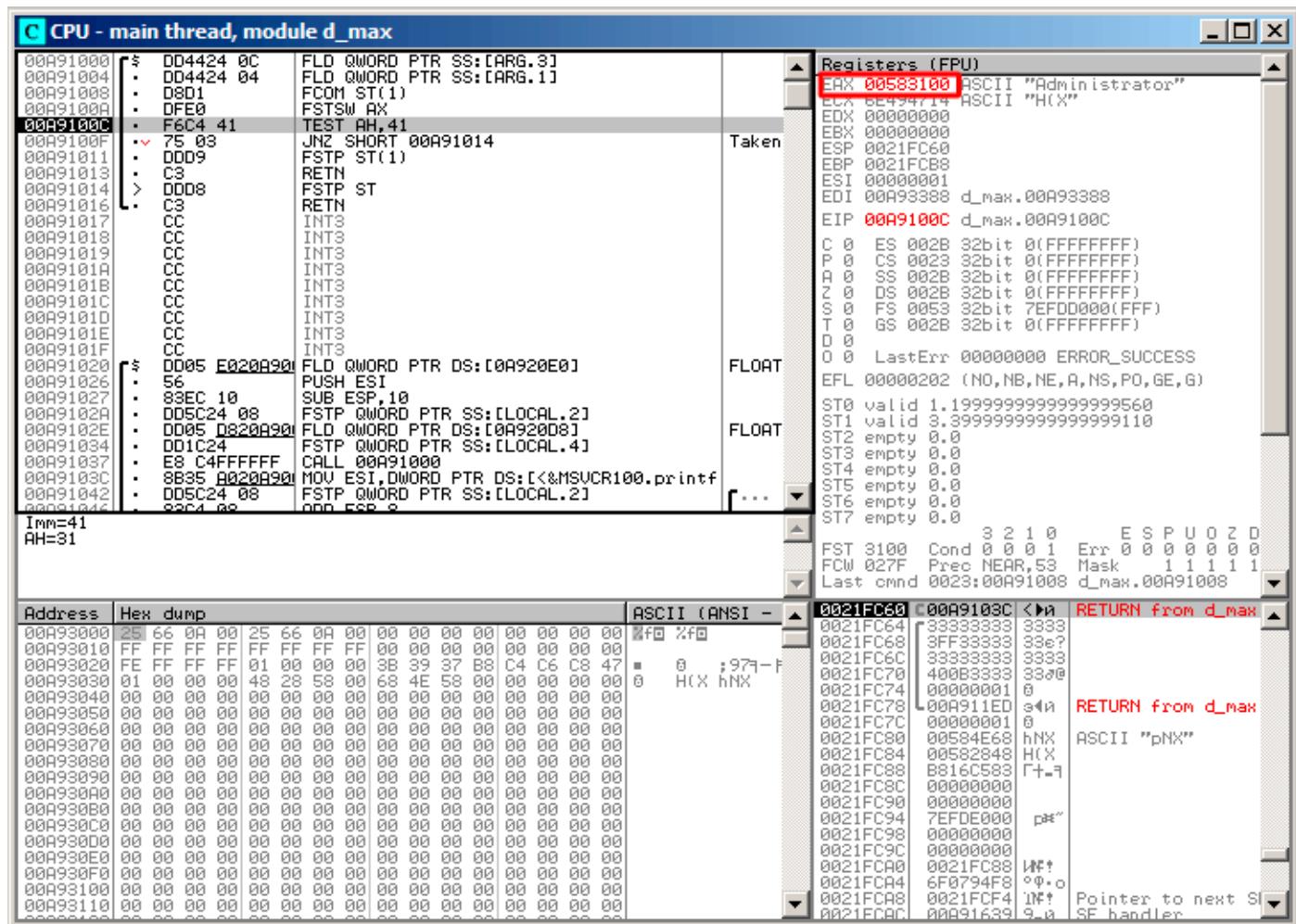


Figure 1.79: OllyDbg: FNSTSW is executed

## 1.19. FLOATING-POINT UNIT

TEST is executed:

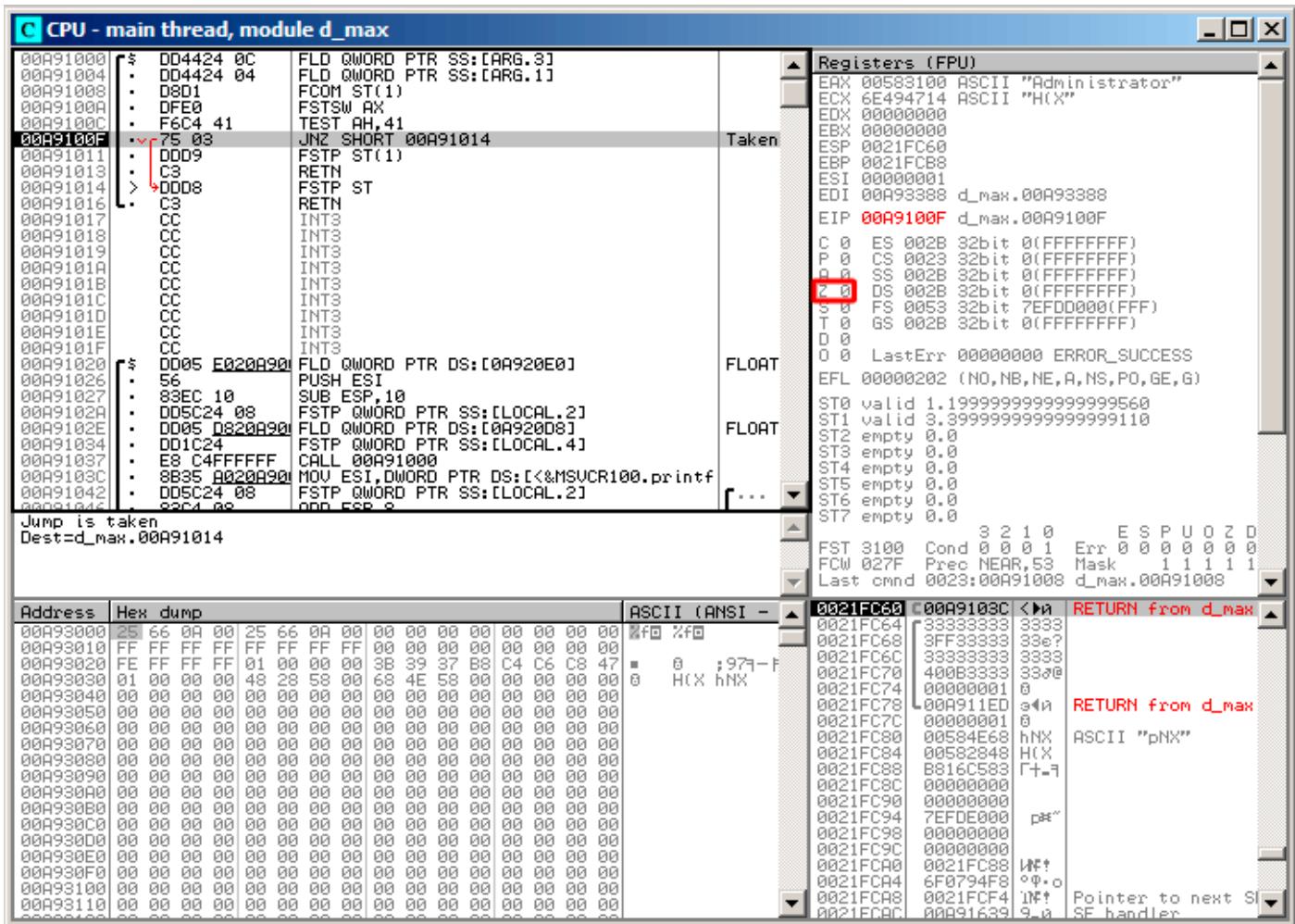


Figure 1.80: OllyDbg: TEST is executed

ZF=0, conditional jump is about to trigger now.

## 1.19. FLOATING-POINT UNIT

FSTP ST (or FSTP ST(0) ) has been executed —1.2 has been popped from the stack, and 3.4 was left on top:

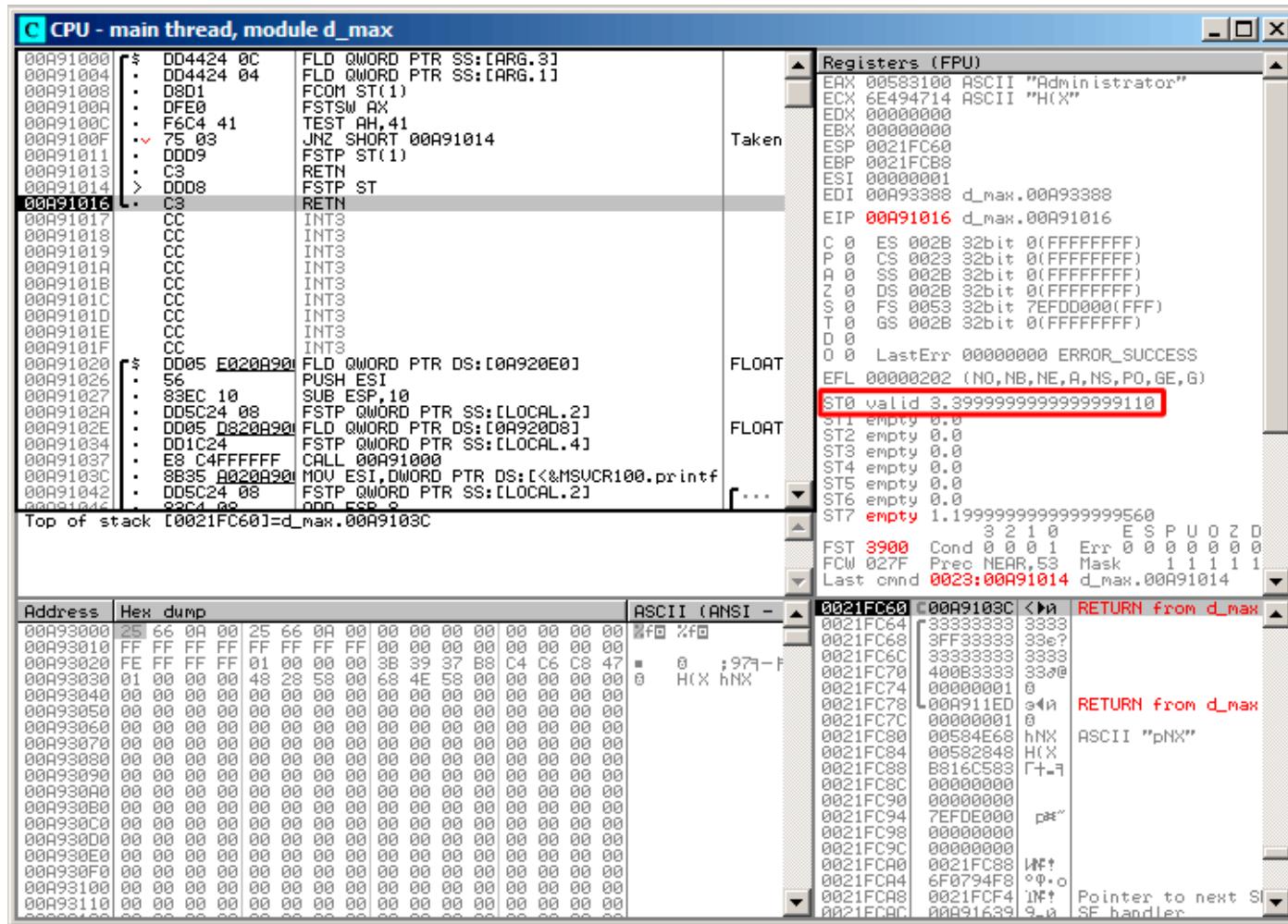


Figure 1.81: OllyDbg: FSTP is executed

We see that the FSTP ST

instruction works just like popping one value from the FPU stack.

## 1.19. FLOATING-POINT UNIT

### Second OllyDbg example: a=5.6 and b=-4

Both **FLD** are executed:

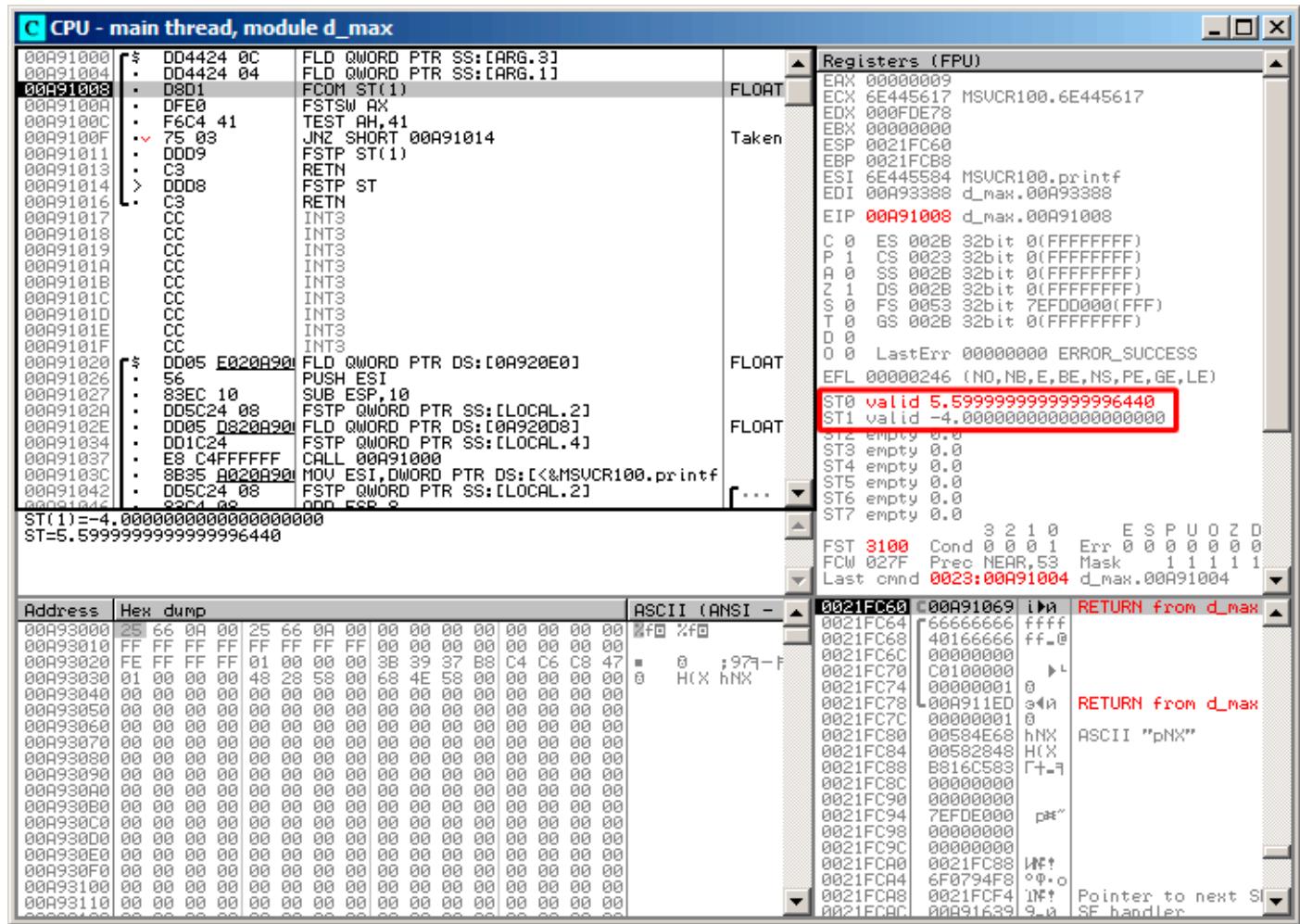


Figure 1.82: OllyDbg: both **FLD** are executed

**FCOM** is about to execute.

## 1.19. FLOATING-POINT UNIT

FCOM has been executed:

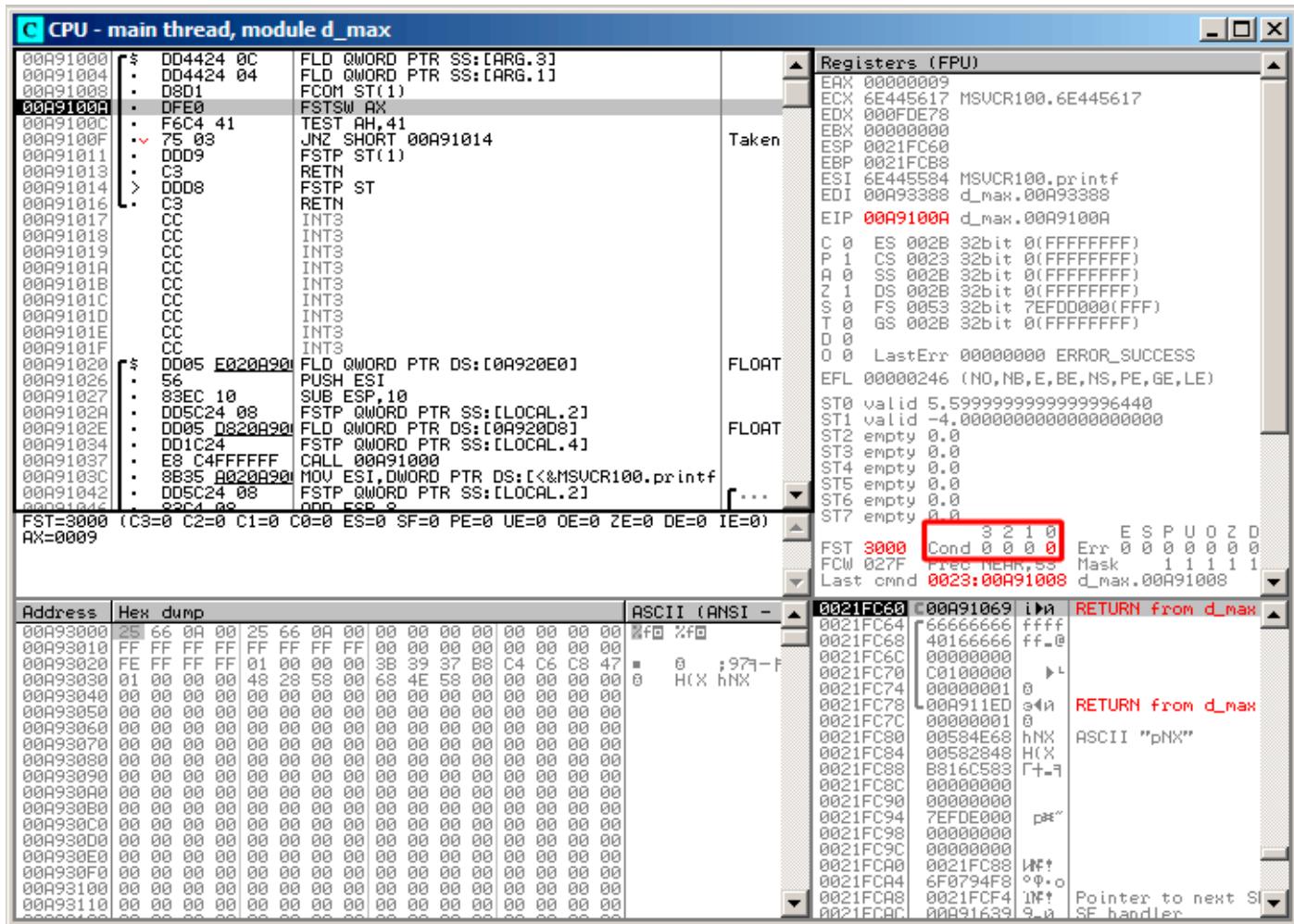


Figure 1.83: OllyDbg: FCOM is finished

All conditional flags are cleared.

## 1.19. FLOATING-POINT UNIT

FNSTSW done, AX =0x3000:

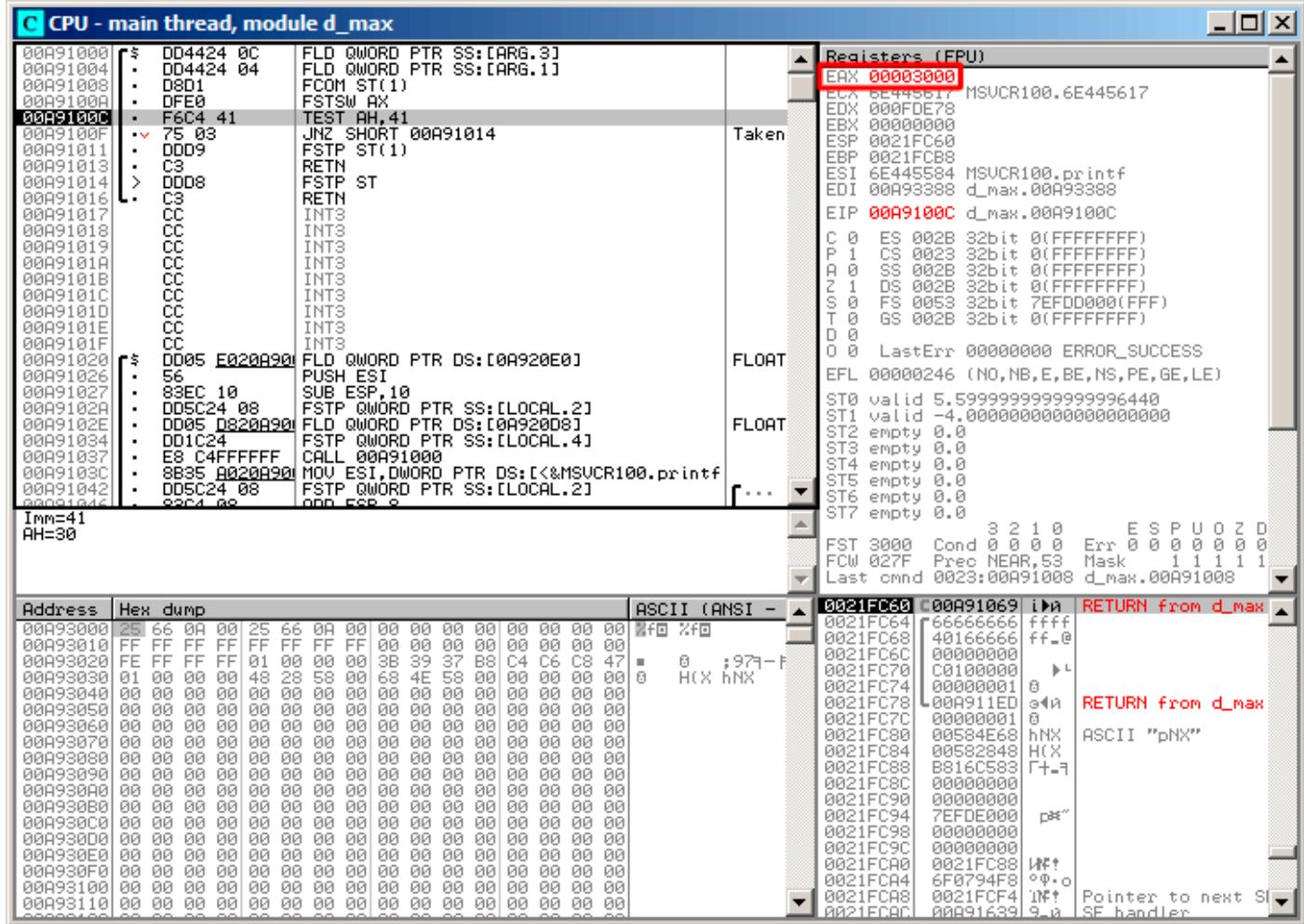


Figure 1.84: OllyDbg: FNSTSW has been executed

## 1.19. FLOATING-POINT UNIT

TEST has been executed:

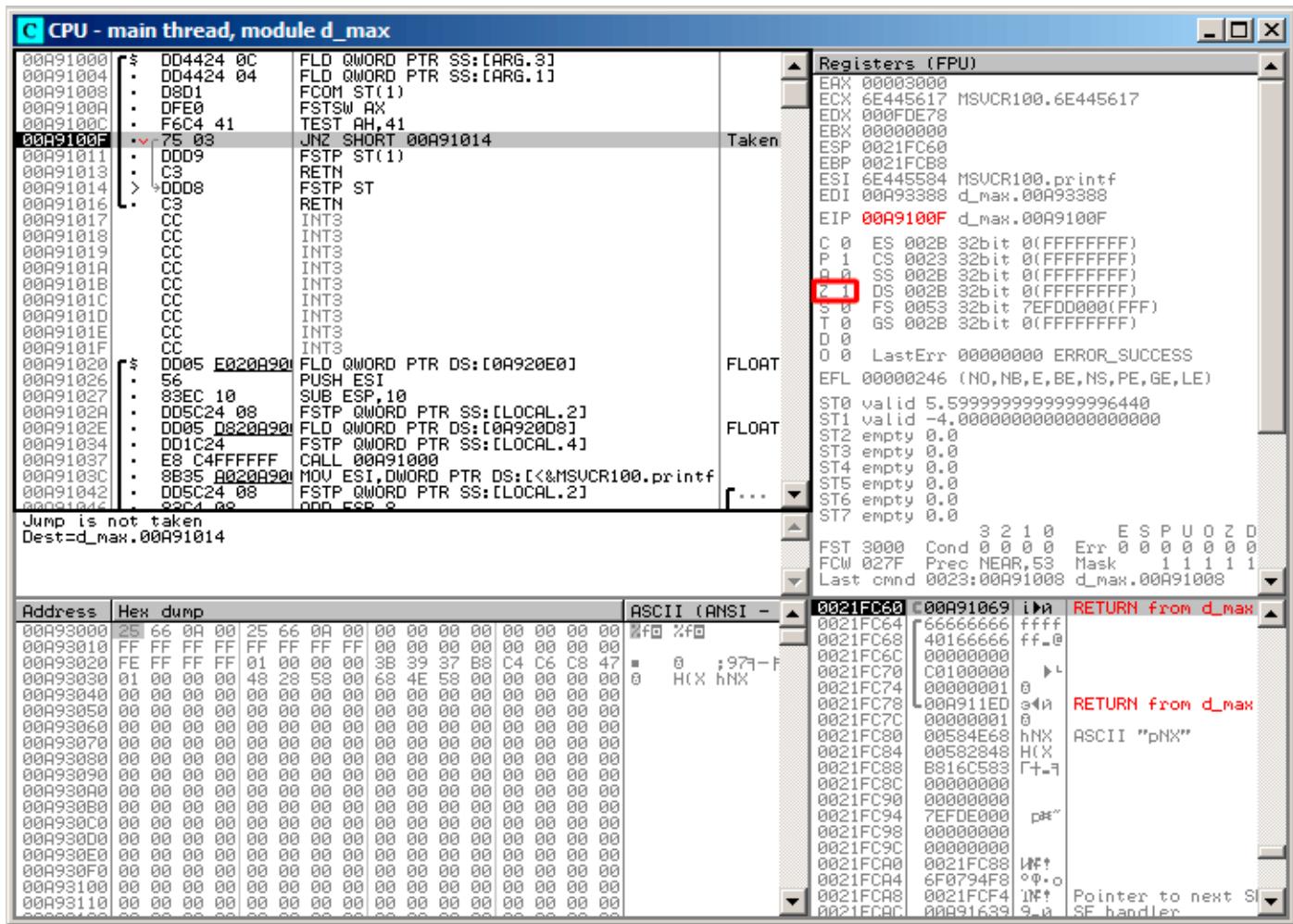


Figure 1.85: OllyDbg: TEST has been executed

ZF=1, jump will not happen now.

## 1.19. FLOATING-POINT UNIT

FSTP ST(1) has been executed: a value of 5.6 is now at the top of the FPU stack.

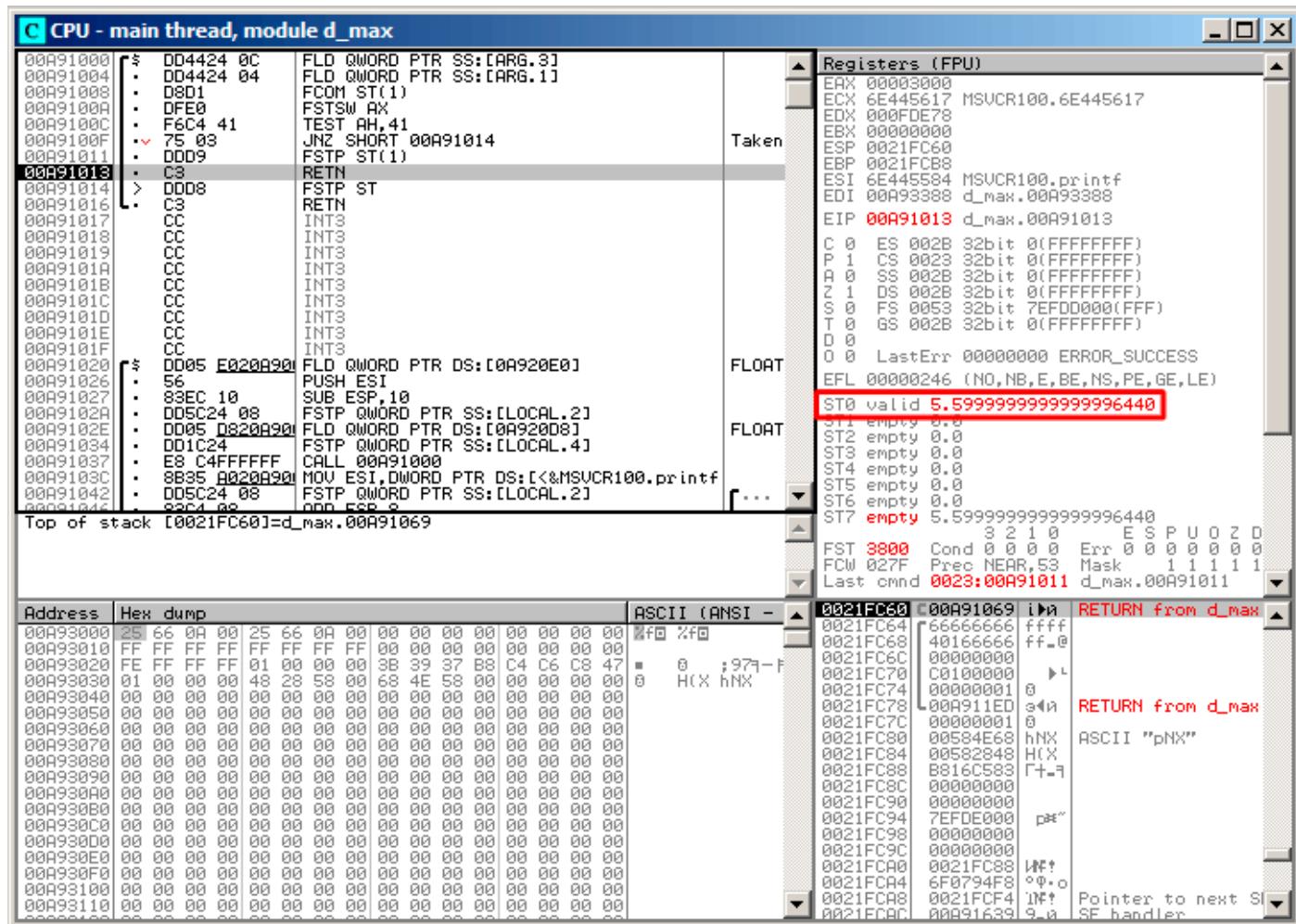


Figure 1.86: OllyDbg: FSTP has been executed

We now see that the FSTP ST(1) instruction works as follows: it leaves what has been at the top of the stack, but clears ST(1).

## GCC 4.4.1

Listing 1.211: GCC 4.4.1

```

d_max proc near

b          = qword ptr -10h
a          = qword ptr -8
a_first_half = dword ptr 8
a_second_half = dword ptr 0Ch
b_first_half = dword ptr 10h
b_second_half = dword ptr 14h

push    ebp
mov     ebp, esp
sub     esp, 10h

; put a and b to local stack:

mov     eax, [ebp+a_first_half]
mov     dword ptr [ebp+a], eax
mov     eax, [ebp+a_second_half]
mov     dword ptr [ebp+a+4], eax

```

## 1.19. FLOATING-POINT UNIT

```

mov      eax, [ebp+b_first_half]
mov      dword ptr [ebp+b], eax
mov      eax, [ebp+b_second_half]
mov      dword ptr [ebp+b+4], eax

; load a and b to FPU stack:

fld      [ebp+a]
fld      [ebp+b]

; current stack state: ST(0) - b; ST(1) - a

fxch    st(1) ; this instruction swapping ST(1) and ST(0)

; current stack state: ST(0) - a; ST(1) - b

fucompp   ; compare a and b and pop two values from stack, i.e., a and b
fnstsw  ax ; store FPU status to AX
sahf      ; load SF, ZF, AF, PF, and CF flags state from AH
setnbe  al ; store 1 to AL, if CF=0 and ZF=0
test    al, al      ; AL==0 ?
jz      short loc_8048453 ; yes
fld      [ebp+a]
jmp      short locret_8048456

loc_8048453:
fld      [ebp+b]

locret_8048456:
leave
retn
d_max endp

```

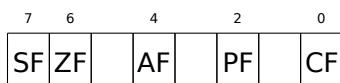
`FUCOMPP` is almost like `FCOM`, but pops both values from the stack and handles “not-a-numbers” differently.

A bit about *not-a-numbers*.

The FPU is able to deal with special values which are *not-a-numbers* or `NaN`<sup>125</sup>. These are infinity, result of division by 0, etc. Not-a-numbers can be “quiet” and “signaling”. It is possible to continue to work with “quiet” NaNs, but if one tries to do any operation with “signaling” NaNs, an exception is to be raised.

`FCOM` raising an exception if any operand is `NaN`. `FUCOM` raising an exception only if any operand is a signaling `NaN` (`SNaN`).

The next instruction is `SAHF` (*Store AH into Flags*) —this is a rare instruction in code not related to the FPU. 8 bits from AH are moved into the lower 8 bits of the CPU flags in the following order:



Let’s recall that `FNSTSW` moves the bits that interest us (`C3 / C2 / C0`) into `AH` and they are in positions 6, 2, 0 of the `AH` register:



In other words, the `fnstsw ax / sahf` instruction pair moves `C3 / C2 / C0` into `ZF`, `PF` and `CF`.

Now let’s also recall the values of `C3 / C2 / C0` in different conditions:

- If  $a$  is greater than  $b$  in our example, then `C3 / C2 / C0` are to be set to: 0, 0, 0.
- if  $a$  is less than  $b$ , then the bits are to be set to: 0, 0, 1.
- If  $a = b$ , then: 1, 0, 0.

In other words, these states of the CPU flags are possible after three

`FUCOMPP / FNSTSW / SAHF` instructions:

<sup>125</sup>[wikipedia.org/wiki/NaN](https://en.wikipedia.org/wiki/NaN)

## 1.19. FLOATING-POINT UNIT

- If  $a > b$ , the CPU flags are to be set as:  $ZF=0, PF=0, CF=0$ .
- If  $a < b$ , then the flags are to be set as:  $ZF=0, PF=0, CF=1$ .
- And if  $a = b$ , then:  $ZF=1, PF=0, CF=0$ .

Depending on the CPU flags and conditions, `SETNBE` stores 1 or 0 to AL. It is almost the counterpart of `JNBE`, with the exception that `SETcc`<sup>126</sup> stores 1 or 0 in AL, but `Jcc` does actually jump or not. `SETNBE` stores 1 only if `CF=0` and `ZF=0`. If it is not true, 0 is to be stored into AL.

Only in one case both `CF` and `ZF` are 0: if  $a > b$ .

Then 1 is to be stored to AL, the subsequent `JZ` is not to be triggered and the function will return `_a`. In all other cases, `_b` is to be returned.

### Optimizing GCC 4.4.1

Listing 1.212: Optimizing GCC 4.4.1

```

d_max      public d_max
            proc near

arg_0       = qword ptr 8
arg_8       = qword ptr 10h

            push    ebp
            mov     ebp, esp
            fld     [ebp+arg_0] ; _a
            fld     [ebp+arg_8] ; _b

; stack state now: ST(0) = _b, ST(1) = _a
            fxch   st(1)

; stack state now: ST(0) = _a, ST(1) = _b
            fucom  st(1) ; compare _a and _b
            fnstsw ax
            sahf
            ja     short loc_8048448

; store ST(0) to ST(0) (idle operation),
; pop value at top of stack,
; leave _b at top
            fstp   st
            jmp    short loc_804844A

loc_8048448:
; store _a to ST(1), pop value at top of stack, leave _a at top
            fstp   st(1)

loc_804844A:
            pop    ebp
            retn
d_max      endp

```

It is almost the same except that `JA` is used after `SAHF`. Actually, conditional jump instructions that check “larger”, “lesser” or “equal” for unsigned number comparison (these are `JA`, `JAE`, `JB`, `JBE`, `JE/JZ`, `JNA`, `JNAE`, `JNB`, `JNBE`, `JNE/JNZ`) check only flags `CF` and `ZF`.

Let's recall where bits `C3/C2/C0` are located in the `AH` register after the execution of `FSTSW/FNSTSW`:



Let's also recall, how the bits from `AH` are stored into the CPU flags the execution of `SAHF`:

<sup>126</sup>cc is condition code



After the comparison, the **C3** and **C0** bits are moved into **ZF** and **CF**, so the conditional jumps are able work after. **JA** is triggering if both **CF** are **ZF** zero.

Thereby, the conditional jumps instructions listed here can be used after a **FNSTSW / SAHF** instruction pair.

Apparently, the FPU **C3 / C2 / C0** status bits were placed there intentionally, to easily map them to base CPU flags without additional permutations?

### GCC 4.8.1 with **-O3** optimization turned on

Some new FPU instructions were added in the P6 Intel family<sup>127</sup>. These are **FUCOMI** (compare operands and set flags of the main CPU) and **FCM0Vcc** (works like **CMOVcc**, but on FPU registers).

Apparently, the maintainers of GCC decided to drop support of pre-P6 Intel CPUs (early Pentiums, 80486, etc.).

And also, the FPU is no longer separate unit in P6 Intel family, so now it is possible to modify/check flags of the main CPU from the FPU.

So what we get is:

Listing 1.213: Optimizing GCC 4.8.1

```

fld    QWORD PTR [esp+4]      ; load "a"
fld    QWORD PTR [esp+12]      ; load "b"
; ST0=b, ST1=a
fxch   st(1)
; ST0=a, ST1=b
; compare "a" and "b"
fucomi st, st(1)
; copy ST1 ("b" here) to ST0 if a<=b
; leave "a" in ST0 otherwise
fcmovbe st, st(1)
; discard value in ST1
fstp   st(1)
ret

```

Hard to guess why **FXCH** (swap operands) is here.

It's possible to get rid of it easily by swapping the first two **FLD** instructions or by replacing **FCM0VBE** (*below or equal*) by **FCM0VA** (*above*). Probably it's a compiler inaccuracy.

So **FUCOMI** compares **ST(0)** (*a*) and **ST(1)** (*b*) and then sets some flags in the main CPU. **FCM0VBE** checks the flags and copies **ST(1)** (*b* here at the moment) to **ST(0)** (*a* here) if **ST0(a) <= ST1(b)**. Otherwise (*a > b*), it leaves *a* in **ST(0)**.

The last **FSTP** leaves **ST(0)** on top of the stack, discarding the contents of **ST(1)**.

Let's trace this function in GDB:

Listing 1.214: Optimizing GCC 4.8.1 and GDB

```

1 dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
2 dennis@ubuntuvm:~/polygon$ gdb d_max
3 GNU gdb (GDB) 7.6.1-ubuntu
4 ...
5 Reading symbols from /home/dennis/polygon/d_max... (no debugging symbols found)...done.
6 (gdb) b d_max
7 Breakpoint 1 at 0x80484a0
8 (gdb) run
9 Starting program: /home/dennis/polygon/d_max
10
11 Breakpoint 1, 0x080484a0 in d_max ()
12 (gdb) ni

```

<sup>127</sup>Starting at Pentium Pro, Pentium-II, etc.

## 1.19. FLOATING-POINT UNIT

```
13 0x080484a4 in d_max ()
14 (gdb) disas $eip
15 Dump of assembler code for function d_max:
16     0x080484a0 <+0>:    fldl   0x4(%esp)
17 => 0x080484a4 <+4>:    fldl   0xc(%esp)
18     0x080484a8 <+8>:    fxch   %st(1)
19     0x080484aa <+10>:   fucomi %st(1),%st
20     0x080484ac <+12>:   fcmovbe %st(1),%st
21     0x080484ae <+14>:   fstp   %st(1)
22     0x080484b0 <+16>:   ret
23 End of assembler dump.
24 (gdb) ni
25 0x080484a8 in d_max ()
26 (gdb) info float
27     R7: Valid      0x3fff999999999999800 +1.19999999999999956
28 =>R6: Valid      0x4000d99999999999800 +3.39999999999999911
29     R5: Empty      0x00000000000000000000000000000000
30     R4: Empty      0x00000000000000000000000000000000
31     R3: Empty      0x00000000000000000000000000000000
32     R2: Empty      0x00000000000000000000000000000000
33     R1: Empty      0x00000000000000000000000000000000
34     R0: Empty      0x00000000000000000000000000000000
35
36 Status Word:      0x3000
37                 TOP: 6
38 Control Word:     0x037f  IM DM ZM OM UM PM
39                           PC: Extended Precision (64-bits)
40                           RC: Round to nearest
41 Tag Word:          0xffff
42 Instruction Pointer: 0x73:0x080484a4
43 Operand Pointer:   0x7b:0xbffff118
44 Opcode:            0x0000
45 (gdb) ni
46 0x080484aa in d_max ()
47 (gdb) info float
48     R7: Valid      0x4000d99999999999800 +3.39999999999999911
49 =>R6: Valid      0x3fff999999999999800 +1.19999999999999956
50     R5: Empty      0x00000000000000000000000000000000
51     R4: Empty      0x00000000000000000000000000000000
52     R3: Empty      0x00000000000000000000000000000000
53     R2: Empty      0x00000000000000000000000000000000
54     R1: Empty      0x00000000000000000000000000000000
55     R0: Empty      0x00000000000000000000000000000000
56
57 Status Word:      0x3000
58                 TOP: 6
59 Control Word:     0x037f  IM DM ZM OM UM PM
60                           PC: Extended Precision (64-bits)
61                           RC: Round to nearest
62 Tag Word:          0xffff
63 Instruction Pointer: 0x73:0x080484a8
64 Operand Pointer:   0x7b:0xbffff118
65 Opcode:            0x0000
66 (gdb) disas $eip
67 Dump of assembler code for function d_max:
68     0x080484a0 <+0>:    fldl   0x4(%esp)
69     0x080484a4 <+4>:    fldl   0xc(%esp)
70     0x080484a8 <+8>:    fxch   %st(1)
71 => 0x080484aa <+10>:   fucomi %st(1),%st
72     0x080484ac <+12>:   fcmovbe %st(1),%st
73     0x080484ae <+14>:   fstp   %st(1)
74     0x080484b0 <+16>:   ret
75 End of assembler dump.
76 (gdb) ni
77 0x080484ac in d_max ()
78 (gdb) info registers
79 eax          0x1          1
80 ecx          0xbffff1c4      -1073745468
81 edx          0x8048340       134513472
82 ebx          0xb7fbf000      -1208225792
```

## 1.19. FLOATING-POINT UNIT

```
83 esp          0xbfffff10c      0xbfffff10c
84 ebp          0xbfffff128      0xbfffff128
85 esi          0x0          0
86 edi          0x0          0
87 eip          0x80484ac      0x80484ac <d_max+12>
88 eflags       0x203      [ CF IF ]
89 cs           0x73        115
90 ss           0x7b        123
91 ds           0x7b        123
92 es           0x7b        123
93 fs           0x0          0
94 gs           0x33        51
95 (gdb) ni
96 0x080484ae in d_max ()
97 (gdb) info float
98   R7: Valid    0x4000d9999999999999800 +3.39999999999999911
99 =>R6: Valid    0x4000d9999999999999800 +3.39999999999999911
100  R5: Empty    0x00000000000000000000000000000000
101  R4: Empty    0x00000000000000000000000000000000
102  R3: Empty    0x00000000000000000000000000000000
103  R2: Empty    0x00000000000000000000000000000000
104  R1: Empty    0x00000000000000000000000000000000
105  R0: Empty    0x00000000000000000000000000000000
106
107 Status Word: 0x3000
108          TOP: 6
109 Control Word: 0x037f IM DM ZM OM UM PM
110          PC: Extended Precision (64-bits)
111          RC: Round to nearest
112 Tag Word: 0xffff
113 Instruction Pointer: 0x73:0x080484ac
114 Operand Pointer: 0x7b:0xbfffff118
115 Opcode: 0x0000
116 (gdb) disas $eip
117 Dump of assembler code for function d_max:
118 0x080484a0 <+0>: fildl 0x4(%esp)
119 0x080484a4 <+4>: fildl 0xc(%esp)
120 0x080484a8 <+8>: fxch %st(1)
121 0x080484aa <+10>: fucomi %st(1),%st
122 0x080484ac <+12>: fcmovebe %st(1),%st
123 => 0x080484ae <+14>: fstp %st(1)
124 0x080484b0 <+16>: ret
125 End of assembler dump.
126 (gdb) ni
127 0x080484b0 in d_max ()
128 (gdb) info float
129 =>R7: Valid    0x4000d9999999999999800 +3.39999999999999911
130  R6: Empty    0x4000d9999999999999800
131  R5: Empty    0x00000000000000000000000000000000
132  R4: Empty    0x00000000000000000000000000000000
133  R3: Empty    0x00000000000000000000000000000000
134  R2: Empty    0x00000000000000000000000000000000
135  R1: Empty    0x00000000000000000000000000000000
136  R0: Empty    0x00000000000000000000000000000000
137
138 Status Word: 0x3800
139          TOP: 7
140 Control Word: 0x037f IM DM ZM OM UM PM
141          PC: Extended Precision (64-bits)
142          RC: Round to nearest
143 Tag Word: 0x3fff
144 Instruction Pointer: 0x73:0x080484ae
145 Operand Pointer: 0x7b:0xbfffff118
146 Opcode: 0x0000
147 (gdb) quit
148 A debugging session is active.
149
150      Inferior 1 [process 30194] will be killed.
151
152 Quit anyway? (y or n) y
```

Using “ni”, let’s execute the first two `FLD` instructions.

Let’s examine the FPU registers (line 33).

As it was mentioned before, the FPU registers set is a circular buffer rather than a stack ([1.19.5 on page 225](#)). And GDB doesn’t show `STx` registers, but internal the FPU registers (`Rx`). The arrow (at line 35) points to the current top of the stack.

You can also see the `TOP` register contents in *Status Word* (line 44)—it is 6 now, so the stack top is now pointing to internal register 6.

The values of *a* and *b* are swapped after `FXCH` is executed (line 54).

`FUCOMI` is executed (line 83). Let’s see the flags: `CF` is set (line 95).

`FCMOVBE` has copied the value of *b* (see line 104).

`FSTP` leaves one value at the top of stack (line 136). The value of `TOP` is now 7, so the FPU stack top is pointing to internal register 7.

## ARM

### Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 1.215: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```

VMOV      D16, R2, R3 ; b
VMOV      D17, R0, R1 ; a
VCMPE.F64 D17, D16
VMRS      APSR_nzcv, FPSCR
VMOVGT.F64 D16, D17 ; copy "a" to D16
VMOV      R0, R1, D16
BX

```

A very simple case. The input values are placed into the `D17` and `D16` registers and then compared using the `VCMPE` instruction.

Just like in the x86 coprocessor, the ARM coprocessor has its own status and flags register ([FPSCR<sup>128</sup>](#)), since there is a necessity to store coprocessor-specific flags. And just like in x86, there are no conditional jump instruction in ARM, that can check bits in the status register of the coprocessor. So there is `VMRS`, which copies 4 bits (N, Z, C, V) from the coprocessor status word into bits of the *general* status register ([APSR<sup>129</sup>](#)).

`VMOVGT` is the analog of the `MOVGT`, instruction for D-registers, it executes if one operand is greater than the other while comparing (*GT—Greater Than*).

If it gets executed, the value of *a* is to be written into `D16` (that is currently stored in `D17`). Otherwise the value of *b* stays in the `D16` register.

The penultimate instruction `VMOV` prepares the value in the `D16` register for returning it via the `R0` and `R1` register pair.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Listing 1.216: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```

VMOV      D16, R2, R3 ; b
VMOV      D17, R0, R1 ; a
VCMPE.F64 D17, D16
VMRS      APSR_nzcv, FPSCR
IT GT

```

<sup>128</sup>(ARM) Floating-Point Status and Control Register

<sup>129</sup>(ARM) Application Program Status Register

## 1.19. FLOATING-POINT UNIT

VMOVGT.F64	D16, D17
VMOV	R0, R1, D16
BX	LR

Almost the same as in the previous example, however slightly different. As we already know, many instructions in ARM mode can be supplemented by condition predicate. But there is no such thing in Thumb mode. There is no space in the 16-bit instructions for 4 more bits in which conditions can be encoded.

However, Thumb-2 was extended to make it possible to specify predicates to old Thumb instructions. Here, in the [IDA](#)-generated listing, we see the `VMOVGT` instruction, as in previous example.

In fact, the usual `VMOV` is encoded there, but [IDA](#) adds the `-GT` suffix to it, since there is a `IT GT` instruction placed right before it.

The `IT` instruction defines a so-called *if-then block*.

After the instruction it is possible to place up to 4 instructions, each of them has a predicate suffix. In our example, `IT GT` implies that the next instruction is to be executed, if the `GT` (*Greater Than*) condition is true.

Here is a more complex code fragment, by the way, from Angry Birds (for iOS):

Listing 1.217: Angry Birds Classic

```
...
ITE NE
VMOVNE      R2, R3, D16
VMOVEQ      R2, R3, D17
BLX         _objc_msgSend ; not suffixed
...
```

`ITE` stands for *if-then-else*

and it encodes suffixes for the next two instructions.

The first instruction executes if the condition encoded in `ITE` (`NE`, *not equal*) is true at, and the second—if the condition is not true. (The inverse condition of `NE` is `EQ` (*equal*)).

The instruction followed after the second `VMOV` (or `VMOVEQ`) is a normal one, not suffixed (`BLX`).

One more that's slightly harder, which is also from Angry Birds:

Listing 1.218: Angry Birds Classic

```
...
ITTTT EQ
MOVEQ      R0, R4
ADDEQ      SP, SP, #0x20
POPEQ.W    {R8,R10}
POPEQ      {R4-R7,PC}
BLX        __stack_chk_fail ; not suffixed
...
```

Four “T” symbols in the instruction mnemonic mean that the four subsequent instructions are to be executed if the condition is true.

That's why [IDA](#) adds the `-EQ` suffix to each one of them.

And if there was, for example, `ITEEE EQ` (*if-then-else-else-else*), then the suffixes would have been set as follows:

```
-EQ
-NE
-NE
-NE
```

Another fragment from Angry Birds:

Listing 1.219: Angry Birds Classic

```
...
```

## 1.19. FLOATING-POINT UNIT

```
CMP.W      R0, #0xFFFFFFFF
ITTE LE    R10, R0, #1
SUBLE.W   R10, R0, #1
NEGLE     R0, R0
MOVGT     R10, R0
MOVS      R6, #0          ; not suffixed
CBZ      R0, loc_1E7E32 ; not suffixed
...
```

### ITTE (*if-then-then-else*)

implies that the 1st and 2nd instructions are to be executed if the **LE** (*Less or Equal*) condition is true, and the 3rd—if the inverse condition (**GT** —*Greater Than*) is true.

Compilers usually don't generate all possible combinations.

For example, in the mentioned Angry Birds game (*classic* version for iOS) only these variants of the **IT** instruction are used: **IT**, **ITE**, **ITT**, **ITTE**, **ITTT**, **ITTTT**. How to learn this? In **IDA** it is possible to produce listing files, so it was created with an option to show 4 bytes for each opcode. Then, knowing the high part of the 16-bit opcode (**IT** is **0xBF**), we do the following using **grep**:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

By the way, if you program in ARM assembly language manually for Thumb-2 mode, and you add conditional suffixes, the assembler will add the **IT** instructions automatically with the required flags where it is necessary.

## Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 1.220: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
b      = -0x20
a      = -0x18
val_to_return = -0x10
saved_R7    = -4

STR    R7, [SP,#saved_R7]!
MOV    R7, SP
SUB   SP, SP, #0x1C
BIC    SP, SP, #7
VMOV  D16, R2, R3
VMOV  D17, R0, R1
VSTR  D17, [SP,#0x20+a]
VSTR  D16, [SP,#0x20+b]
VLDR  D16, [SP,#0x20+a]
VLDR  D17, [SP,#0x20+b]
VCMPE.F64 D16, D17
VMRS  APSR_nzcv, FPSCR
BLE   loc_2E08
VLDR  D16, [SP,#0x20+a]
VSTR  D16, [SP,#0x20+val_to_return]
B     loc_2E10

loc_2E08
VLDR  D16, [SP,#0x20+b]
VSTR  D16, [SP,#0x20+val_to_return]

loc_2E10
VLDR  D16, [SP,#0x20+val_to_return]
VMOV  R0, R1, D16
MOV   SP, R7
LDR   R7, [SP+0x20+b],#4
BX    LR
```

Almost the same as we already saw, but there is too much redundant code because the *a* and *b* variables are stored in the local stack, as well as the return value.

**Optimizing Keil 6/2013 (Thumb mode)**

Listing 1.221: Optimizing Keil 6/2013 (Thumb mode)

```

PUSH {R3-R7,LR}
MOVS R4, R2
MOVS R5, R3
MOVS R6, R0
MOVS R7, R1
BL __aeabi_cdrcmple
BCS loc_1C0
MOVS R0, R6
MOVS R1, R7
POP {R3-R7,PC}

loc_1C0
MOVS R0, R4
MOVS R1, R5
POP {R3-R7,PC}

```

Keil doesn't generate FPU-instructions since it cannot rely on them being supported on the target CPU, and it cannot be done by straightforward bitwise comparing. So it calls an external library function to do the comparison: `__aeabi_cdrcmple`.

N.B. The result of the comparison is to be left in the flags by this function, so the following `BCS` (*Carry set—Greater than or equal*) instruction can work without any additional code.

**ARM64****Optimizing GCC (Linaro) 4.9**

```

d_max:
; D0 - a, D1 - b
    fcmpe d0, d1
    fcsel d0, d0, d1, gt
; now result in D0
    ret

```

The ARM64 ISA has FPU-instructions which set `APSR` the CPU flags instead of `FPSCR` for convenience. The `FPU` is not a separate device here anymore (at least, logically). Here we see `FCMPE`. It compares the two values passed in `D0` and `D1` (which are the first and second arguments of the function) and sets `APSR` flags (N, Z, C, V).

`FCSEL` (*Floating Conditional Select*) copies the value of `D0` or `D1` into `D0` depending on the condition (`GT`—Greater Than), and again, it uses flags in `APSR` register instead of `FPSCR`.

This is much more convenient, compared to the instruction set in older CPUs.

If the condition is true (`GT`), then the value of `D0` is copied into `D0` (i.e., nothing happens). If the condition is not true, the value of `D1` is copied into `D0`.

**Non-optimizing GCC (Linaro) 4.9**

```

d_max:
; save input arguments in "Register Save Area"
    sub sp, sp, #16
    str d0, [sp,8]
    str d1, [sp]
; reload values
    ldr x1, [sp,8]
    ldr x0, [sp]
    fmov d0, x1
    fmov d1, x0

```

## 1.19. FLOATING-POINT UNIT

```
; D0 - a, D1 - b
    fcmpe    d0, d1
    ble      .L76
; a>b; load D0 (a) into X0
    ldr      x0, [sp,8]
    b       .L74
.L76:
; a<=b; load D1 (b) into X0
    ldr      x0, [sp]
.L74:
; result in X0
    fmov    d0, x0
; result in D0
    add     sp, sp, 16
    ret
```

Non-optimizing GCC is more verbose.

First, the function saves its input argument values in the local stack (*Register Save Area*). Then the code reloads these values into registers `X0 / X1` and finally copies them to `D0 / D1` to be compared using `FCMPE`. A lot of redundant code, but that is how non-optimizing compilers work. `FCMPE` compares the values and sets the `APSR` flags. At this moment, the compiler is not thinking yet about the more convenient `FCSEL` instruction, so it proceed using old methods: using the `BLE` instruction (*Branch if Less than or Equal*). In the first case ( $a > b$ ), the value of  $a$  gets loaded into `X0`. In the other case ( $a \leq b$ ), the value of  $b$  gets loaded into `X0`. Finally, the value from `X0` gets copied into `D0`, because the return value needs to be in this register.

### Exercise

As an exercise, you can try optimizing this piece of code manually by removing redundant instructions and not introducing new ones (including `FCSEL`).

### Optimizing GCC (Linaro) 4.9—float

Let's also rewrite this example to use `float` instead of `double`.

```
float f_max (float a, float b)
{
    if (a>b)
        return a;

    return b;
};
```

```
f_max:
; S0 - a, S1 - b
    fcmpe    s0, s1
    fcsel    s0, s0, s1, gt
; now result in S0
    ret
```

It is the same code, but the S-registers are used instead of D- ones. It's because numbers of type `float` are passed in 32-bit S-registers (which are in fact the lower parts of the 64-bit D-registers).

### MIPS

The co-processor of the MIPS processor has a condition bit which can be set in the FPU and checked in the CPU.

Earlier MIPS-es have only one condition bit (called FCC0), later ones have 8 (called FCC7-FCC0).

This bit (or bits) are located in the register called FCCR.

```

d_max:
; set FPU condition bit if $f14<$f12 (b<a):
    c.lt.d  $f14, $f12
    or      $at, $zero ; NOP
; jump to locret_14 if condition bit is set
    bc1t   locret_14
; this instruction is always executed (set return value to "a"):
    mov.d   $f0, $f12 ; branch delay slot
; this instruction is executed only if branch was not taken (i.e., if b>=a)
; set return value to "b":
    mov.d   $f0, $f14

locret_14:
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP

```

`C.LT.D` compares two values. `LT` is the condition “Less Than”. `D` implies values of type *double*. Depending on the result of the comparison, the FCC0 condition bit is either set or cleared.

`BC1T` checks the FCC0 bit and jumps if the bit is set. `T` means that the jump is to be taken if the bit is set (“True”). There is also the instruction `BC1F` which jumps if the bit is cleared (“False”).

Depending on the jump, one of function arguments is placed into `$F0`.

## 1.19.8 Some constants

It’s easy to find representations of some constants in Wikipedia for IEEE 754 encoded numbers. It’s interesting to know that 0.0 in IEEE 754 is represented as 32 zero bits (for single precision) or 64 zero bits (for double). So in order to set a floating point variable to 0.0 in register or memory, one can use `MOV` or `XOR reg, reg` instruction. This is suitable for structures where many variables present of various data types. With usual `memset()` function one can set all integer variables to 0, all boolean variables to *false*, all pointers to `NULL`, and all floating point variables (of any precision) to 0.0.

## 1.19.9 Copying

One may think inertially that `FLD / FST` instructions must be used to load and store (and hence, copy) IEEE 754 values. Nevertheless, same can be achieved easier by usual `MOV` instruction, which, of course, copies values bitwisely.

## 1.19.10 Stack, calculators and reverse Polish notation

Now we understand why some old calculators use reverse Polish notation <sup>130</sup>.

For example, for addition of 12 and 34 one has to enter 12, then 34, then press “plus” sign.

It’s because old calculators were just stack machine implementations, and this was much simpler than to handle complex parenthesized expressions.

## 1.19.11 x64

On how floating point numbers are processed in x86-64, read more here: [1.31 on page 428](#).

## 1.19.12 Exercises

- <http://challenges.re/60>
- <http://challenges.re/61>

<sup>130</sup>[wikipedia.org/wiki/Reverse\\_Polish\\_notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation)

## 1.20 Arrays

An array is just a set of variables in memory that lie next to each other and that have the same type<sup>131</sup>.

### 1.20.1 Simple example

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

**x86**

**MSVC**

Let's compile:

Listing 1.223: MSVC 2008

```
_TEXT      SEGMENT
_i$ = -84          ; size = 4
_a$ = -80          ; size = 80
_main      PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84      ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN6@main
$LN5@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp     DWORD PTR _i$[ebp], 20      ; 00000014H
    jge     SHORT $LN4@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20      ; 00000014H
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    mov     edx, DWORD PTR _a$[ebp+ecx*4]
```

<sup>131</sup>AKA “homogeneous container”

## 1.20. ARRAYS

```
push    edx
mov     eax, DWORD PTR _i$[ebp]
push    eax
push    OFFSET $SG2463
call    _printf
add    esp, 12           ; 0000000cH
jmp    SHORT $LN2@main
$LN1@main:
xor    eax, eax
mov    esp, ebp
pop    ebp
ret    0
_main    ENDP
```

Nothing very special, just two loops: the first is a filling loop and second is a printing loop. The `shl ecx, 1` instruction is used for value multiplication by 2 in `ECX`, more about below [1.18.2 on page 217](#).

80 bytes are allocated on the stack for the array, 20 elements of 4 bytes.

## 1.20. ARRAYS

Let's try this example in OllyDbg.

We see how the array gets filled:

each element is 32-bit word of *int* type and its value is the index multiplied by 2:

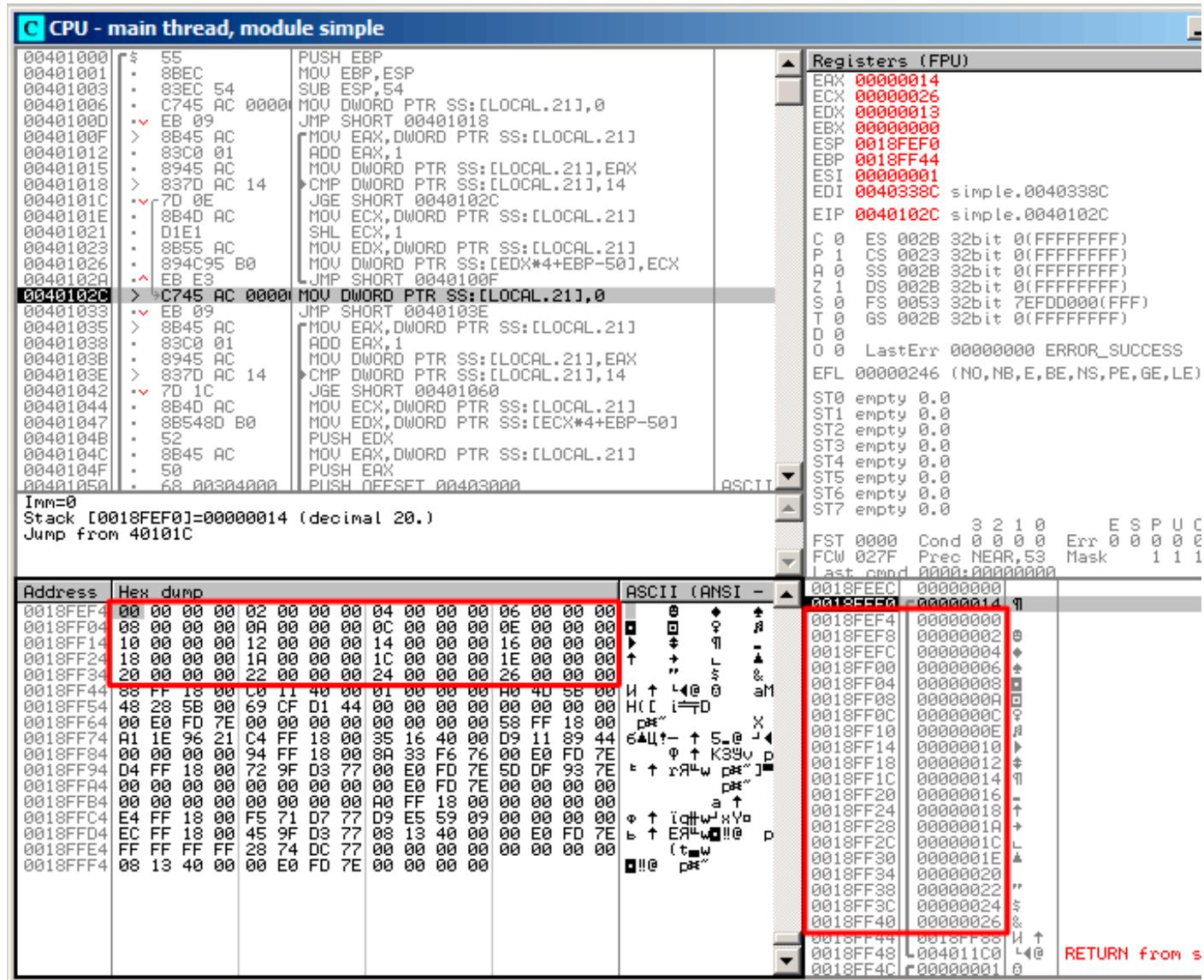


Figure 1.87: OllyDbg: after array filling

Since this array is located in the stack, we can see all its 20 elements there.

GCC

Here is what GCC 4.4.1 does:

Listing 1.224: GCC 4.4.1

```
public main
main    proc near ; DATA XREF: _start+17

var_70     = dword ptr -70h
var_6C     = dword ptr -6Ch
var_68     = dword ptr -68h
i_2        = dword ptr -54h
i          = dword ptr -4

push      ebp
mov       ebp, esp
```

## 1.20. ARRAYS

```

and    esp, 0FFFFFFF0h
sub    esp, 70h
mov    [esp+70h+i], 0           ; i=0
jmp    short loc_804840A

loc_80483F7:
    mov    eax, [esp+70h+i]
    mov    edx, [esp+70h+i]
    add    edx, edx             ; edx=i*2
    mov    [esp+eax*4+70h+i_2], edx
    add    [esp+70h+i], 1         ; i++

loc_804840A:
    cmp    [esp+70h+i], 13h
    jle    short loc_80483F7
    mov    [esp+70h+i], 0
    jmp    short loc_8048441

loc_804841B:
    mov    eax, [esp+70h+i]
    mov    edx, [esp+eax*4+70h+i_2]
    mov    eax, offset aADD ; "a[%d]=%d\n"
    mov    [esp+70h+var_68], edx
    mov    edx, [esp+70h+i]
    mov    [esp+70h+var_6C], edx
    mov    [esp+70h+var_70], eax
    call   _printf
    add    [esp+70h+i], 1

loc_8048441:
    cmp    [esp+70h+i], 13h
    jle    short loc_804841B
    mov    eax, 0
    leave
    retn
main  endp

```

By the way, variable *a* is of type *int\** (the pointer to *int*)—you can pass a pointer to an array to another function, but it's more correct to say that a pointer to the first element of the array is passed (the addresses of rest of the elements are calculated in an obvious way).

If you index this pointer as *a[idx]*, *idx* is just to be added to the pointer and the element placed there (to which calculated pointer is pointing) is to be returned.

An interesting example: a string of characters like “*string*” is an array of characters and it has a type of *const char[]*.

An index can also be applied to this pointer.

And that is why it is possible to write things like “*string*”[*i*] —this is a correct C/C++ expression!

## ARM

### Non-optimizing Keil 6/2013 (ARM mode)

```

EXPORT _main
_main
    STMF D SP!, {R4,LR}
    SUB     SP, SP, #0x50      ; allocate place for 20 int variables

; first loop

    MOV     R4, #0              ; i
    B      loc_4A0

loc_494
    MOV     R0, R4,LSL#1        ; R0=R4*2
    STR     R0, [SP,R4,LSL#2]   ; store R0 to SP+R4<<2 (same as SP+R4*4)
    ADD     R4, R4, #1          ; i=i+1

```

```

loc_4A0
    CMP    R4, #20          ; i<20?
    BLT    loc_494          ; yes, run loop body again

; second loop

    MOV    R4, #0            ; i
    B     loc_4C4

loc_4B0
    LDR    R2, [SP,R4,LSL#2] ; (second printf argument) R2=*(SP+R4<<4) (same as *(SP+✓
    ↳ R4*4))
    MOV    R1, R4            ; (first printf argument) R1=i
    ADR    R0, aADD          ; "a[%d]=%d\n"
    BL    __2printf
    ADD    R4, R4, #1          ; i=i+1

loc_4C4
    CMP    R4, #20          ; i<20?
    BLT    loc_4B0          ; yes, run loop body again
    MOV    R0, #0            ; value to return
    ADD    SP, SP, #0x50      ; deallocate chunk, allocated for 20 int variables
    LDMFD  SP!, {R4,PC}

```

*int* type requires 32 bits for storage (or 4 bytes),

so to store 20 *int* variables 80 (`0x50`) bytes are needed. So that is why the `SUB SP, SP, #0x50` instruction in the function's prologue allocates exactly this amount of space in the stack.

In both the first and second loops, the loop iterator *i* is placed in the `R4` register.

The number that is to be written into the array is calculated as  $i * 2$ , which is effectively equivalent to shifting it left by one bit,

so `MOV R0, R4, LSL#1` instruction does this.

`STR R0, [SP,R4,LSL#2]` writes the contents of `R0` into the array.

Here is how a pointer to array element is calculated: `SP` points to the start of the array, `R4` is *i*.

So shifting *i* left by 2 bits is effectively equivalent to multiplication by 4 (since each array element has a size of 4 bytes) and then it's added to the address of the start of the array.

The second loop has an inverse `LDR R2, [SP,R4,LSL#2]` instruction. It loads the value we need from the array, and the pointer to it is calculated likewise.

## Optimizing Keil 6/2013 (Thumb mode)

```

_main
    PUSH    {R4,R5,LR}
; allocate place for 20 int variables + one more variable
    SUB    SP, SP, #0x54

; first loop

    MOVS   R0, #0            ; i
    MOV    R5, SP            ; pointer to first array element

loc_1CE
    LSLS   R1, R0, #1          ; R1=i<<1 (same as i*2)
    LSLS   R2, R0, #2          ; R2=i<<2 (same as i*4)
    ADDS   R0, R0, #1          ; i=i+1
    CMP    R0, #20            ; i<20?
    STR    R1, [R5,R2]          ; store R1 to *(R5+R2) (same R5+i*4)
    BLT    loc_1CE            ; yes, i<20, run loop body again

; second loop

    MOVS   R4, #0            ; i=0

```

## 1.20. ARRAYS

```
loc_1DC
    LSLS    R0, R4, #2      ; R0=i<<2 (same as i*4)
    LDR    R2, [R5,R0]      ; load from *(R5+R0) (same as R5+i*4)
    MOVS   R1, R4
    ADR    R0, aADD        ; "a[%d]=%d\n"
    BL     __2printf
    ADDS   R4, R4, #1      ; i=i+1
    CMP    R4, #20         ; i<20?
    BLT   loc_1DC          ; yes, i<20, run loop body again
    MOVS   R0, #0          ; value to return
; deallocate chunk, allocated for 20 int variables + one more variable
    ADD    SP, SP, #0x54
    POP    {R4,R5,PC}
```

Thumb code is very similar.

Thumb mode has special instructions for bit shifting (like `LSLS`), which calculates the value to be written into the array and the address of each element in the array as well.

The compiler allocates slightly more space in the local stack, however, the last 4 bytes are not used.

### Non-optimizing GCC 4.9.1 (ARM64)

Listing 1.225: Non-optimizing GCC 4.9.1 (ARM64)

```
.LC0:
    .string "a[%d]=%d\n"
main:
; save FP and LR in stack frame:
    stp    x29, x30, [sp, -112]!
; set stack frame (FP=SP)
    add    x29, sp, 0
; setting initial counter variable at 0 (WZR is the register always holding zero):
    str    wZR, [x29,108]
; jump to loop condition checking code:
    b     .L2
.L3:
; load value of "i" variable:
    ldr    w0, [x29,108]
; multiplicate it by 2:
    lsl    w2, w0, 1
; find a place of an array in local stack:
    add    x0, x29, 24
; load 32-bit integer from local stack and sign extend it to 64-bit one:
    ldrsw  x1, [x29,108]
; calculate address of element (X0+X1<<2=array address+i*4) and store W2 (i*2) there:
    str    w2, [x0,x1,lsl 2]
; increment counter (i):
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L2:
; check if we finished:
    ldr    w0, [x29,108]
    cmp    w0, 19
; jump to L3 (loop body begin) if not:
    ble   .L3
; second part of the function begins here.
; setting initial counter variable at 0.
; by the way, the same place in the local stack was used for counter,
; because the same local variable (i) is being used as counter.
    str    wZR, [x29,108]
    b     .L4
.L5:
; calculate array address:
    add    x0, x29, 24
; load "i" value:
    ldrsw  x1, [x29,108]
; load value from the array at the address (X0+X1<<2 = address of array + i*4)
```

## 1.20. ARRAYS

```
    ldr    w2, [x0,x1,lsl 2]
; load address of the "a[%d]=%d\n" string:
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; load "i" variable to W1 and pass it to printf() as second argument:
    ldr    w1, [x29,108]
; W2 still contains the value of array element which was just loaded.
; call printf():
    bl     printf
; increment "i" variable:
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L4:
; are we finished?
    ldr    w0, [x29,108]
    cmp    w0, 19
; jump to the loop body begin if not:
    ble   .L5
; return 0
    mov    w0, 0
; restore FP and LR:
    ldp    x29, x30, [sp], 112
    ret
```

## MIPS

The function uses a lot of S- registers which must be preserved, so that's why its values are saved in the function prologue and restored in the epilogue.

Listing 1.226: Optimizing GCC 4.4.5 (IDA)

```
main:
var_70      = -0x70
var_68      = -0x68
var_14      = -0x14
var_10      = -0x10
var_C       = -0xC
var_8       = -8
var_4       = -4
; function prologue:
    lui    $gp, (__gnu_local_gp >> 16)
    addiu $sp, -0x80
    la    $gp, (__gnu_local_gp & 0xFFFF)
    sw    $ra, 0x80+var_4($sp)
    sw    $s3, 0x80+var_8($sp)
    sw    $s2, 0x80+var_C($sp)
    sw    $s1, 0x80+var_10($sp)
    sw    $s0, 0x80+var_14($sp)
    sw    $gp, 0x80+var_70($sp)
    addiu $s1, $sp, 0x80+var_68
    move  $v1, $s1
    move  $v0, $zero
; that value will be used as a loop terminator.
; it was precalculated by GCC compiler at compile stage:
    li    $a0, 0x28 # '('

loc_34:          # CODE XREF: main+3C
; store value into memory:
    sw    $v0, 0($v1)
; increase value to be stored by 2 at each iteration:
    addiu $v0, 2
; loop terminator reached?
    bne   $v0, $a0, loc_34
; add 4 to address anyway:
    addiu $v1, 4
; array filling loop is ended
```

## 1.20. ARRAYS

```
; second loop begin
    la      $s3, $LC0          # "a[%d]=%d\n"
; "i" variable will reside in $s0:
    move    $s0, $zero
    li      $s2, 0x14

loc_54:                           # CODE XREF: main+70
; call printf():
    lw      $t9, (printf & 0xFFFF)($gp)
    lw      $a2, 0($s1)
    move    $a1, $s0
    move    $a0, $s3
    jalr   $t9

; increment "i":
    addiu  $s0, 1
    lw      $gp, 0x80+var_70($sp)
; jump to loop body if end is not reached:
    bne    $s0, $s2, loc_54
; move memory pointer to the next 32-bit word:
    addiu  $s1, 4
; function epilogue
    lw      $ra, 0x80+var_4($sp)
    move    $v0, $zero
    lw      $s3, 0x80+var_8($sp)
    lw      $s2, 0x80+var_C($sp)
    lw      $s1, 0x80+var_10($sp)
    lw      $s0, 0x80+var_14($sp)
    jr      $ra
    addiu  $sp, 0x80

$LC0: .ascii "a[%d]=%d\n<0>" # DATA XREF: main+44
```

Something interesting: there are two loops and the first one doesn't need *i*, it needs only  $i * 2$  (increased by 2 at each iteration) and also the address in memory (increased by 4 at each iteration).

So here we see two variables, one (in \$V0) increasing by 2 each time, and another (in \$V1) — by 4.

The second loop is where `printf()` is called and it reports the value of *i* to the user, so there is a variable which is increased by 1 each time (in \$S0) and also a memory address (in \$S1) increased by 4 each time.

That reminds us of loop optimizations we considered earlier: [3.7 on page 489](#).

Their goal is to get rid of of multiplications.

### 1.20.2 Buffer overflow

#### Reading outside array bounds

So, array indexing is just `array[index]`. If you study the generated code closely, you'll probably note the missing index bounds checking, which could check *if it is less than 20*. What if the index is 20 or greater? That's the one C/C++ feature it is often blamed for.

Here is a code that successfully compiles and works:

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    printf ("a[20]=%d\n", a[20]);

    return 0;
}
```

## 1.20. ARRAYS

Compilation results (MSVC 2008):

Listing 1.227: Non-optimizing MSVC 2008

```
$SG2474 DB      'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN2@main
$LN1@main:
    mov     eax, DWORD PTR _a$[ebp+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

The code produced this result:

Listing 1.228: OllyDbg: console output

```
a[20]=1638280
```

It is just *something* that has been lying in the stack near to the array, 80 bytes away from its first element.

## 1.20. ARRAYS

Let's try to find out where did this value come from, using OllyDbg.

Let's load and find the value located right after the last array element:

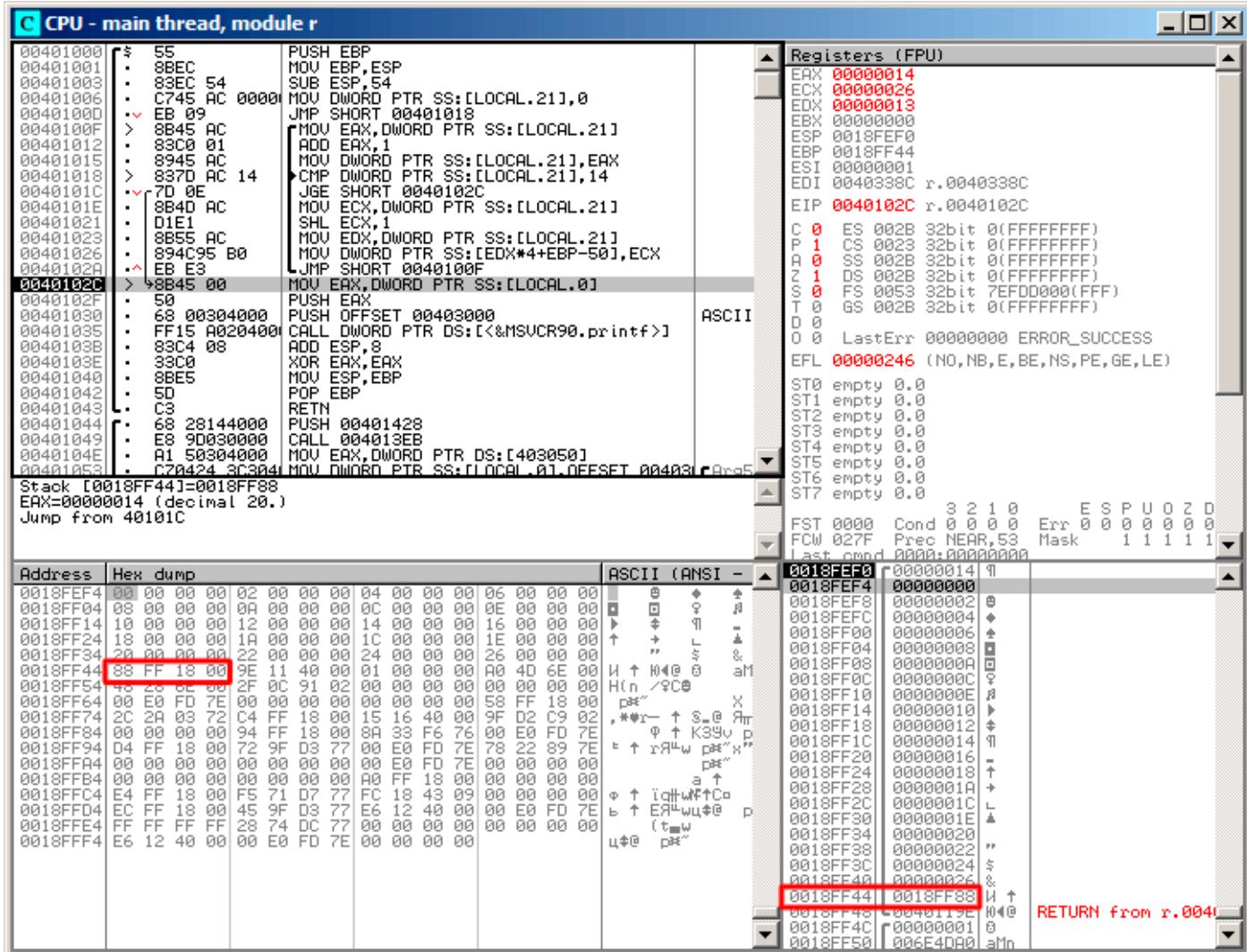


Figure 1.88: OllyDbg: reading of the 20th element and execution of `printf()`

What is this? Judging by the stack layout, this is the saved value of the EBP register.

## 1.20. ARRAYS

Let's trace further and see how it gets restored:

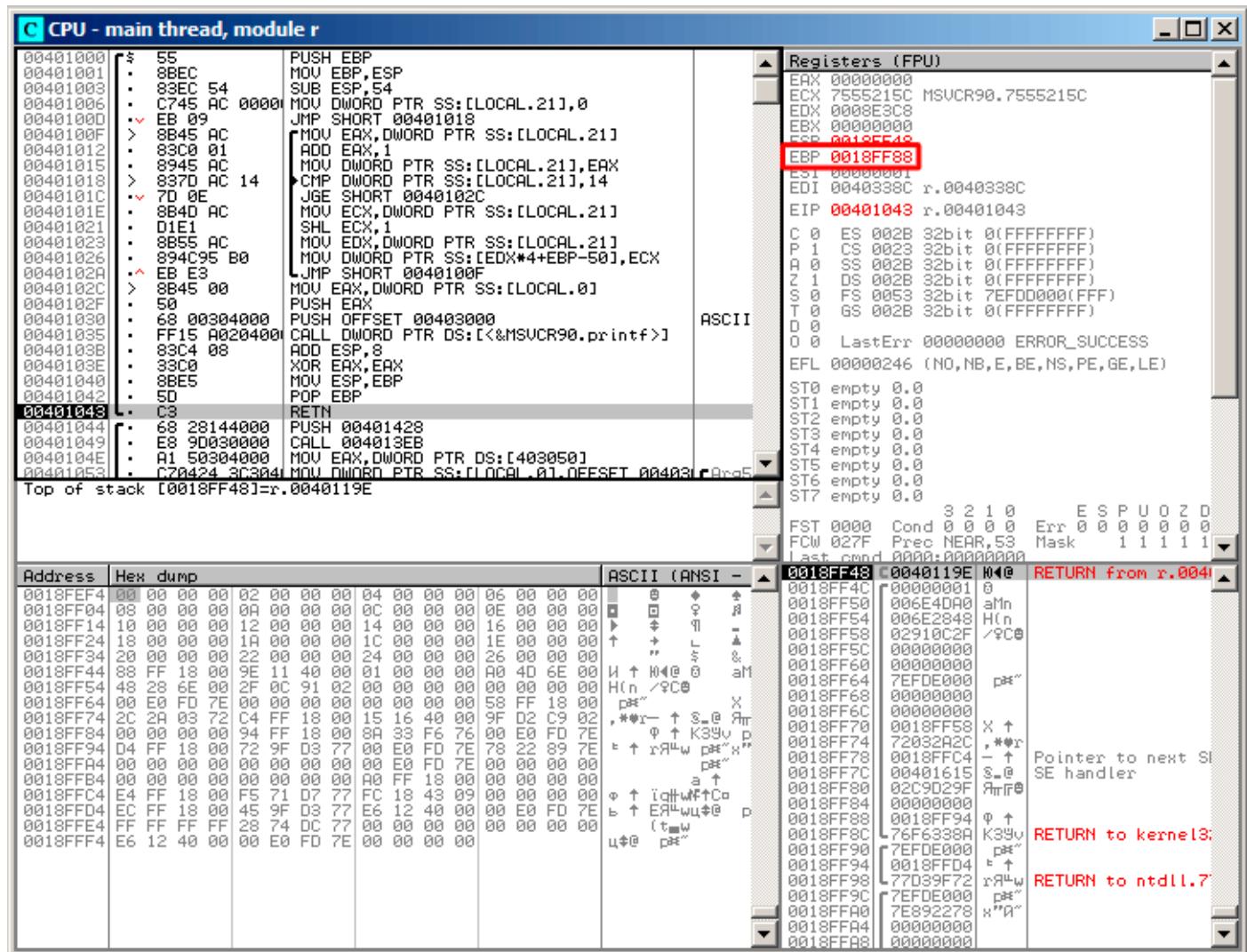


Figure 1.89: OllyDbg: restoring value of EBP

Indeed, how it could be different? The compiler may generate some additional code to check the index value to be always in the array's bounds (like in higher-level programming languages<sup>132</sup>) but this makes the code slower.

## Writing beyond array bounds

OK, we read some values from the stack *illegally*, but what if we could write something to it?

Here is what we have got:

```

#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
}

```

<sup>132</sup>Java, Python, etc.

And what we get:

Listing 1.229: Non-optimizing MSVC 2008

```

_TEXT    SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main   PROC
push    ebp
mov     ebp, esp
sub    esp, 84
mov     DWORD PTR _i$[ebp], 0
jmp     SHORT $LN3@main
$LN2@main:
mov     eax, DWORD PTR _i$[ebp]
add     eax, 1
mov     DWORD PTR _i$[ebp], eax
$LN3@main:
cmp     DWORD PTR _i$[ebp], 30 ; 0000001eH
jge     SHORT $LN1@main
mov     ecx, DWORD PTR _i$[ebp]
mov     edx, DWORD PTR _i$[ebp]      ; that instruction is obviously redundant
mov     DWORD PTR _a$[ebp+ecx*4], edx ; ECX could be used as second operand here instead
jmp     SHORT $LN2@main
$LN1@main:
xor    eax, eax
mov    esp, ebp
pop    ebp
ret    0
_main  ENDP

```

The compiled program crashes after running. No wonder. Let's see where exactly does it crash.

## 1.20. ARRAYS

Let's load it into OllyDbg, and trace until all 30 elements are written:

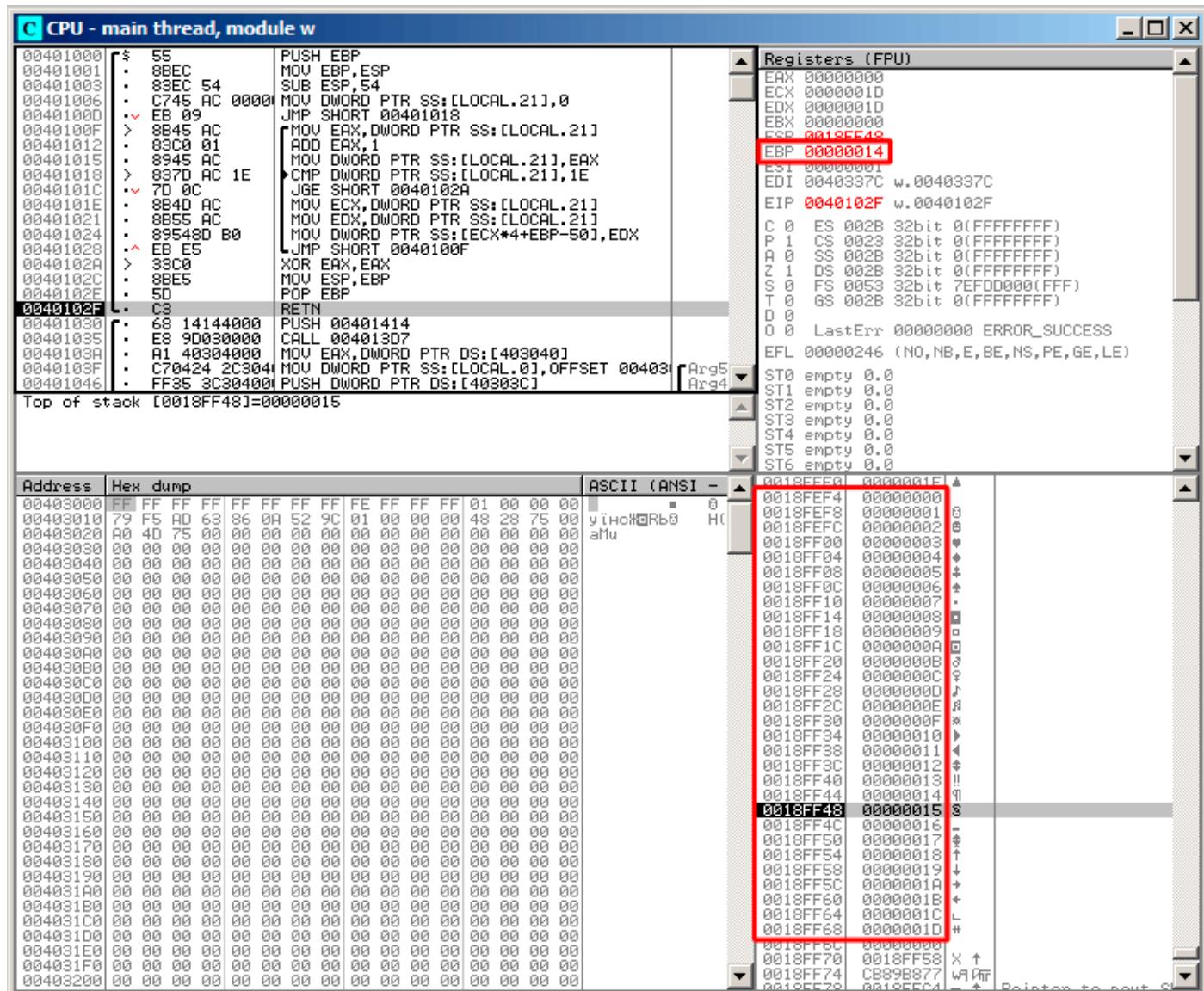


Figure 1.90: OllyDbg: after restoring the value of EBP

## 1.20. ARRAYS

Trace until the function end:

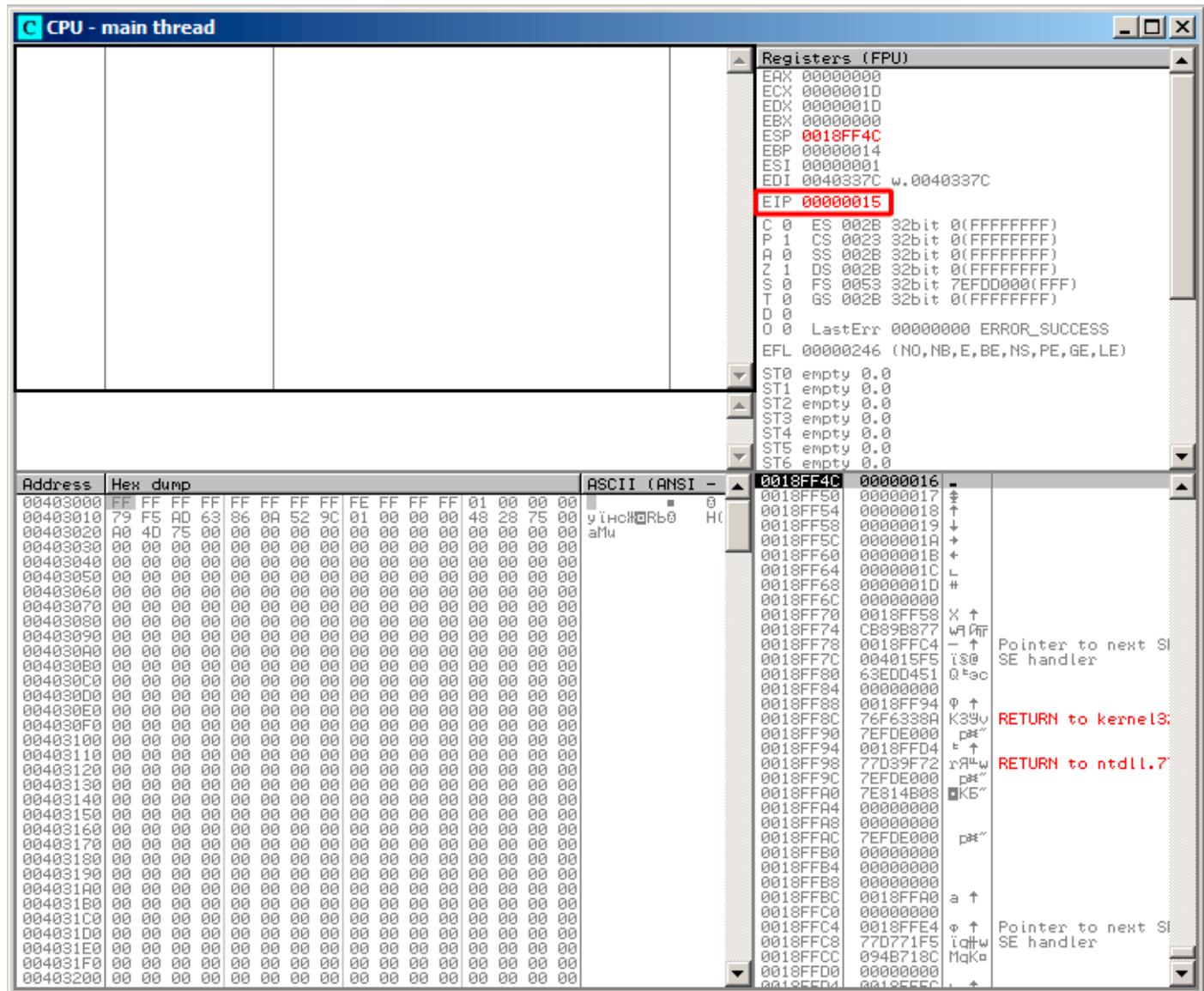


Figure 1.91: OllyDbg: `EIP` has been restored, but OllyDbg can't disassemble at `0x15`

Now please keep your eyes on the registers.

`EIP` is `0x15` now. It is not a legal address for code—at least for win32 code! We got there somehow against our will. It is also interesting that the `EBP` register contains `0x14`, `ECX` and `EDX` contain `0x1D`. Let's study stack layout a bit more.

After the control flow has been passed to `main()`, the value in the `EBP` register was saved on the stack. Then, 84 bytes were allocated for the array and the `i` variable. That's  $(20+1)*\text{sizeof}(\text{int})$ . `ESP` now points to the `_i` variable in the local stack and after the execution of the next `PUSH something`, `something` is appearing next to `_i`.

That's the stack layout while the control is in `main()`:

<code>ESP</code>	4 bytes allocated for <code>i</code> variable
<code>ESP+4</code>	80 bytes allocated for <code>a[20]</code> array
<code>ESP+84</code>	saved <code>EBP</code> value
<code>ESP+88</code>	return address

`a[19]=something` statement writes the last `int` in the bounds of the array (in bounds so far!).

## 1.20. ARRAYS

`a[20]=something` statement writes *something* to the place where the value of `EBP` is saved.

Please take a look at the register state at the moment of the crash. In our case, 20 has been written in the 20th element. At the function end, the function epilogue restores the original `EBP` value. (20 in decimal is `0x14` in hexadecimal). Then `RET` gets executed, which is effectively equivalent to `POP EIP` instruction.

The `RET` instruction takes the return address from the stack (that is the address in `CRT`), which has called `main()`, and 21 is stored there (`0x15` in hexadecimal). The CPU traps at address `0x15`, but there is no executable code there, so exception gets raised.

Welcome! It is called a *buffer overflow*<sup>133</sup>.

Replace the `int` array with a string (`char` array), create a long string deliberately and pass it to the program, to the function, which doesn't check the length of the string and copies it in a short buffer, and you'll be able to point the program to an address to which it must jump. It's not that simple in reality, but that is how it emerged. Classic article about it: [Aleph One, *Smashing The Stack For Fun And Profit*, (1996)]<sup>134</sup>.

## GCC

Let's try the same code in GCC 4.4.1. We get:

```
main          public main
              proc near
a             = dword ptr -54h
i             = dword ptr -4

              push    ebp
              mov     ebp, esp
              sub    esp, 60h ; 96
              mov     [ebp+i], 0
              jmp    short loc_80483D1
loc_80483C3:
              mov     eax, [ebp+i]
              mov     edx, [ebp+i]
              mov     [ebp+eax*4+a], edx
              add    [ebp+i], 1
loc_80483D1:
              cmp     [ebp+i], 1Dh
              jle    short loc_80483C3
              mov     eax, 0
              leave
              retn
main          endp
```

Running this in Linux will produce: `Segmentation fault`.

If we run this in the GDB debugger, we get this:

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x000000016 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xd2f96388      -755407992
edx            0x1d      29
ebx            0x26efff4 2551796
esp            0xbffff4b0      0xbffff4b0
ebp            0x15      0x15
esi            0x0      0
edi            0x0      0
eip            0x16      0x16
eflags          0x10202  [ IF RF ]
```

<sup>133</sup>[wikipedia](#)

<sup>134</sup>Also available as <http://go.yurichev.com/17266>

## 1.20. ARRAYS

```
cs          0x73    115
ss          0x7b    123
ds          0x7b    123
es          0x7b    123
fs          0x0     0
gs          0x33    51
(gdb)
```

The register values are slightly different than in win32 example, since the stack layout is slightly different too.

### 1.20.3 Buffer overflow protection methods

There are several methods to protect against this scourge, regardless of the C/C++ programmers' negligence. MSVC has options like<sup>135</sup>:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

One of the methods is to write a random value between the local variables in stack at function prologue and to check it in function epilogue before the function exits. If value is not the same, do not execute the last instruction `RET`, but stop (or hang). The process will halt, but that is much better than a remote attack to your host.

This random value is called a “canary” sometimes, it is related to the miners’ canary<sup>136</sup>, they were used by miners in the past days in order to detect poisonous gases quickly.

Canaries are very sensitive to mine gases, they become very agitated in case of danger, or even die.

If we compile our very simple array example ([1.20.1 on page 268](#)) in MSVC with RTC1 and RTCs option, you can see a call to `@_RTC_CheckStackVars@8` a function at the end of the function that checks if the “canary” is correct.

Let’s see how GCC handles this. Let’s take an `alloca()` ([1.7.2 on page 35](#)) example:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    sprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
}
```

By default, without any additional options, GCC 4.7.3 inserts a “canary” check into the code:

Listing 1.230: GCC 4.7.3

```
.LC0:
    .string "hi! %d, %d, %d\n"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 676
    lea     ebx, [esp+39]
```

<sup>135</sup>compiler-side buffer overflow protection methods: [wikipedia.org/wiki/Buffer\\_overflow\\_protection](http://wikipedia.org/wiki/Buffer_overflow_protection)

<sup>136</sup>[wikipedia.org/wiki/Domestic\\_canary#Miner.27s\\_canary](http://wikipedia.org/wiki/Domestic_canary#Miner.27s_canary)

## 1.20. ARRAYS

```
and    ebx, -16
mov    DWORD PTR [esp+20], 3
mov    DWORD PTR [esp+16], 2
mov    DWORD PTR [esp+12], 1
mov    DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
mov    DWORD PTR [esp+4], 600
mov    DWORD PTR [esp], ebx
mov    eax, DWORD PTR gs:20      ; canary
mov    DWORD PTR [ebp-12], eax
xor    eax, eax
call   _snprintf
mov    DWORD PTR [esp], ebx
call   puts
mov    eax, DWORD PTR [ebp-12]
xor    eax, DWORD PTR gs:20      ; check canary
jne    .L5
mov    ebx, DWORD PTR [ebp-4]
leave
ret
.L5:
call   __stack_chk_fail
```

The random value is located in `gs:20`. It gets written on the stack and then at the end of the function the value in the stack is compared with the correct “canary” in `gs:20`. If the values are not equal, the `__stack_chk_fail` function is called and we can see in the console something like that (Ubuntu 13.04 x86):

```
*** buffer overflow detected ***: ./2_1 terminated
===== Backtrace: =====
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586 /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0 [vdsol]
b775e000-b777e000 r-xp 00000000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0 [stack]
Aborted (core dumped)
```

`gs` is the so-called segment register. These registers were used widely in MS-DOS and DOS-extenders times. Today, its function is different.

To say it briefly, the `gs` register in Linux always points to the [TLS](#) ([6.2 on page 741](#))—some information specific to thread is stored there. By the way, in win32 the `fs` register plays the same role, pointing to [TIB](#)<sup>137 138</sup>.

<sup>137</sup>Thread Information Block

<sup>138</sup>[wikipedia.org/wiki/Win32\\_Thread\\_Information\\_Block](http://en.wikipedia.org/wiki/Win32_Thread_Information_Block)

## 1.20. ARRAYS

More information can be found in the Linux kernel source code (at least in 3.11 version), in `arch/x86/include/asm/stackprotector.h` this variable is described in the comments.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Let's get back to our simple array example ( [1.20.1 on page 268](#)),

again, now we can see how LLVM checks the correctness of the "canary":

```
_main

var_64      = -0x64
var_60      = -0x60
var_5C      = -0x5C
var_58      = -0x58
var_54      = -0x54
var_50      = -0x50
var_4C      = -0x4C
var_48      = -0x48
var_44      = -0x44
var_40      = -0x40
var_3C      = -0x3C
var_38      = -0x38
var_34      = -0x34
var_30      = -0x30
var_2C      = -0x2C
var_28      = -0x28
var_24      = -0x24
var_20      = -0x20
var_1C      = -0x1C
var_18      = -0x18
canary     = -0x14
var_10      = -0x10

PUSH {R4-R7,LR}
ADD R7, SP, #0xC
STR.W R8, [SP,#0xC+var_10]!
SUB SP, SP, #0x54
MOVW R0, #aobjc_methtype ; "objc_methtype"
MOVS R2, #0
MOVT.W R0, #0
MOVS R5, #0
ADD R0, PC
LDR.W R8, [R0]
LDR.W R0, [R8]
STR R0, [SP,#0x64+canary]
MOVS R0, #2
STR R2, [SP,#0x64+var_64]
STR R0, [SP,#0x64+var_60]
MOVS R0, #4
STR R0, [SP,#0x64+var_5C]
MOVS R0, #6
STR R0, [SP,#0x64+var_58]
MOVS R0, #8
STR R0, [SP,#0x64+var_54]
MOVS R0, #0xA
STR R0, [SP,#0x64+var_50]
MOVS R0, #0xC
STR R0, [SP,#0x64+var_4C]
MOVS R0, #0xE
STR R0, [SP,#0x64+var_48]
MOVS R0, #0x10
STR R0, [SP,#0x64+var_44]
MOVS R0, #0x12
STR R0, [SP,#0x64+var_40]
MOVS R0, #0x14
STR R0, [SP,#0x64+var_3C]
MOVS R0, #0x16
STR R0, [SP,#0x64+var_38]
```

## 1.20. ARRAYS

```
MOVS    R0, #0x18
STR     R0, [SP,#0x64+var_34]
MOVS    R0, #0x1A
STR     R0, [SP,#0x64+var_30]
MOVS    R0, #0x1C
STR     R0, [SP,#0x64+var_2C]
MOVS    R0, #0x1E
STR     R0, [SP,#0x64+var_28]
MOVS    R0, #0x20
STR     R0, [SP,#0x64+var_24]
MOVS    R0, #0x22
STR     R0, [SP,#0x64+var_20]
MOVS    R0, #0x24
STR     R0, [SP,#0x64+var_1C]
MOVS    R0, #0x26
STR     R0, [SP,#0x64+var_18]
MOV     R4, 0xFDA ; "a[%d]=%d\n"
MOV     R0, SP
ADDS   R6, R0, #4
ADD    R4, PC
B      loc_2F1C

; second loop begin

loc_2F14
ADDS   R0, R5, #1
LDR.W  R2, [R6,R5,LSL#2]
MOV    R5, R0

loc_2F1C
MOV    R0, R4
MOV    R1, R5
BLX    _printf
CMP    R5, #0x13
BNE   loc_2F14
LDR.W  R0, [R8]
LDR    R1, [SP,#0x64+canary]
CMP    R0, R1
ITTTT EQ           ; is canary still correct?
MOVEQ  R0, #0
ADDEQ  SP, SP, #0x54
LDREQ.W R8, [SP+0x64+var_64],#4
POPEQ  {R4-R7,PC}
BLX    __stack_chk_fail
```

First of all, as we see, LLVM “unrolled” the loop and all values were written into an array one-by-one, pre-calculated, as LLVM concluded it can work faster. By the way, instructions in ARM mode may help to do this even faster, and finding this could be your homework.

At the function end we see the comparison of the “canaries”—the one in the local stack and the correct one, to which `R8` points.

If they are equal to each other, a 4-instruction block is triggered by `ITTTT EQ`, which contains writing 0 in `R0`, the function epilogue and exit. If the “canaries” are not equal, the block being skipped, and the jump to `__stack_chk_fail` function will occur, which, perhaps will halt execution.

### 1.20.4 One more word about arrays

Now we understand why it is impossible to write something like this in C/C++ code:

```
void f(int size)
{
    int a[size];
...
};
```

## 1.20. ARRAYS

That's just because the compiler must know the exact array size to allocate space for it in the local stack layout on at the compiling stage.

If you need an array of arbitrary size, allocate it by using `malloc()`, then access the allocated memory block as an array of variables of the type you need.

Or use the C99 standard feature [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.5/2], and it works like `alloca()` ([1.7.2 on page 35](#)) internally.

It's also possible to use garbage collecting libraries for C.

And there are also libraries supporting smart pointers for C++.

### 1.20.5 Array of pointers to strings

Here is an example for an array of pointers.

Listing 1.231: Get month name

```
#include <stdio.h>

const char* month1[]=
{
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};

// in 0..11 range
const char* get_month1 (int month)
{
    return month1[month];
}
```

## x64

Listing 1.232: Optimizing MSVC 2013 x64

```
_DATA SEGMENT
month1 DQ     FLAT:$SG3122
        DQ     FLAT:$SG3123
        DQ     FLAT:$SG3124
        DQ     FLAT:$SG3125
        DQ     FLAT:$SG3126
        DQ     FLAT:$SG3127
        DQ     FLAT:$SG3128
        DQ     FLAT:$SG3129
        DQ     FLAT:$SG3130
        DQ     FLAT:$SG3131
        DQ     FLAT:$SG3132
        DQ     FLAT:$SG3133
$SG3122 DB     'January', 00H
$SG3123 DB     'February', 00H
$SG3124 DB     'March', 00H
$SG3125 DB     'April', 00H
$SG3126 DB     'May', 00H
$SG3127 DB     'June', 00H
$SG3128 DB     'July', 00H
$SG3129 DB     'August', 00H
```

## 1.20. ARRAYS

```
$SG3130 DB      'September', 00H
$SG3156 DB      '%s', 0AH, 00H
$SG3131 DB      'October', 00H
$SG3132 DB      'November', 00H
$SG3133 DB      'December', 00H
_DATA    ENDS

month$ = 8
get_month1 PROC
    movsx   rax, ecx
    lea     rcx, OFFSET FLAT:month1
    mov     rax, QWORD PTR [rcx+rax*8]
    ret    0
get_month1 ENDP
```

The code is very simple:

- The first `MOVSDX` instruction copies a 32-bit value from `ECX` (where `month` argument is passed) to `RAX` with sign-extension (because the `month` argument is of type `int`).

The reason for the sign extension is that this 32-bit value is to be used in calculations with other 64-bit values.

Hence, it has to be promoted to 64-bit<sup>139</sup>.

- Then the address of the pointer table is loaded into `RCX`.
- Finally, the input value (`month`) is multiplied by 8 and added to the address. Indeed: we are in a 64-bit environment and all address (or pointers) require exactly 64 bits (or 8 bytes) for storage. Hence, each table element is 8 bytes wide. And that's why to pick a specific element, `month * 8` bytes has to be skipped from the start. That's what `MOV` does. In addition, this instruction also loads the element at this address. For 1, an element would be a pointer to a string that contains "February", etc.

Optimizing GCC 4.9 can do the job even better <sup>140</sup>:

Listing 1.233: Optimizing GCC 4.9 x64

```
movsx   rdi, edi
mov     rax, QWORD PTR month1[0+rdi*8]
ret
```

## 32-bit MSVC

Let's also compile it in the 32-bit MSVC compiler:

Listing 1.234: Optimizing MSVC 2013 x86

```
_month$ = 8
_get_month1 PROC
    mov     eax, DWORD PTR _month$[esp-4]
    mov     eax, DWORD PTR _month1[eax*4]
    ret    0
_get_month1 ENDP
```

The input value does not need to be extended to 64-bit value, so it is used as is.

And it's multiplied by 4, because the table elements are 32-bit (or 4 bytes) wide.

## 32-bit ARM

### ARM in ARM mode

<sup>139</sup>It is somewhat weird, but negative array index could be passed here as `month` (negative array indices will have been explained later: [3.20 on page 597](#)). And if this happens, the negative input value of `int` type is sign-extended correctly and the corresponding element before table is picked. It is not going to work correctly without sign-extension.

<sup>140</sup>"0+" was left in the listing because GCC assembler output is not tidy enough to eliminate it. It's *displacement*, and it's zero here.

Listing 1.235: Optimizing Keil 6/2013 (ARM mode)

```

get_month1 PROC
    LDR    r1, |L0.100|
    LDR    r0,[r1,r0,LSL #2]
    BX    lr
    ENDP

|L0.100|
    DCD    |||.data||

    DCB    "January",0
    DCB    "February",0
    DCB    "March",0
    DCB    "April",0
    DCB    "May",0
    DCB    "June",0
    DCB    "July",0
    DCB    "August",0
    DCB    "September",0
    DCB    "October",0
    DCB    "November",0
    DCB    "December",0

    AREA |||.data||, DATA, ALIGN=2
month1
    DCD    |||.conststring||
    DCD    |||.conststring||+0x8
    DCD    |||.conststring||+0x11
    DCD    |||.conststring||+0x17
    DCD    |||.conststring||+0x1d
    DCD    |||.conststring||+0x21
    DCD    |||.conststring||+0x26
    DCD    |||.conststring||+0x2b
    DCD    |||.conststring||+0x32
    DCD    |||.conststring||+0x3c
    DCD    |||.conststring||+0x44
    DCD    |||.conststring||+0x4d

```

The address of the table is loaded in R1.

All the rest is done using just one `LDR` instruction.

Then input value `month` is shifted left by 2 (which is the same as multiplying by 4), then added to R1 (where the address of the table is) and then a table element is loaded from this address.

The 32-bit table element is loaded into R0 from the table.

## ARM in Thumb mode

The code is mostly the same, but less dense, because the `LSL` suffix cannot be specified in the `LDR` instruction here:

```

get_month1 PROC
    LSLS    r0,r0,#2
    LDR    r1, |L0.64|
    LDR    r0,[r1,r0]
    BX    lr
    ENDP

```

## ARM64

Listing 1.236: Optimizing GCC 4.9 ARM64

```

get_month1:
    adrp    x1, .LANCHOR0
    add     x1, x1, :lo12:.LANCHOR0

```

## 1.20. ARRAYS

```
    ldr    x0, [x1,w0,sxtw 3]
    ret

.LANCHOR0 = . + 0
.type   month1, %object
.size   month1, 96
month1:
    .xword  .LC2
    .xword  .LC3
    .xword  .LC4
    .xword  .LC5
    .xword  .LC6
    .xword  .LC7
    .xword  .LC8
    .xword  .LC9
    .xword  .LC10
    .xword  .LC11
    .xword  .LC12
    .xword  .LC13
.LC2:
    .string "January"
.LC3:
    .string "February"
.LC4:
    .string "March"
.LC5:
    .string "April"
.LC6:
    .string "May"
.LC7:
    .string "June"
.LC8:
    .string "July"
.LC9:
    .string "August"
.LC10:
    .string "September"
.LC11:
    .string "October"
.LC12:
    .string "November"
.LC13:
    .string "December"
```

The address of the table is loaded in X1 using `ADRP / ADD` pair.

Then corresponding element is picked using just one `LDR`, which takes W0 (the register where input argument `month` is), shifts it 3 bits to the left (which is the same as multiplying by 8), sign-extends it (this is what “`sxtw`” suffix implies) and adds to X0. Then the 64-bit value is loaded from the table into X0.

## MIPS

Listing 1.237: Optimizing GCC 4.4.5 (IDA)

```
get_month1:
; load address of table into $v0:
        la      $v0, month1
; take input value and multiply it by 4:
        sll    $a0, 2
; sum up address of table and multiplied value:
        addu   $a0, $v0
; load table element at this address into $v0:
        lw     $v0, 0($a0)
; return
        jr    $ra
        or     $at, $zero ; branch delay slot, NOP
                .data # .data.rel.local
```

```

.globl month1
month1:
    .word aJanuary      # "January"
    .word aFebruary     # "February"
    .word aMarch        # "March"
    .word aApril         # "April"
    .word aMay          # "May"
    .word aJune         # "June"
    .word aJuly         # "July"
    .word aAugust        # "August"
    .word aSeptember    # "September"
    .word aOctober       # "October"
    .word aNovember      # "November"
    .word aDecember      # "December"

    .data # .rodata.str1.4
aJanuary:   .ascii "January"<0>
aFebruary:  .ascii "February"<0>
aMarch:     .ascii "March"<0>
aApril:     .ascii "April"<0>
aMay:       .ascii "May"<0>
aJune:      .ascii "June"<0>
aJuly:      .ascii "July"<0>
aAugust:    .ascii "August"<0>
aSeptember: .ascii "September"<0>
aOctober:   .ascii "October"<0>
aNovember:  .ascii "November"<0>
aDecember:  .ascii "December"<0>

```

## Array overflow

Our function accepts values in the range of 0..11, but what if 12 is passed? There is no element in table at this place.

So the function will load some value which happens to be there, and return it.

Soon after, some other function can try to get a text string from this address and may crash.

Let's compile the example in MSVC for win64 and open it in [IDA](#) to see what the linker has placed after the table:

Listing 1.238: Executable file in IDA

```

off_140011000 dq offset aJanuary_1      ; DATA XREF: .text:0000000140001003
                  ; "January"
    dq offset aFebruary_1    ; "February"
    dq offset aMarch_1       ; "March"
    dq offset aApril_1       ; "April"
    dq offset aMay_1         ; "May"
    dq offset aJune_1        ; "June"
    dq offset aJuly_1        ; "July"
    dq offset aAugust_1      ; "August"
    dq offset aSeptember_1   ; "September"
    dq offset aOctober_1     ; "October"
    dq offset aNovember_1    ; "November"
    dq offset aDecember_1    ; "December"
aJanuary_1    db 'January',0            ; DATA XREF: sub_140001020+4
                  ; .data:off_140011000
aFebruary_1   db 'February',0          ; DATA XREF: .data:0000000140011008
                  align 4
aMarch_1      db 'March',0             ; DATA XREF: .data:0000000140011010
                  align 4
aApril_1      db 'April',0              ; DATA XREF: .data:0000000140011018

```

Month names are came right after.

Our program is tiny, so there isn't much data to pack in the data segment, so it just the month names. But it has to be noted that there might be really *anything* that linker has decided to put by chance.

So what if 12 is passed to the function? The 13th element will be returned.

## 1.20. ARRAYS

Let's see how the CPU treats the bytes there as a 64-bit value:

Listing 1.239: Executable file in IDA

```
off_140011000 dq offset qword_140011060          ; DATA XREF: .text:0000000140001003
                  dq offset aFebruary_1    ; "February"
                  dq offset aMarch_1     ; "March"
                  dq offset aApril_1     ; "April"
                  dq offset aMay_1       ; "May"
                  dq offset aJune_1      ; "June"
                  dq offset aJuly_1      ; "July"
                  dq offset aAugust_1    ; "August"
                  dq offset aSeptember_1 ; "September"
                  dq offset aOctober_1   ; "October"
                  dq offset aNovember_1  ; "November"
                  dq offset aDecember_1  ; "December"
qword_140011060 dq 797261756E614Ah           ; DATA XREF: sub_140001020+4
                  ; .data:off_140011000
aFebruary_1    db 'February',0                 ; DATA XREF: .data:0000000140011008
                  align 4
aMarch_1       db 'March',0                   ; DATA XREF: .data:0000000140011010
```

And this is 0x797261756E614A.

Soon after, some other function (presumably, one that processes strings) may try to read bytes at this address, expecting a C-string there.

Most likely it is about to crash, because this value doesn't look like a valid address.

### Array overflow protection

If something can go wrong, it will

Murphy's Law

It's a bit naïve to expect that every programmer who use your function or library will never pass an argument larger than 11.

There exists the philosophy that says "fail early and fail loudly" or "fail-fast", which teaches to report problems as early as possible and halt.

One such method in C/C++ is assertions.

We can modify our program to fail if an incorrect value is passed:

Listing 1.240: assert() added

```
const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
};
```

The assertion macro checks for valid values at every function start and fails if the expression is false.

Listing 1.241: Optimizing MSVC 2013 x64

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
      DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
      DB      '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5:
    push    rbx
    sub     rsp, 32
    movsxd  rbx, ecx
    cmp     ebx, 12
    jl      SHORT $LN3@get_month1
```

## 1.20. ARRAYS

```
    lea    rdx, OFFSET FLAT:$SG3143
    lea    rcx, OFFSET FLAT:$SG3144
    mov    r8d, 29
    call   _wassert
$LN3@get_month1:
    lea    rcx, OFFSET FLAT:month1
    mov    rax, QWORD PTR [rcx+rbx*8]
    add    rsp, 32
    pop    rbx
    ret    0
get_month1_checked ENDP
```

In fact, assert() is not a function, but macro. It checks for a condition, then passes also the line number and file name to another function which reports this information to the user.

Here we see that both file name and condition are encoded in UTF-16. The line number is also passed (it's 29).

This mechanism is probably the same in all compilers. Here is what GCC does:

Listing 1.242: Optimizing GCC 4.9 x64

```
.LC1:
    .string "month.c"
.LC2:
    .string "month<12"

get_month1_checked:
    cmp    edi, 11
    jg     .L6
    movsx  rdi, edi
    mov    rax, QWORD PTR month1[0+rdi*8]
    ret

.L6:
    push   rax
    mov    ecx, OFFSET FLAT:__PRETTY_FUNCTION__.2423
    mov    edx, 29
    mov    esi, OFFSET FLAT:.LC1
    mov    edi, OFFSET FLAT:.LC2
    call   __assert_fail

__PRETTY_FUNCTION__.2423:
    .string "get_month1_checked"
```

So the macro in GCC also passes the function name for convenience.

Nothing is really free, and this is true for the sanitizing checks as well.

They make your program slower, especially if the assert() macros used in small time-critical functions.

So MSVC, for example, leaves the checks in debug builds, but in release builds they all disappear.

Microsoft Windows NT kernels come in “checked” and “free” builds <sup>141</sup>.

The first has validation checks (hence, “checked”), the second one doesn’t (hence, “free” of checks).

Of course, “checked” kernel works slower because of all these checks, so it is usually used only in debug sessions.

### 1.20.6 Multidimensional arrays

Internally, a multidimensional array is essentially the same thing as a linear array.

Since the computer memory is linear, it is an one-dimensional array. For convenience, this multi-dimensional array can be easily represented as one-dimensional.

For example, this is how the elements of the 3x4 array are placed in one-dimensional array of 12 cells:

<sup>141</sup>[msdn.microsoft.com/en-us/library/windows/hardware/ff543450\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx)

Offset in memory	array element
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Table 1.3: Two-dimensional array represented in memory as one-dimensional

Here is how each cell of 3\*4 array are placed in memory:

0	1	2	3
4	5	6	7
8	9	10	11

Table 1.4: Memory addresses of each cell of two-dimensional array

So, in order to calculate the address of the element we need, we first multiply the first index by 4 (array width) and then add the second index. That's called *row-major order*, and this method of array and matrix representation is used in at least C/C++ and Python. The term *row-major order* in plain English language means: "first, write the elements of the first row, then the second row ...and finally the elements of the last row".

Another method for representation is called *column-major order* (the array indices are used in reverse order) and it is used at least in Fortran, MATLAB and R. *column-major order* term in plain English language means: "first, write the elements of the first column, then the second column ...and finally the elements of the last column".

Which method is better?

In general, in terms of performance and cache memory, the best scheme for data organization is the one, in which the elements are accessed sequentially.

So if your function accesses data per row, *row-major order* is better, and vice versa.

## Two-dimensional array example

We are going to work with an array of type *char*, which implies that each element requires only one byte in memory.

## Row filling example

Let's fill the second row with these values 0..3:

Listing 1.243: Row filling example

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // clear array
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
```

## 1.20. ARRAYS

```
a[x][y]=0;

// fill second row by 0..3:
for (y=0; y<4; y++)
    a[1][y]=y;
};
```

All three rows are marked with red. We see that second row now has values 0, 1, 2 and 3:

Address	Hex dump
00C33370	00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00
00C33380	02 00 00 00 C3 66 47 4E C3 66 47 4E 00 00 00 00
00C33390	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Figure 1.92: OllyDbg: array is filled

### Column filling example

Let's fill the third column with values: 0..2:

Listing 1.244: Column filling example

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // clear array
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // fill third column by 0..2:
    for (x=0; x<3; x++)
        a[x][2]=x;
}
```

The three rows are also marked in red here.

We see that in each row, at third position these values are written: 0, 1 and 2.

Address	Hex dump
01033380	00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390	00 00 00 00 1E AA EF 31 1E AA EF 31 00 00 00 00
010333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Figure 1.93: OllyDbg: array is filled

### Access two-dimensional array as one-dimensional

We can be easily assured that it's possible to access a two-dimensional array as one-dimensional array in at least two ways:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
    return array[a][b];
};
```

## 1.20. ARRAYS

```
char get_by_coordinates2 (char *array, int a, int b)
{
    // treat input array as one-dimensional
    // 4 is array width here
    return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
    // treat input array as pointer,
    // calculate address, get value at it
    // 4 is array width here
    return *(array+a*4+b);
};

int main()
{
    a[2][3]=123;
    printf ("%d\n", get_by_coordinates1(a, 2, 3));
    printf ("%d\n", get_by_coordinates2(a, 2, 3));
    printf ("%d\n", get_by_coordinates3(a, 2, 3));
}
```

Compile and run it: it shows correct values.

What MSVC 2013 did is fascinating, all three routines are just the same!

Listing 1.245: Optimizing MSVC 2013 x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=address of array
; RDX=a
; R8=b
    movsxd    rax, r8d
; EAX=b
    movsxd    r9, edx
; R9=a
    add       rax, rcx
; RAX=b+address of array
    movzx    eax, BYTE PTR [rax+r9*4]
; AL=load byte at address RAX+R9*4=b+address of array+a*4=address of array+a*4+b
    ret      0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
    movsxd    rax, r8d
    movsxd    r9, edx
    add       rax, rcx
    movzx    eax, BYTE PTR [rax+r9*4]
    ret      0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
    movsxd    rax, r8d
    movsxd    r9, edx
    add       rax, rcx
    movzx    eax, BYTE PTR [rax+r9*4]
    ret      0
get_by_coordinates1 ENDP
```

GCC also generates equivalent routines, but slightly different:

```

; RDI=address of array
; RSI=a
; RDX=b

get_by_coordinates1:
; sign-extend input 32-bit int values "a" and "b" to 64-bit ones
    movsx    rsi, esi
    movsx    rdx, edx
    lea     rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=address of array+a*4
    movzx   eax, BYTE PTR [rax+rdx]
; AL=load byte at address RAX+RDX=address of array+a*4+b
    ret

get_by_coordinates2:
    lea     eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
    cdqe
    movzx   eax, BYTE PTR [rdi+rax]
; AL=load byte at address RDI+RAX=address of array+b+a*4
    ret

get_by_coordinates3:
    sal    esi, 2
; ESI=a<<2=a*4
; sign-extend input 32-bit int values "a*4" and "b" to 64-bit ones
    movsx    rdx, edx
    movsx    rsi, esi
    add     rdi, rsi
; RDI=RDI+RSI=address of array+a*4
    movzx   eax, BYTE PTR [rdi+rdx]
; AL=load byte at address RDI+RDX=address of array+a*4+b
    ret

```

### Three-dimensional array example

It's the same for multidimensional arrays.

Now we are going to work with an array of type *int*: each element requires 4 bytes in memory.

Let's see:

Listing 1.247: simple example

```

#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
    a[x][y][z]=value;
}

```

### x86

We get (MSVC 2010):

Listing 1.248: MSVC 2010

```

_DATA  SEGMENT
COMM   _a:DWORD:01770H
_DATA  ENDS
PUBLIC _insert
_TEXT  SEGMENT
_x$ = 8           ; size = 4
_y$ = 12          ; size = 4

```

## 1.20. ARRAYS

```

_z$ = 16          ; size = 4
_value$ = 20      ; size = 4
_insert PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _x$[ebp]
    imul    eax, 2400          ; eax=600*4*x
    mov     ecx, DWORD PTR _y$[ebp]
    imul    ecx, 120           ; ecx=30*4*y
    lea     edx, DWORD PTR _a[eax+ecx] ; edx=a + 600*4*x + 30*4*y
    mov     eax, DWORD PTR _z$[ebp]
    mov     ecx, DWORD PTR _value$[ebp]
    mov     DWORD PTR [edx+eax*4], ecx ; *(edx+z*4)=value
    pop    ebp
    ret    0
_insert ENDP
_TEXT ENDS

```

Nothing special. For index calculation, three input arguments are used in the formula  $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$ , to represent the array as multidimensional. Do not forget that the *int* type is 32-bit (4 bytes), so all coefficients must be multiplied by 4.

Listing 1.249: GCC 4.4.1

```

public insert
insert proc near

x      = dword ptr  8
y      = dword ptr  0Ch
z      = dword ptr  10h
value  = dword ptr  14h

    push    ebp
    mov     ebp, esp
    push    ebx
    mov     ebx, [ebp+x]
    mov     eax, [ebp+y]
    mov     ecx, [ebp+z]
    lea     edx, [eax+eax]    ; edx=y*2
    mov     eax, edx          ; eax=y*2
    shl    eax, 4             ; eax=(y*2)<<4 = y*2*16 = y*32
    sub    eax, edx          ; eax=y*32 - y*2=y*30
    imul   edx, ebx, 600       ; edx=x*600
    add    eax, edx          ; eax=eax+edx=y*30 + x*600
    lea     edx, [eax+ecx]    ; edx=y*30 + x*600 + z
    mov     eax, [ebp+value]
    mov     dword ptr ds:a[edx*4], eax ; *(a+edx*4)=value
    pop    ebx
    pop    ebp
    retn
insert endp

```

The GCC compiler does it differently.

For one of the operations in the calculation ( $30y$ ), GCC produces code without multiplication instructions. This is how it done:  $(y + y) \ll 4 - (y + y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$ . Thus, for the  $30y$  calculation, only one addition operation, one bitwise shift operation and one subtraction operation are used. This works faster.

## ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

Listing 1.250: Non-optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```

_insert
value  = -0x10
z      = -0xC
y      = -8

```

## 1.20. ARRAYS

```
x      = -4

; allocate place in local stack for 4 values of int type
SUB    SP, SP, #0x10
MOV    R9, 0xFC2 ; a
ADD    R9, PC
LDR.W  R9, [R9] ; get pointer to array
STR    R0, [SP,#0x10+x]
STR    R1, [SP,#0x10+y]
STR    R2, [SP,#0x10+z]
STR    R3, [SP,#0x10+value]
LDR    R0, [SP,#0x10+value]
LDR    R1, [SP,#0x10+z]
LDR    R2, [SP,#0x10+y]
LDR    R3, [SP,#0x10+x]
MOV    R12, 2400
MUL.W  R3, R3, R12
ADD    R3, R9
MOV    R9, 120
MUL.W  R2, R2, R9
ADD    R2, R3
LSLS   R1, R1, #2 ; R1=R1<<2
ADD    R1, R2
STR    R0, [R1] ; R1 - address of array element
; deallocate chunk in local stack, allocated for 4 values of int type
ADD    SP, SP, #0x10
BX    LR
```

Non-optimizing LLVM saves all variables in local stack, which is redundant.

The address of the array element is calculated by the formula we already saw.

## ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

Listing 1.251: Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```
_insert
MOVW  R9, #0x10FC
MOV.W R12, #2400
MOVT.W R9, #0
RSB.W R1, R1, R1, LSL#4 ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD   R9, PC
LDR.W R9, [R9] ; R9 = pointer to a array
MLA.W R0, R0, R12, R9 ; R0 - x, R12 - 2400, R9 - pointer to a. R0=x*2400 + ptr to a
ADD.W R0, R0, R1, LSL#3 ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + ptr to a + y*15*8 =
                  ; ptr to a + y*30*4 + x*600*4
STR.W R3, [R0,R2,LSL#2] ; R2 - z, R3 - value. address=R0+z*4 =
                  ; ptr to a + y*30*4 + x*600*4 + z*4
BX    LR
```

The tricks for replacing multiplication by shift, addition and subtraction which we already saw are also present here.

Here we also see a new instruction for us: **RSB** (*Reverse Subtract*).

It works just as **SUB**, but it swaps its operands with each other before execution. Why? **SUB** and **RSB** are instructions, to the second operand of which shift coefficient may be applied: (**LSL#4**).

But this coefficient can be applied only to second operand.

That's fine for commutative operations like addition or multiplication (operands may be swapped there without changing the result).

But subtraction is a non-commutative operation, so **RSB** exist for these cases.

## MIPS

## 1.20. ARRAYS

My example is tiny, so the GCC compiler decided to put the *a* array into the 64KiB area addressable by the Global Pointer.

Listing 1.252: Optimizing GCC 4.4.5 (IDA)

```
insert:  
; $a0=x  
; $a1=y  
; $a2=z  
; $a3=value  
          sll      $v0, $a0, 5  
; $v0 = $a0<<5 = x*32  
          sll      $a0, 3  
; $a0 = $a0<<3 = x*8  
          addu    $a0, $v0  
; $a0 = $a0+$v0 = x*8+x*32 = x*40  
          sll      $v1, $a1, 5  
; $v1 = $a1<<5 = y*32  
          sll      $v0, $a0, 4  
; $v0 = $a0<<4 = x*40*16 = x*640  
          sll      $a1, 1  
; $a1 = $a1<<1 = y*2  
          subu    $a1, $v1, $a1  
; $a1 = $v1-$a1 = y*32-y*2 = y*30  
          subu    $a0, $v0, $a0  
; $a0 = $v0-$a0 = x*640-x*40 = x*600  
          la       $gp, __gnu_local_gp  
          addu    $a0, $a1, $a0  
; $a0 = $a1+$a0 = y*30+x*600  
          addu    $a0, $a2  
; $a0 = $a0+$a2 = y*30+x*600+z  
; load address of table:  
          lw       $v0, (a & 0xFFFF)($gp)  
; multiply index by 4 to seek array element:  
          sll      $a0, 2  
; sum up multiplied index and table address:  
          addu    $a0, $v0, $a0  
; store value into table and return:  
          jr       $ra  
          sw       $a3, 0($a0)  
  
.comm a:0x1770
```

## More examples

The computer screen is represented as a 2D array, but the video-buffer is a linear 1D array. We talk about it here: [8.13.2 on page 916](#).

Another example in this book is Minesweeper game: its field is also two-dimensional array: [8.3](#).

### 1.20.7 Pack of strings as a two-dimensional array

Let's revisit the function that returns the name of a month: listing [1.231](#).

As you see, at least one memory load operation is needed to prepare a pointer to the string that's the month's name.

Is it possible to get rid of this memory load operation?

In fact yes, if you represent the list of strings as a two-dimensional array:

```
#include <stdio.h>  
#include <assert.h>  
  
const char month2[12][10]=  
{  
    { 'J','a','n','u','a','r','y', 0, 0, 0 },  
    { 'F','e','b','r','u','a','r','y', 0, 0 },
```

## 1.20. ARRAYS

```
{ 'M','a','r','c','h', 0, 0, 0, 0, 0 },
{ 'A','p','r','i','l', 0, 0, 0, 0, 0 },
{ 'M','a','y', 0, 0, 0, 0, 0, 0 },
{ 'J','u','n','e', 0, 0, 0, 0, 0 },
{ 'J','u','l','y', 0, 0, 0, 0, 0 },
{ 'A','u','s','t', 0, 0, 0, 0, 0 },
{ 'S','e','p','t','e','m','b','e','r', 0 },
{ 'O','c','t','o','b','e','r', 0, 0, 0 },
{ 'N','o','v','e','m','b','e','r', 0, 0 },
{ 'D','e','c','e','m','b','e','r', 0, 0 }
};

// in 0..11 range
const char* get_month2 (int month)
{
    return &month2[month][0];
}
```

Here is what we've get:

Listing 1.253: Optimizing MSVC 2013 x64

```
month2 DB 04aH
        DB 061H
        DB 06eH
        DB 075H
        DB 061H
        DB 072H
        DB 079H
        DB 00H
        DB 00H
        DB 00H
...
get_month2 PROC
; sign-extend input argument and promote to 64-bit value
    movsx rax, ecx
    lea    rcx, QWORD PTR [rax+rax*4]
; RCX=month+month*4=month*5
    lea    rax, OFFSET FLAT:month2
; RAX=pointer to table
    lea    rax, QWORD PTR [rax+rcx*2]
; RAX=pointer to table + RCX*2=pointer to table + month*5*2=pointer to table + month*10
    ret    0
get_month2 ENDP
```

There are no memory accesses at all.

All this function does is to calculate a point at which the first character of the name of the month is: *pointer\_to\_the\_table + month \* 10*.

There are also two **LEA** instructions, which effectively work as several **MUL** and **MOV** instructions.

The width of the array is 10 bytes.

Indeed, the longest string here—"September"—is 9 bytes, and plus the terminating zero is 10 bytes.

The rest of the month names are padded by zero bytes, so they all occupy the same space (10 bytes).

Thus, our function works even faster, because all string start at an address which can be calculated easily.

Optimizing GCC 4.9 can do it even shorter:

Listing 1.254: Optimizing GCC 4.9 x64

```
movsx rdi, edi
lea    rax, [rdi+rdi*4]
lea    rax, month2[rax+rax]
ret
```

**LEA** is also used here for multiplication by 10.

Non-optimizing compilers do multiplication differently.

Listing 1.255: Non-optimizing GCC 4.9 x64

```
get_month2:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    movsx   rdx, eax
; RDX = sign-extended input value
    mov     rax, rdx
; RAX = month
    sal     rax, 2
; RAX = month<<2 = month*4
    add     rax, rdx
; RAX = RAX+RDX = month*4+month = month*5
    add     rax, rax
; RAX = RAX*2 = month*5*2 = month*10
    add     rax, OFFSET FLAT:month2
; RAX = month*10 + pointer to the table
    pop    rbp
    ret
```

Non-optimizing MSVC just uses `IMUL` instruction:

Listing 1.256: Non-optimizing MSVC 2013 x64

```
month$ = 8
get_month2 PROC
    mov     DWORD PTR [rsp+8], ecx
    movsxd  rax, DWORD PTR month$[rsp]
; RAX = sign-extended input value into 64-bit one
    imul   rax, rax, 10
; RAX = RAX*10
    lea     rcx, OFFSET FLAT:month2
; RCX = pointer to the table
    add     rcx, rax
; RCX = RCX+RAX = pointer to the table+month*10
    mov     rax, rcx
; RAX = pointer to the table+month*10
    mov     ecx, 1
; RCX = 1
    imul   rcx, rcx, 0
; RCX = 1*0 = 0
    add     rax, rcx
; RAX = pointer to the table+month*10 + 0 = pointer to the table+month*10
    ret    0
get_month2 ENDP
```

But one thing is weird here: why add multiplication by zero and adding zero to the final result?

This looks like a compiler code generator quirk, which wasn't caught by the compiler's tests (the resulting code works correctly, after all). We intentionally consider such pieces of code so the reader would understand, that sometimes one shouldn't puzzle over such compiler artifacts.

## 32-bit ARM

Optimizing Keil for Thumb mode uses the multiplication instruction `MULS`:

Listing 1.257: Optimizing Keil 6/2013 (Thumb mode)

```
; R0 = month
    MOVS   r1,#0xa
; R1 = 10
    MULS   r0,r1,r0
; R0 = R1*R0 = 10*month
    LDR    r1,|L0.68|
; R1 = pointer to the table
    ADDS   r0,r0,r1
; R0 = R0+R1 = 10*month + pointer to the table
    BX    lr
```

## 1.20. ARRAYS

Optimizing Keil for ARM mode uses add and shift operations:

Listing 1.258: Optimizing Keil 6/2013 (ARM mode)

```
; R0 = month
    LDR      r1, |L0.104|
; R1 = pointer to the table
    ADD      r0, r0, r0, LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = month*5
    ADD      r0, r1, r0, LSL #1
; R0 = R1+R0<<2 = pointer to the table + month*5*2 = pointer to the table + month*10
    BX      lr
```

## ARM64

Listing 1.259: Optimizing GCC 4.9 ARM64

```
; W0 = month
    sxtw    x0, w0
; X0 = sign-extended input value
    adrp    x1, .LANCHOR1
    add     x1, x1, :lo12:.LANCHOR1
; X1 = pointer to the table
    add     x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
    add     x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = pointer to the table + X0*10
    ret
```

SXTW is used for sign-extension and promoting input 32-bit value into a 64-bit one and storing it in X0.

ADRP / ADD pair is used for loading the address of the table.

The ADD instructions also has a LSL suffix, which helps with multiplications.

## MIPS

Listing 1.260: Optimizing GCC 4.4.5 (IDA)

```
.globl get_month2
get_month2:
; $a0=month
    sll      $v0, $a0, 3
; $v0 = $a0<<3 = month*8
    sll      $a0, 1
; $a0 = $a0<<1 = month*2
    addu   $a0, $v0
; $a0 = month*2+month*8 = month*10
; load address of the table:
    la      $v0, month2
; sum up table address and index we calculated and return:
    jr      $ra
    addu   $v0, $a0

month2:    .ascii "January"<0>
            .byte 0, 0
aFebruary: .ascii "February"<0>
            .byte 0
aMarch:    .ascii "March"<0>
            .byte 0, 0, 0, 0
aApril:    .ascii "April"<0>
            .byte 0, 0, 0, 0
aMay:      .ascii "May"<0>
            .byte 0, 0, 0, 0, 0
aJune:     .ascii "June"<0>
            .byte 0, 0, 0, 0, 0
aJuly:     .ascii "July"<0>
```

## 1.21. BY THE WAY

```
aAugust:      .byte 0, 0, 0, 0, 0  
.ascii "August"<0>  
.byte 0, 0, 0  
aSeptember:   .ascii "September"<0>  
aOctober:    .ascii "October"<0>  
.byte 0, 0  
aNovermber:   .ascii "November"<0>  
.byte 0  
aDecember:    .ascii "December"<0>  
.byte 0, 0, 0, 0, 0, 0, 0, 0, 0
```

## Conclusion

This is a bit old-school technique to store text strings. You may find a lot of it in Oracle RDBMS, for example. It's hard to say if it's worth doing on modern computers. Nevertheless, it is a good example of arrays, so it was added to this book.

### 1.20.8 Conclusion

An array is a pack of values in memory located adjacently.

It's true for any element type, including structures.

Access to a specific array element is just a calculation of its address.

## 1.21 By the way

So, pointer to an array and address of a first element—is the same thing. This is why `ptr[0]` and `*ptr` expressions are equivalent in C/C++. It's interesting to note that Hex-Rays often replaces the first by the second. It does so when it has no idea that it works with pointer to the whole array, and thinks that this is a pointer to single variable.

### 1.21.1 Exercises

- <http://challenges.re/62>
- <http://challenges.re/63>
- <http://challenges.re/64>
- <http://challenges.re/65>
- <http://challenges.re/66>

## 1.22 Manipulating specific bit(s)

A lot of functions define their input arguments as flags in bit fields.

Of course, they could be substituted by a set of *bool*-typed variables, but it is not frugally.

### 1.22.1 Specific bit checking

#### x86

Win32 API example:

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS,
                FILE_ATTRIBUTE_NORMAL, NULL);
```

We get (MSVC 2010):

Listing 1.261: MSVC 2010

```
push    0
push    128           ; 00000080H
push    4
push    0
push    1
push    -1073741824   ; c0000000H
push    OFFSET $SG78813
call    DWORD PTR __imp__CreateFileA@28
mov     DWORD PTR _fh$[ebp], eax
```

Let's take a look in WinNT.h:

Listing 1.262: WinNT.h

```
#define GENERIC_READ          (0x80000000L)
#define GENERIC_WRITE         (0x40000000L)
#define GENERIC_EXECUTE        (0x20000000L)
#define GENERIC_ALL            (0x10000000L)
```

Everything is clear, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, and that value is used as the second argument for the `CreateFile()`<sup>142</sup> function.

How would `CreateFile()` check these flags?

If we look in KERNEL32.DLL in Windows XP SP3 x86, we'll find this fragment of code in `CreateFileW`:

Listing 1.263: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429  test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D  mov     [ebp+var_8], 1
.text:7C83D434  jz     short loc_7C83D417
.text:7C83D436  jmp     loc_7C810817
```

Here we see the `TEST` instruction, however it doesn't take the whole second argument, but only the most significant byte (`ebp+dwDesiredAccess+3`) and checks it for flag `0x40` (which implies the `GENERIC_WRITE` flag here)

`TEST` is basically the same instruction as `AND`, but without saving the result (recall the fact `CMP` is merely the same as `SUB`, but without saving the result ([1.9.4 on page 86](#))).

The logic of this code fragment is as follows:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

If `AND` instruction leaves this bit, the `ZF` flag is to be cleared and the `JZ` conditional jump is not to be triggered. The conditional jump is triggered only if the `0x40000000` bit is absent in `dwDesiredAccess` variable —then the result of `AND` is 0, `ZF` is to be set and the conditional jump is to be triggered.

Let's try GCC 4.4.1 and Linux:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
    int handle;

    handle=open ("file", O_RDWR | O_CREAT);
}
```

<sup>142</sup>[msdn.microsoft.com/en-us/library/aa363858\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx)

## 1.22. MANIPULATING SPECIFIC BIT(S)

We get:

Listing 1.264: GCC 4.4.1

```
public main
main proc near

var_20      = dword ptr -20h
var_1C      = dword ptr -1Ch
var_4       = dword ptr -4

push    ebp
mov     ebp, esp
and    esp, 0FFFFFFF0h
sub    esp, 20h
mov     [esp+20h+var_1C], 42h
mov     [esp+20h+var_20], offset aFile ; "file"
call    _open
mov     [esp+20h+var_4], eax
leave
retn
main endp
```

If we take a look in the `open()` function in the `libc.so.6` library, it is only a syscall:

Listing 1.265: `open()` (`libc.so.6`)

```
.text:000BE69B  mov     edx, [esp+4+mode] ; mode
.text:000BE69F  mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3  mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7  mov     eax, 5
.text:000BE6AC  int     80h           ; LINUX - sys_open
```

So, the bit fields for `open()` are apparently checked somewhere in the Linux kernel.

Of course, it is easy to download both Glibc and the Linux kernel source code, but we are interested in understanding the matter without it.

So, as of Linux 2.6, when the `sys_open` syscall is called, control eventually passes to `do_sys_open`, and from there—to the `do_filp_open()` function (it's located in the kernel source tree in `fs/namei.c`).

N.B. Aside from passing arguments via the stack, there is also a method of passing some of them via registers. This is also called fastcall ([6.1.3 on page 734](#)). This works faster since CPU does not need to access the stack in memory to read argument values. GCC has the option `regparm`<sup>143</sup>, through which it's possible to set the number of arguments that can be passed via registers.

The Linux 2.6 kernel is compiled with `-mregparm=3` option [144 145](#).

What this means to us is that the first 3 arguments are to be passed via registers `EAX`, `EDX` and `ECX`, and the rest via the stack. Of course, if the number of arguments is less than 3, only part of registers set is to be used.

So, let's download Linux Kernel 2.6.31, compile it in Ubuntu: `make vmlinux`, open it in [IDA](#), and find the `do_filp_open()` function. At the beginning, we see (the comments are mine):

Listing 1.266: `do_filp_open()` (linux kernel 2.6.31)

```
do_filp_open proc near
...
push    ebp
mov     ebp, esp
push    edi
push    esi
push    ebx
mov     ebx, ecx
add    ebx, 1
```

<sup>143</sup>[ohse.de/uwe/articles/gcc-attributes.html#func-regparm](http://ohse.de/uwe/articles/gcc-attributes.html#func-regparm)

<sup>144</sup>[kernelnewbies.org/Linux\\_2\\_6\\_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f](http://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f)

<sup>145</sup>See also `arch/x86/include/asm/calling.h` file in kernel tree

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
sub    esp, 98h
mov    esi, [ebp+arg_4] ; acc_mode (5th argument)
test   bl, 3
mov    [ebp+var_80], eax ; dfd (1th argument)
mov    [ebp+var_7C], edx ; pathname (2th argument)
mov    [ebp+var_78], ecx ; open_flag (3th argument)
jnz    short loc_C01EF684
mov    ebx, ecx          ; ebx <- open_flag
```

GCC saves the values of the first 3 arguments in the local stack. If that wasn't done, the compiler would not touch these registers, and that would be too tight environment for the compiler's [register allocator](#).

Let's find this fragment of code:

Listing 1.267: do\_filp\_open() (linux kernel 2.6.31)

```
loc_C01EF6B4:      ; CODE XREF: do_filp_open+4F
    test   bl, 40h           ; 0_CREAT
    jnz    loc_C01EF810
    mov    edi, ebx
    shr    edi, 11h
    xor    edi, 1
    and    edi, 1
    test   ebx, 10000h
    jz    short loc_C01EF6D3
    or     edi, 2
```

0x40—is what the 0\_CREAT macro equals to. open\_flag gets checked for the presence of the 0x40 bit, and if this bit is 1, the next JNZ instruction is triggered.

## ARM

The 0\_CREAT bit is checked differently in Linux kernel 3.8.0.

Listing 1.268: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                           const struct open_flags *op)
{
...
    filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                               struct nameidata *nd, const struct open_flags *op, int flags)
{
...
    error = do_last(nd, &path, file, op, &opened, pathname);
...
}

static int do_last(struct nameidata *nd, struct path *path,
                  struct file *file, const struct open_flags *op,
                  int *opened, struct filename *name)
{
...
    if (!(open_flag & O_CREAT)) {
...
        error = lookup_fast(nd, path, &inode);
...
    } else {
...
        error = complete_walk(nd);
    }
...
}
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

Here is how the kernel compiled for ARM mode looks in IDA:

Listing 1.269: do\_last() from vmlinux (IDA)

```
...
.text:C0169EA8      MOV      R9, R3 ; R3 - (4th argument) open_flag
...
.text:C0169ED4      LDR      R6, [R9] ; R6 - open_flag
...
.text:C0169F68      TST      R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C      BNE      loc_C016A128
.text:C0169F70      LDR      R2, [R4,#0x10]
.text:C0169F74      ADD      R12, R4, #8
.text:C0169F78      LDR      R3, [R4,#0xC]
.text:C0169F7C      MOV      R0, R4
.text:C0169F80      STR      R12, [R11,#var_50]
.text:C0169F84      LDRB     R3, [R2,R3]
.text:C0169F88      MOV      R2, R8
.text:C0169F8C      CMP      R3, #0
.text:C0169F90      ORRNE   R1, R1, #3
.text:C0169F94      STRNE   R1, [R4,#0x24]
.text:C0169F98      ANDS    R3, R6, #0x200000
.text:C0169F9C      MOV      R1, R12
.text:C0169FA0      LDRNE   R3, [R4,#0x24]
.text:C0169FA4      ANDNE   R3, R3, #1
.text:C0169FA8      EORNE   R3, R3, #1
.text:C0169FAC      STR      R3, [R11,#var_54]
.text:C0169FB0      SUB     R3, R11, #-var_38
.text:C0169FB4      BL      lookup_fast
...
.text:C016A128 loc_C016A128 ; CODE XREF: do_last.isra.14+DC
.text:C016A128      MOV      R0, R4
.text:C016A12C      BL      complete_walk
...
```

`TST` is analogous to the `TEST` instruction in x86. We can “spot” visually this code fragment by the fact the `lookup_fast()` is to be executed in one case and `complete_walk()` in the other. This corresponds to the source code of the `do_last()` function. The `O_CREAT` macro equals to `0x40` here too.

### 1.22.2 Setting and clearing specific bits

For example:

```
#include <stdio.h>

#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```

**Non-optimizing MSVC**

We get (MSVC 2010):

Listing 1.270: MSVC 2010

```

_rt$ = -4          ; size = 4
_a$ = 8           ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR _rt$[ebp], eax
    mov     ecx, DWORD PTR _rt$[ebp]
    or      ecx, 16384        ; 00004000H
    mov     DWORD PTR _rt$[ebp], ecx
    mov     edx, DWORD PTR _rt$[ebp]
    and    edx, -513          ; fffffdffH
    mov     DWORD PTR _rt$[ebp], edx
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop    ebp
    ret    0
_f ENDP

```

The **OR** instruction sets one bit to value while ignoring the rest.

**AND** resets one bit. It can be said that **AND** just copies all bits except one. Indeed, in the second **AND** operand only the bits that need to be saved are set, just the one do not want to copy is not (which is 0 in the bitmask). It is the easier way to memorize the logic.

**OllyDbg**

Let's try this example in OllyDbg.

First, let's see the binary form of the constants we are going to use:

**0x200** (0b00000000000000000000000000000000) (i.e., the 10th bit (counting from 1st)).

Inverted **0x200** is **0xFFFFDFF** (0b1111111111111111011111111).

**0x4000** (0b00000000000000000000000000000000) (i.e., the 15th bit).

The input value is: **0x12340678** (0b1001000110100000011001111000). We see how it's loaded:

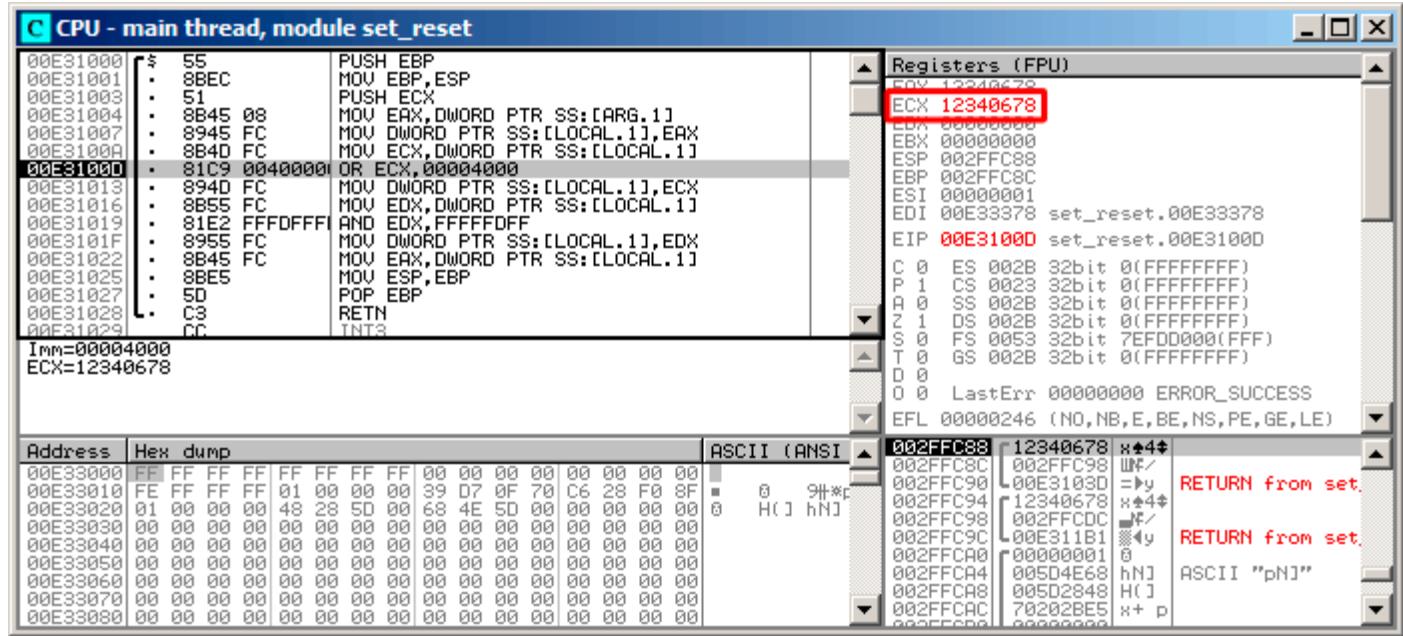


Figure 1.94: OllyDbg: value is loaded into ECX

## 1.22. MANIPULATING SPECIFIC BIT(S)

0R got executed:

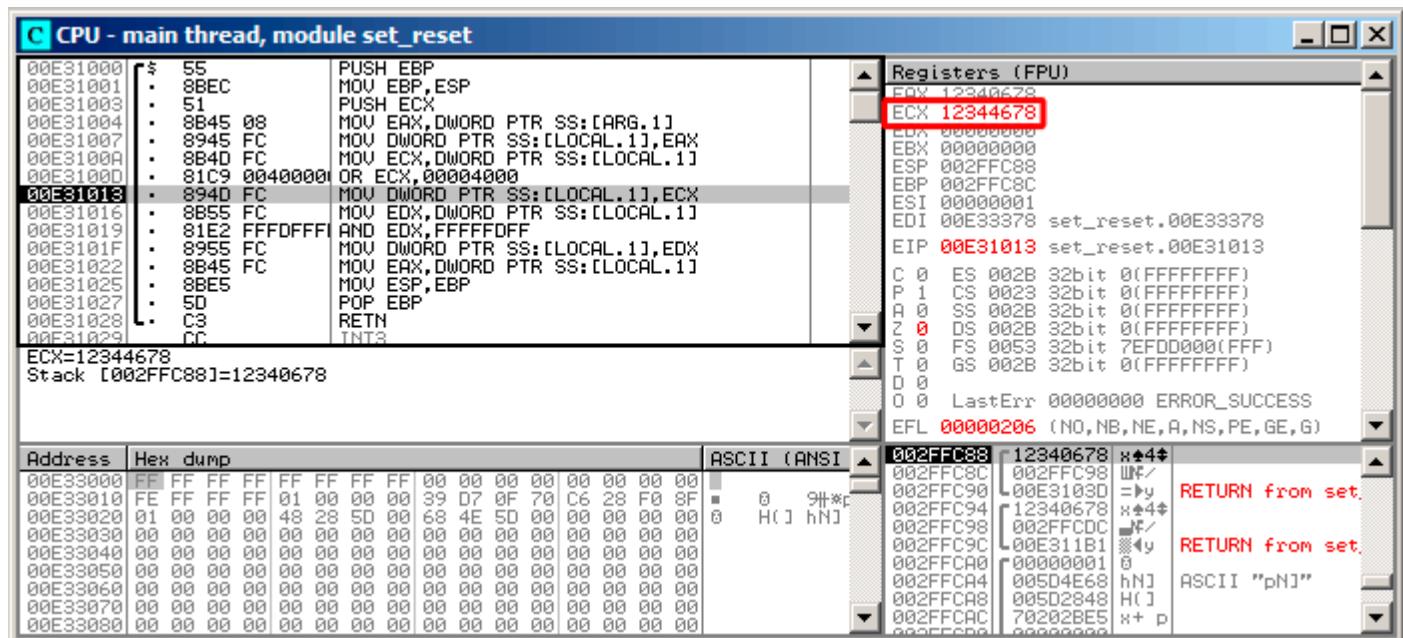


Figure 1.95: OllyDbg: 0R executed

15th bit is set: **0x12344678** (0b10010001101000**1**00011001111000).

## 1.22. MANIPULATING SPECIFIC BIT(S)

The value is reloaded again (because the compiler is not in optimizing mode):

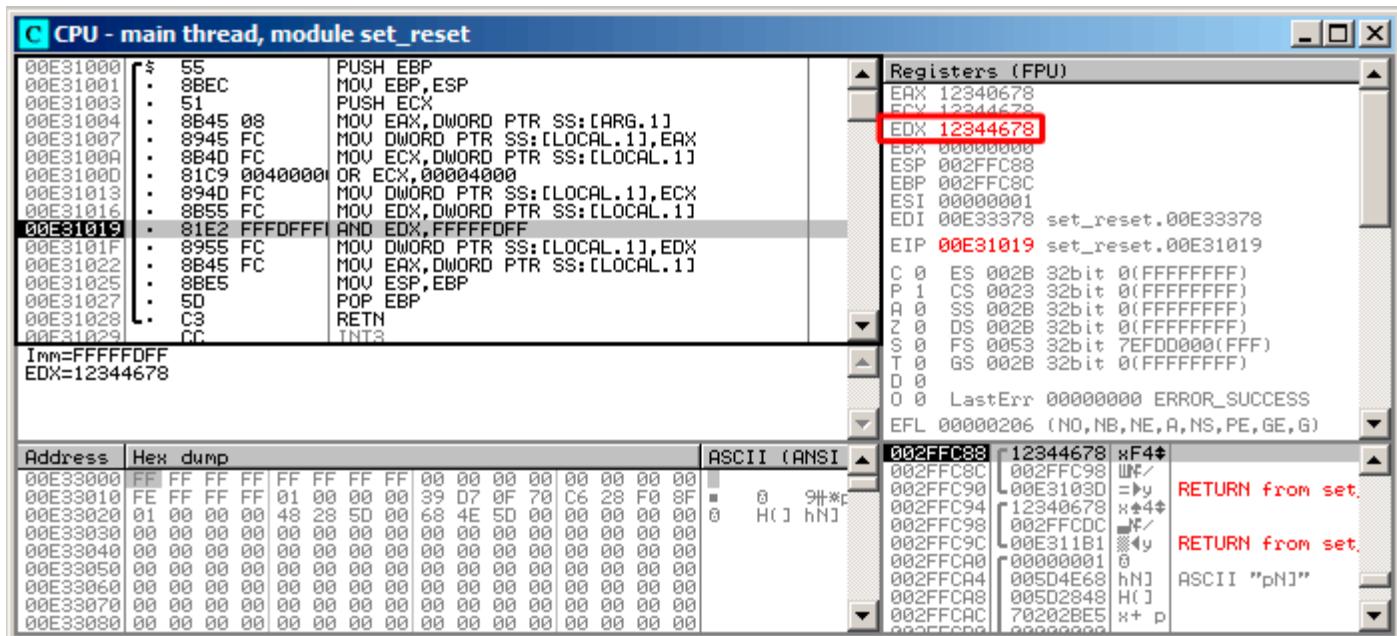


Figure 1.96: OllyDbg: value has been reloaded into EDX

## 1.22. MANIPULATING SPECIFIC BIT(S)

AND got executed:

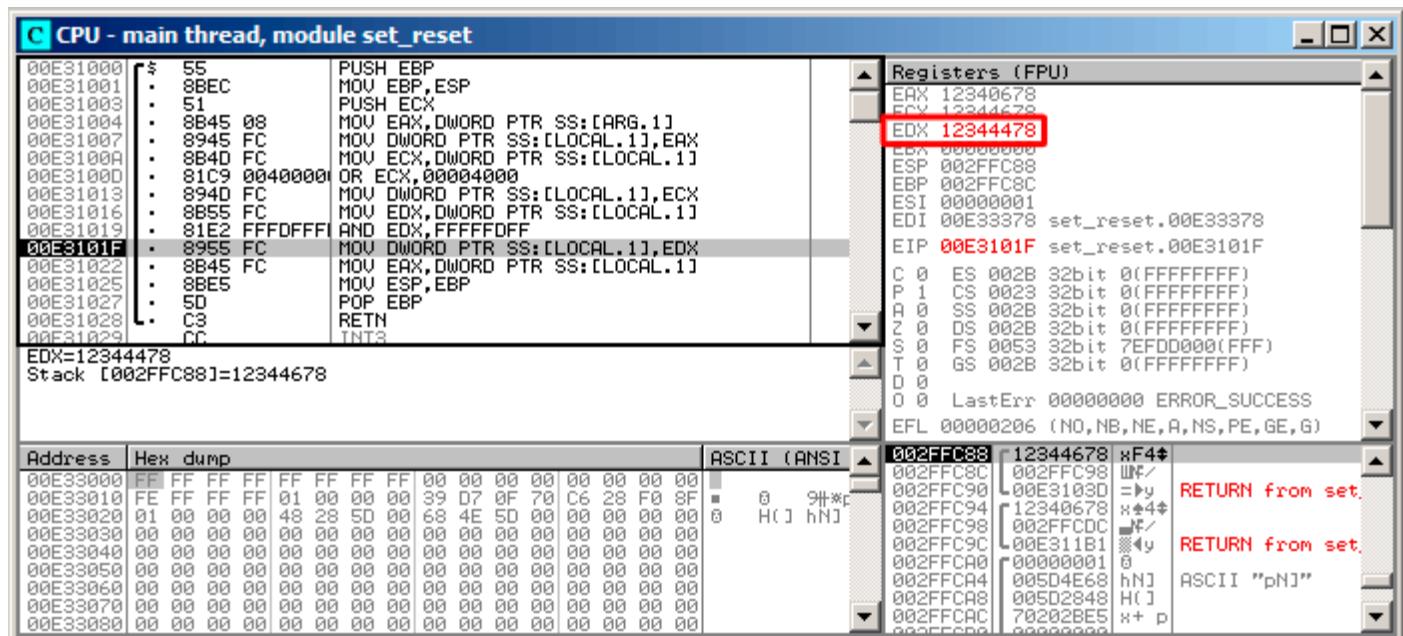


Figure 1.97: OllyDbg: AND executed

The 10th bit has been cleared (or, in other words, all bits were left except the 10th) and the final value now is

**0x12344478** (0b10010001101000100010001111000).

## Optimizing MSVC

If we compile it in MSVC with optimization turned on (**/Ox**), the code is even shorter:

Listing 1.271: Optimizing MSVC

```
_a$ = 8           ; size = 4
_f PROC
    mov    eax, DWORD PTR _a$[esp-4]
    and    eax, -513          ; fffffdffH
    or     eax, 16384         ; 00004000H
    ret    0
_f ENDP
```

## Non-optimizing GCC

Let's try GCC 4.4.1 without optimization:

Listing 1.272: Non-optimizing GCC

```
public f
proc near

var_4      = dword ptr -4
arg_0      = dword ptr  8

push    ebp
mov     ebp, esp
sub    esp, 10h
mov    eax, [ebp+arg_0]
mov    [ebp+var_4], eax
or     [ebp+var_4], 4000h
and    [ebp+var_4], 0FFFFFDFFh
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
f          mov      eax, [ebp+var_4]
           leave
           retn
           endp
```

There is a redundant code present, however, it is shorter than the MSVC version without optimization.

Now let's try GCC with optimization turned on `-O3`:

### Optimizing GCC

Listing 1.273: Optimizing GCC

```
f          public f
           proc near
arg_0       = dword ptr 8
           push    ebp
           mov     ebp, esp
           mov     eax, [ebp+arg_0]
           pop    ebp
           or      ah, 40h
           and    ah, 0FDh
           retn
           endp
```

That's shorter. It is worth noting the compiler works with the `EAX` register part via the `AH` register—that is the `EAX` register part from the 8th to the 15th bits included.

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RAX <sup>x64</sup>							
EAX							
AX							
AH AL							

N.B. The 16-bit CPU 8086 accumulator was named `AX` and consisted of two 8-bit halves—`AL` (lower byte) and `AH` (higher byte). In 80386 almost all registers were extended to 32-bit, the accumulator was named `EAX`, but for the sake of compatibility, its *older parts* may be still accessed as `AX / AH / AL`.

Since all x86 CPUs are successors of the 16-bit 8086 CPU, these *older* 16-bit opcodes are shorter than the newer 32-bit ones. That's why the `or ah, 40h` instruction occupies only 3 bytes. It would be more logical way to emit here `or eax, 04000h` but that is 5 bytes, or even 6 (in case the register in the first operand is not `EAX`).

### Optimizing GCC and regparm

It would be even shorter if to turn on the `-O3` optimization flag and also set `regparm=3`.

Listing 1.274: Optimizing GCC

```
f          public f
           proc near
           push    ebp
           or      ah, 40h
           mov     ebp, esp
           and    ah, 0FDh
           pop    ebp
           retn
           endp
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

Indeed, the first argument is already loaded in `EAX`, so it is possible to work with it in-place. It is worth noting that both the function prologue (`push ebp / mov ebp,esp`) and epilogue (`pop ebp`) can easily be omitted here, but GCC probably is not good enough to do such code size optimizations. However, such short functions are better to be *inlined functions* ( [3.11 on page 506](#) ).

### ARM + Optimizing Keil 6/2013 (ARM mode)

Listing 1.275: Optimizing Keil 6/2013 (ARM mode)

02 0C C0 E3	BIC	R0, R0, #0x200
01 09 80 E3	ORR	R0, R0, #0x4000
1E FF 2F E1	BX	LR

`BIC` (*Bitwise bit Clear*) is an instruction for clearing specific bits. This is just like the `AND` instruction, but with inverted operand. I.e., it's analogous to a `NOT` + `AND` instruction pair.

`ORR` is “logical or”, analogous to `OR` in x86.

So far it's easy.

### ARM + Optimizing Keil 6/2013 (Thumb mode)

Listing 1.276: Optimizing Keil 6/2013 (Thumb mode)

01 21 89 03	MOVS	R1, 0x4000
08 43	ORRS	R0, R1
49 11	ASRS	R1, R1, #5 ; generate 0x200 and place to R1
88 43	BICS	R0, R1
70 47	BX	LR

Seems like Keil decided that the code in Thumb mode, making `0x200` from `0x4000`, is more compact than the code for writing `0x200` to an arbitrary register.

So that is why, with the help of `ASRS` (arithmetic shift right), this value is calculated as `0x4000 >> 5`.

### ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 1.277: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

42 0C C0 E3	BIC	R0, R0, #0x4200
01 09 80 E3	ORR	R0, R0, #0x4000
1E FF 2F E1	BX	LR

The code that was generated by LLVM, in source code form could be something like this:

```
REMOVE_BIT(rt, 0x4200);
SET_BIT(rt, 0x4000);
```

And it does exactly what we need. But why `0x4200`? Perhaps that an artifact from LLVM's optimizer <sup>146</sup>.

Probably a compiler's optimizer error, but the generated code works correctly anyway.

You can read more about compiler anomalies here ( [10.4 on page 989](#) ).

Optimizing Xcode 4.6.3 (LLVM) for Thumb mode generates the same code.

<sup>146</sup>It was LLVM build 2410.2.00 bundled with Apple Xcode 4.6.3

**ARM: more about the BIC instruction**

Let's rework the example slightly:

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

Then the optimizing Keil 5.03 in ARM mode does:

```
f PROC
    BIC      r0, r0, #0x1000
    BIC      r0, r0, #0x234
    BX       lr
ENDP
```

There are two **BIC** instructions, i.e., bits **0x1234** are cleared in two passes.

This is because it's not possible to encode **0x1234** in a **BIC** instruction, but it's possible to encode **0x1000** and **0x234**.

**ARM64: Optimizing GCC (Linaro) 4.9**

Optimizing GCC compiling for ARM64 can use the **AND** instruction instead of **BIC**:

Listing 1.278: Optimizing GCC (Linaro) 4.9

```
f:
    and      w0, w0, -513      ; 0xFFFFFFFFFFFFFFDFF
    orr      w0, w0, 16384     ; 0x4000
    ret
```

**ARM64: Non-optimizing GCC (Linaro) 4.9**

Non-optimizing GCC generates more redundant code, but works just like optimized:

Listing 1.279: Non-optimizing GCC (Linaro) 4.9

```
f:
    sub      sp, sp, #32
    str     w0, [sp,12]
    ldr     w0, [sp,12]
    str     w0, [sp,28]
    ldr     w0, [sp,28]
    orr     w0, w0, 16384    ; 0x4000
    str     w0, [sp,28]
    ldr     w0, [sp,28]
    and     w0, w0, -513     ; 0xFFFFFFFFFFFFFFDFF
    str     w0, [sp,28]
    ldr     w0, [sp,28]
    add     sp, sp, 32
    ret
```

**MIPS**

Listing 1.280: Optimizing GCC 4.4.5 (IDA)

```
f:
; $a0=a
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
ori    $a0, 0x4000
; $a0=a|0x4000
li     $v0, 0xFFFFFDFF
jr     $ra
and    $v0, $a0, $v0
; at finish: $v0 = $a0&$v0 = a|0x4000 & 0xFFFFFDFF
```

**ORI** is, of course, the OR operation. “I” in the instruction name means that the value is embedded in the machine code.

But after that we have **AND**. There is no way to use **ANDI** because it’s not possible to embed the 0xFFFFFDFF number in a single instruction, so the compiler has to load 0xFFFFFDFF into register \$V0 first and then generates **AND** which takes all its values from registers.

### 1.22.3 Shifts

Bit shifts in C/C++ are implemented using `<<` and `>>` operators. The x86 ISA has the SHL (Shift Left) and SHR (Shift Right) instructions for this. Shift instructions are often used in division and multiplications by powers of two:  $2^n$  (e.g., 1, 2, 4, 8, etc.): [1.18.1 on page 213](#), [1.18.2 on page 217](#).

Shifting operations are also so important because they are often used for specific bit isolation or for constructing a value of several scattered bits.

### 1.22.4 Setting and clearing specific bits: FPU example

Here is how bits are located in the *float* type in IEEE 754 form:



The sign of number is in the **MSB<sup>147</sup>**. Will it be possible to change the sign of a floating point number without any FPU instructions?

```
#include <stdio.h>

float my_abs (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) & 0x7FFFFFFF;
    return *(float*)&tmp;
};

float set_sign (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) | 0x80000000;
    return *(float*)&tmp;
};

float negate (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) ^ 0x80000000;
    return *(float*)&tmp;
};

int main()
{
    printf ("my_abs():\n");
    printf ("%f\n", my_abs (123.456));
    printf ("%f\n", my_abs (-456.123));
    printf ("set_sign():\n");
    printf ("%f\n", set_sign (123.456));
    printf ("%f\n", set_sign (-456.123));
    printf ("negate()):\n");
}
```

<sup>147</sup>Most significant bit

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
    printf ("%f\n", negate (123.456));
    printf ("%f\n", negate (-456.123));
};
```

We need this trickery in C/C++ to copy to/from *float* value without actual conversion. So there are three functions: `my_abs()` resets **MSB**; `set_sign()` sets **MSB** and `negate()` flips it.

**XOR** can be used to flip a bit: [2.5 on page 461](#).

### x86

The code is pretty straightforward:

Listing 1.281: Optimizing MSVC 2012

```
_tmp$ = 8
_i$ = 8
_my_abs PROC
    and    DWORD PTR _i$[esp-4], 2147483647 ; 7fffffffH
    fld    DWORD PTR _tmp$[esp-4]
    ret    0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
    or     DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
    fld    DWORD PTR _tmp$[esp-4]
    ret    0
_set_sign ENDP

_tmp$ = 8
_i$ = 8
_negate PROC
    xor   DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
    fld    DWORD PTR _tmp$[esp-4]
    ret    0
_negate ENDP
```

An input value of type *float* is taken from the stack, but treated as an integer value.

**AND** and **OR** reset and set the desired bit. **XOR** flips it.

Finally, the modified value is loaded into **ST0**, because floating-point numbers are returned in this register.

Now let's try optimizing MSVC 2012 for x64:

Listing 1.282: Optimizing MSVC 2012 x64

```
tmp$ = 8
i$ = 8
my_abs PROC
    movss  DWORD PTR [rsp+8], xmm0
    mov    eax, DWORD PTR i$[rsp]
    btr    eax, 31
    mov    DWORD PTR tmp$[rsp], eax
    movss  xmm0, DWORD PTR tmp$[rsp]
    ret    0
my_abs ENDP
_TEXT ENDS

tmp$ = 8
i$ = 8
set_sign PROC
    movss  DWORD PTR [rsp+8], xmm0
    mov    eax, DWORD PTR i$[rsp]
    bts    eax, 31
    mov    DWORD PTR tmp$[rsp], eax
    movss  xmm0, DWORD PTR tmp$[rsp]
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
        ret      0
set_sign ENDP

tmp$ = 8
i$ = 8
negate PROC
    movss  DWORD PTR [rsp+8], xmm0
    mov    eax, DWORD PTR i$[rsp]
    btc   eax, 31
    mov    DWORD PTR tmp$[rsp], eax
    movss  xmm0, DWORD PTR tmp$[rsp]
    ret      0
negate ENDP
```

The input value is passed in `XMM0`, then it is copied into the local stack and then we see some instructions that are new to us: `BTR`, `BTS`, `BTC`.

These instructions are used for resetting (`BTR`), setting (`BTS`) and inverting (or complementing): `BTC`) specific bits. The 31st bit is `MSB`, counting from 0.

Finally, the result is copied into `XMM0`, because floating point values are returned through `XMM0` in Win64 environment.

## MIPS

GCC 4.4.5 for MIPS does mostly the same:

Listing 1.283: Optimizing GCC 4.4.5 (IDA)

```
my_abs:
; move from coprocessor 1:
        mfcl    $v1, $f12
        li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; do AND:
        and     $v0, $v1
; move to coprocessor 1:
        mtcl    $v0, $f0
; return
        jr      $ra
        or      $at, $zero ; branch delay slot

set_sign:
; move from coprocessor 1:
        mfcl    $v0, $f12
        lui     $v1, 0x8000
; $v1=0x80000000
; do OR:
        or      $v0, $v1, $v0
; move to coprocessor 1:
        mtcl    $v0, $f0
; return
        jr      $ra
        or      $at, $zero ; branch delay slot

negate:
; move from coprocessor 1:
        mfcl    $v0, $f12
        lui     $v1, 0x8000
; $v1=0x80000000
; do XOR:
        xor     $v0, $v1, $v0
; move to coprocessor 1:
        mtcl    $v0, $f0
; return
        jr      $ra
        or      $at, $zero ; branch delay slot
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

One single `LUI` instruction is used to load `0x80000000` into a register, because `LUI` is clearing the low 16 bits and these are zeros in the constant, so one `LUI` without subsequent `ORI` is enough.

### ARM

#### Optimizing Keil 6/2013 (ARM mode)

Listing 1.284: Optimizing Keil 6/2013 (ARM mode)

```
my_abs PROC
; clear bit:
    BIC      r0,r0,#0x80000000
    BX       lr
    ENDP

set_sign PROC
; do OR:
    ORR      r0,r0,#0x80000000
    BX       lr
    ENDP

negate PROC
; do XOR:
    EOR      r0,r0,#0x80000000
    BX       lr
    ENDP
```

So far so good.

ARM has the `BIC` instruction, which explicitly clears specific bit(s). `EOR` is the ARM instruction name for `XOR` ("Exclusive OR").

#### Optimizing Keil 6/2013 (Thumb mode)

Listing 1.285: Optimizing Keil 6/2013 (Thumb mode)

```
my_abs PROC
    LSLS      r0,r0,#1
; r0=i<<1
    LSRS      r0,r0,#1
; r0=(i<<1)>>1
    BX       lr
    ENDP

set_sign PROC
    MOVS      r1,#1
; r1=1
    LSLS      r1,r1,#31
; r1=1<<31=0x80000000
    ORRS      r0,r0,r1
; r0=r0 | 0x80000000
    BX       lr
    ENDP

negate PROC
    MOVS      r1,#1
; r1=1
    LSLS      r1,r1,#31
; r1=1<<31=0x80000000
    EORS      r0,r0,r1
; r0=r0 ^ 0x80000000
    BX       lr
    ENDP
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

Thumb mode in ARM offers 16-bit instructions and not much data can be encoded in them, so here a `MOVS / LSLS` instruction pair is used for forming the `0x80000000` constant. It works like this:  $1 \ll 31 = 0x80000000$ .

The code of `my_abs` is weird and it effectively works like this expression:  $(i \ll 1) \gg 1$ . This statement looks meaningless. But nevertheless, when `input << 1` is executed, the **MSB** (sign bit) is just dropped. When the subsequent `result >> 1` statement is executed, all bits are now in their own places, but **MSB** is zero, because all “new” bits appearing from the shift operations are always zeros. That is how the `LSLS / LSRS` instruction pair clears **MSB**.

### Optimizing GCC 4.6.3 (Raspberry Pi, ARM mode)

Listing 1.286: Optimizing GCC 4.6.3 for Raspberry Pi (ARM mode)

```
my_abs
; copy from S0 to R2:
    FMRS    R2, S0
; clear bit:
    BIC     R3, R2, #0x80000000
; copy from R3 to S0:
    FMSR    S0, R3
    BX      LR

set_sign
; copy from S0 to R2:
    FMRS    R2, S0
; do OR:
    ORR     R3, R2, #0x80000000
; copy from R3 to S0:
    FMSR    S0, R3
    BX      LR

negate
; copy from S0 to R2:
    FMRS    R2, S0
; do ADD:
    ADD     R3, R2, #0x80000000
; copy from R3 to S0:
    FMSR    S0, R3
    BX      LR
```

Let's run Raspberry Pi Linux in QEMU and it emulates an ARM FPU, so S-registers are used here for floating point numbers instead of R-registers.

The `FMRS` instruction copies data from **GPR** to the FPU and back.

`my_abs()` and `set_sign()` looks as expected, but `negate()`? Why is there `ADD` instead of `XOR`?

It's hard to believe, but the instruction `ADD register, 0x80000000` works just like `XOR register, 0x80000000`. First of all, what's our goal? The goal is to flip the **MSB**, so let's forget about the `XOR` operation. From school-level mathematics we may recall that adding values like `1000` to other values never affects the last 3 digits. For example:  $1234567 + 10000 = 1244567$  (last 4 digits are never affected).

But here we operate in binary base and `0x80000000` is `0b10000000000000000000000000000000`, i.e., only the highest bit is set.

Adding `0x80000000` to any value never affects the lowest 31 bits, but affects only the **MSB**. Adding `1` to `0` is resulting in `1`.

Adding `1` to `1` is resulting in `0b10` in binary form, but the 32th bit (counting from zero) gets dropped, because our registers are 32 bit wide, so the result is `0`. That's why `XOR` can be replaced by `ADD` here.

It's hard to say why GCC decided to do this, but it works correctly.

## 1.22.5 Counting bits set to 1

Here is a simple example of a function that calculates the number of bits set in the input value. This operation is also called “population count”<sup>148</sup>.

```
#include <stdio.h>

#define IS_SET(flag, bit) ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};

int main()
{
    f(0x12345678); // test
};
```

In this loop, the iteration count value  $i$  is counting from 0 to 31, so the  $1 \ll i$  statement is counting from 1 to  $0x80000000$ . Describing this operation in natural language, we would say *shift 1 by n bits left*. In other words,  $1 \ll i$  statement consequently produces all possible bit positions in a 32-bit number. The freed bit at right is always cleared.

Here is a table of all possible  $1 \ll i$  for  $i = 0 \dots 31$ :

C/C++ expression	Power of two	Decimal form	Hexadecimal form
$1 \ll 0$	$1$	1	1
$1 \ll 1$	$2^1$	2	2
$1 \ll 2$	$2^2$	4	4
$1 \ll 3$	$2^3$	8	8
$1 \ll 4$	$2^4$	16	0x10
$1 \ll 5$	$2^5$	32	0x20
$1 \ll 6$	$2^6$	64	0x40
$1 \ll 7$	$2^7$	128	0x80
$1 \ll 8$	$2^8$	256	0x100
$1 \ll 9$	$2^9$	512	0x200
$1 \ll 10$	$2^{10}$	1024	0x400
$1 \ll 11$	$2^{11}$	2048	0x800
$1 \ll 12$	$2^{12}$	4096	0x1000
$1 \ll 13$	$2^{13}$	8192	0x2000
$1 \ll 14$	$2^{14}$	16384	0x4000
$1 \ll 15$	$2^{15}$	32768	0x8000
$1 \ll 16$	$2^{16}$	65536	0x10000
$1 \ll 17$	$2^{17}$	131072	0x20000
$1 \ll 18$	$2^{18}$	262144	0x40000
$1 \ll 19$	$2^{19}$	524288	0x80000
$1 \ll 20$	$2^{20}$	1048576	0x100000
$1 \ll 21$	$2^{21}$	2097152	0x200000
$1 \ll 22$	$2^{22}$	4194304	0x400000
$1 \ll 23$	$2^{23}$	8388608	0x800000
$1 \ll 24$	$2^{24}$	16777216	0x1000000
$1 \ll 25$	$2^{25}$	33554432	0x2000000
$1 \ll 26$	$2^{26}$	67108864	0x4000000
$1 \ll 27$	$2^{27}$	134217728	0x8000000
$1 \ll 28$	$2^{28}$	268435456	0x10000000
$1 \ll 29$	$2^{29}$	536870912	0x20000000
$1 \ll 30$	$2^{30}$	1073741824	0x40000000
$1 \ll 31$	$2^{31}$	2147483648	0x80000000

<sup>148</sup>modern x86 CPUs (supporting SSE4) even have a POPCNT instruction for it

## 1.22. MANIPULATING SPECIFIC BIT(S)

These constant numbers (bit masks) very often appear in code and a practicing reverse engineer must be able to spot them quickly.

Decimal numbers before 63356 and hexadeciml ones are very easy to memorize. While decimal numbers after 65536, probably, not worth memorizing.

These constants are very often used for mapping flags to specific bits. For example, here is excerpt from `ssl_private.h` from Apache 2.4.6 source code:

```
/**  
 * Define the SSL options  
 */  
#define SSL_OPT_NONE          (0)  
#define SSL_OPT_RELSET        (1<<0)  
#define SSL_OPT_STDENVARS     (1<<1)  
#define SSL_OPT_EXPORTCERTDATA (1<<3)  
#define SSL_OPT_FAKEBASICAUTH (1<<4)  
#define SSL_OPT_STRICTREQUIRE (1<<5)  
#define SSL_OPT_OPTRENEGOTIATE (1<<6)  
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Let's get back to our example.

The `IS_SET` macro checks bit presence in `a`.

The `IS_SET` macro is in fact the logical AND operation (*AND*) and it returns 0 if the specific bit is absent there, or the bit mask, if the bit is present. The `if()` operator in C/C++ triggers if the expression in it is not zero, it might be even 123456, that is why it always works correctly.

### x86

### MSVC

Let's compile (MSVC 2010):

Listing 1.287: MSVC 2010

```
_rt$ = -8           ; size = 4  
_i$ = -4           ; size = 4  
_a$ = 8            ; size = 4  
_f  PROC  
    push  ebp  
    mov   ebp, esp  
    sub   esp, 8  
    mov   DWORD PTR _rt$[ebp], 0  
    mov   DWORD PTR _i$[ebp], 0  
    jmp   SHORT $LN4@f  
$LN3@f:  
    mov   eax, DWORD PTR _i$[ebp] ; increment of i  
    add   eax, 1  
    mov   DWORD PTR _i$[ebp], eax  
$LN4@f:  
    cmp   DWORD PTR _i$[ebp], 32  ; 00000020H  
    jge   SHORT $LN2@f          ; loop finished?  
    mov   edx, 1  
    mov   ecx, DWORD PTR _i$[ebp]  
    shl   edx, cl              ; EDX=EDX<<CL  
    and   edx, DWORD PTR _a$[ebp]  
    je    SHORT $LN1@f          ; result of AND instruction was 0?  
                                ; then skip next instructions  
    mov   eax, DWORD PTR _rt$[ebp]; no, not zero  
    add   eax, 1                ; increment rt  
    mov   DWORD PTR _rt$[ebp], eax  
$LN1@f:  
    jmp   SHORT $LN3@f  
$LN2@f:  
    mov   eax, DWORD PTR _rt$[ebp]  
    mov   esp, ebp
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
pop    ebp  
ret    0  
_f    ENDP
```

# **OllyDbg**

Let's load this example into OllyDbg. Let the input value be `0x12345678`.

For  $i = 1$ , we see how  $i$  is loaded into ECX :

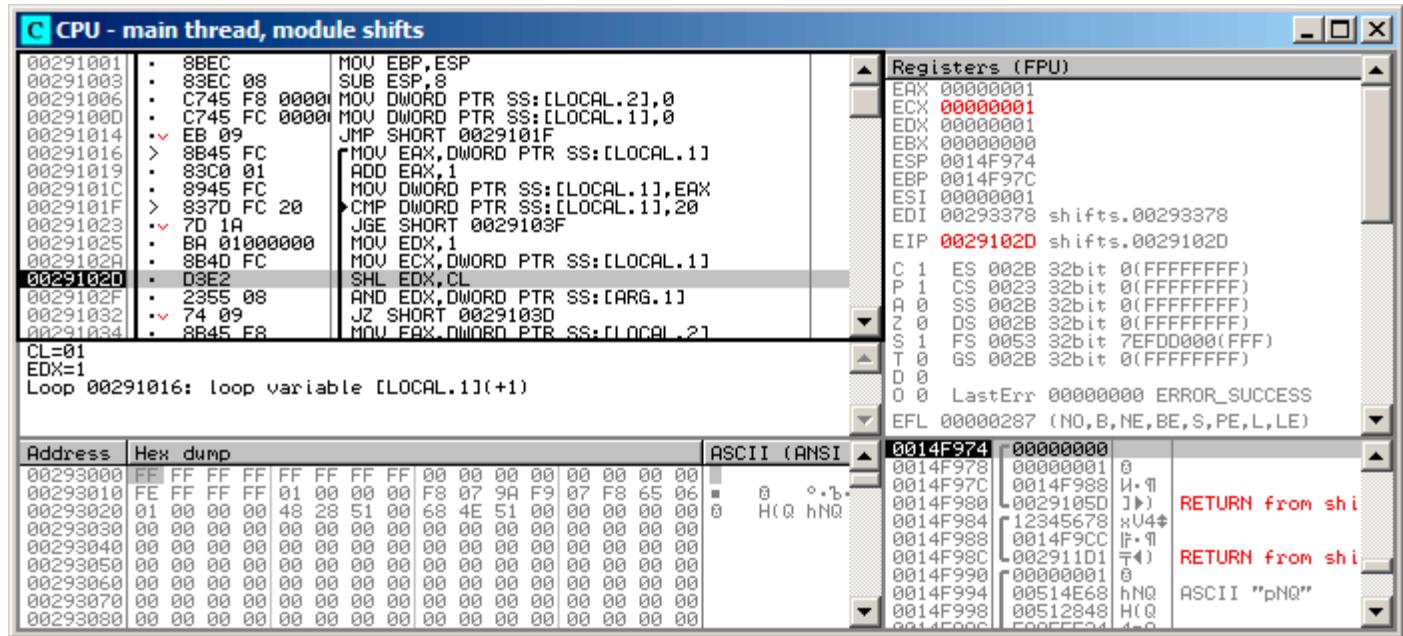


Figure 1.98: OllyDbg:  $i = 1$ ,  $i$  is loaded into ECX

**EDX** is 1. **SHL** is to be executed now.

## 1.22. MANIPULATING SPECIFIC BIT(S)

SHL has been executed:

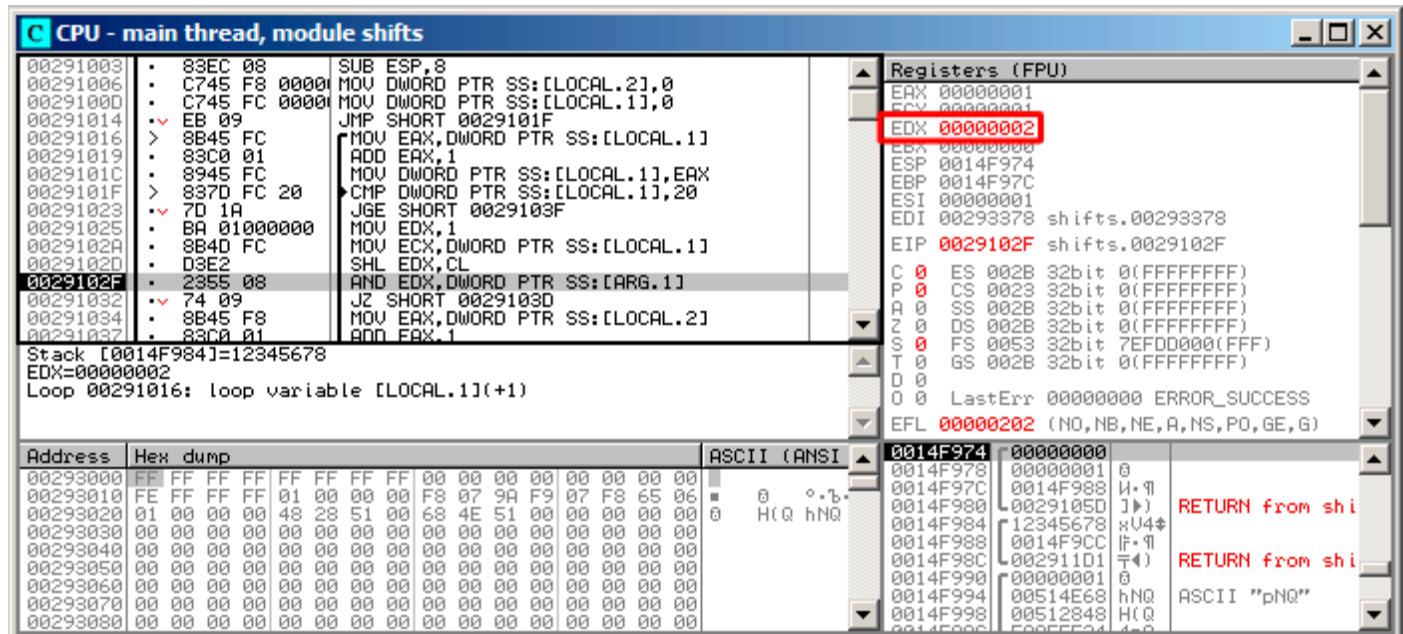


Figure 1.99: OllyDbg:  $i = 1$ ,  $EDX = 1 \ll 1 = 2$

EDX contain  $1 \ll 1$  (or 2). This is a bit mask.

## 1.22. MANIPULATING SPECIFIC BIT(S)

AND sets ZF to 1, which implies that the input value ( 0x12345678 ) ANDed with 2 results in 0:

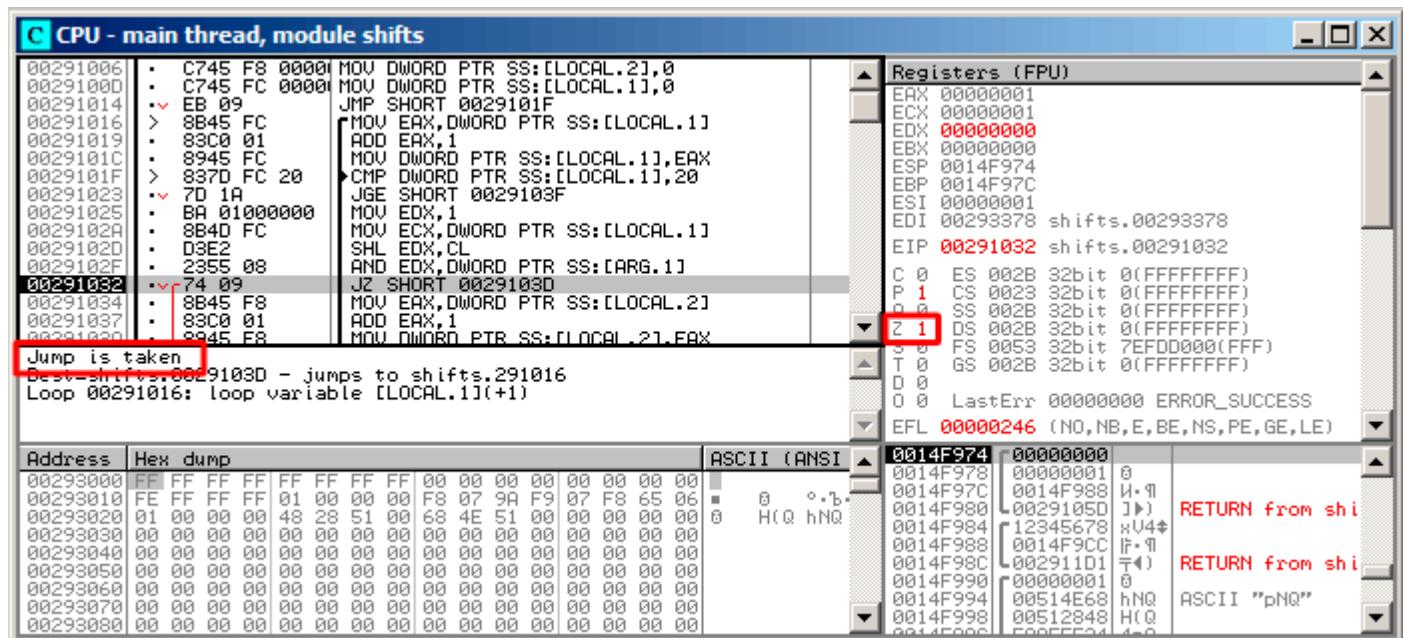


Figure 1.100: OllyDbg:  $i = 1$ , is there that bit in the input value? No. ( $ZF = 1$ )

So, there is no corresponding bit in the input value.

The piece of code, which increments the counter is not to be executed: the JZ instruction bypassing it.

## 1.22. MANIPULATING SPECIFIC BIT(S)

Let's trace a bit further and  $i$  is now 4. `SHL` is to be executed now:

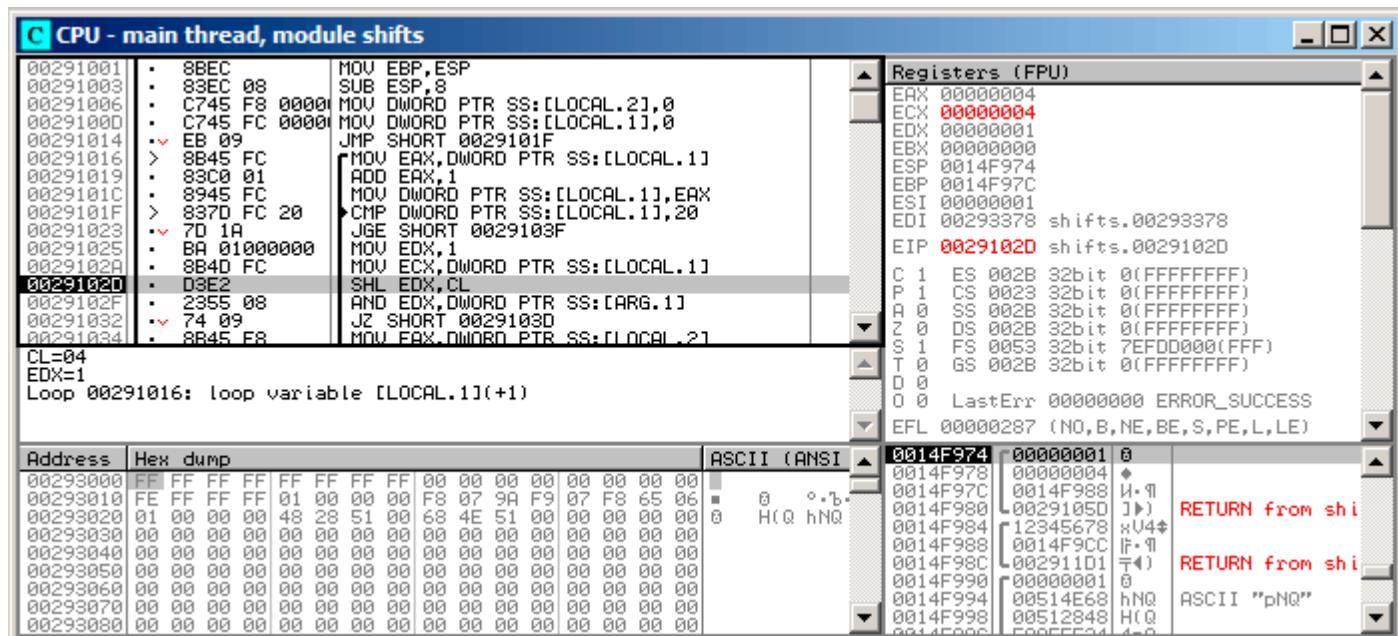


Figure 1.101: OllyDbg:  $i = 4$ ,  $i$  is loaded into ECX

## 1.22. MANIPULATING SPECIFIC BIT(S)

`EDX = 1 << 4 (or 0x10 or 16):`

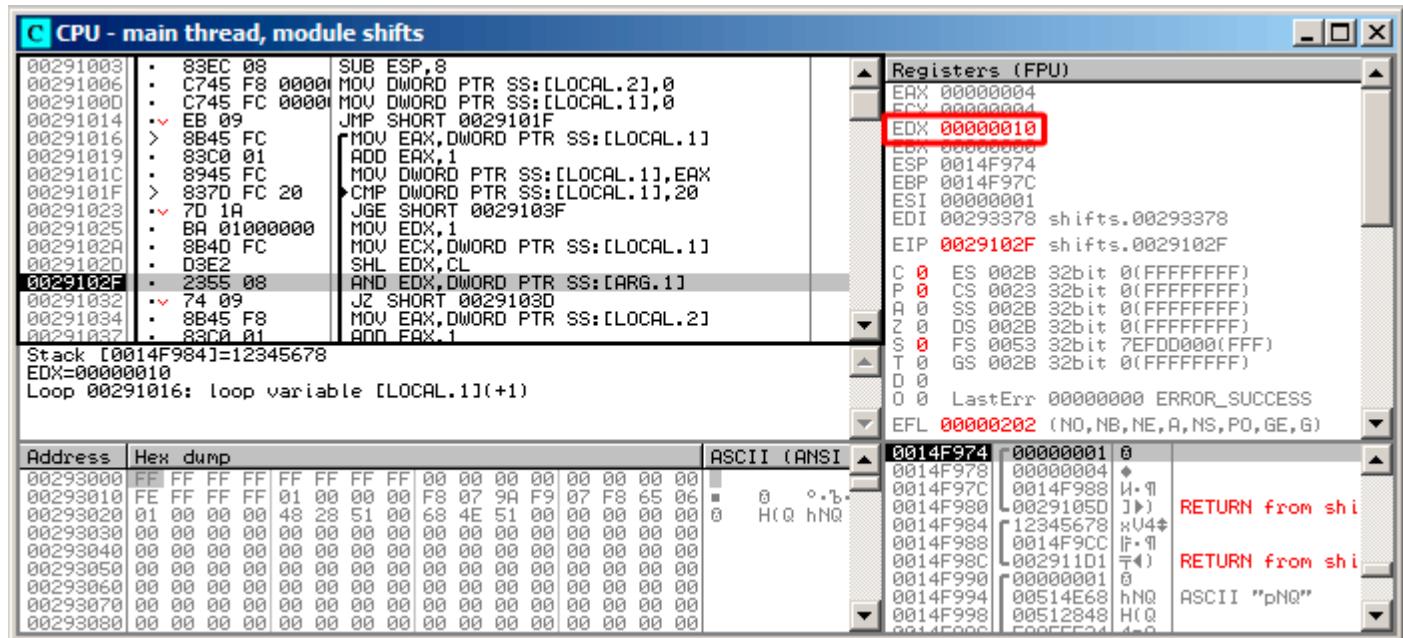


Figure 1.102: OllyDbg:  $i = 4$ ,  $\text{EDX} = 1 \ll 4 = 0x10$

This is another bit mask.

## 1.22. MANIPULATING SPECIFIC BIT(S)

AND is executed:

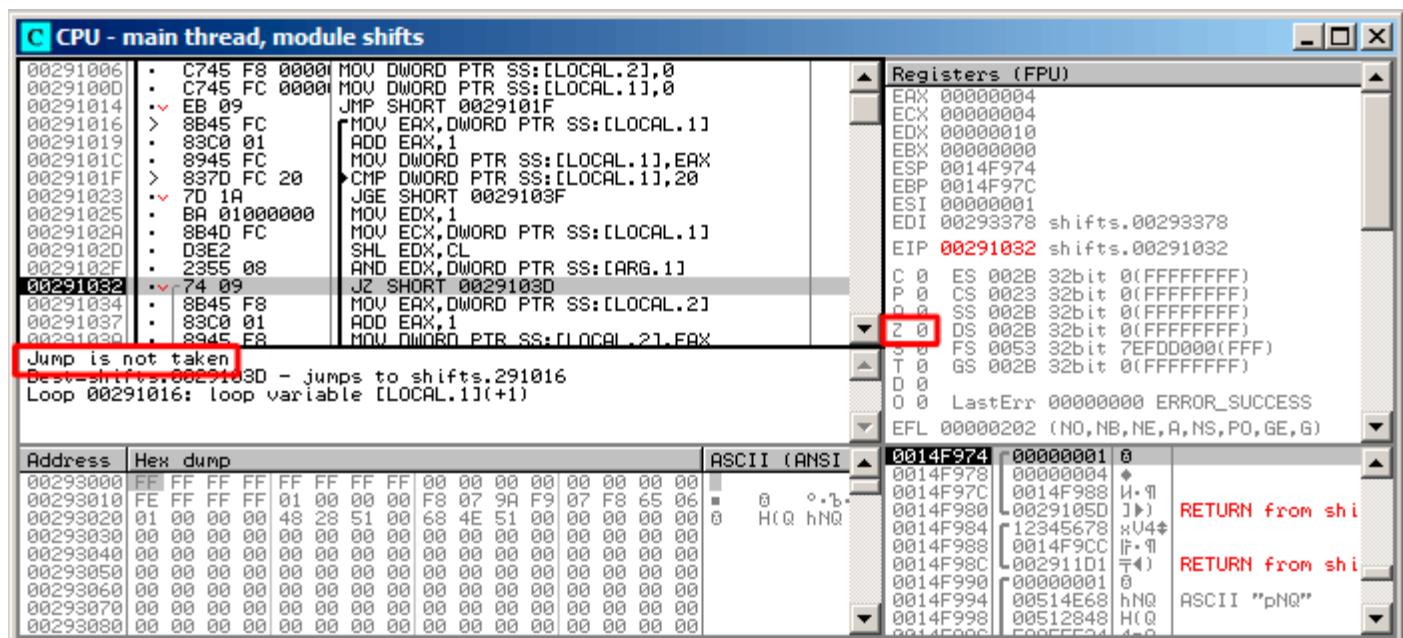


Figure 1.103: OllyDbg:  $i = 4$ , is there that bit in the input value? Yes. ( $ZF = 0$ )

$ZF$  is 0 because this bit is present in the input value.

Indeed,  $0x12345678 \& 0x10 = 0x10$ .

This bit counts: the jump is not triggering and the bit counter [incrementing](#).

The function returns 13. This is total number of bits set in  $0x12345678$ .

## GCC

Let's compile it in GCC 4.4.1:

Listing 1.288: GCC 4.4.1

```

public f
f
proc near

rt          = dword ptr -0Ch
i          = dword ptr -8
arg_0      = dword ptr 8

push    ebp
mov     ebp, esp
push    ebx
sub    esp, 10h
mov     [ebp+rt], 0
mov     [ebp+i], 0
jmp    short loc_80483EF

loc_80483D0:
        mov     eax, [ebp+i]
        mov     edx, 1
        mov     ebx, edx
        mov     ecx, eax
        shl    ebx, cl
        mov     eax, ebx
        and    eax, [ebp+arg_0]
        test   eax, eax
        jz    short loc_80483EB
        add    [ebp+rt], 1

loc_80483EB:
        add    [ebp+i], 1

```

## 1.22. MANIPULATING SPECIFIC BIT(S)

loc\_80483EF:

```
        cmp    [ebp+i], 1Fh
        jle    short loc_80483D0
        mov    eax, [ebp+rt]
        add    esp, 10h
        pop    ebx
        pop    ebp
        retn
f      endp
```

### x64

Let's modify the example slightly to extend it to 64-bit:

```
#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit) ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
}
```

### Non-optimizing GCC 4.8.2

So far so easy.

Listing 1.289: Non-optimizing GCC 4.8.2

```
f:
    push   rbp
    mov    rbp, rsp
    mov    QWORD PTR [rbp-24], rdi ; a
    mov    DWORD PTR [rbp-12], 0    ; rt=0
    mov    QWORD PTR [rbp-8], 0     ; i=0
    jmp    .L2
.L4:
    mov    rax, QWORD PTR [rbp-8]
    mov    rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
    mov    ecx, eax
; ECX = i
    shr    rdx, cl
; RDX = RDX>>CL = a>>i
    mov    rax, rdx
; RAX = RDX = a>>i
    and    eax, 1
; EAX = EAX&1 = (a>>i)&1
    test   rax, rax
; the last bit is zero?
; skip the next ADD instruction, if it was so.
    je     .L3
    add    DWORD PTR [rbp-12], 1    ; rt++
.L3:
    add    QWORD PTR [rbp-8], 1     ; i++
.L2:
    cmp    QWORD PTR [rbp-8], 63    ; i<63?
    jbe    .L4                      ; jump to the loop body begin, if so
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
1 mov     eax, DWORD PTR [rbp-12] ; return rt
2 pop     rbp
3 ret
```

### Optimizing GCC 4.8.2

Listing 1.290: Optimizing GCC 4.8.2

```
1 f:
2     xor    eax, eax      ; rt variable will be in EAX register
3     xor    ecx, ecx      ; i variable will be in ECX register
4 .L3:
5     mov    rsi, rdi       ; load input value
6     lea    edx, [rax+1]   ; EDX=EAX+1
7 ; EDX here is a new version of rt,
8 ; which will be written into rt variable, if the last bit is 1
9     shr    rsi, cl        ; RSI=RSI>>CL
10    and   esi, 1          ; ESI=ESI&1
11 ; the last bit is 1? If so, write new version of rt into EAX
12    cmovne eax, edx
13    add    rcx, 1         ; RCX++
14    cmp    rcx, 64
15    jne    .L3
16    rep    ret            ; AKA fatret
```

This code is terser, but has a quirk.

In all examples that we see so far, we were incrementing the “rt” value after comparing a specific bit, but the code here increments “rt” before (line 6), writing the new value into register `EDX`. Thus, if the last bit is 1, the `CMOVNE` <sup>149</sup> instruction (which is a synonym for `CMOVNZ` <sup>150</sup>) *commits* the new value of “rt” by moving `EDX` (“proposed rt value”) into `EAX` (“current rt” to be returned at the end).

Hence, the incrementing is performed at each step of loop, i.e., 64 times, without any relation to the input value.

The advantage of this code is that it contain only one conditional jump (at the end of the loop) instead of two jumps (skipping the “rt” value increment and at the end of loop). And that might work faster on the modern CPUs with branch predictors: [2.9.1 on page 465](#).

The last instruction is `REP RET` (opcode `F3 C3`) which is also called `FATRET` by MSVC. This is somewhat optimized version of `RET`, which is recommended by AMD to be placed at the end of function, if `RET` goes right after conditional jump: [[*Software Optimization Guide for AMD Family 16h Processors, (2013)*]p.15] <sup>151</sup>.

### Optimizing MSVC 2010

Listing 1.291: MSVC 2010

```
a$ = 8
f
PROC
; RCX = input value
    xor    eax, eax
    mov    edx, 1
    lea    r8d, QWORD PTR [rax+64]
; R8D=64
    npad   5
$L14@f:
    test   rdx, rcx
; there are no such bit in input value?
; skip the next INC instruction then.
    je     SHORT $LN3@f
    inc    eax      ; rt++
```

<sup>149</sup>Conditional MOVE if Not Equal

<sup>150</sup>Conditional MOVE if Not Zero

<sup>151</sup>More information on it: <http://go.yurichev.com/17328>

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
$LN3@f:  
    rol    rdx, 1 ; RDX=RDX<<1  
    dec    r8      ; R8--  
    jne    SHORT $LL4@f  
    fatret 0  
f     ENDP
```

Here the `ROL` instruction is used instead of `SHL`, which is in fact “rotate left” instead of “shift left”, but in this example it works just as `SHL`.

You can read more about the rotate instruction here: [1.6 on page 1017](#).

`R8` here is counting from 64 to 0. It’s just like an inverted  $i$ .

Here is a table of some registers during the execution:

RDX	R8
0x000000000000000000000001	64
0x000000000000000000000002	63
0x000000000000000000000004	62
0x000000000000000000000008	61
...	...
0x400000000000000000000000	2
0x800000000000000000000000	1

At the end we see the `FATRET` instruction, which was explained here: [1.22.5 on the previous page](#).

## Optimizing MSVC 2012

Listing 1.292: MSVC 2012

```
a$ = 8  
f      PROC  
; RCX = input value  
    xor    eax, eax  
    mov    edx, 1  
    lea    r8d, QWORD PTR [rax+32]  
; EDX = 1, R8D = 32  
    npad   5  
$LL4@f:  
; pass 1 -----  
    test   rdx, rcx  
    je     SHORT $LN3@f  
    inc    eax      ; rt++  
$LN3@f:  
    rol    rdx, 1 ; RDX=RDX<<1  
; -----  
; pass 2 -----  
    test   rdx, rcx  
    je     SHORT $LN11@f  
    inc    eax      ; rt++  
$LN11@f:  
    rol    rdx, 1 ; RDX=RDX<<1  
; -----  
    dec    r8      ; R8--  
    jne    SHORT $LL4@f  
    fatret 0  
f     ENDP
```

Optimizing MSVC 2012 does almost the same job as optimizing MSVC 2010, but somehow, it generates two identical loop bodies and the loop count is now 32 instead of 64.

To be honest, it’s not possible to say why. Some optimization trick? Maybe it’s better for the loop body to be slightly longer?

Anyway, such code is relevant here to show that sometimes the compiler output may be really weird and illogical, but perfectly working.

Listing 1.293: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```

    MOV      R1, R0
    MOV      R0, #0
    MOV      R2, #1
    MOV      R3, R0
loc_2E54
    TST      R1, R2,LSL R3 ; set flags according to R1 & (R2<<R3)
    ADD      R3, R3, #1     ; R3++
    ADDNE   R0, R0, #1     ; if ZF flag is cleared by TST, then R0++
    CMP      R3, #32
    BNE      loc_2E54
    BX       LR

```

`TST` is the same things as `TEST` in x86.

As was noted before ([3.9.3 on page 498](#)), there are no separate shifting instructions in ARM mode. However, there are modifiers LSL (*Logical Shift Left*), LSR (*Logical Shift Right*), ASR (*Arithmetic Shift Right*), ROR (*Rotate Right*) and RRX (*Rotate Right with Extend*), which may be added to such instructions as `MOV`, `TST`, `CMP`, `ADD`, `SUB`, `RSB`<sup>152</sup>.

These modicators define how to shift the second operand and by how many bits.

Thus the “`TST R1, R2,LSL R3`” instruction works here as  $R1 \wedge (R2 \ll R3)$ .

### ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Almost the same, but here are two `LSL.W` / `TST` instructions are used instead of a single `TST`, because in Thumb mode it is not possible to define `LSL` modifier directly in `TST`.

```

    MOV      R1, R0
    MOVS   R0, #0
    MOV.W  R9, #1
    MOVS   R3, #0
loc_2F7A
    LSL.W  R2, R9, R3
    TST    R2, R1
    ADD.W  R3, R3, #1
    IT NE
    ADDNE R0, #1
    CMP    R3, #32
    BNE    loc_2F7A
    BX     LR

```

### ARM64 + Optimizing GCC 4.9

Let's take the 64-bit example which has been already used: [1.22.5 on page 331](#).

Listing 1.294: Optimizing GCC (Linaro) 4.8

```

f:
    mov    w2, 0          ; rt=0
    mov    x5, 1
    mov    w1, w2
.L2:
    lsl    x4, x5, x1    ; w4 = w5<<w1 = 1<<i
    add    w3, w2, 1      ; new_rt=rt+1
    tst    x4, x0          ; (1<<i) & a
    add    w1, w1, 1      ; i++
; result of TST was non-zero?
; then w2=w3 or rt=new_rt.
; otherwise: w2=w2 or rt=rt (idle operation)

```

<sup>152</sup>These instructions are also called “data processing instructions”

## 1.22. MANIPULATING SPECIFIC BIT(S)

```
cSEL    w2, w3, w2, ne
cmp    w1, 64          ; i<64?
bne    .L2            ; yes
mov    w0, w2          ; return rt
ret
```

The result is very similar to what GCC generates for x64: [1.290 on page 332](#).

The `CSEL` instruction is “Conditional SELect”. It just chooses one variable of two depending on the flags set by `TST` and copies the value into `W2`, which holds the “rt” variable.

## ARM64 + Non-optimizing GCC 4.9

And again, we’ll work on the 64-bit example which was already used: [1.22.5 on page 331](#). The code is more verbose, as usual.

Listing 1.295: Non-optimizing GCC (Linaro) 4.8

```
f:
    sub    sp, sp, #32
    str    x0, [sp,8]      ; store "a" value to Register Save Area
    str    wzr, [sp,24]     ; rt=0
    str    wzr, [sp,28]     ; i=0
    b     .L2
.L4:
    ldr    w0, [sp,28]
    mov    x1, 1
    lsl    x0, x1, x0      ; X0 = X1<<X0 = 1<<i
    mov    x1, x0
; X1 = 1<<i
    ldr    x0, [sp,8]
; X0 = a
    and    x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; X0 contain zero? then jump to .L3, skipping "rt" increment
    cmp    x0, xzr
    beq    .L3
; rt++
    ldr    w0, [sp,24]
    add    w0, w0, 1
    str    w0, [sp,24]
.L3:
; i++
    ldr    w0, [sp,28]
    add    w0, w0, 1
    str    w0, [sp,28]
.L2:
; i<=63? then jump to .L4
    ldr    w0, [sp,28]
    cmp    w0, 63
    ble    .L4
; return rt
    ldr    w0, [sp,24]
    add    sp, sp, 32
    ret
```

## MIPS

### Non-optimizing GCC

Listing 1.296: Non-optimizing GCC 4.4.5 (IDA)

```
f:
; IDA is not aware of variable names, we gave them manually:
rt        = -0x10
i         = -0xC
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

```

var_4      = -4
a          = 0

        addiu   $sp, -0x18
        sw      $fp, 0x18+var_4($sp)
        move    $fp, $sp
        sw      $a0, 0x18+a($fp)
; initialize rt and i variables to zero:
        sw      $zero, 0x18+rt($fp)
        sw      $zero, 0x18+i($fp)
; jump to loop check instructions:
        b       loc_68
        or      $at, $zero ; branch delay slot, NOP

loc_20:
        li      $v1, 1
        lw      $v0, 0x18+i($fp)
        or      $at, $zero ; load delay slot, NOP
        sllv   $v0, $v1, $v0
; $v0 = 1<<i
        move   $v1, $v0
        lw      $v0, 0x18+a($fp)
        or      $at, $zero ; load delay slot, NOP
        and   $v0, $v1, $v0
; $v0 = a&(1<<i)
; is a&(1<<i) equals to zero? jump to loc_58 then:
        beqz  $v0, loc_58
        or      $at, $zero
; no jump occurred, that means a&(1<<i)!=0, so increment "rt" then:
        lw      $v0, 0x18+rt($fp)
        or      $at, $zero ; load delay slot, NOP
        addiu $v0, 1
        sw      $v0, 0x18+rt($fp)

loc_58:
; increment i:
        lw      $v0, 0x18+i($fp)
        or      $at, $zero ; load delay slot, NOP
        addiu $v0, 1
        sw      $v0, 0x18+i($fp)

loc_68:
; load i and compare it with 0x20 (32).
; jump to loc_20 if it is less than 0x20 (32):
        lw      $v0, 0x18+i($fp)
        or      $at, $zero ; load delay slot, NOP
        slti   $v0, 0x20 # ''
        bnez   $v0, loc_20
        or      $at, $zero ; branch delay slot, NOP
; function epilogue. return rt:
        lw      $v0, 0x18+rt($fp)
        move   $sp, $fp ; load delay slot
        lw      $fp, 0x18+var_4($sp)
        addiu $sp, 0x18 ; load delay slot
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP

```

That is verbose: all local variables are located in the local stack and reloaded each time they're needed.

The `SLLV` instruction is “Shift Word Left Logical Variable”, it differs from `SLL` only in that the shift amount is encoded in the `SLL` instruction (and is fixed, as a consequence), but `SLLV` takes shift amount from a register.

## Optimizing GCC

That is terser. There are two shift instructions instead of one. Why?

It's possible to replace the first `SLLV` instruction with an unconditional branch instruction that jumps right

## 1.22. MANIPULATING SPECIFIC BIT(S)

to the second `SLLV`. But this is another branching instruction in the function, and it's always favorable to get rid of them: [2.9.1 on page 465](#).

Listing 1.297: Optimizing GCC 4.4.5 (IDA)

```
f:  
; $a0=a  
; rt variable will reside in $v0:  
    move    $v0, $zero  
; i variable will reside in $v1:  
    move    $v1, $zero  
    li      $t0, 1  
    li      $a3, 32  
    sllv    $a1, $t0, $v1  
; $a1 = $t0<<$v1 = 1<<i  
  
loc_14:  
    and     $a1, $a0  
; $a1 = a&(1<<i)  
; increment i:  
    addiu   $v1, 1  
; jump to loc_28 if a&(1<<i)==0 and increment rt:  
    beqz   $a1, loc_28  
    addiu   $a2, $v0, 1  
; if BEQZ was not triggered, save updated rt into $v0:  
    move    $v0, $a2  
  
loc_28:  
; if i!=32, jump to loc_14 and also prepare next shifted value:  
    bne    $v1, $a3, loc_14  
    sllv    $a1, $t0, $v1  
; return  
    jr     $ra  
    or     $at, $zero ; branch delay slot, NOP
```

### 1.22.6 Conclusion

Analogous to the C/C++ shifting operators `<<` and `>>`, the shift instructions in x86 are `SHR / SHL` (for unsigned values) and `SAR / SHL` (for signed values).

The shift instructions in ARM are `LSR / LSL` (for unsigned values) and `ASR / LSL` (for signed values).

It's also possible to add shift suffix to some instructions (which are called "data processing instructions").

### Check for specific bit (known at compile stage)

Test if the `0b1000000` bit (`0x40`) is present in the register's value:

Listing 1.298: C/C++

```
if (input&0x40)  
...
```

Listing 1.299: x86

```
TEST REG, 40h  
JNZ is_set  
; bit is not set
```

Listing 1.300: x86

```
TEST REG, 40h  
JZ is_cleared  
; bit is set
```

## 1.22. MANIPULATING SPECIFIC BIT(S)

Listing 1.301: ARM (ARM mode)

```
TST REG, #0x40
BNE is_set
; bit is not set
```

Sometimes, `AND` is used instead of `TEST`, but the flags that are set are the same.

### Check for specific bit (specified at runtime)

This is usually done by this C/C++ code snippet (shift value by  $n$  bits right, then cut off lowest bit):

Listing 1.302: C/C++

```
if ((value>>n)&1)
....
```

This is usually implemented in x86 code as:

Listing 1.303: x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Or (shift 1 bit  $n$  times left, isolate this bit in input value and check if it's not zero):

Listing 1.304: C/C++

```
if (value & (1<<n))
....
```

This is usually implemented in x86 code as:

Listing 1.305: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

### Set specific bit (known at compile stage)

Listing 1.306: C/C++

```
value=value|0x40;
```

Listing 1.307: x86

```
OR REG, 40h
```

Listing 1.308: ARM (ARM mode) and ARM64

```
ORR R0, R0, #0x40
```

### Set specific bit (specified at runtime)

Listing 1.309: C/C++

```
value=value|(1<<n);
```

This is usually implemented in x86 code as:

Listing 1.310: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

## 1.23. LINEAR CONGRUENTIAL GENERATOR

### Clear specific bit (known at compile stage)

Just apply **AND** operation with the inverted value:

Listing 1.311: C/C++

```
value=value&(~0x40);
```

Listing 1.312: x86

```
AND REG, 0FFFFFFFBFh
```

Listing 1.313: x64

```
AND REG, 0xFFFFFFFFFFFFFFBFh
```

This is actually leaving all bits set except one.

ARM in ARM mode has **BIC** instruction, which works like the **NOT** + **AND** instruction pair:

Listing 1.314: ARM (ARM mode)

```
BIC R0, R0, #0x40
```

### Clear specific bit (specified at runtime)

Listing 1.315: C/C++

```
value=value&(~(1<<n));
```

Listing 1.316: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

## 1.22.7 Exercises

- <http://challenges.re/67>
- <http://challenges.re/68>
- <http://challenges.re/69>
- <http://challenges.re/70>

## 1.23 Linear congruential generator as pseudorandom number generator

Perhaps, the linear congruential generator is the simplest possible way to generate random numbers.

It's not in favour nowadays<sup>153</sup>, but it's so simple (just one multiplication, one addition and one AND operation), we can use it as an example.

```
#include <stdint.h>

// constants from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;
```

<sup>153</sup>Mersenne twister is better

```

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

```

There are two functions: the first one is used to initialize the internal state, and the second one is called to generate pseudorandom numbers.

We see that two constants are used in the algorithm. They are taken from [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)].

Let's define them using a `#define` C/C++ statement. It's a macro.

The difference between a C/C++ macro and a constant is that all macros are replaced with their value by C/C++ preprocessor, and they don't take any memory, unlike variables.

In contrast, a constant is a read-only variable.

It's possible to take a pointer (or address) of a constant variable, but impossible to do so with a macro.

The last AND operation is needed because by C-standard `my_rand()` has to return a value in the 0..32767 range.

If you want to get 32-bit pseudorandom values, just omit the last AND operation.

### 1.23.1 x86

Listing 1.317: Optimizing MSVC 2013

```

_BSS      SEGMENT
_rand_state DD  01H DUP (?)
_BSS      ENDS

_init$ = 8
_srand  PROC
    mov     eax, DWORD PTR _init$[esp-4]
    mov     DWORD PTR _rand_state, eax
    ret     0
_srand  ENDP

_TEXT    SEGMENT
_rand   PROC
    imul   eax, DWORD PTR _rand_state, 1664525
    add    eax, 1013904223 ; 3c6ef35fH
    mov    DWORD PTR _rand_state, eax
    and    eax, 32767       ; 00007fffH
    ret     0
_rand   ENDP

_TEXT    ENDS

```

Here we see it: both constants are embedded into the code. There is no memory allocated for them.

The `my_srand()` function just copies its input value into the internal `rand_state` variable.

`my_rand()` takes it, calculates the next `rand_state`, cuts it and leaves it in the EAX register.

The non-optimized version is more verbose:

## 1.23. LINEAR CONGRUENTIAL GENERATOR

Listing 1.318: Non-optimizing MSVC 2013

```
_BSS SEGMENT
_rand_state DD 01H DUP (?)
_BSS ENDS

_init$ = 8
_srand PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _init$[ebp]
    mov     DWORD PTR _rand_state, eax
    pop    ebp
    ret    0
_srand ENDP

_TEXT SEGMENT
_rand PROC
    push    ebp
    mov     ebp, esp
    imul   eax, DWORD PTR _rand_state, 1664525
    mov     DWORD PTR _rand_state, eax
    mov     ecx, DWORD PTR _rand_state
    add    ecx, 1013904223 ; 3c6ef35fH
    mov     DWORD PTR _rand_state, ecx
    mov     eax, DWORD PTR _rand_state
    and    eax, 32767      ; 00007fffH
    pop    ebp
    ret    0
_rand ENDP

_TEXT ENDS
```

### 1.23.2 x64

The x64 version is mostly the same and uses 32-bit registers instead of 64-bit ones (because we are working with *int* values here).

But `my_srand()` takes its input argument from the `ECX` register rather than from stack:

Listing 1.319: Optimizing MSVC 2013 x64

```
_BSS SEGMENT
rand_state DD 01H DUP (?)
_BSS ENDS

init$ = 8
my_srand PROC
; ECX = input argument
    mov     DWORD PTR rand_state, ecx
    ret    0
my_srand ENDP

_TEXT SEGMENT
my_rand PROC
    imul   eax, DWORD PTR rand_state, 1664525 ; 0019660dH
    add    eax, 1013904223 ; 3c6ef35fH
    mov     DWORD PTR rand_state, eax
    and    eax, 32767      ; 00007fffH
    ret    0
my_rand ENDP

_TEXT ENDS
```

GCC compiler generates mostly the same code.

### 1.23.3 32-bit ARM

## 1.23. LINEAR CONGRUENTIAL GENERATOR

Listing 1.320: Optimizing Keil 6/2013 (ARM mode)

```

my_srand PROC
    LDR    r1, |L0.52| ; load pointer to rand_state
    STR    r0,[r1,#0] ; save rand_state
    BX    lr
    ENDP

my_rand PROC
    LDR    r0, |L0.52| ; load pointer to rand_state
    LDR    r2, |L0.56| ; load RNG_a
    LDR    r1,[r0,#0] ; load rand_state
    MUL    r1,r2,r1
    LDR    r2, |L0.60| ; load RNG_c
    ADD    r1,r1,r2
    STR    r1,[r0,#0] ; save rand_state
; AND with 0x7FFF:
    LSL    r0,r1,#17
    LSR    r0,r0,#17
    BX    lr
    ENDP

|L0.52|
    DCD    ||.data||
|L0.56|
    DCD    0x0019660d
|L0.60|
    DCD    0x3c6ef35f

    AREA ||.data||, DATA, ALIGN=2

rand_state
    DCD    0x00000000

```

It's not possible to embed 32-bit constants into ARM instructions, so Keil has to place them externally and load them additionally. One interesting thing is that it's not possible to embed the 0x7FFF constant as well. So what Keil does is shifting `rand_state` left by 17 bits and then shifting it right by 17 bits. This is analogous to the `(rand_state << 17) >> 17` statement in C/C++. It seems to be useless operation, but what it does is clearing the high 17 bits, leaving the low 15 bits intact, and that's our goal after all.

Optimizing Keil for Thumb mode generates mostly the same code.

### 1.23.4 MIPS

Listing 1.321: Optimizing GCC 4.4.5 (IDA)

```

my_srand:
; store $a0 to rand_state:
    lui    $v0, (rand_state >> 16)
    jr    $ra
    sw    $a0, rand_state

my_rand:
; load rand_state to $v0:
    lui    $v1, (rand_state >> 16)
    lw    $v0, rand_state
    or    $at, $zero ; load delay slot
; multiplicate rand_state in $v0 by 1664525 (RNG_a):
    sll    $a1, $v0, 2
    sll    $a0, $v0, 4
    addu   $a0, $a1, $a0
    sll    $a1, $a0, 6
    subu   $a0, $a1, $a0
    addu   $a0, $v0
    sll    $a1, $a0, 5
    addu   $a0, $a1
    sll    $a0, 3
    addu   $v0, $a0, $v0
    sll    $a0, $v0, 2

```

## 1.23. LINEAR CONGRUENTIAL GENERATOR

```
        addu    $v0, $a0
; add 1013904223 (RNG_c)
; the LI instruction is coalesced by IDA from LUI and ORI
        li      $a0, 0x3C6EF35F
        addu    $v0, $a0
; store to rand_state:
        sw      $v0, (rand_state & 0xFFFF)($v1)
        jr      $ra
        andi    $v0, 0x7FFF ; branch delay slot
```

Wow, here we see only one constant (0x3C6EF35F or 1013904223). Where is the other one (1664525)?

It seems that multiplication by 1664525 is performed by just using shifts and additions! Let's check this assumption:

```
#define RNG_a 1664525

int f (int a)
{
    return a*RNG_a;
}
```

Listing 1.322: Optimizing GCC 4.4.5 (IDA)

```
f:
        sll    $v1, $a0, 2
        sll    $v0, $a0, 4
        addu   $v0, $v1, $v0
        sll    $v1, $v0, 6
        subu   $v0, $v1, $v0
        addu   $v0, $a0
        sll    $v1, $v0, 5
        addu   $v0, $v1
        sll    $v0, 3
        addu   $a0, $v0, $a0
        sll    $v0, $a0, 2
        jr     $ra
        addu   $v0, $a0, $v0 ; branch delay slot
```

Indeed!

## MIPS relocations

We will also focus on how such operations as load from memory and store to memory actually work.

The listings here are produced by IDA, which hides some details.

We'll run objdump twice: to get a disassembled listing and also relocations list:

Listing 1.323: Optimizing GCC 4.4.5 (objdump)

```
# objdump -D rand_03.o

...
00000000 <my_srand>:
 0: 3c020000    lui    v0,0x0
 4: 03e00008    jr     ra
 8: ac440000    sw     a0,0(v0)

0000000c <my_rand>:
 c: 3c030000    lui    v1,0x0
10: 8c620000    lw     v0,0(v1)
14: 00200825    move   at,at
18: 00022880    sll    a1,v0,0x2
1c: 00022100    sll    a0,v0,0x4
20: 00a42021    addu   a0,a1,a0
24: 00042980    sll    a1,a0,0x6
28: 00a42023    subu   a0,a1,a0
2c: 00822021    addu   a0,a0,v0
```

### 1.23. LINEAR CONGRUENTIAL GENERATOR

```
30: 00042940      sll    a1,a0,0x5
34: 00852021      addu   a0,a0,a1
38: 000420c0      sll    a0,a0,0x3
3c: 00821021      addu   v0,a0,v0
40: 00022080      sll    a0,v0,0x2
44: 00441021      addu   v0,v0,a0
48: 3c043c6e      lui    a0,0x3c6e
4c: 3484f35f      ori    a0,a0,0xf35f
50: 00441021      addu   v0,v0,a0
54: ac620000      sw     v0,0(v1)
58: 03e00008      jr    ra
5c: 30427fff      andi   v0,v0,0x7fff

...
# objdump -r rand_03.o
...

RELOCATION RECORDS FOR [.text]:
OFFSET  TYPE      VALUE
00000000 R_MIPS_HI16 .bss
00000008 R_MIPS_L016 .bss
0000000c R_MIPS_HI16 .bss
00000010 R_MIPS_L016 .bss
00000054 R_MIPS_L016 .bss

...
```

Let's consider the two relocations for the `my_srand()` function.

The first one, for address 0 has a type of `R_MIPS_HI16` and the second one for address 8 has a type of `R_MIPS_L016`.

That implies that address of the beginning of the `.bss` segment is to be written into the instructions at address of 0 (high part of address) and 8 (low part of address).

The `rand_state` variable is at the very start of the `.bss` segment.

So we see zeros in the operands of instructions `LUI` and `SW`, because nothing is there yet—the compiler don't know what to write there.

The linker will fix this, and the high part of the address will be written into the operand of `LUI` and the low part of the address—to the operand of `SW`.

`SW` will sum up the low part of the address and what is in register `$V0` (the high part is there).

It's the same story with the `my_rand()` function: `R_MIPS_HI16` relocation instructs the linker to write the high part of the `.bss` segment address into instruction `LUI`.

So the high part of the `rand_state` variable address is residing in register `$V1`.

The `LW` instruction at address `0x10` sums up the high and low parts and loads the value of the `rand_state` variable into `$V0`.

The `SW` instruction at address `0x54` do the summing again and then stores the new value to the `rand_state` global variable.

IDA processes relocations while loading, thus hiding these details, but we should keep them in mind.

#### 1.23.5 Thread-safe version of the example

The thread-safe version of the example is to be demonstrated later: [6.2.1 on page 742](#).

## 1.24 Structures

A C/C++ structure, with some assumptions, is just a set of variables, always stored in memory together, not necessary of the same type <sup>154</sup>.

### 1.24.1 MSVC: SYSTEMTIME example

Let's take the SYSTEMTIME<sup>155</sup> win32 structure that describes time.

This is how it's defined:

Listing 1.324: WinBase.h

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Let's write a C function to get the current time:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t.wYear, t.wMonth, t.wDay,
            t.wHour, t.wMinute, t.wSecond);

    return;
}
```

We get (MSVC 2010):

Listing 1.325: MSVC 2010 /GS-

```
t$ = -16 ; size = 16
_main      PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    lea     eax, DWORD PTR _t$[ebp]
    push    eax
    call    DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
    push    ecx
    movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
    push    edx
    movzx  eax, WORD PTR _t$[ebp+8] ; wHour
    push    eax
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push    ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push    edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push    eax
    push    OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
```

<sup>154</sup> AKA "heterogeneous container"

<sup>155</sup> MSDN: SYSTEMTIME structure

## 1.24. STRUCTURES

```
call   _printf
add    esp, 28
xor    eax, eax
mov    esp, ebp
pop    ebp
ret    0
_main  ENDP
```

16 bytes are allocated for this structure in the local stack —that is exactly `sizeof(WORD)*8` (there are 8 WORD variables in the structure).

Pay attention to the fact that the structure begins with the `wYear` field. It can be said that a pointer to the SYSTEMTIME structure is passed to the `GetSystemTime()` <sup>156</sup>, but it is also can be said that a pointer to the `wYear` field is passed, and that is the same! `GetSystemTime()` writes the current year to the WORD pointer pointing to, then shifts 2 bytes ahead, writes current month, etc., etc.

---

<sup>156</sup>[MSDN: SYSTEMTIME structure](#)

**OllyDbg**

Let's compile this example in MSVC 2010 with `/GS- /MD` keys and run it in OllyDbg.

Let's open windows for data and stack at the address which is passed as the first argument of the `GetSystemTime()` function, and let's wait until it's executed. We see this:

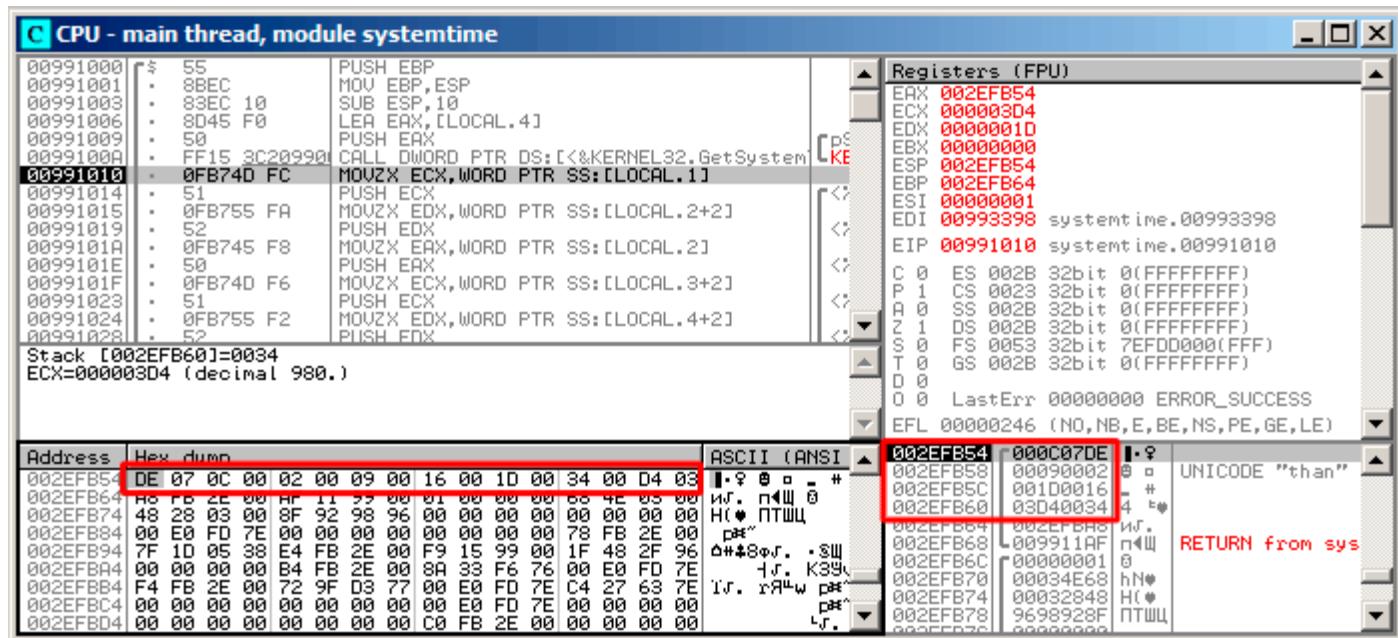


Figure 1.104: OllyDbg: `GetSystemTime()` just executed

The system time of the function execution on my computer is 9 December 2014, 22:29:52:

Listing 1.326: `printf()` output

2014-12-09 22:29:52

So we see these 16 bytes in the data window:

DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03

Each two bytes represent one field of the structure. Since the [endianness](#) is *little endian*, we see the low byte first and then the high one.

Hence, these are the values currently stored in memory:

Hexadecimal number	decimal number	field name
0x07DE	2014	wYear
0x000C	12	wMonth
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliseconds

The same values are seen in the stack window, but they are grouped as 32-bit values.

And then `printf()` just takes the values it needs and outputs them to the console.

Some values aren't output by `printf()` (`wDayOfWeek` and `wMilliseconds`), but they are in memory right now, available for use.

**Replacing the structure with array**

The fact that the structure fields are just variables located side-by-side, can be easily demonstrated by doing the following. Keeping in mind the `SYSTEMTIME` structure description, it's possible to rewrite this simple example like this:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
            array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */);

    return;
}
```

The compiler grumbles a bit:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

But nevertheless, it produces this code:

Listing 1.327: Non-optimizing MSVC 2010

```
$SG78573 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16    ; size = 16
_main PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    lea     eax, DWORD PTR _array$[ebp]
    push    eax
    call    DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _array$[ebp+12] ; wSecond
    push    ecx
    movzx  edx, WORD PTR _array$[ebp+10] ; wMinute
    push    edx
    movzx  eax, WORD PTR _array$[ebp+8] ; wHour
    push    eax
    movzx  ecx, WORD PTR _array$[ebp+6] ; wDay
    push    ecx
    movzx  edx, WORD PTR _array$[ebp+2] ; wMonth
    push    edx
    movzx  eax, WORD PTR _array$[ebp] ; wYear
    push    eax
    push    OFFSET $SG78573
    call    _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main ENDP
```

And it works just as the same!

It is very interesting that the result in assembly form cannot be distinguished from the result of the previous compilation.

So by looking at this code, one cannot say for sure if there was a structure declared, or an array.

Nevertheless, no sane person would do it, as it is not convenient.

Also the structure fields may be changed by developers, swapped, etc.

## 1.24. STRUCTURES

We will not study this example in OllyDbg, because it will be just the same as in the case with the structure.

### 1.24.2 Let's allocate space for a structure using malloc()

Sometimes it is simpler to place structures not the in local stack, but in the [heap](#):

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
           t->wYear, t->wMonth, t->wDay,
           t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
}
```

Let's compile it now with optimization (`/Ox`) so it would be easy to see what we need.

Listing 1.328: Optimizing MSVC

```
_main PROC
    push  esi
    push  16
    call  _malloc
    add   esp, 4
    mov   esi, eax
    push  esi
    call  DWORD PTR __imp__GetSystemTime@4
    movzx eax, WORD PTR [esi+12] ; wSecond
    movzx ecx, WORD PTR [esi+10] ; wMinute
    movzx edx, WORD PTR [esi+8] ; wHour
    push  eax
    movzx eax, WORD PTR [esi+6] ; wDay
    push  ecx
    movzx ecx, WORD PTR [esi+2] ; wMonth
    push  edx
    movzx edx, WORD PTR [esi] ; wYear
    push  eax
    push  ecx
    push  edx
    push  OFFSET $SG78833
    call  _printf
    push  esi
    call  _free
    add   esp, 32
    xor   eax, eax
    pop   esi
    ret   0
_main ENDP
```

So, `sizeof(SYSTEMTIME) = 16` and that is exact number of bytes to be allocated by `malloc()`. It returns a pointer to a freshly allocated memory block in the `EAX` register, which is then moved into the `ESI` register. `GetSystemTime()` win32 function takes care of saving value in `ESI`, and that is why it is not saved here and continues to be used after the `GetSystemTime()` call.

New instruction — `MOVZX` (*Move with Zero eXtend*). It may be used in most cases as `MOVVSX`, but it sets the remaining bits to 0. That's because `printf()` requires a 32-bit `int`, but we got a `WORD` in the structure —

## 1.24. STRUCTURES

that is 16-bit unsigned type. That's why by copying the value from a WORD into *int*, bits from 16 to 31 must be cleared, because a random noise may be there, which is left from the previous operations on the register(s).

In this example, it's possible to represent the structure as an array of 8 WORDs:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
            t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
}
```

We get:

Listing 1.329: Optimizing MSVC

```
$SG78594 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main PROC
    push    esi
    push    16
    call    _malloc
    add     esp, 4
    mov     esi, eax
    push    esi
    call    DWORD PTR _imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12]
    movzx  ecx, WORD PTR [esi+10]
    movzx  edx, WORD PTR [esi+8]
    push    eax
    movzx  eax, WORD PTR [esi+6]
    push    ecx
    movzx  ecx, WORD PTR [esi+2]
    push    edx
    movzx  edx, WORD PTR [esi]
    push    eax
    push    ecx
    push    edx
    push    OFFSET $SG78594
    call    _printf
    push    esi
    call    _free
    add     esp, 32
    xor     eax, eax
    pop     esi
    ret     0
_main ENDP
```

Again, we got the code that cannot be distinguished from the previous one.

And again it has to be noted, you haven't to do this in practice, unless you really know what you are doing.

**1.24.3 UNIX: struct tm****Linux**

Let's take the `tm` structure from `time.h` in Linux for example:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
}
```

Let's compile it in GCC 4.4.1:

Listing 1.330: GCC 4.4.1

```
main proc near
push    ebp
mov     ebp, esp
and    esp, 0FFFFFFF0h
sub    esp, 40h
mov    dword ptr [esp], 0 ; first argument for time()
call   time
mov    [esp+3Ch], eax
lea     eax, [esp+3Ch] ; take pointer to what time() returned
lea     edx, [esp+10h] ; at ESP+10h struct tm will begin
mov    [esp+4], edx ; pass pointer to the structure begin
mov    [esp], eax ; pass pointer to result of time()
call   localtime_r
mov    eax, [esp+24h] ; tm_year
lea     edx, [eax+76Ch] ; edx=eax+1900
mov    eax, offset format ; "Year: %d\n"
mov    [esp+4], edx
mov    [esp], eax
call   printf
mov    edx, [esp+20h] ; tm_mon
mov    eax, offset aMonthD ; "Month: %d\n"
mov    [esp+4], edx
mov    [esp], eax
call   printf
mov    edx, [esp+1Ch] ; tm_mday
mov    eax, offset aDayD ; "Day: %d\n"
mov    [esp+4], edx
mov    [esp], eax
call   printf
mov    edx, [esp+18h] ; tm_hour
mov    eax, offset aHourD ; "Hour: %d\n"
mov    [esp+4], edx
mov    [esp], eax
call   printf
mov    edx, [esp+14h] ; tm_min
mov    eax, offset aMinutesD ; "Minutes: %d\n"
mov    [esp+4], edx
mov    [esp], eax
call   printf
mov    edx, [esp+10h]
```

## 1.24. STRUCTURES

```
mov    eax, offset aSecondsD ; "Seconds: %d\n"
mov    [esp+4], edx          ; tm_sec
mov    [esp], eax
call   printf
leave
retn
main endp
```

Somehow, [IDA](#) did not write the local variables' names in the local stack. But since we already are experienced reverse engineers :-) we may do it without this information in this simple example.

Please also pay attention to the `lea edx, [eax+76Ch]` —this instruction just adds `0x76C` (1900) to value in `EAX`, but doesn't modify any flags. See also the relevant section about `LEA` ([.1.6 on page 1011](#)).

## GDB

Let's try to load the example into GDB [157](#):

Listing 1.331: GDB

```
dennis@ubuntuvm:~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvm:~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvm:~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/GCC_tm... (no debugging symbols found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm

Breakpoint 1, __printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29      printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc: 0x080484c3 0x080485c0 0x000007de 0x00000000
0xbffff0ec: 0x08048301 0x538c93ed 0x00000025 0x0000000a
0xbffff0fc: 0x00000012 0x00000002 0x00000005 0x00000072
0xbffff10c: 0x00000001 0x00000098 0x00000001 0x00002a30
0xbffff11c: 0x0804b090 0x08048530 0x00000000 0x00000000
(gdb)
```

We can easily find our structure in the stack. First, let's see how it's defined in `time.h`:

Listing 1.332: time.h

```
struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

Pay attention that 32-bit `int` is used here instead of WORD in SYSTEMTIME. So, each field occupies 32-bit.

Here are the fields of our structure in the stack:

0xbffff0dc:	0x080484c3	0x080485c0	0x000007de	0x00000000
0xbffff0ec:	0x08048301	0x538c93ed	0x00000025 sec	0x0000000a min
0xbffff0fc:	0x00000012 hour	0x00000002 mday	0x00000005 mon	0x00000072 year

<sup>157</sup>The `date` result is slightly corrected for demonstration purposes. Of course, it's not possible to run GDB that quickly, in the same second.

## 1.24. STRUCTURES

0xbfffff10c:	0x00000001	wday	0x00000098	yday	0x00000001	isdst	0x00002a30
0xbfffff11c:	0x0804b090		0x08048530		0x00000000		0x00000000

Or as a table:

Hexadecimal number	decimal number	field name
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

Just like SYSTEMTIME ( [1.24.1 on page 345](#)),

there are also other fields available that are not used, like tm\_wday, tm\_yday, tm\_isdst.

## ARM

### Optimizing Keil 6/2013 (Thumb mode)

Same example:

Listing 1.333: Optimizing Keil 6/2013 (Thumb mode)

```

var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer = -0xC

PUSH {LR}
MOVS R0, #0          ; timer
SUB SP, SP, #0x34
BL time
STR R0, [SP,#0x38+timer]
MOV R1, SP          ; tp
ADD R0, SP, #0x38+timer ; timer
BL localtime_r
LDR R1, =0x76C
LDR R0, [SP,#0x38+var_24]
ADDS R1, R0, R1
ADR R0, aYearD      ; "Year: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_28]
ADR R0, aMonthD     ; "Month: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_2C]
ADR R0, aDayD       ; "Day: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_30]
ADR R0, aHourD      ; "Hour: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_34]
ADR R0, aMinutesD   ; "Minutes: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_38]
ADR R0, aSecondsD   ; "Seconds: %d\n"
BL __2printf
ADD SP, SP, #0x34
POP {PC}

```

**Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)**

IDA “knows” the `tm` structure (because IDA “knows” the types of the arguments of library functions like `localtime_r()`),

so it shows here structure elements accesses and their names.

Listing 1.334: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```

var_38 = -0x38
var_34 = -0x34

PUSH {R7,LR}
MOV R7, SP
SUB SP, SP, #0x30
MOVS R0, #0 ; time_t *
BLX _time
ADD R1, SP, #0x38+var_34 ; struct tm *
STR R0, [SP,#0x38+var_38]
MOV R0, SP ; time_t *
BLX _localtime_r
LDR R1, [SP,#0x38+var_34.tm_year]
MOV R0, 0xF44 ; "Year: %d\n"
ADD R0, PC ; char *
ADDW R1, R1, #0x76C
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_mon]
MOV R0, 0xF3A ; "Month: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_mday]
MOV R0, 0xF35 ; "Day: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_hour]
MOV R0, 0xF2E ; "Hour: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_min]
MOV R0, 0xF28 ; "Minutes: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34]
MOV R0, 0xF25 ; "Seconds: %d\n"
ADD R0, PC ; char *
BLX _printf
ADD SP, SP, #0x30
POP {R7,PC}

...
00000000 tm      struc ; (sizeof=0x2C, standard type)
00000000 tm_sec  DCD ?
00000004 tm_min  DCD ?
00000008 tm_hour DCD ?
0000000C tm_mday DCD ?
00000010 tm_mon  DCD ?
00000014 tm_year DCD ?
00000018 tm_wday DCD ?
0000001C tm_yday DCD ?
00000020 tm_isdst DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone  DCD ? ; offset
0000002C tm      ends

```

Listing 1.335: Optimizing GCC 4.4.5 (IDA)

```

1 main:
2
3 ; IDA is not aware of structure field names, we named them manually:
4
5 var_40      = -0x40
6 var_38      = -0x38
7 seconds     = -0x34
8 minutes    = -0x30
9 hour        = -0x2C
10 day         = -0x28
11 month       = -0x24
12 year        = -0x20
13 var_4       = -4
14
15         lui      $gp, (_gnu_local_gp >> 16)
16         addiu   $sp, -0x50
17         la       $gp, (_gnu_local_gp & 0xFFFF)
18         sw       $ra, 0x50+var_4($sp)
19         sw       $gp, 0x50+var_40($sp)
20         lw       $t9, (time & 0xFFFF)($gp)
21         or       $at, $zero ; load delay slot, NOP
22         jalr    $t9
23         move    $a0, $zero ; branch delay slot, NOP
24         lw       $gp, 0x50+var_40($sp)
25         addiu   $a0, $sp, 0x50+var_38
26         lw       $t9, (localtime_r & 0xFFFF)($gp)
27         addiu   $a1, $sp, 0x50+seconds
28         jalr    $t9
29         sw       $v0, 0x50+var_38($sp) ; branch delay slot
30         lw       $gp, 0x50+var_40($sp)
31         lw       $a1, 0x50+year($sp)
32         lw       $t9, (printf & 0xFFFF)($gp)
33         la       $a0, $LC0      # "Year: %d\n"
34         jalr    $t9
35         addiu   $a1, 1900 ; branch delay slot
36         lw       $gp, 0x50+var_40($sp)
37         lw       $a1, 0x50+month($sp)
38         lw       $t9, (printf & 0xFFFF)($gp)
39         lui     $a0, ($LC1 >> 16) # "Month: %d\n"
40         jalr    $t9
41         la       $a0, ($LC1 & 0xFFFF) # "Month: %d\n" ; branch delay slot
42         lw       $gp, 0x50+var_40($sp)
43         lw       $a1, 0x50+day($sp)
44         lw       $t9, (printf & 0xFFFF)($gp)
45         lui     $a0, ($LC2 >> 16) # "Day: %d\n"
46         jalr    $t9
47         la       $a0, ($LC2 & 0xFFFF) # "Day: %d\n" ; branch delay slot
48         lw       $gp, 0x50+var_40($sp)
49         lw       $a1, 0x50+hour($sp)
50         lw       $t9, (printf & 0xFFFF)($gp)
51         lui     $a0, ($LC3 >> 16) # "Hour: %d\n"
52         jalr    $t9
53         la       $a0, ($LC3 & 0xFFFF) # "Hour: %d\n" ; branch delay slot
54         lw       $gp, 0x50+var_40($sp)
55         lw       $a1, 0x50+minutes($sp)
56         lw       $t9, (printf & 0xFFFF)($gp)
57         lui     $a0, ($LC4 >> 16) # "Minutes: %d\n"
58         jalr    $t9
59         la       $a0, ($LC4 & 0xFFFF) # "Minutes: %d\n" ; branch delay slot
60         lw       $gp, 0x50+var_40($sp)
61         lw       $a1, 0x50+seconds($sp)
62         lw       $t9, (printf & 0xFFFF)($gp)
63         lui     $a0, ($LC5 >> 16) # "Seconds: %d\n"
64         jalr    $t9
65         la       $a0, ($LC5 & 0xFFFF) # "Seconds: %d\n" ; branch delay slot
66         lw       $ra, 0x50+var_4($sp)
67         or       $at, $zero ; load delay slot, NOP
68         jr       $ra
69         addiu   $sp, 0x50

```

## 1.24. STRUCTURES

```
70
71 $LC0:          .ascii "Year: %d\n"<0>
72 $LC1:          .ascii "Month: %d\n"<0>
73 $LC2:          .ascii "Day: %d\n"<0>
74 $LC3:          .ascii "Hour: %d\n"<0>
75 $LC4:          .ascii "Minutes: %d\n"<0>
76 $LC5:          .ascii "Seconds: %d\n"<0>
```

This is an example where the branch delay slots can confuse us.

For example, there is the instruction `addiu $a1, 1900` at line 35 which adds 1900 to the year number.

It's executed before the corresponding `JALR` at line 34, do not forget about it.

### Structure as a set of values

In order to illustrate that the structure is just variables laying side-by-side in one place, let's rework our example while looking at the `tm` structure definition again: listing 1.332.

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
}
```

N.B. The pointer to the `tm_sec` field is passed into `localtime_r`, i.e., to the first element of the “structure”.

The compiler warns us:

Listing 1.336: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type [enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of type 'int *'
```

But nevertheless, it generates this:

Listing 1.337: GCC 4.7.3

```
main      proc near

var_30     = dword ptr -30h
var_2C     = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec     = dword ptr -18h
tm_min     = dword ptr -14h
tm_hour    = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon     = dword ptr -8
tm_year    = dword ptr -4

        push    ebp
```

## 1.24. STRUCTURES

```
mov    ebp, esp
and   esp, 0FFFFFFF0h
sub    esp, 30h
call   __main
mov    [esp+30h+var_30], 0 ; arg 0
call   time
mov    [esp+30h+unix_time], eax
lea    eax, [esp+30h+tm_sec]
mov    [esp+30h+var_2C], eax
lea    eax, [esp+30h+unix_time]
mov    [esp+30h+var_30], eax
call   localtime_r
mov    eax, [esp+30h+tm_year]
add    eax, 1900
mov    [esp+30h+var_2C], eax
mov    [esp+30h+var_30], offset aYearD ; "Year: %d\n"
call   printf
mov    eax, [esp+30h+tm_mon]
mov    [esp+30h+var_2C], eax
mov    [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
call   printf
mov    eax, [esp+30h+tm_mday]
mov    [esp+30h+var_2C], eax
mov    [esp+30h+var_30], offset aDayD ; "Day: %d\n"
call   printf
mov    eax, [esp+30h+tm_hour]
mov    [esp+30h+var_2C], eax
mov    [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
call   printf
mov    eax, [esp+30h+tm_min]
mov    [esp+30h+var_2C], eax
mov    [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
call   printf
mov    eax, [esp+30h+tm_sec]
mov    [esp+30h+var_2C], eax
mov    [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
call   printf
leave
retn
main  endp
```

This code is identical to what we saw previously and it is not possible to say, was it a structure in original source code or just a pack of variables.

And this works. However, it is not recommended to do this in practice.

Usually, non-optimizing compilers allocates variables in the local stack in the same order as they were declared in the function.

Nevertheless, there is no guarantee.

By the way, some other compiler may warn about the `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` variables, but not `tm_sec` are used without being initialized.

Indeed, the compiler is not aware that these are to be filled by `localtime_r()` function.

We chose this example, since all structure fields are of type `int`.

This would not work if structure fields are 16-bit (`WORD`), like in the case of the `SYSTEMTIME` structure—`GetSystemTime()` will fill them incorrectly (because the local variables are aligned on a 32-bit boundary). Read more about it in next section: “Fields packing in structure” ( [1.24.4 on page 360](#) ).

So, a structure is just a pack of variables laying in one place, side-by-side. We could say that the structure is the instruction to the compiler, directing it to hold variables in one place. By the way, in some very early C versions (before 1972), there were no structures at all [Dennis M. Ritchie, *The development of the C language*, (1993)]<sup>158</sup>.

There is no debugger example here: it is just the same as you already saw.

<sup>158</sup>Also available as <http://go.yurichev.com/17264>

**Structure as an array of 32-bit words**

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    };
}
```

We just *cast* a pointer to structure to an array of *int*'s. And that works! We run the example at 23:51:45 26-July-2014.

```
0x00000002D (45)
0x000000033 (51)
0x000000017 (23)
0x00000001A (26)
0x000000006 (6)
0x000000072 (114)
0x000000006 (6)
0x0000000CE (206)
0x000000001 (1)
```

The variables here are in the same order as they are enumerated in the definition of the structure: [1.332 on page 352](#).

Here is how it gets compiled:

**Listing 1.338: Optimizing GCC 4.8.1**

```
main          proc near
    push    ebp
    mov     ebp, esp
    push    esi
    push    ebx
    and    esp, 0FFFFFFF0h
    sub    esp, 40h
    mov    dword ptr [esp], 0 ; timer
    lea    ebx, [esp+14h]
    call   _time
    lea    esi, [esp+38h]
    mov    [esp+4], ebx      ; tp
    mov    [esp+10h], eax
    lea    eax, [esp+10h]
    mov    [esp], eax       ; timer
    call   _localtime_r
    nop
    lea    esi, [esi+0]     ; NOP
loc_80483D8:
; EBX here is pointer to structure, ESI is the pointer to the end of it.
    mov    eax, [ebx]       ; get 32-bit word from array
    add    ebx, 4           ; next field in structure
    mov    dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
    mov    dword ptr [esp], 1
    mov    [esp+0Ch], eax   ; pass value to printf()
    mov    [esp+8], eax     ; pass value to printf()
    call   __printf_chk
    cmp    ebx, esi         ; meet structure end?
```

## 1.24. STRUCTURES

```
jnz    short loc_80483D8      ; no - load next value then
lea     esp, [ebp-8]
pop    ebx
pop    esi
pop    ebp
retn
main
endp
```

Indeed: the space in the local stack is first treated as a structure, and then it's treated as an array.

It's even possible to modify the fields of the structure through this pointer.

And again, it's dubiously hackish way to do things, not recommended for use in production code.

### Exercise

As an exercise, try to modify (increase by 1) the current month number, treating the structure as an array.

### Structure as an array of bytes

We can go even further. Let's cast the pointer to an array of bytes and dump it:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    };
}
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00
```

We also run this example at 23:51:45 26-July-2014 <sup>159</sup>. The values are just the same as in the previous dump ([1.24.3 on the preceding page](#)), and of course, the lowest byte goes first, because this is a little-endian architecture ([2.7 on page 463](#)).

Listing 1.339: Optimizing GCC 4.8.1

```
main      proc near
push    ebp
mov     ebp, esp
push    edi
push    esi
push    ebx
```

<sup>159</sup>The time and date are the same for demonstration purposes. Byte values are fixed up.

## 1.24. STRUCTURES

```
and    esp, 0FFFFFFF0h
sub    esp, 40h
mov    dword ptr [esp], 0 ; timer
lea    esi, [esp+14h]
call   _time
lea    edi, [esp+38h] ; struct end
mov    [esp+4], esi ; tp
mov    [esp+10h], eax
lea    eax, [esp+10h]
mov    [esp], eax ; timer
call   _localtime_r
lea    esi, [esi+0] ; NOP
; ESI here is the pointer to structure in local stack. EDI is the pointer to structure end.
loc_8048408:
xor    ebx, ebx ; j=0

loc_804840A:
movzx  eax, byte ptr [esi+ebx] ; load byte
add    ebx, 1 ; j=j+1
mov    dword ptr [esp+4], offset a0x02x ; "0x%02X "
mov    dword ptr [esp], 1
mov    [esp+8], eax ; pass loaded byte to printf()
call   __printf_chk
cmp    ebx, 4
jnz    short loc_804840A
; print carriage return character (CR)
mov    dword ptr [esp], 0Ah ; c
add    esi, 4
call   _putchar
cmp    esi, edi ; meet struct end?
jnz    short loc_8048408 ; j=0
lea    esp, [ebp-0Ch]
pop    ebx
pop    esi
pop    edi
pop    ebp
retn
main
endp
```

### 1.24.4 Fields packing in structure

One important thing is fields packing in structures<sup>160</sup>.

Let's take a simple example:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
}

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
```

<sup>160</sup>See also: [Wikipedia: Data structure alignment](#)

## 1.24. STRUCTURES

```
1 tmp.d=4;  
2 f(tmp);  
3};
```

As we see, we have two *char* fields (each is exactly one byte) and two more —*int* (each — 4 bytes).

### x86

This compiles to:

Listing 1.340: MSVC 2012 /GS- /Ob0

```
1 _tmp$ = -16  
2 _main PROC  
3     push    ebp  
4     mov     ebp, esp  
5     sub     esp, 16  
6     mov     BYTE PTR _tmp$[ebp], 1      ; set field a  
7     mov     DWORD PTR _tmp$[ebp+4], 2    ; set field b  
8     mov     BYTE PTR _tmp$[ebp+8], 3    ; set field c  
9     mov     DWORD PTR _tmp$[ebp+12], 4   ; set field d  
10    sub    esp, 16                  ; allocate place for temporary structure  
11    mov    eax, esp  
12    mov    ecx, DWORD PTR _tmp$[ebp]  ; copy our structure to the temporary one  
13    mov    DWORD PTR [eax], ecx  
14    mov    edx, DWORD PTR _tmp$[ebp+4]  
15    mov    DWORD PTR [eax+4], edx  
16    mov    ecx, DWORD PTR _tmp$[ebp+8]  
17    mov    DWORD PTR [eax+8], ecx  
18    mov    edx, DWORD PTR _tmp$[ebp+12]  
19    mov    DWORD PTR [eax+12], edx  
20    call   _f  
21    add    esp, 16  
22    xor    eax, eax  
23    mov    esp, ebp  
24    pop    ebp  
25    ret    0  
26 _main ENDP  
27  
28 _s$ = 8 ; size = 16  
29 ?f@@YAXUs@@@Z PROC ; f  
30     push    ebp  
31     mov     ebp, esp  
32     mov     eax, DWORD PTR _s$[ebp+12]  
33     push    eax  
34     movsx  ecx, BYTE PTR _s$[ebp+8]  
35     push    ecx  
36     mov     edx, DWORD PTR _s$[ebp+4]  
37     push    edx  
38     movsx  eax, BYTE PTR _s$[ebp]  
39     push    eax  
40     push    OFFSET $SG3842  
41     call   _printf  
42     add    esp, 20  
43     pop    ebp  
44     ret    0  
45 ?f@@YAXUs@@@Z ENDP ; f  
46 _TEXT ENDS
```

We pass the structure as a whole, but in fact, as we can see, the structure is being copied to a temporary one (a place in stack is allocated in line 10 for it, and then all 4 fields, one by one, are copied in lines 12 ... 19), and then its pointer (address) is to be passed.

The structure is copied because it's not known whether the `f()` function going to modify the structure or not. If it gets changed, then the structure in `main()` has to remain as it has been.

We could use C/C++ pointers, and the resulting code will be almost the same, but without the copying.

## 1.24. STRUCTURES

As we can see, each field's address is aligned on a 4-byte boundary. That's why each *char* occupies 4 bytes here (like *int*). Why? Because it is easier for the CPU to access memory at aligned addresses and to cache data from it.

However, it is not very economical.

Let's try to compile it with option (`/Zp1`) (`/Zp[n]` pack structures on *n*-byte boundary).

Listing 1.341: MSVC 2012 /GS- /Zp1

```
1 _main    PROC
2     push   ebp
3     mov    ebp, esp
4     sub    esp, 12
5     mov    BYTE PTR _tmp$[ebp], 1      ; set field a
6     mov    DWORD PTR _tmp$[ebp+1], 2   ; set field b
7     mov    BYTE PTR _tmp$[ebp+5], 3   ; set field c
8     mov    DWORD PTR _tmp$[ebp+6], 4   ; set field d
9     sub    esp, 12                  ; allocate place for temporary structure
10    mov    eax, esp
11    mov    ecx, DWORD PTR _tmp$[ebp]  ; copy 10 bytes
12    mov    DWORD PTR [eax], ecx
13    mov    edx, DWORD PTR _tmp$[ebp+4]
14    mov    DWORD PTR [eax+4], edx
15    mov    cx, WORD PTR _tmp$[ebp+8]
16    mov    WORD PTR [eax+8], cx
17    call   _f
18    add    esp, 12
19    xor    eax, eax
20    mov    esp, ebp
21    pop    ebp
22    ret    0
23 _main    ENDP
24
25 _TEXT    SEGMENT
26 _s$ = 8 ; size = 10
27 ?f@@YAXUs@@@Z PROC      ; f
28     push   ebp
29     mov    ebp, esp
30     mov    eax, DWORD PTR _s$[ebp+6]
31     push   eax
32     movsx  ecx, BYTE PTR _s$[ebp+5]
33     push   ecx
34     mov    edx, DWORD PTR _s$[ebp+1]
35     push   edx
36     movsx  eax, BYTE PTR _s$[ebp]
37     push   eax
38     push   OFFSET $SG3842
39     call   _printf
40     add    esp, 20
41     pop    ebp
42     ret    0
43 ?f@@YAXUs@@@Z ENDP      ; f
```

Now the structure takes only 10 bytes and each *char* value takes 1 byte. What does it give to us? Size economy. And as drawback —the CPU accessing these fields slower than it could.

The structure is also copied in `main()`. Not field-by-field, but directly 10 bytes, using three pairs of `MOV`. Why not 4?

The compiler decided that it's better to copy 10 bytes using 3 `MOV` pairs than to copy two 32-bit words and two bytes using 4 `MOV` pairs.

By the way, such copy implementation using `MOV` instead of calling the `memcpy()` function is widely used, because it's faster than a call to `memcpy()` —for short blocks, of course: [3.11.1 on page 511](#).

As it can be easily guessed, if the structure is used in many source and object files, all these must be compiled with the same convention about structures packing.

## 1.24. STRUCTURES

Aside from MSVC `/Zp` option which sets how to align each structure field, there is also the `#pragma pack` compiler option, which can be defined right in the source code. It is available in both MSVC<sup>161</sup> and GCC<sup>162</sup>.

Let's get back to the `SYSTEMTIME` structure that consists of 16-bit fields. How does our compiler know to pack them on 1-byte alignment boundary?

`WinNT.h` file has this:

Listing 1.342: WinNT.h

```
#include "pshpack1.h"
```

And this:

Listing 1.343: WinNT.h

```
#include "pshpack4.h" // 4 byte packing is the default
```

The file `PshPack1.h` looks like:

Listing 1.344: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#endif (_MSC_VER >= 800 && !_defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#endif
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

This tell the compiler how to pack the structures defined after `#pragma pack`.

<sup>161</sup> [MSDN: Working with Packing Structures](#)

<sup>162</sup> [Structure-Packing Pragmas](#)

**OllyDbg + fields are packed by default**

Let's try our example (where the fields are aligned by default (4 bytes)) in OllyDbg:

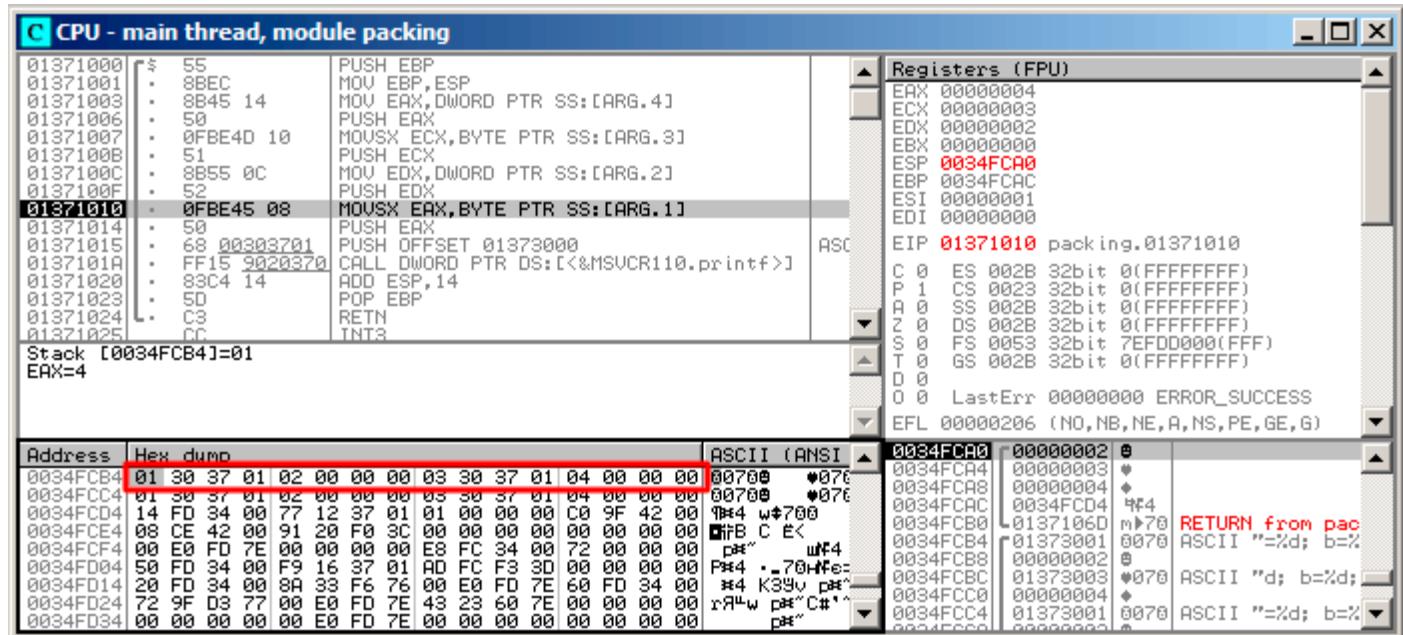


Figure 1.105: OllyDbg: Before `printf()` execution

We see our 4 fields in the data window.

But where do the random bytes (0x30, 0x37, 0x01) come from, that are next to the first (a) and third (c) fields?

By looking at our listing [1.340 on page 361](#), we can see that the first and third fields are `char`, therefore only one byte is written, 1 and 3 respectively (lines 6 and 8).

The remaining 3 bytes of the 32-bit words are not being modified in memory! Hence, random garbage is left there.

This garbage doesn't influence the `printf()` output in any way, because the values for it are prepared using the `MOVsx` instruction, which takes bytes, not words: listing [1.340](#) (lines 34 and 38).

By the way, the `MOVsx` (sign-extending) instruction is used here, because `char` is signed by default in MSVC and GCC. If the `unsigned char` data type or `uint8_t` was used here, `MOVzx` instruction would have been used instead.

## 1.24. STRUCTURES

### OllyDbg + fields aligning on 1 byte boundary

Things are much clearer here: 4 fields occupy 10 bytes and the values are stored side-by-side

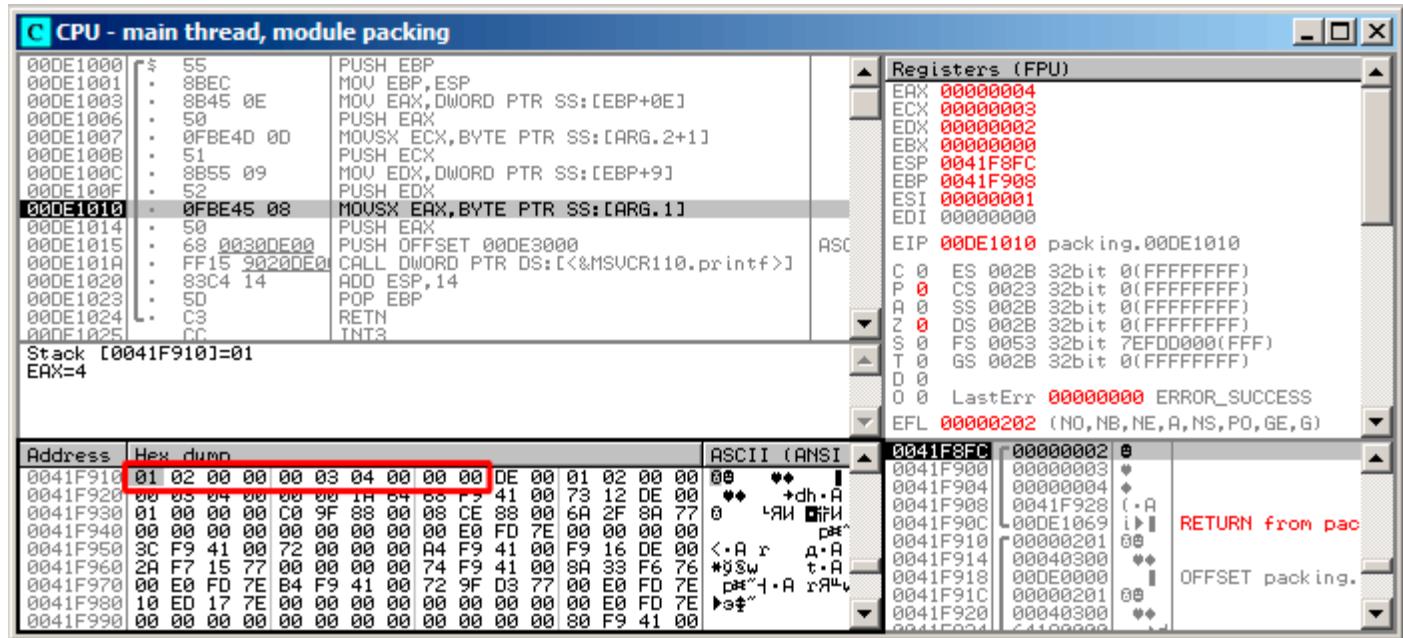


Figure 1.106: OllyDbg: Before `printf()` execution

## ARM

### Optimizing Keil 6/2013 (Thumb mode)

Listing 1.345: Optimizing Keil 6/2013 (Thumb mode)

```
.text:0000003E          exit ; CODE XREF: f+16
.text:0000003E 05 B0    ADD     SP, SP, #0x14
.text:00000040 00 BD    POP    {PC}

.text:00000280          f
.text:00000280
.text:00000280          var_18 = -0x18
.text:00000280          a      = -0x14
.text:00000280          b      = -0x10
.text:00000280          c      = -0xC
.text:00000280          d      = -8
.text:00000280
.text:00000280 0F B5    PUSH   {R0-R3,LR}
.text:00000282 81 B0    SUB    SP, SP, #4
.text:00000284 04 98    LDR    R0, [SP,#16] ; d
.text:00000286 02 9A    LDR    R2, [SP,#8]  ; b
.text:00000288 00 90    STR    R0, [SP]
.text:0000028A 68 46    MOV    R0, SP
.text:0000028C 03 7B    LDRB   R3, [R0,#12] ; c
.text:0000028E 01 79    LDRB   R1, [R0,#4]  ; a
.text:00000290 59 A0    ADR    R0, aADBDCDDD ; "a=%d; b=%d; c=%d; d=%d\n"
.text:00000292 05 F0 AD FF    BL    __2printf
.text:00000296 D2 E6    B     exit
```

As we may recall, here a structure is passed instead of pointer to one, and since the first 4 function arguments in ARM are passed via registers, the structure's fields are passed via R0-R3.

`LDRB` loads one byte from memory and extends it to 32-bit, taking its sign into account. This is similar to `MOVsx` in x86. Here it is used to load fields `a` and `c` from the structure.

## 1.24. STRUCTURES

One more thing we spot easily is that instead of function epilogue, there is jump to another function's epilogue! Indeed, that was quite different function, not related in any way to ours, however, it has exactly the same epilogue (probably because, it hold 5 local variables too ( $5 * 4 = 0x14$ )).

Also it is located nearby (take a look at the addresses).

Indeed, it doesn't matter which epilogue gets executed, if it works just as we need.

Apparently, Keil decides to reuse a part of another function to economize.

The epilogue takes 4 bytes while jump—only 2.

### ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Listing 1.346: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
var_C = -0xC

PUSH {R7,LR}
MOV R7, SP
SUB SP, SP, #4
MOV R9, R1 ; b
MOV R1, R0 ; a
MOVW R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
SXTB R1, R1 ; prepare a
MOVT.W R0, #0
STR R3, [SP,#0xC+var_C] ; place d to stack for printf()
ADD R0, PC ; format-string
SXTB R3, R2 ; prepare c
MOV R2, R9 ; b
BLX _printf
ADD SP, SP, #4
POP {R7,PC}
```

SXTB (*Signed Extend Byte*) is analogous to `MOVVSX` in x86. All the rest—just the same.

### MIPS

Listing 1.347: Optimizing GCC 4.4.5 (IDA)

```
1 f:
2
3 var_18      = -0x18
4 var_10      = -0x10
5 var_4       = -4
6 arg_0       = 0
7 arg_4       = 4
8 arg_8       = 8
9 arg_C       = 0xC
10
11 ; $a0=s.a
12 ; $a1=s.b
13 ; $a2=s.c
14 ; $a3=s.d
15 lui      $gp, (__gnu_local_gp >> 16)
16 addiu   $sp, -0x28
17 la       $gp, (__gnu_local_gp & 0xFFFF)
18 sw       $ra, 0x28+var_4($sp)
19 sw       $gp, 0x28+var_10($sp)
20 ; prepare byte from 32-bit big-endian integer:
21 sra      $t0, $a0, 24
22 move    $v1, $a1
23 ; prepare byte from 32-bit big-endian integer:
24 sra      $v0, $a2, 24
25 lw       $t9, (printf & 0xFFFF)($gp)
26 sw       $a0, 0x28+arg_0($sp)
27 lui      $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d\n"
28 sw       $a3, 0x28+var_18($sp)
```

## 1.24. STRUCTURES

```
29          sw      $a1, 0x28+arg_4($sp)
30          sw      $a2, 0x28+arg_8($sp)
31          sw      $a3, 0x28+arg_C($sp)
32          la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d\n"
33          move   $a1, $t0
34          move   $a2, $v1
35          jalr   $t9
36          move   $a3, $v0 ; branch delay slot
37          lw      $ra, 0x28+var_4($sp)
38          or      $at, $zero ; load delay slot, NOP
39          jr      $ra
40          addiu $sp, 0x28 ; branch delay slot
41
42 $LC0:    .ascii "a=%d; b=%d; c=%d; d=%d\n"<0>
```

Structure fields come in registers \$A0..\$A3 and then get reshuffled into \$A1..\$A3 for `printf()`, while 4th field (from \$A3) is passed via local stack using `SW`.

But there are two SRA (“Shift Word Right Arithmetic”) instructions, which prepare *char* fields. Why?

MIPS is a big-endian architecture by default [2.7 on page 463](#), and the Debian Linux we work in is big-endian as well.

So when byte variables are stored in 32-bit structure slots, they occupy the high 31..24 bits.

And when a *char* variable needs to be extended into a 32-bit value, it must be shifted right by 24 bits.

*char* is a signed type, so an arithmetical shift is used here instead of logical.

### One more word

Passing a structure as a function argument (instead of a passing pointer to structure) is the same as passing all structure fields one by one.

If the structure fields are packed by default, the `f()` function can be rewritten as:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

And that leads to the same code.

### 1.24.5 Nested structures

Now what about situations when one structure is defined inside of another?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
           s.a, s.b, s.c.a, s.c.b, s.d, s.e);
```

## 1.24. STRUCTURES

```
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
}
```

...in this case, both `inner_struct` fields are to be placed between the a,b and d,e fields of the `outer_struct`.

Let's compile (MSVC 2010):

Listing 1.348: Optimizing MSVC 2010 /Ob0

```
$SG2802 DB      'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT      SEGMENT
_s$ = 8
_f      PROC
    mov     eax, DWORD PTR _s$[esp+16]
    movsx  ecx, BYTE PTR _s$[esp+12]
    mov    edx, DWORD PTR _s$[esp+8]
    push   eax
    mov    eax, DWORD PTR _s$[esp+8]
    push   ecx
    mov    ecx, DWORD PTR _s$[esp+8]
    push   edx
    movsx  edx, BYTE PTR _s$[esp+8]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call   _printf
    add    esp, 28
    ret    0
_f      ENDP

_s$ = -24
_main   PROC
    sub    esp, 24
    push   ebx
    push   esi
    push   edi
    mov    ecx, 2
    sub    esp, 24
    mov    eax, esp
    mov    BYTE PTR _s$[esp+60], 1
    mov    ebx, DWORD PTR _s$[esp+60]
    mov    DWORD PTR [eax], ebx
    mov    DWORD PTR [eax+4], ecx
    lea    edx, DWORD PTR [ecx+98]
    lea    esi, DWORD PTR [ecx+99]
    lea    edi, DWORD PTR [ecx+2]
    mov    DWORD PTR [eax+8], edx
    mov    BYTE PTR _s$[esp+76], 3
    mov    ecx, DWORD PTR _s$[esp+76]
    mov    DWORD PTR [eax+12], esi
    mov    DWORD PTR [eax+16], ecx
    mov    DWORD PTR [eax+20], edi
    call   _f
    add    esp, 24
    pop    edi
    pop    esi
    xor    eax, eax
```

## 1.24. STRUCTURES

```
pop    ebx
add    esp, 24
ret    0
_main  ENDP
```

One curious thing here is that by looking onto this assembly code, we do not even see that another structure was used inside of it! Thus, we would say, nested structures are unfolded into *linear* or *one-dimensional* structure.

Of course, if we replace the `struct inner_struct c;` declaration with `struct inner_struct *c;` (thus making a pointer here) the situation will be quite different.

**OllyDbg**

Let's load the example into OllyDbg and take a look at `outer_struct` in memory:

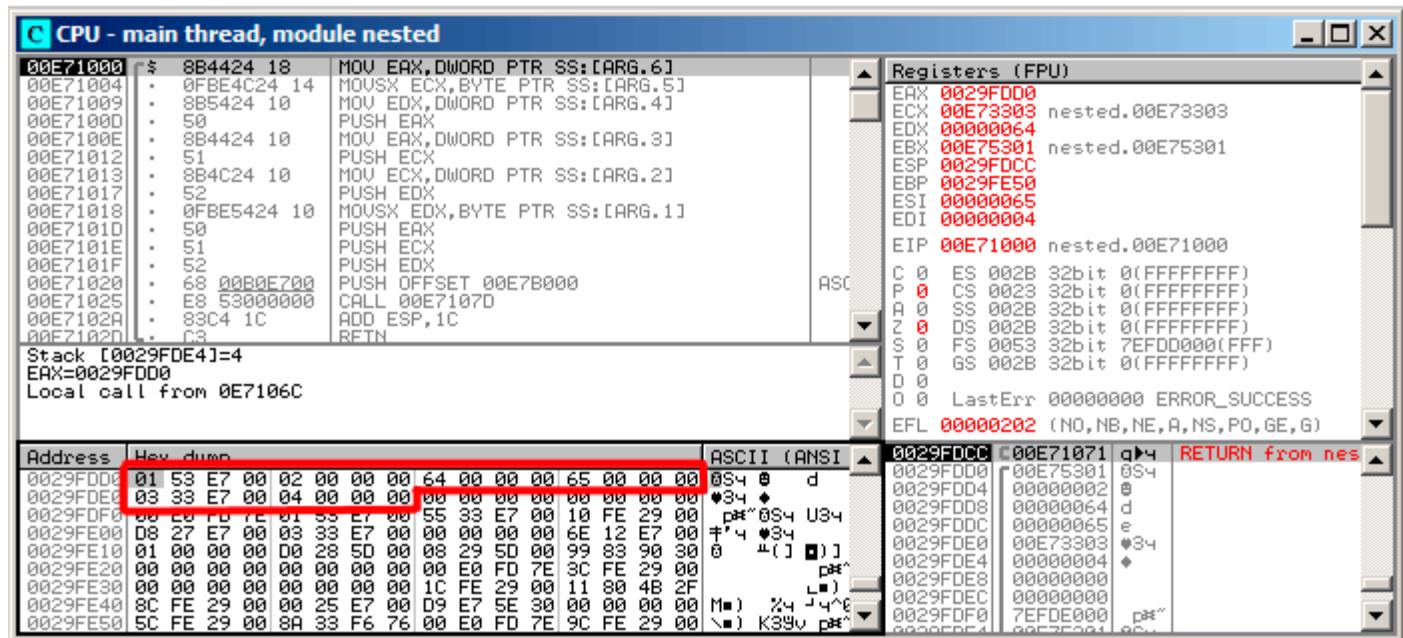


Figure 1.107: OllyDbg: Before `printf()` execution

That's how the values are located in memory:

- (`outer_struct.a`) (byte) 1 + 3 bytes of random garbage;
- (`outer_struct.b`) (32-bit word) 2;
- (`inner_struct.a`) (32-bit word) 0x64 (100);
- (`inner_struct.b`) (32-bit word) 0x65 (101);
- (`outer_struct.d`) (byte) 3 + 3 bytes of random garbage;
- (`outer_struct.e`) (32-bit word) 4.

## 1.24.6 Bit fields in a structure

### CPUID example

The C/C++ language allows to define the exact number of bits for each structure field. It is very useful if one needs to save memory space. For example, one bit is enough for a `bool` variable. But of course, it is not rational if speed is important.

Let's consider the `CPUID`<sup>163</sup> instruction example. This instruction returns information about the current CPU and its features.

If the `EAX` is set to 1 before the instruction's execution, `CPUID` returning this information packed into the `EAX` register:

3:0 (4 bits)	Stepping
7:4 (4 bits)	Model
11:8 (4 bits)	Family
13:12 (2 bits)	Processor Type
19:16 (4 bits)	Extended Model
27:20 (8 bits)	Extended Family

<sup>163</sup>wikipedia

## 1.24. STRUCTURES

MSVC 2010 has `CPUID` macro, but GCC 4.4.1 does not. So let's make this function by ourselves for GCC with the help of its built-in assembler<sup>164</sup>.

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
    asm volatile("cpuid": "=a"(*a), "=b"(*b), "=c"(*c), "=d"(*d): "a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid(1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
}
```

After `CPUID` fills `EAX` / `EBX` / `ECX` / `EDX`, these registers are to be written in the `b[]` array. Then, we have a pointer to the `CPUID_1_EAX` structure and we point it to the value in `EAX` from the `b[]` array.

In other words, we treat a 32-bit `int` value as a structure. Then we read specific bits from the structure.

## MSVC

Let's compile it in MSVC 2008 with `/Ox` option:

Listing 1.349: Optimizing MSVC 2008

```
_b$ = -16 ; size = 16
_main    PROC
        sub    esp, 16
```

<sup>164</sup>[More about internal GCC assembler](#)

## 1.24. STRUCTURES

```

push    ebx
xor    ecx, ecx
mov    eax, 1
cpuid
push    esi
lea    esi, DWORD PTR _b$[esp+24]
mov    DWORD PTR [esi], eax
mov    DWORD PTR [esi+4], ebx
mov    DWORD PTR [esi+8], ecx
mov    DWORD PTR [esi+12], edx

mov    esi, DWORD PTR _b$[esp+24]
mov    eax, esi
and    eax, 15
push    eax
push    OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
call    _printf

mov    ecx, esi
shr    ecx, 4
and    ecx, 15
push    ecx
push    OFFSET $SG15436 ; 'model=%d', 0aH, 00H
call    _printf

mov    edx, esi
shr    edx, 8
and    edx, 15
push    edx
push    OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
call    _printf

mov    eax, esi
shr    eax, 12
and    eax, 3
push    eax
push    OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
call    _printf

mov    ecx, esi
shr    ecx, 16
and    ecx, 15
push    ecx
push    OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
call    _printf

shr    esi, 20
and    esi, 255
push    esi
push    OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
call    _printf
add    esp, 48
pop    esi

xor    eax, eax
pop    ebx

add    esp, 16
ret    0
_main   ENDP

```

The **SHR** instruction shifting the value in **EAX** by the number of bits that must be *skipped*, e.g., we ignore some bits *at the right side*.

The **AND** instruction clears the unneeded bits *on the left*, or, in other words, leaves only those bits in the **EAX** register we need.

Let's load our example into OllyDbg and see, what values are set in EAX/EBX/ECX/EDX after the execution of CPUID:

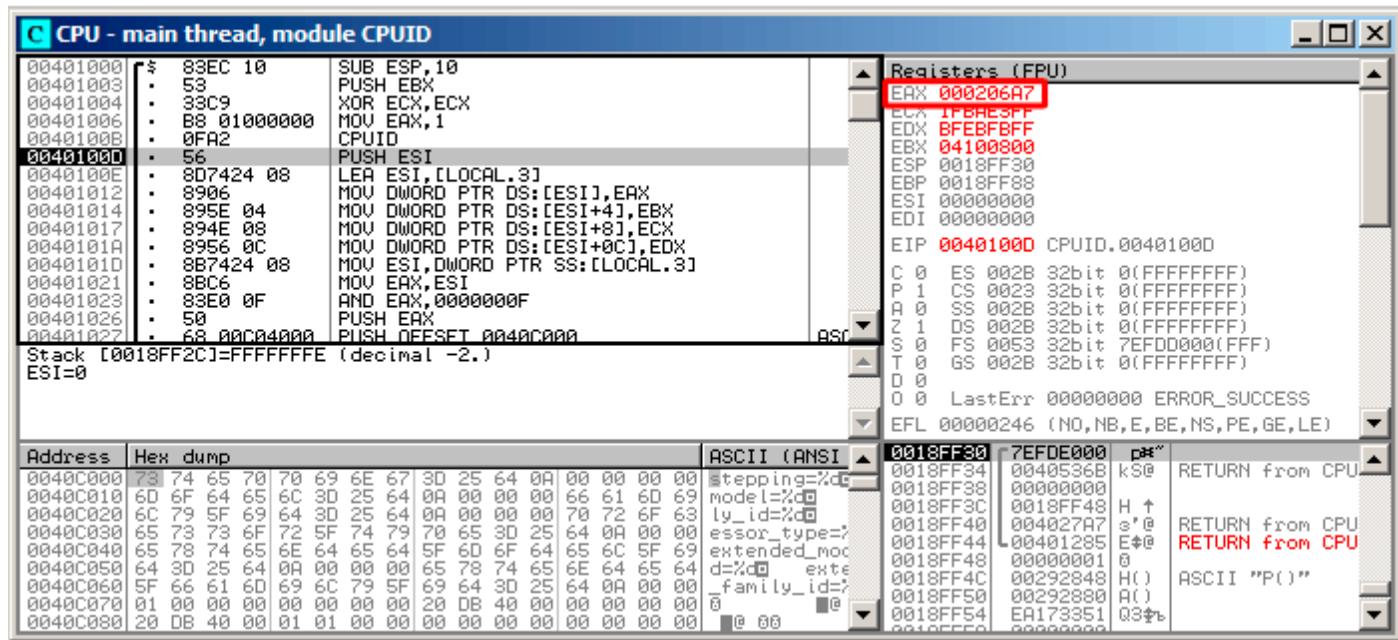


Figure 1.108: OllyDbg: After CPUID execution

EAX has **0x000206A7** (my CPU is Intel Xeon E3-1220).

This is 0b0000000000000000100000011010100111 in binary form.

Here is how the bits are distributed by fields:

field	in binary form	in decimal form
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

Listing 1.350: Console output

```
stepping=7
model=10
family_id=6
processor_type=0
extended_model_id=2
extended_family_id=0
```

## GCC

Let's try GCC 4.4.1 with **-O3** option.

Listing 1.351: Optimizing GCC 4.4.1

```
main proc near ; DATA XREF: _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    push    esi
```

## 1.24. STRUCTURES

```
mov    esi, 1
push   ebx
mov    eax, esi
sub    esp, 18h
cpuid
mov    esi, eax
and    eax, 0Fh
mov    [esp+8], eax
mov    dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
mov    dword ptr [esp], 1
call   __printf_chk
mov    eax, esi
shr    eax, 4
and    eax, 0Fh
mov    [esp+8], eax
mov    dword ptr [esp+4], offset aModelD ; "model=%d\n"
mov    dword ptr [esp], 1
call   __printf_chk
mov    eax, esi
shr    eax, 8
and    eax, 0Fh
mov    [esp+8], eax
mov    dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
mov    dword ptr [esp], 1
call   __printf_chk
mov    eax, esi
shr    eax, 0Ch
and    eax, 3
mov    [esp+8], eax
mov    dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
mov    dword ptr [esp], 1
call   __printf_chk
mov    eax, esi
shr    eax, 10h
shr    esi, 14h
and    eax, 0Fh
and    esi, 0FFh
mov    [esp+8], eax
mov    dword ptr [esp+4], offset aExtended_model ; "extended_model_id=%d\n"
mov    dword ptr [esp], 1
call   __printf_chk
mov    [esp+8], esi
mov    dword ptr [esp+4], offset unk_80486D0
mov    dword ptr [esp], 1
call   __printf_chk
add    esp, 18h
xor    eax, eax
pop    ebx
pop    esi
mov    esp, ebp
pop    ebp
retn
main          endp
```

Almost the same. The only thing worth noting is that GCC somehow combines the calculation of `extended_model_id` and `extended_family_id` into one block, instead of calculating them separately before each `printf()` call.

### Handling float data type as a structure

As we already noted in the section about FPU ([1.19 on page 218](#)), both `float` and `double` types consist of a *sign*, a *significand* (or *fraction*) and an *exponent*. But will we be able to work with these fields directly? Let's try this with `float`.



```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8; // exponent + 0x3FF
    unsigned int sign : 1; // sign bit
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multiply d by  $2^n$  (n here is 2)

    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
}
```

The `float_as_struct` structure occupies the same amount of memory as `float`, i.e., 4 bytes or 32 bits.

Now we are setting the negative sign in the input value and also, by adding 2 to the exponent, we thereby multiply the whole number by  $2^2$ , i.e., by 4.

Let's compile in MSVC 2008 without optimization turned on:

Listing 1.352: Non-optimizing MSVC 2008

```
_t$ = -8 ; size = 4
_f$ = -4 ; size = 4
__in$ = 8 ; size = 4
?f@@YAMM@Z PROC ; f
    push    ebp
    mov     ebp, esp
    sub     esp, 8

    fld     DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push    4
    lea     eax, DWORD PTR _f$[ebp]
    push    eax
    lea     ecx, DWORD PTR _t$[ebp]
    push    ecx
    call    _memcpy
    add    esp, 12

    mov     edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - set minus sign
    mov     DWORD PTR _t$[ebp], edx

    mov     eax, DWORD PTR _t$[ebp]
    shr     eax, 23           ; 00000017H - drop significand
    and     eax, 255          ; 000000ffH - leave here only exponent
```

## 1.24. STRUCTURES

```

add    eax, 2          ; add 2 to it
and    eax, 255        ; 000000ffH
shl    eax, 23         ; 00000017H - shift result to place of bits 30:23
mov    ecx, DWORD PTR _t$[ebp]
and    ecx, -2139095041 ; 807fffffH - drop exponent

; add original value without exponent with new calculated exponent:
or     ecx, eax
mov    DWORD PTR _t$[ebp], ecx

push   4
lea    edx, DWORD PTR _t$[ebp]
push   edx
lea    eax, DWORD PTR _f$[ebp]
push   eax
call   _memcpy
add    esp, 12

fld    DWORD PTR _f$[ebp]

mov    esp, ebp
pop    ebp
ret    0
?f@@YAMM@Z ENDP    ; f

```

A bit redundant. If it was compiled with `/Ox` flag there would be no `memcpy()` call, the `f` variable is used directly. But it is easier to understand by looking at the unoptimized version.

What would GCC 4.4.1 with `-O3` do?

Listing 1.353: Optimizing GCC 4.4.1

```

; f(float)
public _Z1ff
_Z1ff proc near

var_4 = dword ptr -4
arg_0 = dword ptr 8

    push    ebp
    mov     ebp, esp
    sub    esp, 4
    mov    eax, [ebp+arg_0]
    or     eax, 80000000h ; set minus sign
    mov    edx, eax
    and    eax, 807FFFFFFh ; leave only sign and significand in EAX
    shr    edx, 23          ; prepare exponent
    add    edx, 2           ; add 2
    movzx  edx, dl          ; clear all bits except 7:0 in EAX
    shl    edx, 23          ; shift new calculated exponent to its place
    or     eax, edx          ; join new exponent and original value without exponent
    mov    [ebp+var_4], eax
    fld    [ebp+var_4]
    leave
    retn
_Z1ff endp

public main
main  proc near
    push    ebp
    mov     ebp, esp
    and    esp, 0FFFFFFF0h
    sub    esp, 10h
    fld    ds:dword_8048614 ; -4.936
    fstp   qword ptr [esp+8]
    mov    dword ptr [esp+4], offset asc_8048610 ; "%f\n"
    mov    dword ptr [esp], 1
    call   __printf_chk
    xor    eax, eax
    leave

```

## 1.25. UNIONS

```
    retn  
main  endp
```

The `f()` function is almost understandable. However, what is interesting is that GCC was able to calculate the result of `f(1.234)` during compilation despite all this hodge-podge with the structure fields and prepared this argument to `printf()` as precalculated at compile time!

### 1.24.7 Exercises

- <http://challenges.re/71>
- <http://challenges.re/72>

## 1.25 Unions

C/C++ *union* is mostly used for interpreting a variable (or memory block) of one data type as a variable of another data type.

### 1.25.1 Pseudo-random number generator example

If we need float random numbers between 0 and 1, the simplest thing is to use a [PRNG](#) like the Mersenne twister. It produces random unsigned 32-bit values (in other words, it produces random 32 bits). Then we can transform this value to *float* and then divide it by `RAND_MAX` (`0xFFFFFFFF` in our case)—we getting a value in the 0..1 interval.

But as we know, division is slow. Also, we would like to issue as few FPU operations as possible. Can we get rid of the division?

Let's recall what a floating point number consists of: sign bit, significand bits and exponent bits. We just have to store random bits in all significand bits to get a random float number!

The exponent cannot be zero (the floating number is denormalized in this case), so we are storing 0b01111111 to exponent—this means that the exponent is 1. Then we filling the significand with random bits, set the sign bit to 0 (which means a positive number) and voilà. The generated numbers is to be between 1 and 2, so we must also subtract 1.

A very simple linear congruential random numbers generator is used in my example<sup>165</sup>, it produces 32-bit numbers. The [PRNG](#) is initialized with the current time in UNIX timestamp format.

Here we represent the *float* type as an *union*—it is the C/C++ construction that enables us to interpret a piece of memory as different types. In our case, we are able to create a variable of type *union* and then access to it as it is *float* or as it is *uint32\_t*. It can be said, it is just a hack. A dirty one.

The integer [PRNG](#) code is the same as we already considered: [1.23 on page 339](#). So this code in compiled form is omitted.

```
#include <stdio.h>  
#include <stdint.h>  
#include <time.h>  
  
// integer PRNG definitions, data and routines:  
  
// constants from the Numerical Recipes book  
const uint32_t RNG_a=1664525;  
const uint32_t RNG_c=1013904223;  
uint32_t RNG_state; // global variable  
  
void my_srand(uint32_t i)  
{  
    RNG_state=i;  
}
```

<sup>165</sup>the idea was taken from: <http://go.yurichev.com/17308>

## 1.25. UNIONS

```
uint32_t my_rand()
{
    RNG_state=RNG_state*RNG_a+RNG_c;
    return RNG_state;
};

// FPU PRNG definitions and routines:

union uint32_t_float
{
    uint32_t i;
    float f;
};

float float_rand()
{
    union uint32_t_float tmp;
    tmp.i=my_rand() & 0x007fffff | 0x3F800000;
    return tmp.f-1;
};

// test

int main()
{
    my_srand(time(NULL)); // PRNG initialization

    for (int i=0; i<100; i++)
        printf ("%f\n", float_rand());

    return 0;
};
```

## x86

Listing 1.354: Optimizing MSVC 2010

```
$SG4238 DB      '%f', 0aH, 00H

__real@3ff00000000000000 DQ 03ff00000000000000r ; 1

tv130 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
    push    ecx
    call    ?my_rand@@YAIXZ
; EAX=pseudorandom value
    and     eax, 8388607          ; 007fffffH
    or      eax, 1065353216       ; 3f800000H
; EAX=pseudorandom value & 0x007fffff | 0x3f800000
; store it into local stack:
    mov     DWORD PTR _tmp$[esp+4], eax
; reload it as float point number:
    fld     DWORD PTR _tmp$[esp+4]
; subtract 1.0:
    fsub   QWORD PTR __real@3ff00000000000000
; store value we got into local stack and reload it:
    fstp   DWORD PTR tv130[esp+4] ; \ these instructions are redundant
    fld     DWORD PTR tv130[esp+4] ;
    pop    ecx
    ret    0
?float_rand@@YAMXZ ENDP

_main  PROC
    push   esi
    xor    eax, eax
    call   _time
    push   eax
```

## 1.25. UNIONS

```
call    ?my_srand@@YAXI@Z
add     esp, 4
mov     esi, 100
$LL3@main:
call    ?float_rand@@YAMXZ
sub     esp, 8
fstp   QWORD PTR [esp]
push   OFFSET $SG4238
call    _printf
add     esp, 12
dec     esi
jne    SHORT $LL3@main
xor     eax, eax
pop     esi
ret     0
_main  ENDP
```

Function names are so strange here because this example was compiled as C++ and this is name mangling in C++, we will talk about it later: [3.19.1 on page 546](#). If we compile this in MSVC 2012, it uses the SIMD instructions for the FPU, read more about it here: [1.31.5 on page 440](#).

## MIPS

Listing 1.355: Optimizing GCC 4.4.5

```
float_rand:
var_10      = -0x10
var_4       = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu  $sp, -0x20
        la     $gp, (__gnu_local_gp & 0xFFFF)
        sw     $ra, 0x20+var_4($sp)
        sw     $gp, 0x20+var_10($sp)
; call my_rand():
        jal     my_rand
        or     $at, $zero ; branch delay slot, NOP
; $v0=32-bit pseudorandom value
        li     $v1, 0x7FFFFFFF
; $v1=0x7FFFFFFF
        and     $v1, $v0, $v1
; $v1=pseudorandom value & 0x7FFFFFFF
        lui     $a0, 0x3F80
; $a0=0x3F800000
        or     $v1, $a0
; $v1=pseudorandom value & 0x7FFFFFFF | 0x3F800000
; matter of the following instruction is still hard to get:
        lui     $v0, ($LC0 >> 16)
; load 1.0 into $f0:
        lwcl   $f0, $LC0
; move value from $v1 to coprocessor 1 (into register $f2)
; it behaves like bitwise copy, no conversion done:
        mtcl   $v1, $f2
        lw     $ra, 0x20+var_4($sp)
; subtract 1.0. leave result in $f0:
        sub.s  $f0, $f2, $f0
        jr     $ra
        addiu $sp, 0x20 ; branch delay slot

main:
var_18      = -0x18
var_10      = -0x10
var_C       = -0xC
var_8       = -8
var_4       = -4
```

## 1.25. UNIONS

```

lui      $gp, (_gnu_local_gp >> 16)
addiu   $sp, -0x28
la       $gp, (_gnu_local_gp & 0xFFFF)
sw       $ra, 0x28+var_4($sp)
sw       $s2, 0x28+var_8($sp)
sw       $s1, 0x28+var_C($sp)
sw       $s0, 0x28+var_10($sp)
sw       $gp, 0x28+var_18($sp)
lw       $t9, (time & 0xFFFF)($gp)
or       $at, $zero ; load delay slot, NOP
jalr    $t9
move    $a0, $zero ; branch delay slot
lui    $s2, ($LC1 >> 16) # "%f\n"
move    $a0, $v0
la       $s2, ($LC1 & 0xFFFF) # "%f\n"
move    $s0, $zero
jal     my_srand
li       $s1, 0x64 # 'd' ; branch delay slot

loc_104:
jal     float_rand
addiu  $s0, 1
lw      $gp, 0x28+var_18($sp)
; convert value we got from float_rand() to double type (printf() need it):
cvt.d.s $f2, $f0
lw      $t9, (printf & 0xFFFF)($gp)
mfcl   $a3, $f2
mfcl   $a2, $f3
jalr    $t9
move    $a0, $s2
bne    $s0, $s1, loc_104
move    $v0, $zero
lw      $ra, 0x28+var_4($sp)
lw      $s2, 0x28+var_8($sp)
lw      $s1, 0x28+var_C($sp)
lw      $s0, 0x28+var_10($sp)
jr      $ra
addiu  $sp, 0x28 ; branch delay slot

$LC1:    .ascii "%f\n"<0>
$LC0:    .float 1.0

```

There is also an useless `LUI` instruction added for some weird reason. We considered this artifact earlier: [1.19.5 on page 230](#).

## ARM (ARM mode)

Listing 1.356: Optimizing GCC 4.6.3 (IDA)

```

float_rand
        STMFD  SP!, {R3,LR}
        BL     my_rand
; R0=pseudorandom value
        FLDS   S0, =1.0
; S0=1.0
        BIC    R3, R0, #0xFF000000
        BIC    R3, R3, #0x800000
        ORR    R3, R3, #0x3F800000
; R3=pseudorandom value & 0x007fffff | 0x3f800000
; copy from R3 to FPU (register S15).
; it behaves like bitwise copy, no conversion done:
        FMSR   S15, R3
; subtract 1.0 and leave result in S0:
        FSUBS  S0, S15, S0
        LDMFD  SP!, {R3,PC}

flt_5C    DCFS 1.0

```

## 1.25. UNIONS

```

main
    STMFD   SP!, {R4,LR}
    MOV     R0, #0
    BL      time
    BL      my_srand
    MOV     R4, #0x64 ; 'd'

loc_78
    BL      float_rand
; S0=pseudorandom value
    LDR     R0, =aF          ; "%f"
; convert float type value into double type value (printf() will need it):
    FCVTDS D7, S0
; bitwise copy from D7 into R2/R3 pair of registers (for printf()):
    FMRRD  R2, R3, D7
    BL      printf
    SUBS   R4, R4, #1
    BNE    loc_78
    MOV    R0, R4
    LDMFD  SP!, {R4,PC}

aF      DCB "%f",0xA,0

```

We'll also make a dump in objdump and we'll see that the FPU instructions have different names than in IDA. Apparently, IDA and binutils developers used different manuals? Perhaps it would be good to know both instruction name variants.

Listing 1.357: Optimizing GCC 4.6.3 (objdump)

```

00000038 <float_rand>:
38: e92d4008      push   {r3, lr}
3c: ebfffffe      bl     10 <my_rand>
40: ed9f0a05      vldr   s0, [pc, #20] ; 5c <float_rand+0x24>
44: e3c034ff      bic    r3, r0, #-16777216 ; 0xff000000
48: e3c33502      bic    r3, r3, #8388608 ; 0x800000
4c: e38335fe      orr    r3, r3, #1065353216 ; 0x3f800000
50: ee073a90      vmov   s15, r3
54: ee370ac0      vsub.f32 s0, s15, s0
58: e8bd8008      pop    {r3, pc}
5c: 3f800000      svccc  0x00800000

00000000 <main>:
0:  e92d4010      push   {r4, lr}
4:  e3a00000      mov    r0, #0
8:  ebfffffe      bl     0 <time>
c:  ebfffffe      bl     0 <main>
10: e3a04064     mov    r4, #100 ; 0x64
14: ebfffffe      bl     38 <main+0x38>
18: e59f0018      ldr    r0, [pc, #24] ; 38 <main+0x38>
1c: eeb77ac0      vcvt.f64.f32 d7, s0
20: ec532b17      vmov   r2, r3, d7
24: ebfffffe      bl     0 <printf>
28: e2544001      subs   r4, r4, #1
2c: 1afffff8      bne    14 <main+0x14>
30: e1a00004      mov    r0, r4
34: e8bd8010      pop    {r4, pc}
38: 00000000      andeq r0, r0, r0

```

The instructions at 0x5c in `float_rand()` and at 0x38 in `main()` are (pseudo-)random noise.

### 1.25.2 Calculating machine epsilon

The machine epsilon is the smallest possible value the FPU can work with. The more bits allocated for floating point number, the smaller the machine epsilon. It is  $2^{-23} = 1.19e-07$  for `float` and  $2^{-52} = 2.22e-16$  for `double`. See also: [Wikipedia article](#).

It's interesting, how easy it's to calculate the machine epsilon:

```
#include <stdio.h>
#include <stdint.h>

union uint_float
{
    uint32_t i;
    float f;
};

float calculate_machine_epsilon(float start)
{
    union uint_float v;
    v.f=start;
    v.i++;
    return v.f-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
}
```

What we do here is just treat the fraction part of the IEEE 754 number as integer and add 1 to it. The resulting floating number is equal to *starting\_value+machine\_epsilon*, so we just have to subtract the starting value (using floating point arithmetic) to measure, what difference one bit reflects in the single precision (*float*). The *union* serves here as a way to access IEEE 754 number as a regular integer. Adding 1 to it in fact adds 1 to the *fraction* part of the number, however, needless to say, overflow is possible, which will add another 1 to the exponent part.

## x86

Listing 1.358: Optimizing MSVC 2010

```
tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
    fld    DWORD PTR _start$[esp-4]
    fst    DWORD PTR _v$[esp-4]      ; this instruction is redundant
    inc    DWORD PTR _v$[esp-4]
    fsubr   DWORD PTR _v$[esp-4]
    fstp    DWORD PTR tv130[esp-4]    ; \ this instruction pair is also redundant
    fld    DWORD PTR tv130[esp-4]    ; /
    ret    0
_calculate_machine_epsilon ENDP
```

The second **FST** instruction is redundant: there is no necessity to store the input value in the same place (the compiler decided to allocate the *v* variable at the same point in the local stack as the input argument). Then it is incremented with **INC**, as it is a normal integer variable. Then it is loaded into the FPU as a 32-bit IEEE 754 number, **FSUBR** does the rest of job and the resulting value is stored in **ST0**. The last **FSTP / FLD** instruction pair is redundant, but the compiler didn't optimize it out.

## ARM64

Let's extend our example to 64-bit:

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
    uint64_t i;
    double d;
} uint_double;
```

```

double calculate_machine_epsilon(double start)
{
    uint_double v;
    v.d=start;
    v.i++;
    return v.d-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
}

```

ARM64 has no instruction that can add a number to a FPU D-register, so the input value (that came in `D0`) is first copied into `GPR`, incremented, copied to FPU register `D1`, and then subtraction occurs.

Listing 1.359: Optimizing GCC 4.9 ARM64

```

calculate_machine_epsilon:
    fmov    x0, d0      ; load input value of double type into X0
    add     x0, x0, 1    ; X0++
    fmov    d1, x0      ; move it to FPU register
    fsub    d0, d1, d0  ; subtract
    ret

```

See also this example compiled for x64 with SIMD instructions: [1.31.4 on page 439](#).

## MIPS

The new instruction here is `MTC1` (“Move To Coprocessor 1”), it just transfers data from `GPR` to the FPU’s registers.

Listing 1.360: Optimizing GCC 4.4.5 (IDA)

```

calculate_machine_epsilon:
    mfcl    $v0, $f12
    or      $at, $zero ; NOP
    addiu   $v1, $v0, 1
    mtcl    $v1, $f2
    jr      $ra
    sub.s   $f0, $f2, $f12 ; branch delay slot

```

## Conclusion

It’s hard to say whether someone may need this trickery in real-world code, but as was mentioned many times in this book, this example serves well for explaining the IEEE 754 format and *unions* in C/C++.

## 1.26 FSCALE replacement

Agner Fog in his *Optimizing subroutines in assembly language / An optimization guide for x86 platforms* work <sup>166</sup> states that `FSCALE` `FPU` instruction (calculating  $2^n$ ) may be slow on many CPUs, and he offers faster replacement.

Here is my translation of his assembly code to C/C++:

```

#include <stdint.h>
#include <stdio.h>

union uint_float
{
    uint32_t i;

```

<sup>166</sup>[http://www.agner.org/optimize/optimizing\\_assembly.pdf](http://www.agner.org/optimize/optimizing_assembly.pdf)

## 1.26. FSCALE REPLACEMENT

```
        float f;
};

float flt_2n(int N)
{
    union uint_float tmp;

    tmp.i=(N<<23)+0x3f800000;
    return tmp.f;
};

struct float_as_struct
{
    unsigned int fraction : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
};

float flt_2n_v2(int N)
{
    struct float_as_struct tmp;

    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x7f;
    return *(float*)(&tmp);
};

union uint64_double
{
    uint64_t i;
    double d;
};

double dbl_2n(int N)
{
    union uint64_double tmp;

    tmp.i=((uint64_t)N<<52)+0x3ff00000000000000UL;
    return tmp.d;
};

struct double_as_struct
{
    uint64_t fraction : 52;
    int exponent : 11;
    int sign : 1;
};

double dbl_2n_v2(int N)
{
    struct double_as_struct tmp;

    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x3ff;
    return *(double*)(&tmp);
};

int main()
{
    // 2^11=2048
    printf ("%f\n", flt_2n(11));
    printf ("%f\n", flt_2n_v2(11));
    printf ("%lf\n", dbl_2n(11));
    printf ("%lf\n", dbl_2n_v2(11));
};
```

FSCALE instruction may be faster in your environment, but still, it's a good example of *union*'s and the

## 1.27. POINTERS TO FUNCTIONS

fact that exponent is stored in  $2^n$  form, so an input  $n$  value is shifted to the exponent in IEEE 754 encoded number. Then exponent is then corrected with addition of 0x3f800000 or 0x3ff0000000000000.

The same can be done without shift using *struct*, but internally, shift operations still occurred.

### 1.26.1 Fast square root calculation

Another well-known algorithm where *float* is interpreted as integer is fast calculation of square root.

Listing 1.361: DEQuellcode stammt aus der WikipediaThe source code is taken from Wikipedia: <http://go.yurichev.com/17364>

```
* and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     *
     * (((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m
     *
     * where
     *
     * b = exponent bias
     * m = number of mantissa bits
     *
     */
    val_int -= 1 << 23; /* Subtract 2^m. */
    val_int >>= 1; /* Divide by 2. */
    val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

    return *(float*)&val_int; /* Interpret again as float */
}
```

As an exercise, you can try to compile this function and to understand, how it works.

There is also well-known algorithm of fast calculation of  $\frac{1}{\sqrt{x}}$ . Algorithm became popular, supposedly, because it was used in Quake III Arena.

Algorithm description can be found in Wikipedia: <http://go.yurichev.com/17360>.

## 1.27 Pointers to functions

A pointer to a function, as any other pointer, is just the address of the function's start in its code segment. They are often used for calling callback functions<sup>167</sup>.

Well-known examples are:

- `qsort()`<sup>168</sup>, `atexit()`<sup>169</sup> from the standard C library;
- \*NIX OS signals<sup>170</sup>;
- thread starting: `CreateThread()` (win32), `pthread_create()` (POSIX);
- lots of win32 functions, like `EnumChildWindows()`<sup>171</sup>.
- lots of places in the Linux kernel, for example the filesystem driver functions are called via callbacks: <http://go.yurichev.com/17076>

<sup>167</sup>[wikipedia](#)

<sup>168</sup>[wikipedia](#)

<sup>169</sup><http://go.yurichev.com/17073>

<sup>170</sup>[wikipedia](#)

<sup>171</sup>[MSDN](#)

## 1.27. POINTERS TO FUNCTIONS

- The GCC plugin functions are also called via callbacks: <http://go.yurichev.com/17077>
- Another example of function pointers is a table in the “dwm” Linux window manager that defines shortcuts. Each shortcut has a corresponding function to call if a specific key is pressed: [GitHub](#). As we can see, such table is easier to handle than a large switch() statement.

So, the `qsort()` function is an implementation of quicksort in the C/C++ standard library. The function is able to sort anything, any type of data, as long as you have a function to compare these two elements and `qsort()` is able to call it.

The comparison function can be defined as:

```
int (*compare)(const void *, const void *)
```

Let's use the following example:

```
1 /* ex3 Sorting ints with qsort */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int comp(const void * _a, const void * _b)
7 {
8     const int *a=(const int *)_a;
9     const int *b=(const int *)_b;
10
11    if (*a==*b)
12        return 0;
13    else
14        if (*a < *b)
15            return -1;
16        else
17            return 1;
18 }
19
20 int main(int argc, char* argv[])
21 {
22     int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
23     int i;
24
25     /* Sort the array */
26     qsort(numbers,10,sizeof(int),comp) ;
27     for (i=0;i<9;i++)
28         printf("Number = %d\n",numbers[ i ]) ;
29     return 0;
30 }
```

### 1.27.1 MSVC

Let's compile it in MSVC 2010 (some parts were omitted for the sake of brevity) with `/Ox` option:

Listing 1.362: Optimizing MSVC 2010: `/GS- /MD`

```
_a$ = 8                                ; size = 4
_b$ = 12                                ; size = 4
_comp PROC
    mov    eax, DWORD PTR __a$[esp-4]
    mov    ecx, DWORD PTR __b$[esp-4]
    mov    eax, DWORD PTR [eax]
    mov    ecx, DWORD PTR [ecx]
    cmp    eax, ecx
    jne    SHORT $LN4@comp
    xor    eax, eax
    ret    0
$LN4@comp:
    xor    edx, edx
    cmp    eax, ecx
    setge dl
```

## 1.27. POINTERS TO FUNCTIONS

```

    lea    eax, DWORD PTR [edx+edx-1]
    ret    0
_comp ENDP

_numbers$ = -40                                ; size = 40
_argc$ = 8                                     ; size = 4
_argv$ = 12                                    ; size = 4
_main  PROC
    sub    esp, 40                               ; 000000028H
    push   esi
    push   OFFSET _comp
    push   4
    lea    eax, DWORD PTR _numbers$[esp+52]
    push   10                                    ; 00000000aH
    push   eax
    mov    DWORD PTR _numbers$[esp+60], 1892      ; 00000764H
    mov    DWORD PTR _numbers$[esp+64], 45         ; 00000002dH
    mov    DWORD PTR _numbers$[esp+68], 200        ; 0000000c8H
    mov    DWORD PTR _numbers$[esp+72], -98       ; ffffff9eH
    mov    DWORD PTR _numbers$[esp+76], 4087      ; 00000ff7H
    mov    DWORD PTR _numbers$[esp+80], 5          ; ffffffc7H
    mov    DWORD PTR _numbers$[esp+84], -12345     ; 0000043fH
    mov    DWORD PTR _numbers$[esp+88], 1087       ; 00000058H
    mov    DWORD PTR _numbers$[esp+92], 88         ; 0000000000
    mov    DWORD PTR _numbers$[esp+96], -100000     ; fffe7960H
    call   _qsort
    add    esp, 16                               ; 000000010H
...

```

Nothing surprising so far. As a fourth argument, the address of label `_comp` is passed, which is just a place where `comp()` is located, or, in other words, the address of the very first instruction of that function.

How does `qsort()` call it?

Let's take a look at this function, located in MSVCR80.DLL (a MSVC DLL module with C standard library functions):

Listing 1.363: MSVCR80.DLL

```

.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int (__cdecl *)(const void *,
    void *, const void *))
.text:7816CBF0                 public _qsort
.text:7816CBF0 _qsort          proc near
.text:7816CBF0
.text:7816CBF0 lo              = dword ptr -104h
.text:7816CBF0 hi              = dword ptr -100h
.text:7816CBF0 var_FC          = dword ptr -0FCh
.text:7816CBF0 stkptr          = dword ptr -0F8h
.text:7816CBF0 lostk           = dword ptr -0F4h
.text:7816CBF0 histk            = dword ptr -7Ch
.text:7816CBF0 base             = dword ptr 4
.text:7816CBF0 num               = dword ptr 8
.text:7816CBF0 width             = dword ptr 0Ch
.text:7816CBF0 comp              = dword ptr 10h
.text:7816CBF0
.text:7816CBF0 sub    esp, 100h
...
.text:7816CCE0 loc_7816CCE0:          ; CODE XREF: _qsort+B1
.text:7816CCE0 shr    eax, 1
.text:7816CCE2 imul   eax, ebp
.text:7816CCE5 add    eax, ebx
.text:7816CCE7 mov    edi, eax
.text:7816CCE9 push   edi
.text:7816CCEA push   ebx
.text:7816CCEB call   [esp+118h+comp]
.text:7816CCF2 add    esp, 8

```

## 1.27. POINTERS TO FUNCTIONS

```
.text:7816CCF5          test    eax, eax  
.text:7816CCF7          jle     short loc_7816CD04
```

`comp`—is the fourth function argument. Here the control gets passed to the address in the `comp` argument. Before it, two arguments are prepared for `comp()`. Its result is checked after its execution.

That's why it is dangerous to use pointers to functions. First of all, if you call `qsort()` with an incorrect function pointer, `qsort()` may pass control flow to an incorrect point, the process may crash and this bug will be hard to find.

The second reason is that the callback function types must comply strictly, calling the wrong function with wrong arguments of wrong types may lead to serious problems, however, the crashing of the process is not a problem here—the problem is how to determine the reason for the crash—because the compiler may be silent about the potential problems while compiling.

Let's load our example into OllyDbg and set a breakpoint on `comp()`. We can see how the values are compared at the first `comp()` call:

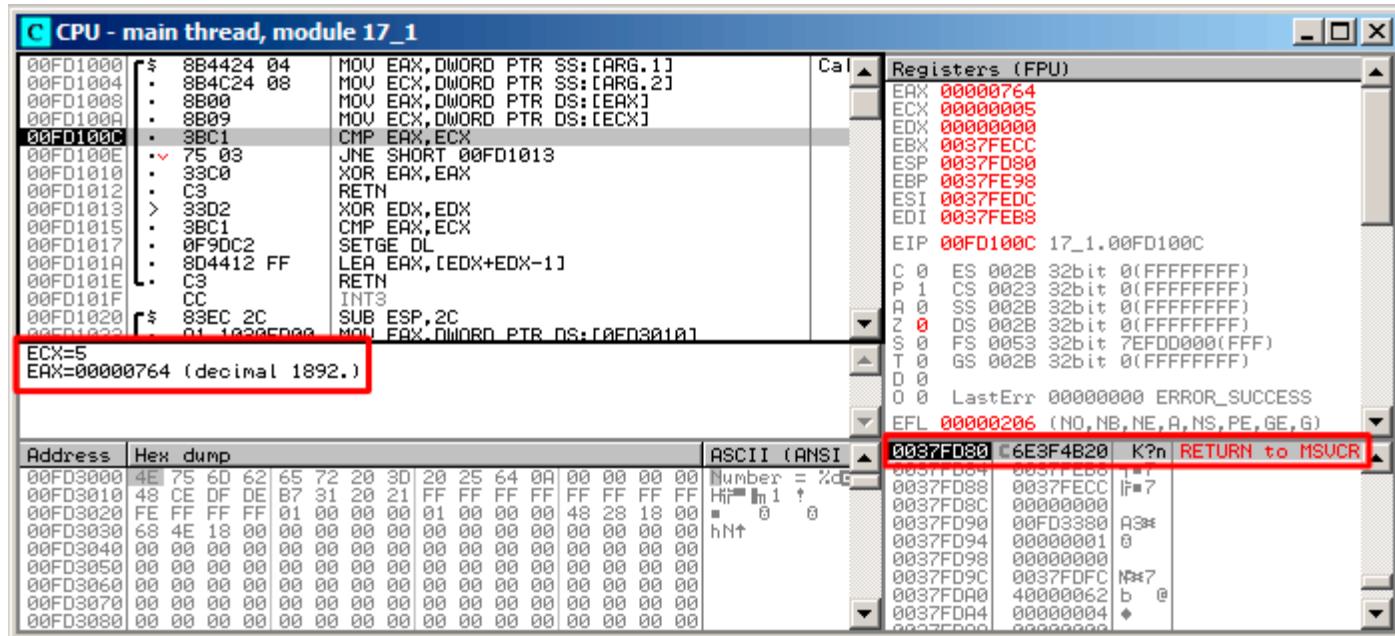


Figure 1.109: OllyDbg: first call of `comp()`

OllyDbg shows the compared values in the window under the code window, for convenience. We can also see that the `SP` points to `RA`, where the `qsort()` function is (located in `MSVCR100.DLL`).

## 1.27. POINTERS TO FUNCTIONS

By tracing (F8) until the RETN instruction and pressing F8 one more time, we return to the `qsort()` function:

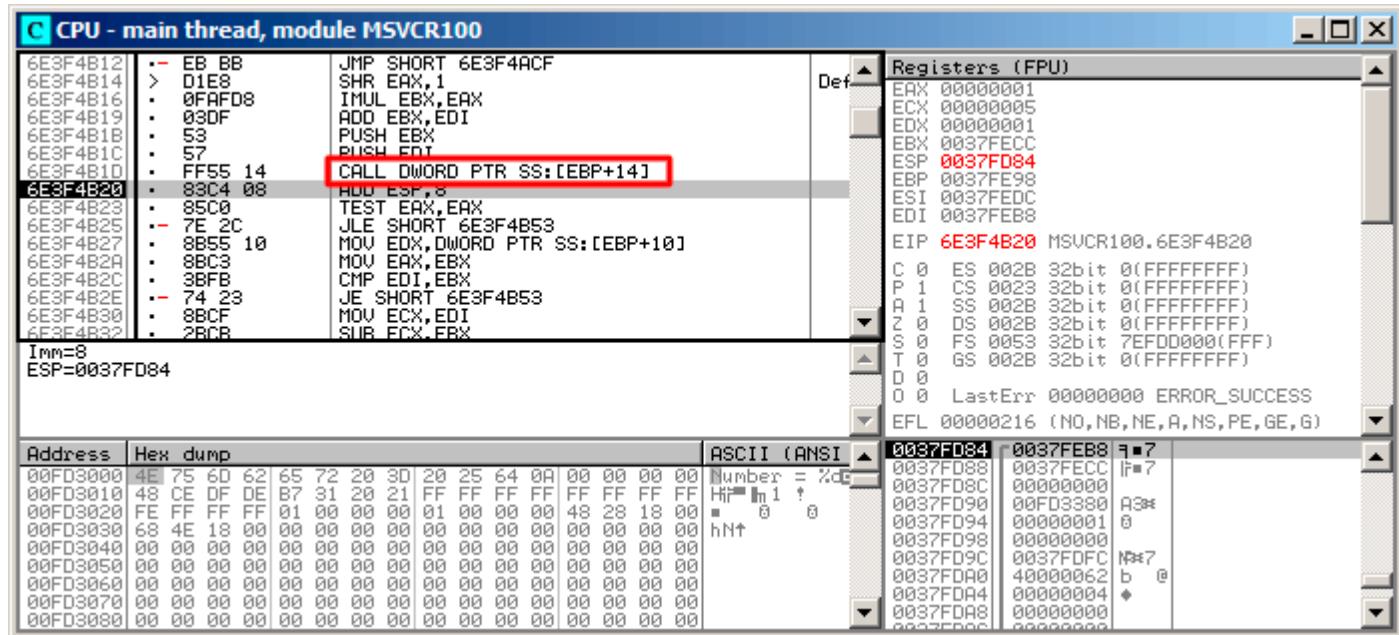


Figure 1.110: OllyDbg: the code in `qsort()` right after `comp()` call

That has been a call to the comparison function.

## 1.27. POINTERS TO FUNCTIONS

Here is also a screenshot of the moment of the second call of `comp()` — now values that have to be compared are different:

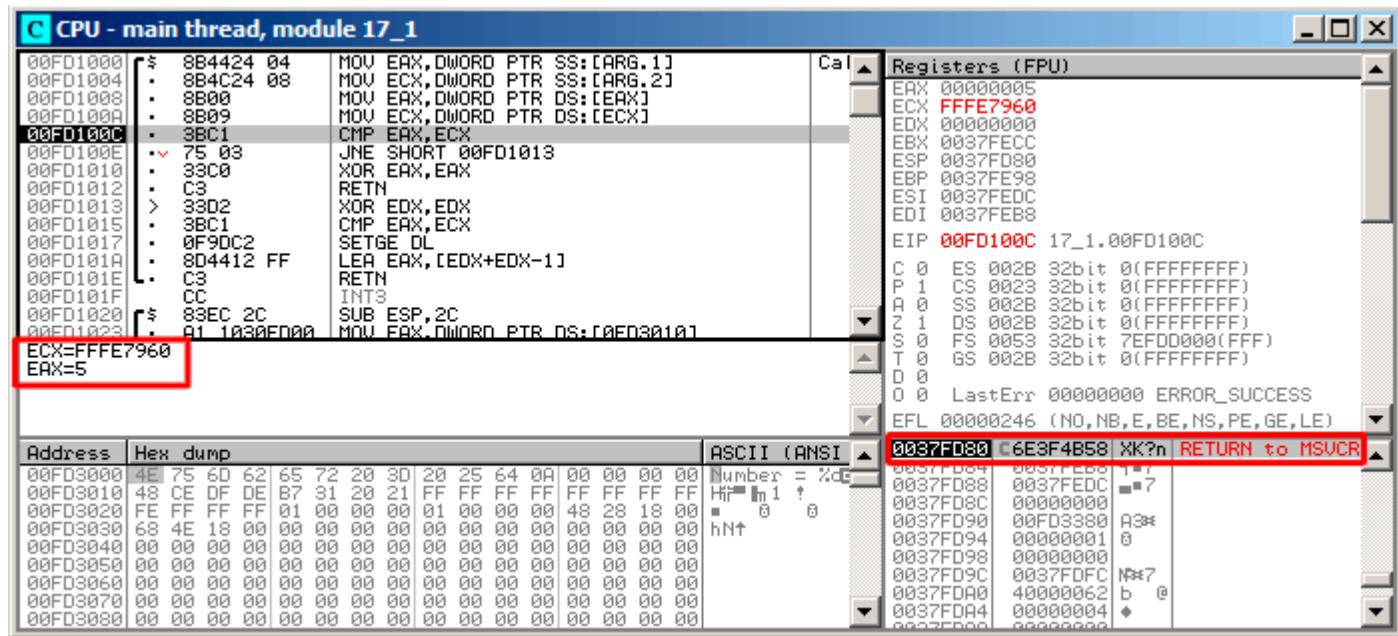


Figure 1.111: OllyDbg: second call of `comp()`

## MSVC + tracer

Let's also see which pairs are compared. These 10 numbers are being sorted: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

We got the address of the first `CMP` instruction in `comp()`, it is `0x0040100C` and we've set a breakpoint on it:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Now we get some information about the registers at the breakpoint:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xffffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Let's filter out `EAX` and `ECX` and we got:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xffffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
```

## 1.27. POINTERS TO FUNCTIONS

```
EAX=0xfffffcfc7 ECX=0x00000005  
EAX=0x000000c8 ECX=0x00000005  
EAX=0xfffffff9e ECX=0x00000005  
EAX=0x00000ff7 ECX=0x00000005  
EAX=0x00000ff7 ECX=0x00000005  
EAX=0xfffffff9e ECX=0x00000005  
EAX=0xfffffff9e ECX=0x00000005  
EAX=0xfffffcfc7 ECX=0xffffe7960  
EAX=0x00000005 ECX=0xfffffcfc7  
EAX=0xfffffff9e ECX=0x00000005  
EAX=0xfffffcfc7 ECX=0xffffe7960  
EAX=0xfffffff9e ECX=0xfffffcfc7  
EAX=0xfffffcfc7 ECX=0xffffe7960  
EAX=0x00000005 ECX=0x00000ff7  
EAX=0x0000002d ECX=0x00000ff7  
EAX=0x0000043f ECX=0x00000ff7  
EAX=0x00000058 ECX=0x00000ff7  
EAX=0x00000764 ECX=0x00000ff7  
EAX=0x000000c8 ECX=0x00000764  
EAX=0x0000002d ECX=0x00000764  
EAX=0x0000043f ECX=0x00000764  
EAX=0x00000058 ECX=0x00000764  
EAX=0x000000c8 ECX=0x00000058  
EAX=0x0000002d ECX=0x000000c8  
EAX=0x0000043f ECX=0x000000c8  
EAX=0x000000c8 ECX=0x00000058  
EAX=0x0000002d ECX=0x000000c8  
EAX=0x0000002d ECX=0x00000058
```

That's 34 pairs. Therefore, the quick sort algorithm needs 34 comparison operations to sort these 10 numbers.

We can also use the tracer's feature to collect all possible register values and show them in [IDA](#).

Let's trace all instructions in `comp()`:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

We get an .idc-script for loading into [IDA](#) and load it:

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare proc near ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0 = dword ptr 4
.text:00401000 arg_4 = dword ptr 8
.text:00401000
.text:00401000     mov    eax, [esp+arg_0] ; [ESP+4]=0x45F7ec..0x45F810(step=4), L"?\\x04?
.text:00401004     mov    ecx, [esp+arg_4] ; [ESP+8]=0x45F7ec..0x45F7f4(step=4), 0x45F7fc
.text:00401008     mov    eax, [eax] ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43F, 0x764, 0xFF
.text:0040100A     mov    ecx, [ecx] ; [ECX]=5, 0x58, 0xc8, 0x764, 0xFF7, 0xFFFFe7960
.text:0040100C     cmp    eax, ecx ; EAX=5, 0x2d, 0x58, 0xc8, 0x43F, 0x764, 0xFF7,
.text:0040100E     jnz    short loc_401013 ; ZF=false
.text:00401010     xor    eax, eax
.text:00401012     retn
.text:00401013 ;
.text:00401013 loc_401013: ; CODE XREF: PtFuncCompare+E↑j
.text:00401013     xor    edx, edx
.text:00401015     cmp    eax, ecx ; EAX=5, 0x2d, 0x58, 0xc8, 0x43F, 0x764, 0xFF7,
.text:00401017     setnl dl ; SF=false,true OF=False
.text:0040101A     lea    eax, [edx+edx-1]
.text:0040101E     retn ; EAX=1, 0xFFFFFFFF
.text:0040101E PtFuncCompare endp
.text:0040101F
```

Figure 1.112: tracer and IDA. N.B.: some values are cut at right

[IDA](#) gave the function a name (`PtFuncCompare`)—because [IDA](#) sees that the pointer to this function is passed to `qsort()`.

We see that the *a* and *b* pointers are pointing to various places in the array, but the step between them is 4, as 32-bit values are stored in the array.

We see that the instructions at `0x401010` and `0x401012` were never executed (so they left as white): indeed, `comp()` has never returned 0, because there no equal elements in the array.

## 1.27.2 GCC

Not a big difference:

Listing 1.364: GCC

```
lea    eax, [esp+40h+var_28]
mov   [esp+40h+var_40], eax
mov   [esp+40h+var_28], 764h
mov   [esp+40h+var_24], 2Dh
mov   [esp+40h+var_20], 0C8h
mov   [esp+40h+var_1C], 0FFFFFF9Eh
mov   [esp+40h+var_18], 0FF7h
mov   [esp+40h+var_14], 5
mov   [esp+40h+var_10], 0FFFFFCFC7h
mov   [esp+40h+var_C], 43Fh
mov   [esp+40h+var_8], 58h
mov   [esp+40h+var_4], 0FFE7960h
mov   [esp+40h+var_34], offset comp
mov   [esp+40h+var_38], 4
mov   [esp+40h+var_3C], 0Ah
```

## 1.27. POINTERS TO FUNCTIONS

```
call    _qsort
```

comp() function:

```
comp          public comp
comp          proc near

arg_0         = dword ptr  8
arg_4         = dword ptr  0Ch

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_4]
        mov     ecx, [ebp+arg_0]
        mov     edx, [eax]
        xor     eax, eax
        cmp     [ecx], edx
        jnz    short loc_8048458
        pop     ebp
        retn

loc_8048458:
        setnl  al
        movzx  eax, al
        lea    eax, [eax+eax-1]
        pop    ebp
        retn
comp          endp
```

The implementation of `qsort()` is located in `libc.so.6` and it is in fact just a wrapper<sup>172</sup> for `qsort_r()`.

In turn, it is calling `quicksort()`, where our defined function is called via a passed pointer:

Listing 1.365: (file `libc.so.6`, glibc version—2.10.1)

```
...
.text:0002DDF6          mov     edx, [ebp+arg_10]
.text:0002DDF9          mov     [esp+4], esi
.text:0002DDFD          mov     [esp], edi
.text:0002DE00          mov     [esp+8], edx
.text:0002DE04          call    [ebp+arg_C]
...
...
```

## GCC + GDB (with source code)

Obviously, we have the C-source code of our example ([1.27 on page 386](#)), so we can set a breakpoint (`b`) on line number (11—the line where the first comparison occurs). We also have to compile the example with debugging information included (`-g`), so the table with addresses and corresponding line numbers is present.

We can also print values using variable names (`p`): the debugging information also has tells us which register and/or local stack element contains which variable.

We can also see the stack (`bt`) and find out that there is some intermediate function `msort_with_tmp()` used in Glibc.

Listing 1.366: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
```

<sup>172</sup>a concept like [thunk function](#)

## 1.27. POINTERS TO FUNCTIONS

Starting program: /home/dennis/polygon./a.out

```
Breakpoint 1, comp (_a=0xfffff0f8, _b=_b@entry=0xfffff0fc) at 17_1.c:11
11          if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xfffff104, _b=_b@entry=0xfffff108) at 17_1.c:11
11          if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xfffff0f8, _b=_b@entry=0xfffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xfffff07c, b=b@entry=0xfffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xfffff0f8, p=0xfffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xfffff07c, b=b@entry=0xfffff0f8, n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xfffff0f8, p=0xfffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xfffff07c, b=b@entry=0xfffff0f8, n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xfffff0f8, p=0xfffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xfffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xfffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

## GCC + GDB (no source code)

But often there is no source code at all, so we can disassemble the `comp()` function (`disas`), find the very first `CMP` instruction and set a breakpoint (`b`) at that address.

At each breakpoint, we are going to dump all register contents (`info registers`). The stack information is also available (`bt`),

but partially: there is no line number information for `comp()`.

Listing 1.367: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
0x0804844d <+0>:    push   ebp
0x0804844e <+1>:    mov    ebp,esp
0x08048450 <+3>:    sub    esp,0x10
0x08048453 <+6>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048456 <+9>:    mov    DWORD PTR [ebp-0x8],eax
0x08048459 <+12>:   mov    eax,DWORD PTR [ebp+0xc]
0x0804845c <+15>:   mov    DWORD PTR [ebp-0x4],eax
0x0804845f <+18>:   mov    eax,DWORD PTR [ebp-0x8]
0x08048462 <+21>:   mov    edx,DWORD PTR [eax]
0x08048464 <+23>:   mov    eax,DWORD PTR [ebp-0x4]
0x08048467 <+26>:   mov    eax,DWORD PTR [eax]
0x08048469 <+28>:   cmp    edx,eax
0x0804846b <+30>:   jne    0x8048474 <comp+39>
0x0804846d <+32>:   mov    eax,0x0
```

## 1.27. POINTERS TO FUNCTIONS

```
0x08048472 <+37>: jmp    0x804848e <comp+65>
0x08048474 <+39>: mov    eax,DWORD PTR [ebp-0x8]
0x08048477 <+42>: mov    edx,DWORD PTR [eax]
0x08048479 <+44>: mov    eax,DWORD PTR [ebp-0x4]
0x0804847c <+47>: mov    eax,DWORD PTR [eax]
0x0804847e <+49>: cmp    edx, eax
0x08048480 <+51>: jge    0x8048489 <comp+60>
0x08048482 <+53>: mov    eax,0xffffffff
0x08048487 <+58>: jmp    0x804848e <comp+65>
0x08048489 <+60>: mov    eax,0x1
0x0804848e <+65>: leave
0x0804848f <+66>: ret
```

End of assembler dump.

(gdb) b \*0x08048469

Breakpoint 1 at 0x8048469

(gdb) run

Starting program: /home/dennis/polygon/.a.out

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

eax	0x2d	45
ecx	0xbffff0f8	-1073745672
edx	0x764	1892
ebx	0xb7fc0000	-1208221696
esp	0xbffffeeb8	0xbffffeeb8
ebp	0xbffffeec8	0xbffffeec8
esi	0xbffff0fc	-1073745668
edi	0xbffff010	-1073745904
eip	0x8048469	0x8048469 <comp+28>
eflags	0x286	[ PF SF IF ]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x33	51

(gdb) c

Continuing.

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

eax	0xff7	4087
ecx	0xbffff104	-1073745660
edx	0xfffffff9e	-98
ebx	0xb7fc0000	-1208221696
esp	0xbffffe58	0xbffffe58
ebp	0xbffffe68	0xbffffe68
esi	0xbffff108	-1073745656
edi	0xbffff010	-1073745904
eip	0x8048469	0x8048469 <comp+28>
eflags	0x282	[ SF IF ]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x33	51

(gdb) c

Continuing.

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

eax	0xfffffff9e	-98
ecx	0xbffff100	-1073745664
edx	0xc8	200
ebx	0xb7fc0000	-1208221696
esp	0xbffffeeb8	0xbffffeeb8
ebp	0xbffffeec8	0xbffffeec8
esi	0xbffff104	-1073745660
edi	0xbffff010	-1073745904

## 1.28. 64-BIT VALUES IN 32-BIT ENVIRONMENT

```
eip          0x8048469      0x8048469 <comp+28>
eflags       0x286      [ PF SF IF ]
cs           0x73       115
ss           0x7b       123
ds           0x7b       123
es           0x7b       123
fs           0x0        0
gs           0x33       51
(gdb) bt
#0 0x08048469 in comp ()
#1 0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
  at msort.c:65
#2 0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#3 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4 0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#5 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6 0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#7 __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <comp>,
  arg=arg@entry=0x0) at msort.c:297
#8 0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9 0x0804850d in main ()
```

### 1.27.3 Danger of pointers to functions

As we can see, `qsort()` function expects a pointer to function which takes two `void*` arguments and returning integer. If you have several comparison functions in your code (one compares string, another—integers, etc), it's very easy to mix them up with each other. You could try to sort array of string using function which compares integers, and compiler will not warn you about bug.

## 1.28 64-bit values in 32-bit environment

In a 32-bit environment, GPR's are 32-bit, so 64-bit values are stored and passed as 32-bit value pairs <sup>173</sup>.

### 1.28.1 Returning of 64-bit value

```
#include <stdint.h>

uint64_t f ()
{
    return 0x1234567890ABCDEF;
```

## x86

In a 32-bit environment, 64-bit values are returned from functions in the `EDX : EAX` register pair.

Listing 1.368: Optimizing MSVC 2010

```
_f    PROC
      mov    eax, -1867788817 ; 90abcdefH
      mov    edx, 305419896   ; 12345678H
      ret    0
_f    ENDP
```

<sup>173</sup>By the way, 32-bit values are passed as pairs in 16-bit environment in the same way: [3.30.4 on page 657](#)

A 64-bit value is returned in the `R0 - R1` register pair (`R1` is for the high part and `R0` for the low part):

Listing 1.369: Optimizing Keil 6/2013 (ARM mode)

```
||f|| PROC
    LDR    r0, |L0.12|
    LDR    r1, |L0.16|
    BX    lr
ENDP

|L0.12| DCD    0x90abcdef
|L0.16| DCD    0x12345678
```

**MIPS**

A 64-bit value is returned in the `V0 - V1` (\$2-\$3) register pair (`V0` (\$2) is for the high part and `V1` (\$3) for the low part):

Listing 1.370: Optimizing GCC 4.4.5 (assembly listing)

```
li    $3, -1867841536    # 0xffffffff90ab0000
li    $2, 305397760      # 0x12340000
ori   $3,$3,0xcdef
j     $31
ori   $2,$2,0x5678
```

Listing 1.371: Optimizing GCC 4.4.5 (IDA)

```
lui   $v1, 0x90AB
lui   $v0, 0x1234
li    $v1, 0x90ABCDEF
jr    $ra
li    $v0, 0x12345678
```

**1.28.2 Arguments passing, addition, subtraction**

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
    return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
    printf ("%lld\n", f_add(12345678901234, 2345678901234));
#else
    printf ("%I64d\n", f_add(12345678901234, 2345678901234));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
    return a-b;
};
```

Listing 1.372: Optimizing MSVC 2012 /Ob1

```

_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f_add PROC
    mov     eax, DWORD PTR _a$[esp-4]
    add     eax, DWORD PTR _b$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    adc     edx, DWORD PTR _b$[esp]
    ret     0
_f_add ENDP

_f_add_test PROC
    push    5461           ; 00001555H
    push    1972608889      ; 75939f79H
    push    2874            ; 00000b3aH
    push    1942892530      ; 73ce2ff_subH
    call    _f_add
    push    edx
    push    eax
    push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
    call    _printf
    add    esp, 28
    ret     0
_f_add_test ENDP

_f_sub PROC
    mov     eax, DWORD PTR _a$[esp-4]
    sub     eax, DWORD PTR _b$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    sbb     edx, DWORD PTR _b$[esp]
    ret     0
_f_sub ENDP

```

We can see in the `f_add_test()` function that each 64-bit value is passed using two 32-bit values, high part first, then low part.

Addition and subtraction occur in pairs as well.

In addition, the low 32-bit part are added first. If carry has been occurred while adding, the `CF` flag is set.

The following `ADC` instruction adds the high parts of the values, and also adds 1 if  $CF = 1$ .

Subtraction also occurs in pairs. The first `SUB` may also turn on the `CF` flag, which is to be checked in the subsequent `SBB` instruction: if the carry flag is on, then 1 is also to be subtracted from the result.

It is easy to see how the `f_add()` function result is then passed to `printf()`.

Listing 1.373: GCC 4.8.1 -O1 -fno-inline

```

_f_add:
    mov     eax, DWORD PTR [esp+12]
    mov     edx, DWORD PTR [esp+16]
    add     eax, DWORD PTR [esp+4]
    adc     edx, DWORD PTR [esp+8]
    ret

_f_add_test:
    sub    esp, 28
    mov     DWORD PTR [esp+8], 1972608889      ; 75939f79H
    mov     DWORD PTR [esp+12], 5461            ; 00001555H
    mov     DWORD PTR [esp], 1942892530        ; 73ce2ff_subH
    mov     DWORD PTR [esp+4], 2874            ; 00000b3aH
    call    _f_add
    mov     DWORD PTR [esp+4], eax
    mov     DWORD PTR [esp+8], edx
    mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\12\0"
    call    _printf

```

## 1.28. 64-BIT VALUES IN 32-BIT ENVIRONMENT

```
    add    esp, 28
    ret

_f_sub:
    mov    eax, DWORD PTR [esp+4]
    mov    edx, DWORD PTR [esp+8]
    sub    eax, DWORD PTR [esp+12]
    sbb    edx, DWORD PTR [esp+16]
    ret
```

GCC code is the same.

## ARM

Listing 1.374: Optimizing Keil 6/2013 (ARM mode)

```
f_add PROC
    ADDS    r0,r0,r2
    ADC     r1,r1,r3
    BX     lr
    ENDP

f_sub PROC
    SUBS    r0,r0,r2
    SBC     r1,r1,r3
    BX     lr
    ENDP

f_add_test PROC
    PUSH   {r4,lr}
    LDR    r2,|L0.68| ; 0x75939f79
    LDR    r3,|L0.72| ; 0x000001555
    LDR    r0,|L0.76| ; 0x73ce2ff2
    LDR    r1,|L0.80| ; 0x000000b3a
    BL     f_add
    POP    {r4,lr}
    MOV    r2,r0
    MOV    r3,r1
    ADR    r0,|L0.84| ; "%I64d\n"
    B     __2printf
    ENDP

|L0.68|
    DCD    0x75939f79
|L0.72|
    DCD    0x000001555
|L0.76|
    DCD    0x73ce2ff2
|L0.80|
    DCD    0x000000b3a
|L0.84|
    DCB    "%I64d\n",0
```

The first 64-bit value is passed in R0 and R1 register pair, the second in R2 and R3 register pair. ARM has the ADC instruction as well (which counts carry flag) and SBC ("subtract with carry"). Important thing: when the low parts are added/subtracted, ADDS and SUBS instructions with -S suffix are used. The -S suffix stands for "set flags", and flags (esp. carry flag) is what consequent ADC / SBC instructions definitely need. Otherwise, instructions without the -S suffix would do the job (ADD and SUB).

## MIPS

Listing 1.375: Optimizing GCC 4.4.5 (IDA)

```
f_add:
; $a0 - high part of a
```

## 1.28. 64-BIT VALUES IN 32-BIT ENVIRONMENT

```

; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
        addu    $v1, $a3, $a1 ; sum up low parts
        addu    $a0, $a2, $a0 ; sum up high parts
; will carry generated while summing up low parts?
; if yes, set $v0 to 1
        sltu    $v0, $v1, $a3
        jr     $ra
; add 1 to high part of result if carry should be generated:
        addu    $v0, $a0 ; branch delay slot
; $v0 - high part of result
; $v1 - low part of result

f_sub:
; $a0 - high part of a
; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
        subu    $v1, $a1, $a3 ; subtract low parts
        subu    $v0, $a0, $a2 ; subtract high parts
; will carry generated while subtracting low parts?
; if yes, set $a0 to 1
        sltu    $a1, $v1
        jr     $ra
; subtract 1 from high part of result if carry should be generated:
        subu    $v0, $a1 ; branch delay slot
; $v0 - high part of result
; $v1 - low part of result

f_add_test:

var_10      = -0x10
var_4       = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu   $sp, -0x20
        la      $gp, (__gnu_local_gp & 0xFFFF)
        sw      $ra, 0x20+var_4($sp)
        sw      $gp, 0x20+var_10($sp)
        lui     $a1, 0x73CE
        lui     $a3, 0x7593
        li      $a0, 0xB3A
        li      $a3, 0x75939F79
        li      $a2, 0x1555
        jal    f_add
        li      $a1, 0x73CE2FF2
        lw      $gp, 0x20+var_10($sp)
        lui     $a0, ($LC0 >> 16) # "%lld\n"
        lw      $t9, (printf & 0xFFFF)($gp)
        lw      $ra, 0x20+var_4($sp)
        la      $a0, ($LC0 & 0xFFFF) # "%lld\n"
        move   $a3, $v1
        move   $a2, $v0
        jr     $t9
        addiu   $sp, 0x20

$LC0:      .ascii "%lld\n<0>

```

MIPS has no flags register, so there is no such information present after the execution of arithmetic operations. So there are no instructions like x86's **ADC** and **SBB**. To know if the carry flag would be set, a comparison (using **SLTU** instruction) also occurs, which sets the destination register to 1 or 0. This 1 or 0 is then added or subtracted to/from the final result.

### 1.28.3 Multiplication, division

```
#include <stdint.h>
```

```

uint64_t f_mul (uint64_t a, uint64_t b)
{
    return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
    return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
    return a % b;
};

```

**x86**

Listing 1.376: Optimizing MSVC 2013 /Ob1

```

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __allmul ; long long multiplication
    pop    ebp
    ret    0
_f_mul ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __aulldiv ; unsigned long long division
    pop    ebp
    ret    0
_f_div ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]

```

## 1.28. 64-BIT VALUES IN 32-BIT ENVIRONMENT

```
push    eax
call    __aullrem ; unsigned long long remainder
pop    ebp
ret    0
_f_rem ENDP
```

Multiplication and division are more complex operations, so usually the compiler embeds calls to a library functions doing that.

These functions are described here: [.5 on page 1026](#).

Listing 1.377: Optimizing GCC 4.8.1 -fno-inline

```
_f_mul:
push    ebx
mov     edx, DWORD PTR [esp+8]
mov     eax, DWORD PTR [esp+16]
mov     ebx, DWORD PTR [esp+12]
mov     ecx, DWORD PTR [esp+20]
imul    ebx, eax
imul    ecx, edx
mul    edx
add    ecx, ebx
add    edx, ecx
pop    ebx
ret

_f_div:
sub    esp, 28
mov     eax, DWORD PTR [esp+40]
mov     edx, DWORD PTR [esp+44]
mov     DWORD PTR [esp+8], eax
mov     eax, DWORD PTR [esp+32]
mov     DWORD PTR [esp+12], edx
mov     edx, DWORD PTR [esp+36]
mov     DWORD PTR [esp], eax
mov     DWORD PTR [esp+4], edx
call    __udivdi3 ; unsigned division
add    esp, 28
ret

_f_rem:
sub    esp, 28
mov     eax, DWORD PTR [esp+40]
mov     edx, DWORD PTR [esp+44]
mov     DWORD PTR [esp+8], eax
mov     eax, DWORD PTR [esp+32]
mov     DWORD PTR [esp+12], edx
mov     edx, DWORD PTR [esp+36]
mov     DWORD PTR [esp], eax
mov     DWORD PTR [esp+4], edx
call    __umoddi3 ; unsigned modulo
add    esp, 28
ret
```

GCC does the expected, but the multiplication code is inlined right in the function, thinking it could be more efficient. GCC has different library function names: [.4 on page 1026](#).

## ARM

Keil for Thumb mode inserts library subroutine calls:

Listing 1.378: Optimizing Keil 6/2013 (Thumb mode)

```
||f_mul|| PROC
PUSH    {r4,lr}
BL     __aeabi_lmul
POP    {r4,pc}
ENDP
```

```
||f_div|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_ulddivmod
    POP    {r4,pc}
    ENDP

||f_rem|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_ulddivmod
    MOVS   r0,r2
    MOVS   r1,r3
    POP    {r4,pc}
    ENDP
```

Keil for ARM mode, on the other hand, is able to produce 64-bit multiplication code:

Listing 1.379: Optimizing Keil 6/2013 (ARM mode)

```
||f_mul|| PROC
    PUSH    {r4,lr}
    UMULL  r12,r4,r0,r2
    MLA    r1,r2,r1,r4
    MLA    r1,r0,r3,r1
    MOV    r0,r12
    POP    {r4,pc}
    ENDP

||f_div|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_ulddivmod
    POP    {r4,pc}
    ENDP

||f_rem|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_ulddivmod
    MOV    r0,r2
    MOV    r1,r3
    POP    {r4,pc}
    ENDP
```

## MIPS

Optimizing GCC for MIPS can generate 64-bit multiplication code, but has to call a library routine for 64-bit division:

Listing 1.380: Optimizing GCC 4.4.5 (IDA)

```
f_mul:
    mult  $a2, $a1
    mflo $v0
    or    $at, $zero ; NOP
    or    $at, $zero ; NOP
    mult  $a0, $a3
    mflo $a0
    addu $v0, $a0
    or    $at, $zero ; NOP
    multu $a3, $a1
    mfhi $a2
    mflo $v1
    jr    $ra
    addu $v0, $a2

f_div:
var_10 = -0x10
var_4  = -4
```

## 1.28. 64-BIT VALUES IN 32-BIT ENVIRONMENT

```
lui      $gp, (_gnu_local_gp >> 16)
addiu   $sp, -0x20
la       $gp, (_gnu_local_gp & 0xFFFF)
sw       $ra, 0x20+var_4($sp)
sw       $gp, 0x20+var_10($sp)
lw       $t9, (_udivdi3 & 0xFFFF)($gp)
or       $at, $zero
jalr    $t9
or       $at, $zero
lw       $ra, 0x20+var_4($sp)
or       $at, $zero
jr       $ra
addiu   $sp, 0x20

f_rem:

var_10 = -0x10
var_4  = -4

lui      $gp, (_gnu_local_gp >> 16)
addiu   $sp, -0x20
la       $gp, (_gnu_local_gp & 0xFFFF)
sw       $ra, 0x20+var_4($sp)
sw       $gp, 0x20+var_10($sp)
lw       $t9, (_umoddi3 & 0xFFFF)($gp)
or       $at, $zero
jalr    $t9
or       $at, $zero
lw       $ra, 0x20+var_4($sp)
or       $at, $zero
jr       $ra
addiu   $sp, 0x20
```

There are a lot of [NOPs](#), probably delay slots filled after the multiplication instruction (it's slower than other instructions, after all).

### 1.28.4 Shifting right

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
    return a>>7;
}
```

## x86

Listing 1.381: Optimizing MSVC 2012 /Ob1

```
_a$ = 8      ; size = 8
_f          PROC
    mov     eax, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    shrd   eax, edx, 7
    shr    edx, 7
    ret    0
_f          ENDP
```

Listing 1.382: Optimizing GCC 4.8.1 -fno-inline

```
_f:
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+4]
    shrd   eax, edx, 7
    shr    edx, 7
    ret
```

## 1.28. 64-BIT VALUES IN 32-BIT ENVIRONMENT

Shifting also occurs in two passes: first the lower part is shifted, then the higher part. But the lower part is shifted with the help of the `SHRD` instruction, it shifts the value of `EAX` by 7 bits, but pulls new bits from `EDX`, i.e., from the higher part. In other words, 64-bit value from `EDX:EAX` register's pair, as a whole, is shifted by 7 bits and lowest 32 bits of result are placed into `EAX`. The higher part is shifted using the much more popular `SHR` instruction: indeed, the freed bits in the higher part must be filled with zeros.

### ARM

ARM doesn't have such instruction as `SHRD` in x86, so the Keil compiler ought to do this using simple shifts and `OR` operations:

Listing 1.383: Optimizing Keil 6/2013 (ARM mode)

```
||f|| PROC
    LSR    r0,r0,#7
    ORR    r0,r0,r1,LSL #25
    LSR    r1,r1,#7
    BX    lr
ENDP
```

Listing 1.384: Optimizing Keil 6/2013 (Thumb mode)

```
||f|| PROC
    LSLS    r2,r1,#25
    LSRS    r0,r0,#7
    ORRS    r0,r0,r2
    LSRS    r1,r1,#7
    BX    lr
ENDP
```

### MIPS

GCC for MIPS follows the same algorithm as Keil does for Thumb mode:

Listing 1.385: Optimizing GCC 4.4.5 (IDA)

```
f:
    sll    $v0, $a0, 25
    srl    $v1, $a1, 7
    or     $v1, $v0, $v1
    jr     $ra
    srl    $v0, $a0, 7
```

## 1.28.5 Converting 32-bit value into 64-bit one

```
#include <stdint.h>

int64_t f (int32_t a)
{
    return a;
};
```

### x86

Listing 1.386: Optimizing MSVC 2012

```
_a$ = 8
_f      PROC
        mov    eax, DWORD PTR _a$[esp-4]
        cdq
        ret    0
_f      ENDP
```

## 1.29. SIMD

Here we also run into necessity to extend a 32-bit signed value into a 64-bit signed one. Unsigned values are converted straightforwardly: all bits in the higher part must be set to 0. But this is not appropriate for signed data types: the sign has to be copied into the higher part of the resulting number.

The `CDQ` instruction does that here, it takes its input value in `EAX`, extends it to 64-bit and leaves it in the `EDX : EAX` register pair. In other words, `CDQ` gets the number sign from `EAX` (by getting the most significant bit in `EAX`), and depending of it, sets all 32 bits in `EDX` to 0 or 1. Its operation is somewhat similar to the `M0VSX` instruction.

### ARM

Listing 1.387: Optimizing Keil 6/2013 (ARM mode)

```
||f|| PROC
    ASR      r1,r0,#31
    BX       lr
ENDP
```

Keil for ARM is different: it just arithmetically shifts right the input value by 31 bits. As we know, the sign bit is **MSB**, and the arithmetical shift copies the sign bit into the “emerged” bits. So after “`ASR r1,r0,#31`”, `R1` containing `0xFFFFFFFF` if the input value has been negative and 0 otherwise. `R1` contains the high part of the resulting 64-bit value. In other words, this code just copies the **MSB** (sign bit) from the input value in `R0` to all bits of the high 32-bit part of the resulting 64-bit value.

### MIPS

GCC for MIPS does the same as Keil did for ARM mode:

Listing 1.388: Optimizing GCC 4.4.5 (IDA)

```
f:
    sra      $v0, $a0, 31
    jr       $ra
    move    $v1, $a0
```

## 1.29 SIMD

**SIMD** is an acronym: *Single Instruction, Multiple Data*.

As its name implies, it processes multiple data using only one instruction.

Like the **FPU**, that **CPU** subsystem looks like a separate processor inside x86.

SIMD began as MMX in x86. 8 new 64-bit registers appeared: MM0-MM7.

Each MMX register can hold 2 32-bit values, 4 16-bit values or 8 bytes. For example, it is possible to add 8 8-bit values (bytes) simultaneously by adding two values in MMX registers.

One simple example is a graphics editor that represents an image as a two dimensional array. When the user changes the brightness of the image, the editor must add or subtract a coefficient to/from each pixel value. For the sake of brevity if we say that the image is grayscale and each pixel is defined by one 8-bit byte, then it is possible to change the brightness of 8 pixels simultaneously.

By the way, this is the reason why the *saturation* instructions are present in SIMD.

When the user changes the brightness in the graphics editor, overflow and underflow are not desirable, so there are addition instructions in SIMD which are not adding anything if the maximum value is reached, etc.

When MMX appeared, these registers were actually located in the FPU’s registers. It was possible to use either FPU or MMX at the same time. One might think that Intel saved on transistors, but in fact the reason of such symbiosis was simpler —older **OSes** that are not aware of the additional CPU registers would not save them at the context switch, but saving the FPU registers. Thus, MMX-enabled CPU + old **OS** + process utilizing MMX features will still work.

## 1.29. SIMD

SSE—is extension of the SIMD registers to 128 bits, now separate from the FPU.

AVX—another extension, to 256 bits.

Now about practical usage.

Of course, this is memory copy routines (`memcpy`), memory comparing (`memcmp`) and so on.

One more example: the DES encryption algorithm takes a 64-bit block and a 56-bit key, encrypt the block and produces a 64-bit result. The DES algorithm may be considered as a very large electronic circuit, with wires and AND/OR/NOT gates.

Bitslice DES<sup>174</sup> —is the idea of processing groups of blocks and keys simultaneously. Let's say, variable of type *unsigned int* on x86 can hold up to 32 bits, so it is possible to store there intermediate results for 32 block-key pairs simultaneously, using 64+56 variables of type *unsigned int*.

There is an utility to brute-force Oracle RDBMS passwords/hashes (ones based on DES), using slightly modified bitslice DES algorithm for SSE2 and AVX—now it is possible to encrypt 128 or 256 block-keys pairs simultaneously.

<http://go.yurichev.com/17313>

### 1.29.1 Vectorization

Vectorization<sup>175</sup> is when, for example, you have a loop taking couple of arrays for input and producing one array. The loop body takes values from the input arrays, does something and puts the result into the output array. Vectorization is to process several elements simultaneously.

Vectorization is not very fresh technology: the author of this textbook saw it at least on the Cray Y-MP supercomputer line from 1988 when he played with its “lite” version Cray Y-MP EL<sup>176</sup>.

For example:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

This fragment of code takes elements from A and B, multiplies them and saves the result into C.

If each array element we have is 32-bit *int*, then it is possible to load 4 elements from A into a 128-bit XMM-register, from B to another XMM-registers, and by executing *PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*) and *PMULHW* (*Multiply Packed Signed Integers and Store High Result*), it is possible to get 4 64-bit products at once.

Thus, loop body execution count is 1024/4 instead of 1024, that is 4 times less and, of course, faster.

### Addition example

Some compilers can do vectorization automatically in simple cases, e.g., Intel C++<sup>177</sup>.

Here is tiny function:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];

    return 0;
};
```

<sup>174</sup><http://go.yurichev.com/17329>

<sup>175</sup>[Wikipedia: vectorization](#)

<sup>176</sup>Remotely. It is installed in the museum of supercomputers: <http://go.yurichev.com/17081>

<sup>177</sup>More about Intel C++ automatic vectorization: [Excerpt: Effective Automatic Vectorization](#)

Let's compile it with Intel C++ 11.1.051 win32:

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

We got (in [IDA](#)):

```
; int __cdecl f(int, int *, int *, int *)
    public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz      = dword ptr 4
ar1     = dword ptr 8
ar2     = dword ptr 0Ch
ar3     = dword ptr 10h

    push    edi
    push    esi
    push    ebx
    push    esi
    mov     edx, [esp+10h+sz]
    test   edx, edx
    jle    loc_15B
    mov     eax, [esp+10h+ar3]
    cmp     edx, 6
    jle    loc_143
    cmp     eax, [esp+10h+ar2]
    jbe    short loc_36
    mov     esi, [esp+10h+ar2]
    sub     esi, eax
    lea     ecx, ds:0[edx*4]
    neg     esi
    cmp     ecx, esi
    jbe    short loc_55

loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
    cmp     eax, [esp+10h+ar2]
    jnb    loc_143
    mov     esi, [esp+10h+ar2]
    sub     esi, eax
    lea     ecx, ds:0[edx*4]
    cmp     esi, ecx
    jb     loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
    cmp     eax, [esp+10h+ar1]
    jbe    short loc_67
    mov     esi, [esp+10h+ar1]
    sub     esi, eax
    neg     esi
    cmp     ecx, esi
    jbe    short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
    cmp     eax, [esp+10h+ar1]
    jnb    loc_143
    mov     esi, [esp+10h+ar1]
    sub     esi, eax
    cmp     esi, ecx
    jb     loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
    mov     edi, eax        ; edi = ar3
    and     edi, 0Fh        ; is ar3 16-byte aligned?
    jz     short loc_9A     ; yes
    test   edi, 3
    jnz    loc_162
    neg     edi
```

## 1.29. SIMD

```
add    edi, 10h
shr    edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
lea    ecx, [edi+4]
cmp    edx, ecx
jl    loc_162
mov    ecx, edx
sub    ecx, edi
and    ecx, 3
neg    ecx
add    ecx, edx
test   edi, edi
jbe    short loc_D6
mov    ebx, [esp+10h+ar2]
mov    [esp+10h+var_10], ecx
mov    ecx, [esp+10h+ar1]
xor    esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
mov    edx, [ecx+esi*4]
add    edx, [ebx+esi*4]
mov    [eax+esi*4], edx
inc    esi
cmp    esi, edi
jb    short loc_C1
mov    ecx, [esp+10h+var_10]
mov    edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
mov    esi, [esp+10h+ar2]
lea    esi, [esi+edi*4] ; is ar2+i*4 16-byte aligned?
test   esi, 0Fh
jz    short loc_109 ; yes!
mov    ebx, [esp+10h+ar1]
mov    esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to ↴
↳ XMM0
padddd xmm1, xmm0
movdqqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
add    edi, 4
cmp    edi, ecx
jb    short loc_ED
jmp    short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
mov    ebx, [esp+10h+ar1]
mov    esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
movdqu xmm0, xmmword ptr [ebx+edi*4]
padddd xmm0, xmmword ptr [esi+edi*4]
movdqqa xmmword ptr [eax+edi*4], xmm0
add    edi, 4
cmp    edi, ecx
jb    short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
; f(int,int *,int *,int *)+164
cmp    ecx, edx
jnb    short loc_15B
mov    esi, [esp+10h+ar1]
mov    edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
mov    ebx, [esi+ecx*4]
add    ebx, [edi+ecx*4]
```

## 1.29. SIMD

```
mov    [eax+ecx*4], ebx
inc    ecx
cmp    ecx, edx
jb     short loc_133
jmp    short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
          ; f(int,int *,int *,int *)+3A ...
mov    esi, [esp+10h+ar1]
mov    edi, [esp+10h+ar2]
xor    ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
mov    ebx, [esi+ecx*4]
add    ebx, [edi+ecx*4]
mov    [eax+ecx*4], ebx
inc    ecx
cmp    ecx, edx
jb     short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
          ; f(int,int *,int *,int *)+129 ...
xor    eax, eax
pop    ecx
pop    ebx
pop    esi
pop    edi
retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
          ; f(int,int *,int *,int *)+9F
xor    ecx, ecx
jmp    short loc_127
?f@@YAHHPAH00@Z endp
```

The SSE2-related instructions are:

- **MOVDQU** (*Move Unaligned Double Quadword*)—just loads 16 bytes from memory into a XMM-register.
- **PADDD** (*Add Packed Integers*)—adds 4 pairs of 32-bit numbers and leaves the result in the first operand. By the way, no exception is raised in case of overflow and no flags are to be set, just the low 32 bits of the result are to be stored. If one of **PADDD**'s operands is the address of a value in memory, then the address must be aligned on a 16-byte boundary. If it is not aligned, an exception will be triggered <sup>178</sup>.
- **MOVDQA** (*Move Aligned Double Quadword*) is the same as **MOVDQU**, but requires the address of the value in memory to be aligned on a 16-bit boundary. If it is not aligned, exception will be raised. **MOVDQA** works faster than **MOVDQU**, but requires aforesaid.

So, these SSE2-instructions are to be executed only in case there are more than 4 pairs to work on and the pointer **ar3** is aligned on a 16-byte boundary.

Also, if **ar2** is aligned on a 16-byte boundary as well, this fragment of code is to be executed:

```
movdqu xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

Otherwise, the value from **ar2** is to be loaded into **XMM0** using **MOVDQU**, which does not require aligned pointer, but may work slower:

```
movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to XMM0
paddd  xmm1, xmm0
movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

In all other cases, non-SSE2 code is to be executed.

<sup>178</sup>More about data alignment: [Wikipedia: Data structure alignment](#)

GCC may also vectorize in simple cases<sup>179</sup>, if the `-O3` option is used and SSE2 support is turned on: `-msse2`.

What we get (GCC 4.4.1):

```
; f(int, int *, int *, int *)
    public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

push        ebp
mov         ebp, esp
push        edi
push        esi
push        ebx
sub         esp, 0Ch
mov         ecx, [ebp+arg_0]
mov         esi, [ebp+arg_4]
mov         edi, [ebp+arg_8]
mov         ebx, [ebp+arg_C]
test        ecx, ecx
jle         short loc_80484D8
cmp         ecx, 6
lea         eax, [ebx+10h]
ja          short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
; f(int,int *,int *,int *)+61 ...
xor         eax, eax
nop
lea         esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
mov         edx, [edi+eax*4]
add         edx, [esi+eax*4]
mov         [ebx+eax*4], edx
add         eax, 1
cmp         eax, ecx
jnz         short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
; f(int,int *,int *,int *)+A5
add         esp, 0Ch
xor         eax, eax
pop         ebx
pop         esi
pop         edi
pop         ebp
ret

align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
test        bl, 0Fh
jnz         short loc_80484C1
lea         edx, [esi+10h]
cmp         ebx, edx
jbe         loc_8048578
```

<sup>179</sup>More about GCC vectorization support: <http://go.yurichev.com/17083>

## 1.29. SIMD

```
loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
    lea    edx, [edi+10h]
    cmp    ebx, edx
    ja     short loc_8048503
    cmp    edi, eax
    jbe    short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
    mov    eax, ecx
    shr    eax, 2
    mov    [ebp+var_14], eax
    shl    eax, 2
    test   eax, eax
    mov    [ebp+var_10], eax
    jz    short loc_8048547
    mov    [ebp+var_18], ecx
    mov    ecx, [ebp+var_14]
    xor    eax, eax
    xor    edx, edx
    nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
    movdqu xmm1, xmmword ptr [edi+eax]
    movdqu xmm0, xmmword ptr [esi+eax]
    add    edx, 1
    paddd xmm0, xmm1
    movdqa xmmword ptr [ebx+eax], xmm0
    add    eax, 10h
    cmp    edx, ecx
    jb    short loc_8048520
    mov    ecx, [ebp+var_18]
    mov    eax, [ebp+var_10]
    cmp    ecx, eax
    jz    short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
    lea    edx, ds:0[eax*4]
    add    esi, edx
    add    edi, edx
    add    ebx, edx
    lea    esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
    mov    edx, [edi]
    add    eax, 1
    add    edi, 4
    add    edx, [esi]
    add    esi, 4
    mov    [ebx], edx
    add    ebx, 4
    cmp    ecx, eax
    jg    short loc_8048558
    add    esp, 0Ch
    xor    eax, eax
    pop    ebx
    pop    esi
    pop    edi
    pop    ebp
    retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
    cmp    eax, esi
    jnb    loc_80484C1
    jmp    loc_80484F8
_Z1fiPiS_S_ endp
```

Almost the same, however, not as meticulously as Intel C++.

## Memory copy example

Let's revisit the simple `memcpy()` example ([1.16.2 on page 196](#)):

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
}
```

And that's what optimizations GCC 4.9.1 did:

Listing 1.389: Optimizing GCC 4.9.1 x64

```
my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block
    test    rdx, rdx
    je     .L41
    lea     rax, [rdi+16]
    cmp    rsi, rax
    lea     rax, [rsi+16]
    setae  cl
    cmp    rdi, rax
    setae  al
    or     cl, al
    je     .L13
    cmp    rdx, 22
    jbe   .L13
    mov    rcx, rsi
    push   rbp
    push   rbx
    neg    rcx
    and   ecx, 15
    cmp    rcx, rdx
    cmova rcx, rdx
    xor    eax, eax
    test   rcx, rcx
    je     .L4
    movzx  eax, BYTE PTR [rsi]
    cmp    rcx, 1
    mov    BYTE PTR [rdi], al
    je     .L15
    movzx  eax, BYTE PTR [rsi+1]
    cmp    rcx, 2
    mov    BYTE PTR [rdi+1], al
    je     .L16
    movzx  eax, BYTE PTR [rsi+2]
    cmp    rcx, 3
    mov    BYTE PTR [rdi+2], al
    je     .L17
    movzx  eax, BYTE PTR [rsi+3]
    cmp    rcx, 4
    mov    BYTE PTR [rdi+3], al
    je     .L18
    movzx  eax, BYTE PTR [rsi+4]
    cmp    rcx, 5
    mov    BYTE PTR [rdi+4], al
    je     .L19
    movzx  eax, BYTE PTR [rsi+5]
    cmp    rcx, 6
    mov    BYTE PTR [rdi+5], al
    je     .L20
    movzx  eax, BYTE PTR [rsi+6]
    cmp    rcx, 7
    mov    BYTE PTR [rdi+6], al
```

```

je      .L21
movzx  eax, BYTE PTR [rsi+7]
cmp    rcx, 8
mov    BYTE PTR [rdi+7], al
je      .L22
movzx  eax, BYTE PTR [rsi+8]
cmp    rcx, 9
mov    BYTE PTR [rdi+8], al
je      .L23
movzx  eax, BYTE PTR [rsi+9]
cmp    rcx, 10
mov   BYTE PTR [rdi+9], al
je      .L24
movzx  eax, BYTE PTR [rsi+10]
cmp    rcx, 11
mov   BYTE PTR [rdi+10], al
je      .L25
movzx  eax, BYTE PTR [rsi+11]
cmp    rcx, 12
mov   BYTE PTR [rdi+11], al
je      .L26
movzx  eax, BYTE PTR [rsi+12]
cmp    rcx, 13
mov   BYTE PTR [rdi+12], al
je      .L27
movzx  eax, BYTE PTR [rsi+13]
cmp    rcx, 15
mov   BYTE PTR [rdi+13], al
jne     .L28
movzx  eax, BYTE PTR [rsi+14]
mov   BYTE PTR [rdi+14], al
mov    eax, 15

.L4:
mov    r10, rdx
lea     r9, [rdx-1]
sub    r10, rcx
lea     r8, [r10-16]
sub    r9, rcx
shr    r8, 4
add    r8, 1
mov    r11, r8
sal     r11, 4
cmp    r9, 14
jbe     .L6
lea     rbp, [rsi+rcx]
xor    r9d, r9d
add    rcx, rdi
xor    ebx, ebx

.L7:
movdqa xmm0, XMMWORD PTR [rbp+0+r9]
add    rbx, 1
movups XMMWORD PTR [rcx+r9], xmm0
add    r9, 16
cmp    rbx, r8
jb     .L7
add    rax, r11
cmp    r10, r11
je      .L1

.L6:
movzx  ecx, BYTE PTR [rsi+rax]
mov    BYTE PTR [rdi+rax], cl
lea     rcx, [rax+1]
cmp    rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+1+rax]
mov    BYTE PTR [rdi+1+rax], cl
lea     rcx, [rax+2]
cmp    rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+2+rax]

```

## 1.29. SIMD

```

mov    BYTE PTR [rdi+2+rax], cl
lea    rcx, [rax+3]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+3+rax]
mov    BYTE PTR [rdi+3+rax], cl
lea    rcx, [rax+4]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+4+rax]
mov    BYTE PTR [rdi+4+rax], cl
lea    rcx, [rax+5]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+5+rax]
mov    BYTE PTR [rdi+5+rax], cl
lea    rcx, [rax+6]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+6+rax]
mov    BYTE PTR [rdi+6+rax], cl
lea    rcx, [rax+7]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+7+rax]
mov    BYTE PTR [rdi+7+rax], cl
lea    rcx, [rax+8]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+8+rax]
mov    BYTE PTR [rdi+8+rax], cl
lea    rcx, [rax+9]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+9+rax]
mov    BYTE PTR [rdi+9+rax], cl
lea    rcx, [rax+10]
cmp    rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+10+rax]
mov   BYTE PTR [rdi+10+rax], cl
lea   rcx, [rax+11]
cmp   rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+11+rax]
mov   BYTE PTR [rdi+11+rax], cl
lea   rcx, [rax+12]
cmp   rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+12+rax]
mov   BYTE PTR [rdi+12+rax], cl
lea   rcx, [rax+13]
cmp   rdx, rcx
jbe   .L1
movzx ecx, BYTE PTR [rsi+13+rax]
mov   BYTE PTR [rdi+13+rax], cl
lea   rcx, [rax+14]
cmp   rdx, rcx
jbe   .L1
movzx edx, BYTE PTR [rsi+14+rax]
mov   BYTE PTR [rdi+14+rax], dl
.L1:
pop    rbx
pop    rbp
.L41:
rep ret
.L13:
xor    eax, eax
.L3:
movzx ecx, BYTE PTR [rsi+rax]

```

## 1.29. SIMD

```
    mov    BYTE PTR [rdi+rax], cl
    add    rax, 1
    cmp    rax, rdx
    jne    .L3
    rep    ret
.L28:
    mov    eax, 14
    jmp    .L4
.L15:
    mov    eax, 1
    jmp    .L4
.L16:
    mov    eax, 2
    jmp    .L4
.L17:
    mov    eax, 3
    jmp    .L4
.L18:
    mov    eax, 4
    jmp    .L4
.L19:
    mov    eax, 5
    jmp    .L4
.L20:
    mov    eax, 6
    jmp    .L4
.L21:
    mov    eax, 7
    jmp    .L4
.L22:
    mov    eax, 8
    jmp    .L4
.L23:
    mov    eax, 9
    jmp    .L4
.L24:
    mov    eax, 10
    jmp   .L4
.L25:
    mov    eax, 11
    jmp   .L4
.L26:
    mov    eax, 12
    jmp   .L4
.L27:
    mov    eax, 13
    jmp   .L4
```

### 1.29.2 SIMD `strlen()` implementation

It has to be noted that the SIMD instructions can be inserted in C/C++ code via special macros<sup>180</sup>. For MSVC, some of them are located in the `intrin.h` file.

It is possible to implement the `strlen()` function<sup>181</sup> using SIMD instructions that works 2-2.5 times faster than the common implementation. This function loads 16 characters into a XMM-register and check each against zero<sup>182</sup>.

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFFF0) == (unsigned int)str;
```

<sup>180</sup>MSDN: MMX, SSE, and SSE2 Intrinsics

<sup>181</sup>`strlen()` —standard C library function for calculating string length

<sup>182</sup>The example is based on source code from: <http://go.yurichev.com/17330>.

## 1.29. SIMD

```
if (str_is_aligned==false)
    return strlen (str);

__m128i xmm0 = _mm_setzero_si128();
__m128i xmm1;
int mask = 0;

for (;;)
{
    xmm1 = _mm_load_si128((__m128i *)s);
    xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
    if ((mask = _mm_movemask_epi8(xmm1)) != 0)
    {
        unsigned long pos;
        _BitScanForward(&pos, mask);
        len += (size_t)pos;

        break;
    }
    s += sizeof(__m128i);
    len += sizeof(__m128i);
};

return len;
}
```

Let's compile it in MSVC 2010 with `/Ox` option:

Listing 1.390: Optimizing MSVC 2010

```
.pos$75552 = -4          ; size = 4
._str$ = 8                ; size = 4
?strlen_sse2@YAIPBD@Z PROC ; strlen_sse2

push    ebp
mov     ebp, esp
and    esp, -16           ; ffffffff0H
mov     eax, DWORD PTR _str$[ebp]
sub    esp, 12             ; 00000000cH
push    esi
mov     esi, eax
and    esi, -16           ; ffffffff0H
xor    edx, edx
mov     ecx, eax
cmp    esi, eax
je     SHORT $LN4@strlen_sse
lea     edx, DWORD PTR [eax+1]
npad   3 ; align next label
$LL11@strlen_sse:
    mov    cl, BYTE PTR [eax]
    inc    eax
    test   cl, cl
    jne    SHORT $LL11@strlen_sse
    sub    eax, edx
    pop    esi
    mov    esp, ebp
    pop    ebp
    ret    0
$LN4@strlen_sse:
    movdqa xmm1, XMMWORD PTR [eax]
    pxor   xmm0, xmm0
    pcmpeqb xmm1, xmm0
    pmovmskb eax, xmm1
    test   eax, eax
    jne    SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa xmm1, XMMWORD PTR [ecx+16]
    add    ecx, 16            ; 0000000010H
    pcmpeqb xmm1, xmm0
    add    edx, 16            ; 0000000010H
```

## 1.29. SIMD

```
pmovmskb eax, xmm1
test    eax, eax
je      SHORT $LL3@strlen_sse
$LN9@strlen_sse:
bsf    eax, eax
mov    ecx, eax
mov    DWORD PTR _pos$75552[esp+16], eax
lea     eax, DWORD PTR [ecx+edx]
pop    esi
mov    esp, ebp
pop    ebp
ret    0
(strlen_sse2@@YAIIPBD@Z ENDP ; strlen_sse2
```

How it works? First of all, we must understand goal of the function. It calculates C-string length, but we can use different terms: it's task is searching for zero byte, and then calculating its position relatively to string start.

First, we check if the `str` pointer is aligned on a 16-byte boundary. If not, we call the generic `strlen()` implementation.

Then, we load the next 16 bytes into the `XMM1` register using `MOVDQA`.

An observant reader might ask, why can't `MOVDQU` be used here since it can load data from the memory regardless pointer alignment?

Yes, it might be done in this way: if the pointer is aligned, load data using `MOVDQA`, if not —use the slower `MOVDQU`.

But here we are may hit another caveat:

In the [Windows NT](#) line of [OS](#) (but not limited to it), memory is allocated by pages of 4 KiB (4096 bytes). Each win32-process has 4 GiB available, but in fact, only some parts of the address space are connected to real physical memory. If the process is accessing an absent memory block, an exception is to be raised. That's how [VM](#) works<sup>183</sup>.

So, a function loading 16 bytes at once may step over the border of an allocated memory block. Let's say that the [OS](#) has allocated 8192 (0x2000) bytes at address 0x008c0000. Thus, the block is the bytes starting from address 0x008c0000 to 0x008c1fff inclusive.

After the block, that is, starting from address 0x008c2000 there is nothing at all, e.g. the [OS](#) not allocated any memory there. Any attempt to access memory starting from that address will raise an exception.

And let's consider the example in which the program is holding a string that contains 5 characters almost at the end of a block, and that is not a crime.

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	random noise
0x008c1fff	random noise

So, in normal conditions the program calls `strlen()`, passing it a pointer to the string `'hello'` placed in memory at address 0x008c1ff8. `strlen()` reads one byte at a time until 0x008c1ffd, where there's a zero byte, and then it stops.

Now if we implement our own `strlen()` reading 16 byte at once, starting at any address, aligned or not, `MOVDQU` may attempt to load 16 bytes at once at address 0x008c1ff8 up to 0x008c2008, and then an exception will be raised. That situation is to be avoided, of course.

So then we'll work only with the addresses aligned on a 16 byte boundary, which in combination with the knowledge that the [OS](#)' page size is usually aligned on a 16-byte boundary gives us some warranty that our function will not read from unallocated memory.

Let's get back to our function.

<sup>183</sup>[wikipedia](#)

## 1.29. SIMD

`_mm_setzero_si128()`—is a macro generating `pxor xmm0, xmm0`—it just clears the `XMM0` register.

`_mm_load_si128()`—is a macro for `MOVDQA`, it just loads 16 bytes from the address into the `XMM1` register.

`_mm_cmpeq_epi8()`—is a macro for `PCMPEQB`, an instruction that compares two XMM-registers bytewise.

And if some byte is equals to the one in the other register, there will be `0xff` at this point in the result or 0 if otherwise.

For example:

`XMM1: 0x11223344556677880000000000000000`

`XMM0: 0x11ab3444007877881111111111111111`

After the execution of `pcmpeqb xmm1, xmm0`, the `XMM1` register contains:

`XMM1: 0xff0000ff0000ffff0000000000000000`

In our case, this instruction compares each 16-byte block with a block of 16 zero-bytes, which has been set in the `XMM0` register by `pxor xmm0, xmm0`.

The next macro is `_mm_movemask_epi8()`—that is the `PMOVMSKB` instruction.

It is very useful with `PCMPEQB`.

`pmovmskb eax, xmm1`

This instruction sets first `EAX` bit to 1 if the most significant bit of the first byte in `XMM1` is 1. In other words, if the first byte of the `XMM1` register is `0xff`, then the first bit of `EAX` is to be 1, too.

If the second byte in the `XMM1` register is `0xff`, then the second bit in `EAX` is to be set to 1. In other words, the instruction is answering the question “which bytes in `XMM1` are `0xff`?” and returns 16 bits in the `EAX` register. The other bits in the `EAX` register are to be cleared.

By the way, do not forget about this quirk of our algorithm. There might be 16 bytes in the input like:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
'h'	'e'	'l'	'l'	'o'	0	garbage				0	garbage				

It is the `'hello'` string, terminating zero, and some random noise in memory.

If we load these 16 bytes into `XMM1` and compare them with the zeroed `XMM0`, we are getting something like <sup>184</sup>:

`XMM1: 0x0000ff00000000000000ff0000000000`

This means that the instruction found two zero bytes, and it is not surprising.

`PMOVMSKB` in our case will set `EAX` to  
`0b001000000100000`.

Obviously, our function must take only the first zero bit and ignore the rest.

The next instruction is `BSF` (*Bit Scan Forward*).

This instruction finds the first bit set to 1 and stores its position into the first operand.

`EAX=0b001000000100000`

After the execution of `bsf eax, eax`, `EAX` contains 5, meaning 1 has been found at the 5th bit position (starting from zero).

MSVC has a macro for this instruction: `_BitScanForward`.

Now it is simple. If a zero byte has been found, its position is added to what we have already counted and now we have the return result.

Almost all.

By the way, it is also has to be noted that the MSVC compiler emitted two loop bodies side by side, for optimization.

<sup>184</sup>An order from `MSB` to `LSB`<sup>185</sup> is used here.

## 1.30. 64 BITS

By the way, SSE 4.2 (that appeared in Intel Core i7) offers more instructions where these string manipulations might be even easier: <http://go.yurichev.com/17331>

# 1.30 64 bits

## 1.30.1 x86-64

It is a 64-bit extension to the x86 architecture.

From the reverse engineer's perspective, the most important changes are:

- Almost all registers (except FPU and SIMD) were extended to 64 bits and got a R- prefix. 8 additional registers were added. Now GPR's are: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

It is still possible to access the *older* register parts as usual. For example, it is possible to access the lower 32-bit part of the RAX register using EAX:

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RAX <sup>x64</sup>							
EAX							
AX							
AH AL							

The new R8-R15 registers also have their *lower parts*: R8D-R15D (lower 32-bit parts), R8W-R15W (lower 16-bit parts), R8L-R15L (lower 8-bit parts).

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
R8							
R8D							
R8W							
R8L							

The number of SIMD registers was doubled from 8 to 16: XMM0 - XMM15.

- In Win64, the function calling convention is slightly different, somewhat resembling fastcall ([6.1.3 on page 734](#)). The first 4 arguments are stored in the RCX, RDX, R8, R9 registers, the rest—in the stack. The **caller** function must also allocate 32 bytes so the **callee** may save there 4 first arguments and use these registers for its own needs. Short functions may use arguments just from registers, but larger ones may save their values on the stack.

System V AMD64 ABI (Linux, \*BSD, Mac OS X)[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>186</sup> also somewhat resembles fastcall, it uses 6 registers RDI, RSI, RDX, RCX, R8, R9 for the first 6 arguments. All the rest are passed via the stack.

See also the section on calling conventions ([6.1 on page 733](#)).

- The C/C++ *int* type is still 32-bit for compatibility.
- All pointers are 64-bit now.

Since now the number of registers is doubled, the compilers have more space for maneuvering called **register allocation**. For us this implies that the emitted code containing less number of local variables.

For example, the function that calculates the first S-box of the DES encryption algorithm processes 32/64/128/256 values at once (depending on **DES\_type** type (uint32, uint64, SSE2 or AVX)) using the bitslice DES method (read more about this technique here ([1.29 on page 408](#))):

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
```

<sup>186</sup>Also available as <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

```

/*
 * Produced by Matthew Kwan - March 1998
 */

#ifndef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
    x30 = x9 ^ x24;
    x31 = x18 & ~x30;
    x32 = a2 & x31;
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
    x37 = x18 & ~x36;
}

```

### 1.30. 64 BITS

```
x38 = a2 | x3;
x39 = x37 ^ x38;
x40 = a3 | x31;
x41 = x24 & ~x37;
x42 = x41 | x3;
x43 = x42 & ~a2;
x44 = x40 ^ x43;
x45 = a1 & ~x44;
x46 = x39 ^ ~x45;
*out1 ^= x46;
x47 = x33 & ~x9;
x48 = x47 ^ x39;
x49 = x4 ^ x36;
x50 = x49 & ~x5;
x51 = x42 | x18;
x52 = x51 ^ a5;
x53 = a2 & ~x52;
x54 = x50 ^ x53;
x55 = a1 | x54;
x56 = x48 ^ ~x55;
*out3 ^= x56;
}
```

There are a lot of local variables. Of course, not all those going into the local stack. Let's compile it with MSVC 2008 with `/Ox` option:

Listing 1.391: Optimizing MSVC 2008

```
PUBLIC    _s1
; Function compile flags: /Ogtpy
TEXT      SEGMENT
_x6$ = -20          ; size = 4
_x3$ = -16          ; size = 4
_x1$ = -12          ; size = 4
_x8$ = -8           ; size = 4
_x4$ = -4           ; size = 4
_a1$ = 8            ; size = 4
_a2$ = 12           ; size = 4
_a3$ = 16           ; size = 4
_x33$ = 20          ; size = 4
_x7$ = 20            ; size = 4
_a4$ = 20           ; size = 4
_a5$ = 24           ; size = 4
tv326 = 28          ; size = 4
_x36$ = 28          ; size = 4
_x28$ = 28          ; size = 4
_a6$ = 28           ; size = 4
_out1$ = 32          ; size = 4
_x24$ = 36           ; size = 4
_out2$ = 36          ; size = 4
_out3$ = 40          ; size = 4
_out4$ = 44          ; size = 4
_s1    PROC
    sub    esp, 20          ; 00000014H
    mov    edx, DWORD PTR _a5$[esp+16]
    push   ebx
    mov    ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov    esi, DWORD PTR _a3$[esp+28]
    push   edi
    mov    edi, ebx
    not    edi
    mov    ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov    ecx, edx
    not    ecx
    and    ebp, esi
    mov    eax, ecx
    and    eax, esi
```

```

and    ecx, ebx
mov    DWORD PTR _x1$[esp+36], eax
xor    eax, ebx
mov    esi, ebp
or     esi, edx
mov    DWORD PTR _x4$[esp+36], esi
and    esi, DWORD PTR _a6$[esp+32]
mov    DWORD PTR _x7$[esp+32], ecx
mov    edx, esi
xor    edx, eax
mov    DWORD PTR _x6$[esp+36], edx
mov    edx, DWORD PTR _a3$[esp+32]
xor    edx, ebx
mov    ebx, esi
xor    ebx, DWORD PTR _a5$[esp+32]
mov    DWORD PTR _x8$[esp+36], edx
and    ebx, edx
mov    ecx, edx
mov    edx, ebx
xor    edx, ebp
or     edx, DWORD PTR _a6$[esp+32]
not    ecx
and    ecx, DWORD PTR _a6$[esp+32]
xor    edx, edi
mov    edi, edx
or     edi, DWORD PTR _a2$[esp+32]
mov    DWORD PTR _x3$[esp+36], ebp
mov    ebp, DWORD PTR _a2$[esp+32]
xor    edi, ebx
and    edi, DWORD PTR _a1$[esp+32]
mov    ebx, ecx
xor    ebx, DWORD PTR _x7$[esp+32]
not    edi
or     ebx, ebp
xor    edi, ebx
mov    ebx, edi
mov    edi, DWORD PTR _out2$[esp+32]
xor    ebx, DWORD PTR [edi]
not    eax
xor    ebx, DWORD PTR _x6$[esp+36]
and    eax, edx
mov    DWORD PTR [edi], ebx
mov    ebx, DWORD PTR _x7$[esp+32]
or     ebx, DWORD PTR _x6$[esp+36]
mov    edi, esi
or     edi, DWORD PTR _x1$[esp+36]
mov    DWORD PTR _x28$[esp+32], ebx
xor    edi, DWORD PTR _x8$[esp+36]
mov    DWORD PTR _x24$[esp+32], edi
xor    edi, ecx
not    edi
and    edi, edx
mov    ebx, edi
and    ebx, ebp
xor    ebx, DWORD PTR _x28$[esp+32]
xor    ebx, eax
not    eax
mov    DWORD PTR _x33$[esp+32], ebx
and    ebx, DWORD PTR _a1$[esp+32]
and    eax, ebp
xor    eax, ebx
mov    ebx, DWORD PTR _out4$[esp+32]
xor    eax, DWORD PTR [ebx]
xor    eax, DWORD PTR _x24$[esp+32]
mov    DWORD PTR [ebx], eax
mov    eax, DWORD PTR _x28$[esp+32]
and    eax, DWORD PTR _a3$[esp+32]
mov    ebx, DWORD PTR _x3$[esp+36]
or     edi, DWORD PTR _a3$[esp+32]
mov    DWORD PTR _x36$[esp+32], eax

```

### 1.30. 64 BITS

```
not    eax
and    eax, edx
or     ebx, ebp
xor    ebx, eax
not    eax
and    eax, DWORD PTR _x24$[esp+32]
not    ebp
or     eax, DWORD PTR _x3$[esp+36]
not    esi
and    ebp, eax
or     eax, edx
xor    eax, DWORD PTR _a5$[esp+32]
mov    edx, DWORD PTR _x36$[esp+32]
xor    edx, DWORD PTR _x4$[esp+36]
xor    ebp, edi
mov    edi, DWORD PTR _out1$[esp+32]
not    eax
and    eax, DWORD PTR _a2$[esp+32]
not    ebp
and    ebp, DWORD PTR _a1$[esp+32]
and    edx, esi
xor    eax, edx
or     eax, DWORD PTR _a1$[esp+32]
not    ebp
xor    ebp, DWORD PTR [edi]
not    ecx
and    ecx, DWORD PTR _x33$[esp+32]
xor    ebp, ebx
not    eax
mov    DWORD PTR [edi], ebp
xor    eax, ecx
mov    ecx, DWORD PTR _out3$[esp+32]
xor    eax, DWORD PTR [ecx]
pop    edi
pop    esi
xor    eax, ebx
pop    ebp
mov    DWORD PTR [ecx], eax
pop    ebx
add    esp, 20
ret    0
_s1    ENDP
```

5 variables were allocated in the local stack by the compiler.

Now let's try the same thing in the 64-bit version of MSVC 2008:

Listing 1.392: Optimizing MSVC 2008

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1    PROC
$LN3:
    mov    QWORD PTR [rsp+24], rbx
    mov    QWORD PTR [rsp+32], rbp
    mov    QWORD PTR [rsp+16], rdx
    mov    QWORD PTR [rsp+8], rcx
    push   rsi
    push   rdi
    push   r12
    push   r13
    push   r14
```

### 1.30. 64 BITS

```
push    r15
mov     r15, QWORD PTR a5$[rsp]
mov     rcx, QWORD PTR a6$[rsp]
mov     rbp, r8
mov     r10, r9
mov     rax, r15
mov     rdx, rbp
not    rax
xor    rdx, r9
not    r10
mov     r11, rax
and    rax, r9
mov     rsi, r10
mov     QWORD PTR x36$1$[rsp], rax
and    r11, r8
and    rsi, r8
and    r10, r15
mov     r13, rdx
mov     rbx, r11
xor    rbx, r9
mov     r9, QWORD PTR a2$[rsp]
mov     r12, rsi
or     r12, r15
not    r13
and    r13, rcx
mov     r14, r12
and    r14, rcx
mov     rax, r14
mov     r8, r14
xor    r8, rbx
xor    rax, r15
not    rbx
and    rax, rdx
mov     rdi, rax
xor    rdi, rsi
or     rdi, rcx
xor    rdi, r10
and    rbx, rdi
mov     rcx, rdi
or     rcx, r9
xor    rcx, rax
mov     rax, r13
xor    rax, QWORD PTR x36$1$[rsp]
and    rcx, QWORD PTR a1$[rsp]
or     rax, r9
not    rcx
xor    rcx, rax
mov     rax, QWORD PTR out2$[rsp]
xor    rcx, QWORD PTR [rax]
xor    rcx, r8
mov     QWORD PTR [rax], rcx
mov     rax, QWORD PTR x36$1$[rsp]
mov     rcx, r14
or     rax, r8
or     rcx, r11
mov     r11, r9
xor    rcx, rdx
mov     QWORD PTR x36$1$[rsp], rax
mov     r8, rsi
mov     rdx, rcx
xor    rdx, r13
not    rdx
and    rdx, rdi
mov     r10, rdx
and    r10, r9
xor    r10, rax
xor    r10, rbx
not    rbx
and    rbx, r9
mov     rax, r10
```

### 1.30. 64 BITS

```
and    rax, QWORD PTR a1$[rsp]
xor    rbx, rax
mov    rax, QWORD PTR out4$[rsp]
xor    rbx, QWORD PTR [rax]
xor    rbx, rcx
mov    QWORD PTR [rax], rbx
mov    rbx, QWORD PTR x36$1$[rsp]
and    rbp, rbp
mov    r9, rbx
not    r9
and    r9, rdi
or     r8, r11
mov    rax, QWORD PTR out1$[rsp]
xor    r8, r9
not    r9
and    r9, rcx
or     rdx, rbp
mov    rbp, QWORD PTR [rsp+80]
or     r9, rsi
xor    rbx, r12
mov    rcx, r11
not    rcx
not    r14
not    r13
and    rcx, r9
or     r9, rdi
and    rbx, r14
xor    r9, r15
xor    rcx, rdx
mov    rdx, QWORD PTR a1$[rsp]
not    r9
not    rcx
and    r13, r10
and    r9, r11
and    rcx, rdx
xor    r9, rbx
mov    rbx, QWORD PTR [rsp+72]
not    rcx
xor    rcx, QWORD PTR [rax]
or     r9, rdx
not    r9
xor    rcx, r8
mov    QWORD PTR [rax], rcx
mov    rax, QWORD PTR out3$[rsp]
xor    r9, r13
xor    r9, QWORD PTR [rax]
xor    r9, r8
mov    QWORD PTR [rax], r9
pop    r15
pop    r14
pop    r13
pop    r12
pop    rdi
pop    rsi
ret    0
s1    ENDP
```

Nothing was allocated in the local stack by the compiler, `x36` is synonym for `a5`.

By the way, there are CPUs with much more [GPR's](#), e.g. Itanium (128 registers).

## 1.30.2 ARM

64-bit instructions appeared in ARMv8.

### 1.30.3 Float point numbers

How floating point numbers are processed in x86-64 is explained here: [1.31](#).

### 1.30.4 64-bit architecture criticism

Some people has irritation sometimes: now one needs twice as much memory for storing pointers, including cache memory, despite the fact that x64 CPUs can address only 48 bits of external RAM.

Some people made their own memory allocators. It's interesting to know about CryptoMiniSat<sup>187</sup> case. This program rarely uses more than 4GiB of RAM, but it uses pointers heavily. So it requires less memory on 32-bit architecture than on 64-bit one. To mitigate this problem, author made his own allocator (in `clauseallocator.(h|cpp)` files), which allows to have access to allocated memory using 32-bit identifiers instead of 64-bit pointers.

## 1.31 Working with floating point numbers using SIMD

Of course, the FPU has remained in x86-compatible processors when the SIMD extensions were added.

The SIMD extensions (SSE2) offer an easier way to work with floating-point numbers.

The number format remains the same (IEEE 754).

So, modern compilers (including those generating for x86-64) usually use SIMD instructions instead of FPU ones.

It can be said that it's good news, because it's easier to work with them.

We are going to reuse the examples from the FPU section here: [1.19 on page 218](#).

### 1.31.1 Simple example

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
}

int main()
{
    printf ("%f\n", f(1.2, 3.4));
}
```

### x64

Listing 1.393: Optimizing MSVC 2012 x64

```
_real@4010666666666666 DQ 0401066666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14

a$ = 8
b$ = 16
f
    PROC
        divsd    xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd    xmm1, QWORD PTR __real@4010666666666666
        addsd    xmm0, xmm1
        ret     0
    ENDP
```

The input floating point values are passed in the XMM0 - XMM3 registers, all the rest—via the stack<sup>188</sup>.

<sup>187</sup><https://github.com/msoos/cryptominisat/>

<sup>188</sup>MSDN: Parameter Passing

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

a is passed in `XMM0`, b—via `XMM1`.

The XMM-registers are 128-bit (as we know from the section about [SIMD: 1.29 on page 407](#)), but the *double* values are 64 bit, so only lower register half is used.

`DIVSD` is an SSE-instruction that stands for “Divide Scalar Double-Precision Floating-Point Values”, it just divides one value of type *double* by another, stored in the lower halves of operands.

The constants are encoded by compiler in IEEE 754 format.

`MULSD` and `ADDSD` work just as the same, but do multiplication and addition.

The result of the function’s execution in type *double* is left in the in `XMM0` register.

That is how non-optimizing MSVC works:

Listing 1.394: MSVC 2012 x64

```
_real@4010666666666666 DQ 0401066666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14

a$ = 8
b$ = 16
_f
    PROC
        movsd QWORD PTR [rsp+16], xmm1
        movsd QWORD PTR [rsp+8], xmm0
        movsd xmm0, QWORD PTR a$[rsp]
        divsd xmm0, QWORD PTR __real@40091eb851eb851f
        movsd xmm1, QWORD PTR b$[rsp]
        mulsd xmm1, QWORD PTR __real@4010666666666666
        addsd xmm0, xmm1
        ret 0
_f
    ENDP
```

Slightly redundant. The input arguments are saved in the “shadow space” ([1.10.2 on page 101](#)), but only their lower register halves, i.e., only 64-bit values of type *double*. GCC produces the same code.

## x86

Let’s also compile this example for x86. Despite the fact it’s generating for x86, MSVC 2012 uses SSE2 instructions:

Listing 1.395: Non-optimizing MSVC 2012 x86

```
tv70 = -8      ; size = 8
_a$ = 8        ; size = 8
_b$ = 16       ; size = 8
_f
    PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        movsd  xmm0, QWORD PTR _a$[ebp]
        divsd  xmm0, QWORD PTR __real@40091eb851eb851f
        movsd  xmm1, QWORD PTR _b$[ebp]
        mulsd  xmm1, QWORD PTR __real@4010666666666666
        addsd  xmm0, xmm1
        movsd  QWORD PTR tv70[ebp], xmm0
        fld    QWORD PTR tv70[ebp]
        mov    esp, ebp
        pop    ebp
        ret    0
_f
    ENDP
```

Listing 1.396: Optimizing MSVC 2012 x86

```
tv67 = 8      ; size = 8
_a$ = 8        ; size = 8
_b$ = 16       ; size = 8
_f
    PROC
        movsd  xmm1, QWORD PTR _a$[esp-4]
```

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

```
divsd    xmm1, QWORD PTR __real@40091eb851eb851f
movsd    xmm0, QWORD PTR _b$[esp-4]
mulsd    xmm0, QWORD PTR __real@4010666666666666
addsd    xmm1, xmm0
movsd    QWORD PTR tv67[esp-4], xmm1
fld      QWORD PTR tv67[esp-4]
ret      0
_f      ENDP
```

It's almost the same code, however, there are some differences related to calling conventions: 1) the arguments are passed not in XMM registers, but in the stack, like in the FPU examples ([1.19 on page 218](#)); 2) the result of the function is returned in `ST(0)` — in order to do so, it's copied (through local variable `tv`) from one of the XMM registers to `ST(0)`.

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

Let's try the optimized example in OllyDbg:

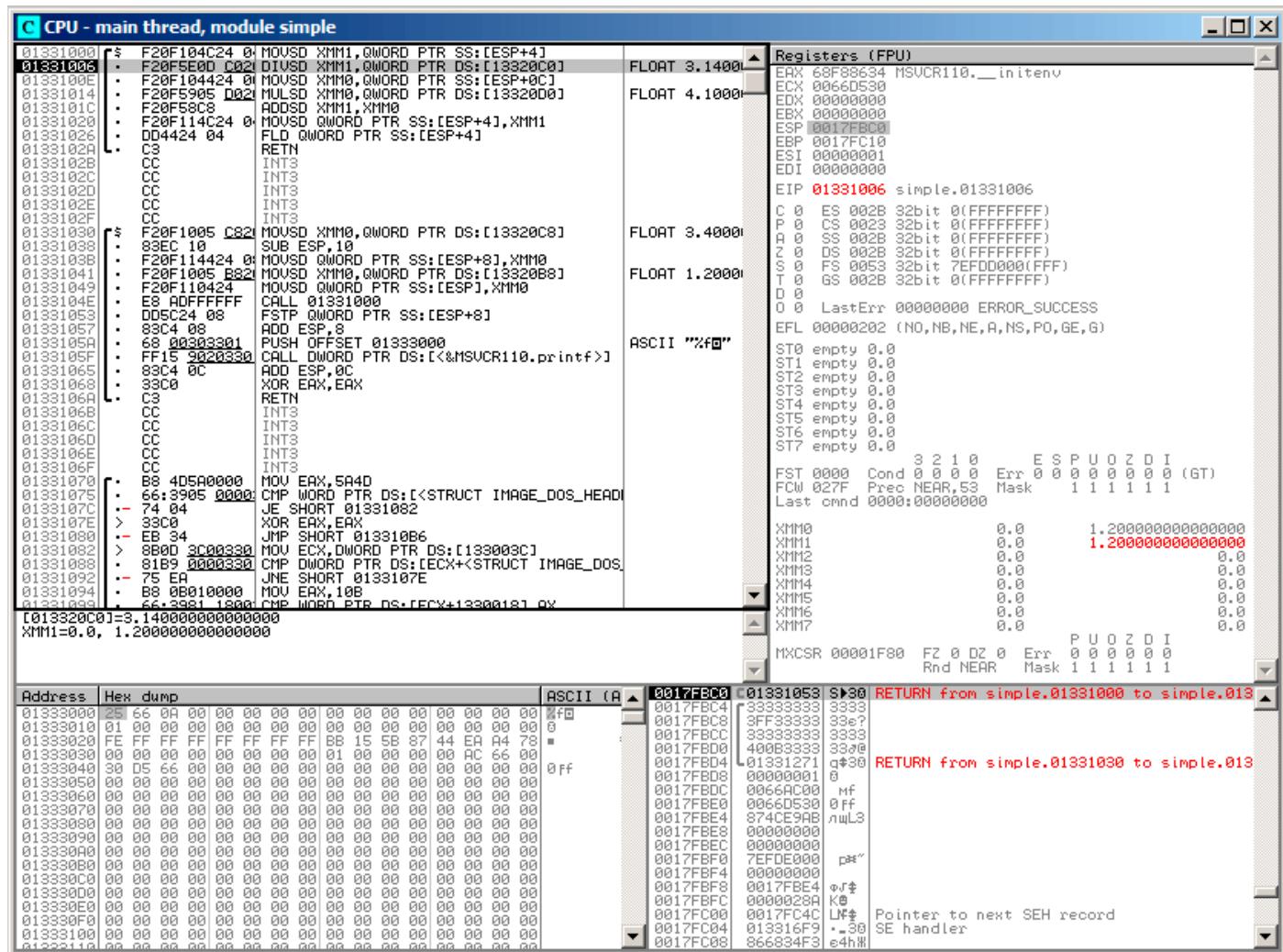


Figure 1.113: OllyDbg: MOVSD loads the value of *a* into XMM1

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

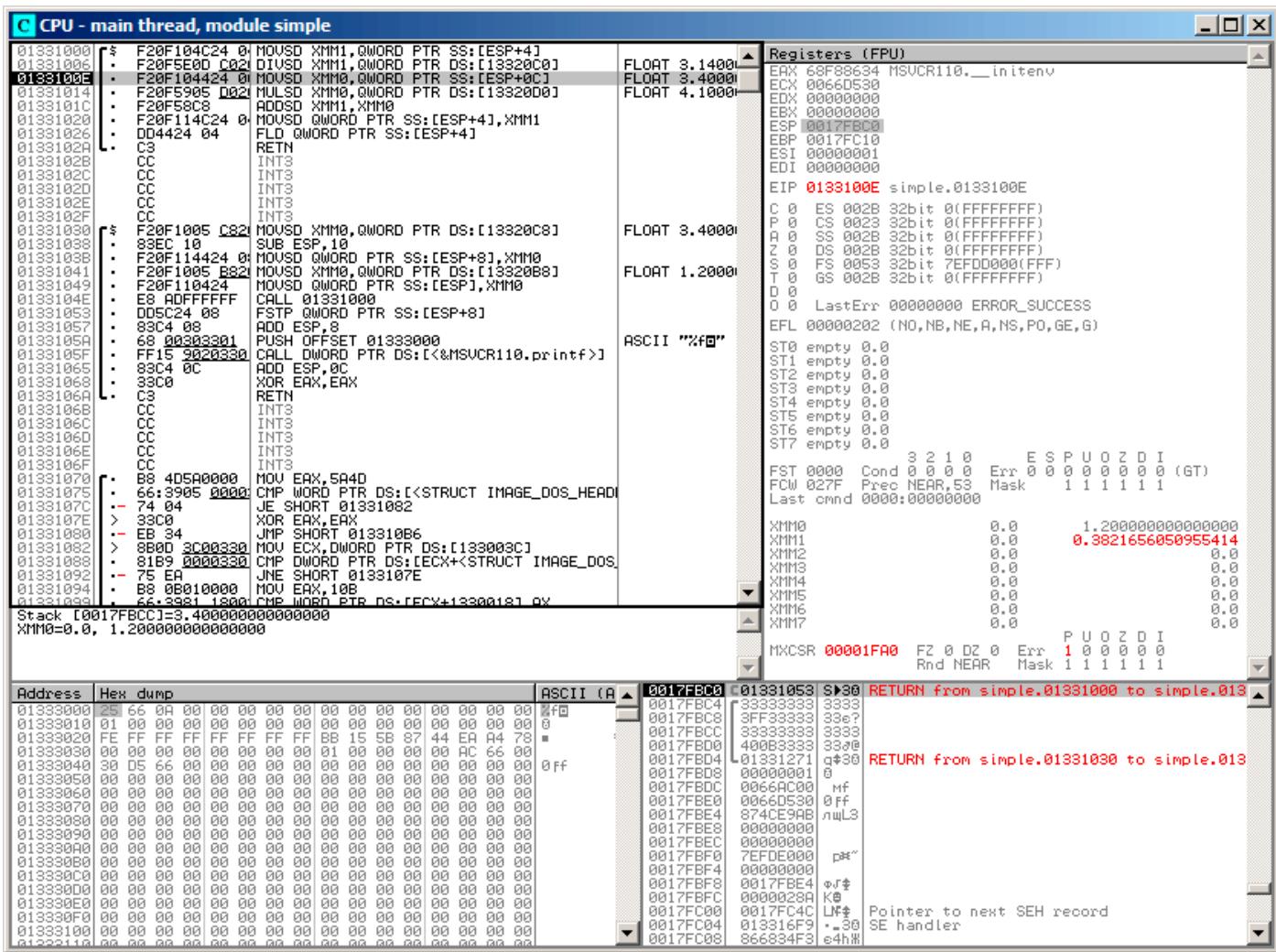


Figure 1.114: OllyDbg: DIVSD calculated quotient and stored it in XMM1

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

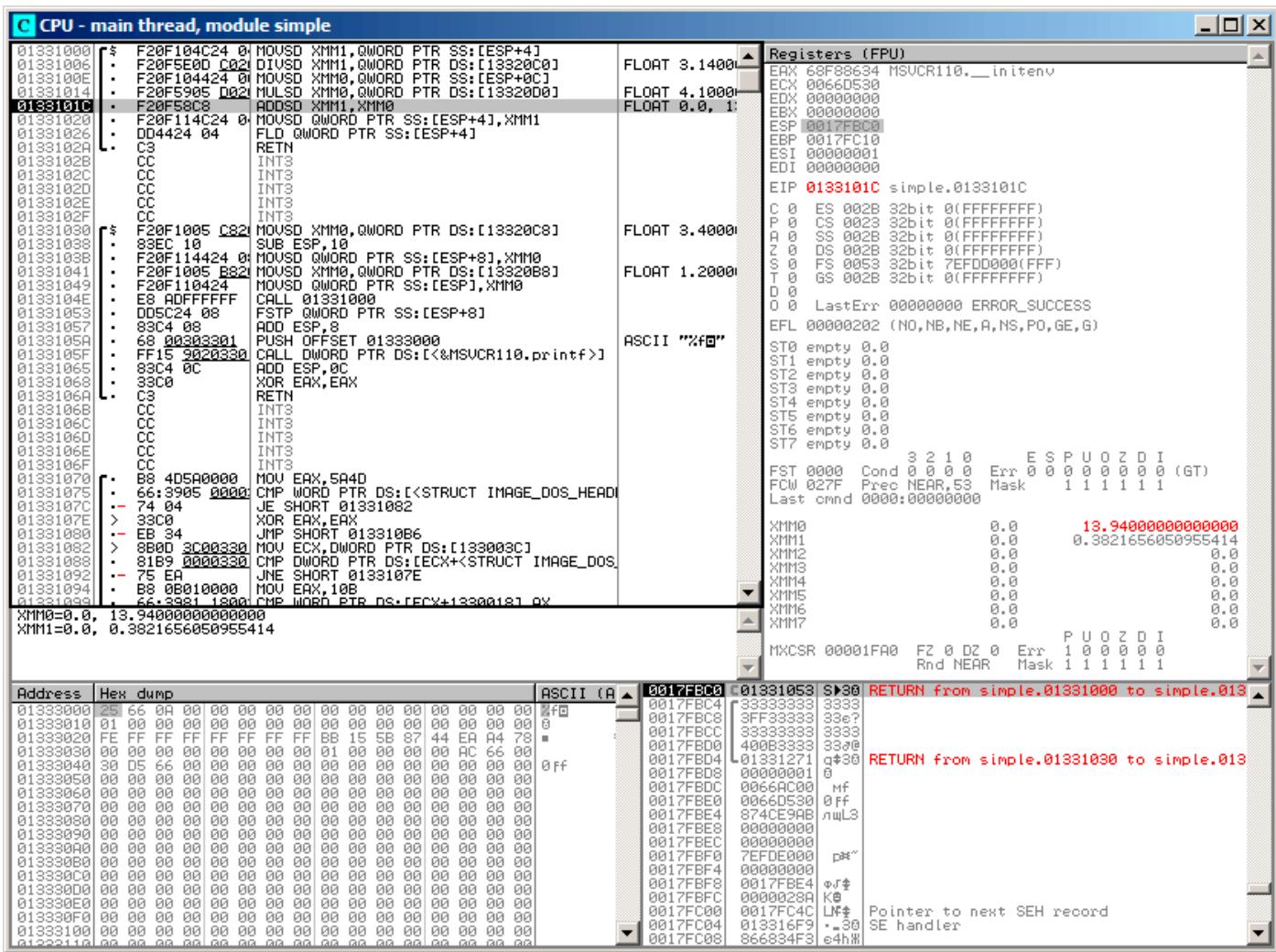


Figure 1.115: OllyDbg: MULSD calculated product and stored it in XMM0

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

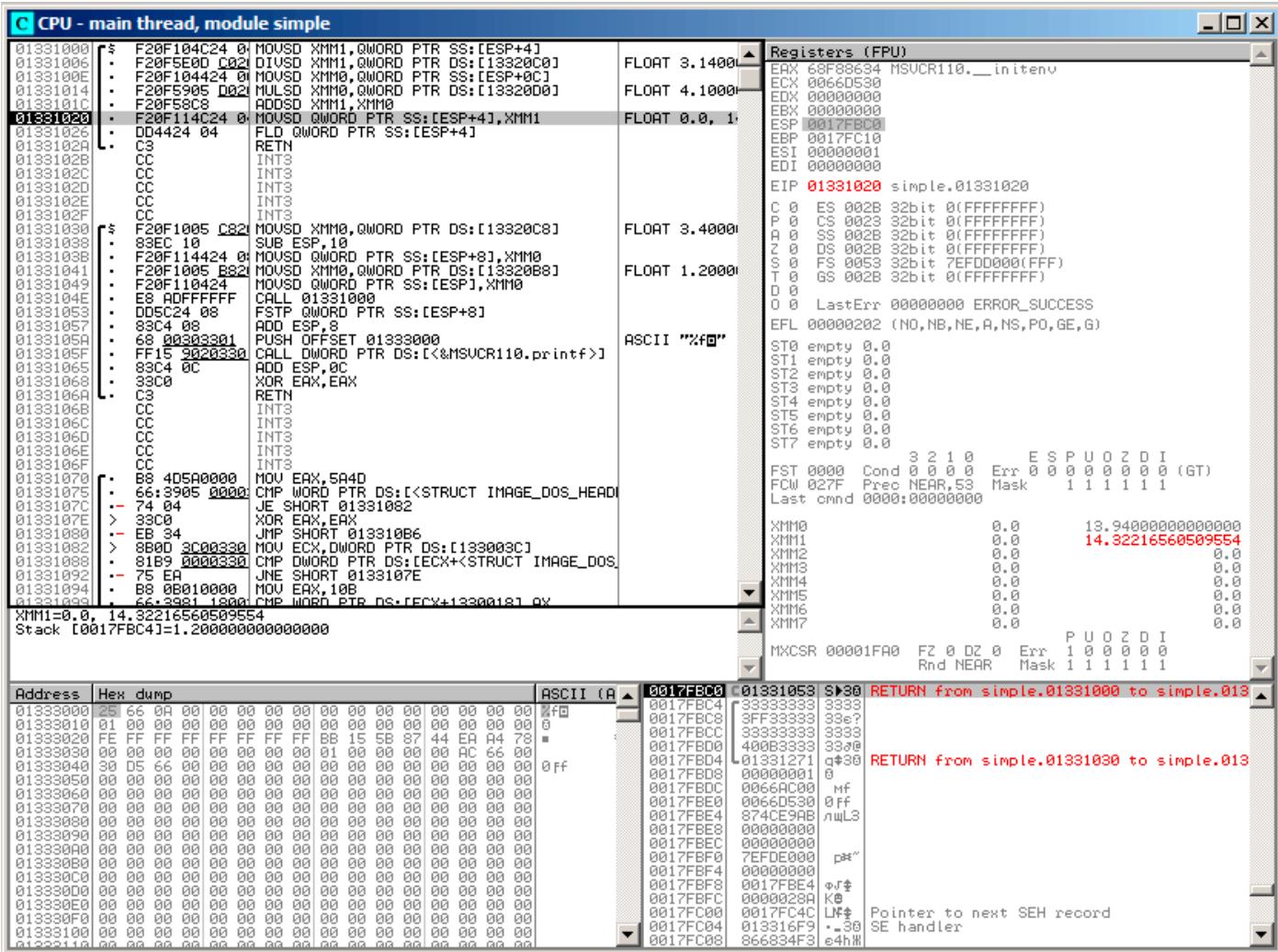


Figure 1.116: OllyDbg: ADDSD adds value in XMM0 to XMM1

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

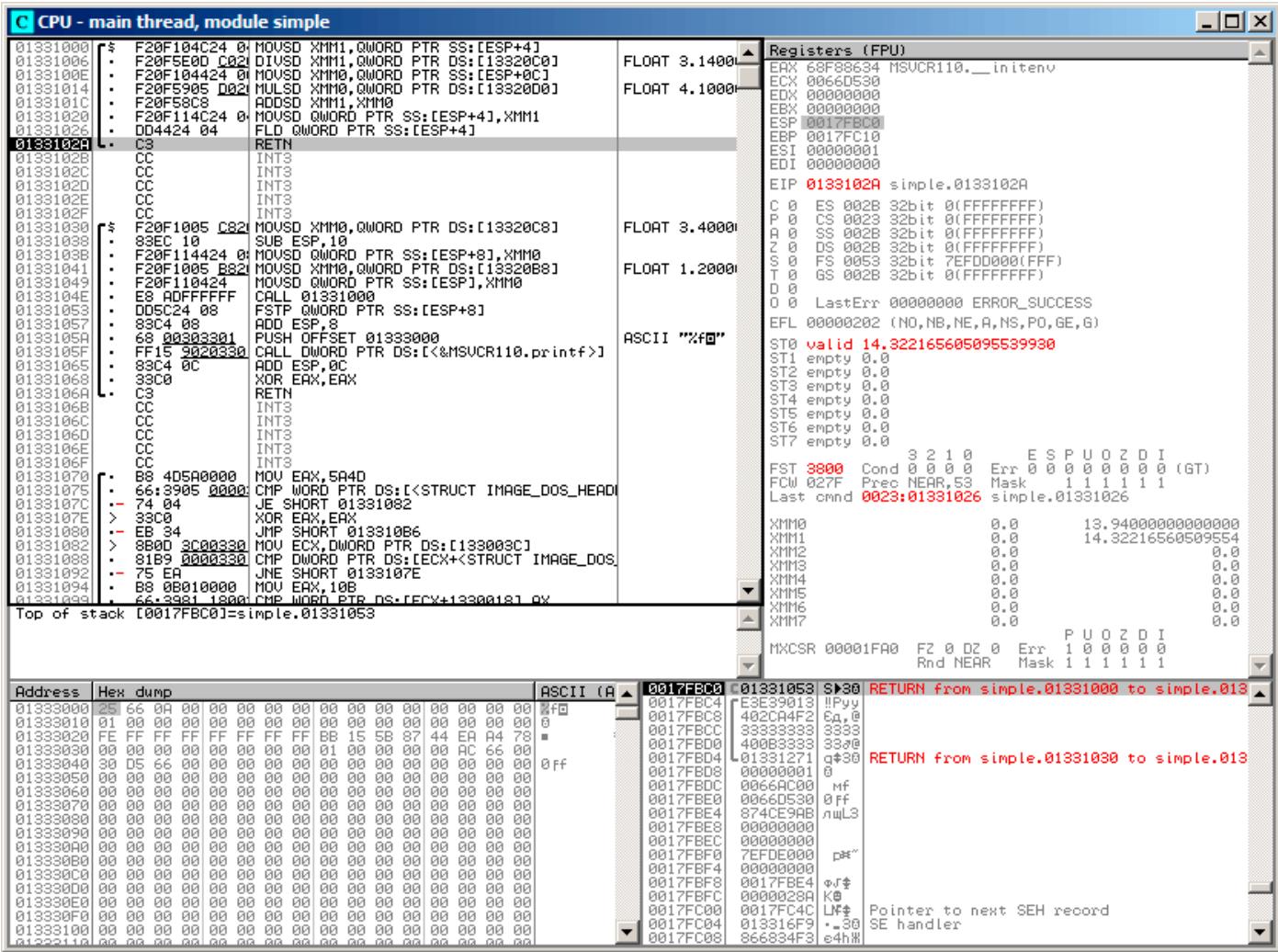


Figure 1.117: OllyDbg: FLD left function result in ST(0)

We see that OllyDbg shows the XMM registers as pairs of *double* numbers, but only the *lower* part is used. Apparently, OllyDbg shows them in that format because the SSE2 instructions (suffixed with `-SD`) are executed right now.

But of course, it's possible to switch the register format and to see their contents as 4 *float*-numbers or just as 16 bytes.

### 1.31.2 Passing floating point number via arguments

```
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}
```

They are passed in the lower halves of the `XMM0 - XMM3` registers.

Listing 1.397: Optimizing MSVC 2012 x64

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r ; 1.54

main PROC
    sub    rsp, 40                                ; 00000028H
    movsd  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
    movsd  xmm0, QWORD PTR __real@40400147ae147ae1
    call   pow
    lea    rcx, OFFSET FLAT:$SG1354
    movaps xmm1, xmm0
    movd   rdx, xmm1
    call   printf
    xor    eax, eax
    add    rsp, 40                                ; 00000028H
    ret    0
main ENDP
```

There is no `MOVSDX` instruction in Intel and AMD manuals ([11.1.4 on page 996](#)), there it is called just `MOVSD`. So there are two instructions sharing the same name in x86 (about the other see: [.1.6 on page 1012](#)). Apparently, Microsoft developers wanted to get rid of the mess, so they renamed it to `MOVSDX`. It just loads a value into the lower half of a XMM register.

`pow()` takes arguments from `XMM0` and `XMM1`, and returns result in `XMM0`. It is then moved to `RDX` for `printf()`. Why? Maybe because `printf()` —is a variable arguments function?

Listing 1.398: Optimizing GCC 4.4.6 x64

```
.LC2:
    .string "32.01 ^ 1.54 = %lf\n"
main:
    sub    rsp, 8
    movsd xmm1, QWORD PTR .LC0[rip]
    movsd xmm0, QWORD PTR .LC1[rip]
    call   pow
    ; result is now in XMM0
    mov    edi, OFFSET FLAT:.LC2
    mov    eax, 1 ; number of vector registers passed
    call   printf
    xor    eax, eax
    add    rsp, 8
    ret
.LC0:
    .long  171798692
    .long  1073259479
.LC1:
    .long  2920577761
    .long  1077936455
```

GCC generates clearer output. The value for `printf()` is passed in `XMM0`. By the way, here is a case when 1 is written into `EAX` for `printf()` —this implies that one argument will be passed in vector reg-

### 1.31. WORKING WITH FLOATING POINT NUMBERS USING SIMD

isters, just as the standard requires [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] <sup>189</sup>.

#### 1.31.3 Comparison example

```
#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
}

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
}
```

#### x64

Listing 1.399: Optimizing MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max PROC
    comisd xmm0, xmm1
    ja SHORT $LN2@d_max
    movaps xmm0, xmm1
$LN2@d_max:
    fatret 0
d_max ENDP
```

Optimizing MSVC generates a code very easy to understand.

`COMISD` is “Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS”. Essentially, that is what it does.

Non-optimizing MSVC generates more redundant code, but it is still not hard to understand:

Listing 1.400: MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max PROC
    movsdx QWORD PTR [rsp+16], xmm1
    movsdx QWORD PTR [rsp+8], xmm0
    movsdx xmm0, QWORD PTR a$[rsp]
    comisd xmm0, QWORD PTR b$[rsp]
    jbe SHORT $LN1@d_max
    movsdx xmm0, QWORD PTR a$[rsp]
    jmp SHORT $LN2@d_max
$LN1@d_max:
    movsdx xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
    fatret 0
d_max ENDP
```

However, GCC 4.4.6 did more optimizations and used the `MAXSD` (“Return Maximum Scalar Double-Precision Floating-Point Value”) instruction, which just choose the maximum value!

<sup>189</sup>Also available as <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

```
d_max:  
    maxsd    xmm0, xmm1  
    ret
```

Let's compile this example in MSVC 2012 with optimization turned on:

Listing 1.402: Optimizing MSVC 2012 x86

```
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_d_max PROC
    movsd xmm0, QWORD PTR _a$[esp-4]
    comisd xmm0, QWORD PTR _b$[esp-4]
    jbe SHORT $LN1@_d_max
    fld  QWORD PTR _a$[esp-4]
    ret  0
$LN1@_d_max:
    fld  QWORD PTR _b$[esp-4]
    ret  0
_d_max ENDP
```

Almost the same, but the values of *a* and *b* are taken from the stack and the function result is left in ST(0).

If we load this example in OllyDbg, we can see how the COMISD instruction compares values and sets/clears the CF and PF flags:

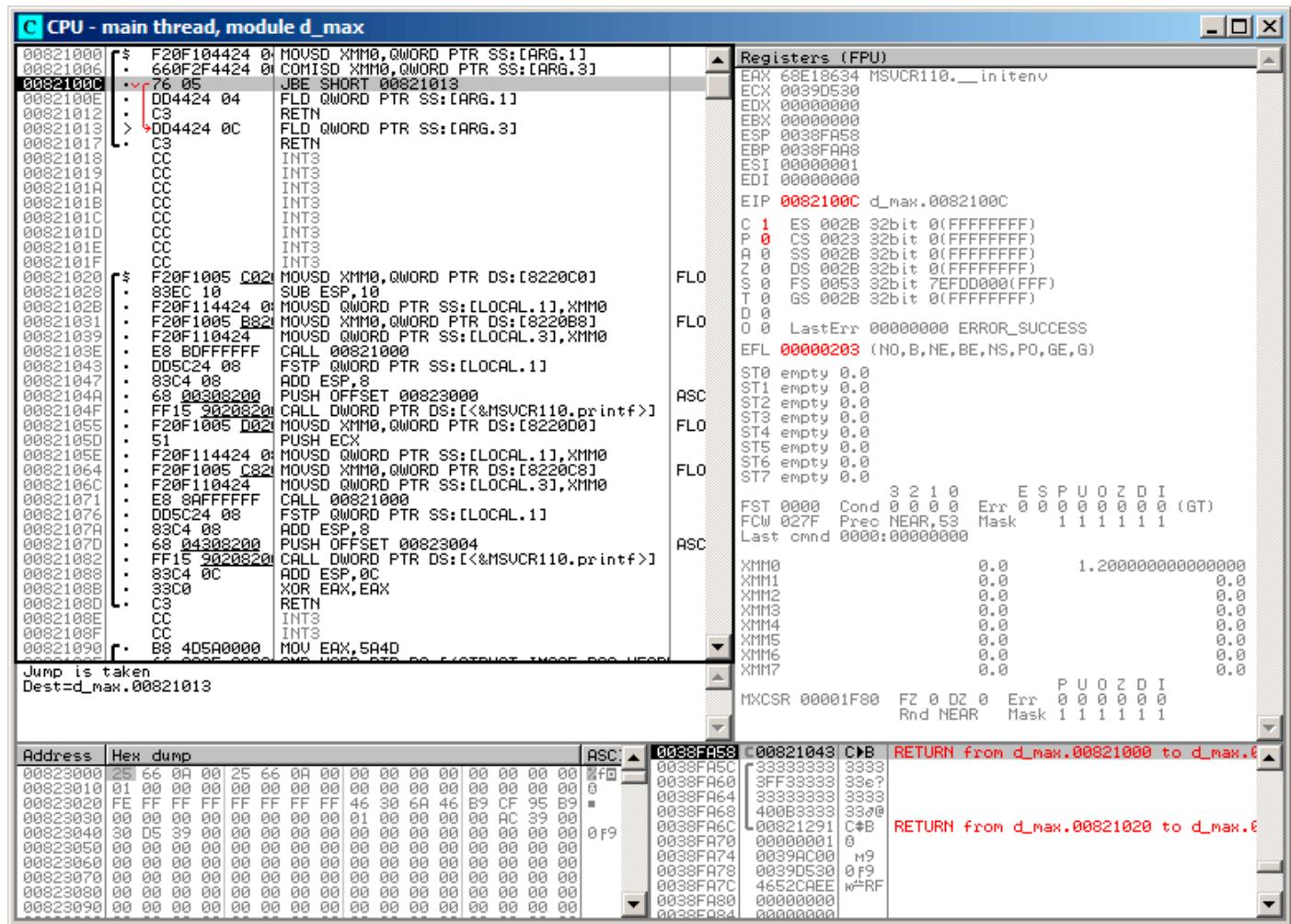


Figure 1.118: OllyDbg: COMISD changed CF and PF flags

### 1.31.4 Calculating machine epsilon: x64 and SIMD

Let's revisit the “calculating machine epsilon” example for *double* listing 1.25.2.

Now we compile it for x64:

```
v$ = 8
calculate_machine_epsilon PROC
    movsdx QWORD PTR v$[rsp], xmm0
    movaps xmm1, xmm0
    inc QWORD PTR v$[rsp]
    movsdx xmm0, QWORD PTR v$[rsp]
    subsd xmm0, xmm1
    ret 0
calculate_machine_epsilon ENDP
```

There is no way to add 1 to a value in 128-bit XMM register, so it must be placed into memory.

There is, however, the `ADDSD` instruction (*Add Scalar Double-Precision Floating-Point Values*) which can add a value to the lowest 64-bit half of a XMM register while ignoring the higher one, but MSVC 2012 probably is not that good yet <sup>190</sup>.

Nevertheless, the value is then reloaded to a XMM register and subtraction occurs. `SUBSD` is “Subtract Scalar Double-Precision Floating-Point Values”, i.e., it operates on the lower 64-bit part of 128-bit XMM register. The result is returned in the XMM0 register.

### 1.31.5 Pseudo-random number generator example revisited

Let's revisit “pseudo-random number generator example” example listing [1.25.1](#).

If we compile this in MSVC 2012, it will use the SIMD instructions for the FPU.

```
_real@3f800000 DD 03f800000r ; 1

tv128 = -4
_tmp$ = -4
?float_rand@YAMXZ PROC
    push    ecx
    call    ?my_rand@@YAIXZ
; EAX=pseudorandom value
    and     eax, 8388607 ; 007fffffH
    or      eax, 1065353216 ; 3f800000H
; EAX=pseudorandom value & 0x007fffff | 0x3f800000
; store it into local stack:
    mov    DWORD PTR _tmp$[esp+4], eax
; reload it as float point number:
    movss  xmm0, DWORD PTR _tmp$[esp+4]
; subtract 1.0:
    subss  xmm0, DWORD PTR __real@3f800000
; move value to ST0 by placing it in temporary variable...
    movss  DWORD PTR tv128[esp+4], xmm0
; ... and reloading it into ST0:
    fld    DWORD PTR tv128[esp+4]
    pop    ecx
    ret    0
?float_rand@YAMXZ ENDP
```

All instructions have the `-SS` suffix, which stands for “Scalar Single”.

“Scalar” implies that only one value is stored in the register.

“Single” stands for *float* data type.

### 1.31.6 Summary

Only the lower half of XMM registers is used in all examples here, to store number in IEEE 754 format.

Essentially, all instructions prefixed by `-SD` (“Scalar Double-Precision”)—are instructions working with floating point numbers in IEEE 754 format, stored in the lower 64-bit half of a XMM register.

<sup>190</sup>As an exercise, you may try to rework this code to eliminate the usage of the local stack.

## 1.32. ARM-SPECIFIC DETAILS

And it is easier than in the FPU, probably because the SIMD extensions were evolved in a less chaotic way than the FPU ones in the past. The stack register model is not used.

If you would try to replace *double* with *float*

in these examples, the same instructions will be used, but prefixed with `-SS` ("Scalar Single-Precision"), for example, `MOVSS`, `COMISS`, `ADDSS`, etc.

"Scalar" implies that the SIMD register containing only one value instead of several.

Instructions working with several values in a register simultaneously have "Packed" in their name.

Needless to say, the SSE2 instructions work with 64-bit IEEE 754 numbers (*double*), while the internal representation of the floating-point numbers in FPU is 80-bit numbers.

Hence, the FPU may produce less round-off errors and as a consequence, FPU may give more precise calculation results.

## 1.32 ARM-specific details

### 1.32.1 Number sign (#) before number

The Keil compiler, [IDA](#) and objdump precede all numbers with the "#" number sign, for example: listing.[1.16.1](#).

But when GCC 4.9 generates assembly language output, it doesn't, for example: listing.[3.15](#).

The ARM listings in this book are somewhat mixed.

It's hard to say, which method is right. Supposedly, one has to obey the rules accepted in environment he/she works in.

### 1.32.2 Addressing modes

This instruction is possible in ARM64:

```
ldr    x0, [x29,24]
```

This means add 24 to the value in X29 and load the value from this address.

Please note that 24 is inside the brackets. The meaning is different if the number is outside the brackets:

```
ldr    w4, [x1],28
```

This means load the value at the address in X1, then add 28 to X1.

ARM allows you to add or subtract a constant to/from the address used for loading.

And it's possible to do that both before and after loading.

There is no such addressing mode in x86, but it is present in some other processors, even on PDP-11.

There is a legend that the pre-increment, post-increment, pre-decrement and post-decrement modes in PDP-11,

were "guilty" for the appearance of such C language (which developed on PDP-11) constructs as `*ptr++`, `++ptr`, `*ptr--`, `--ptr`.

By the way, this is one of the hard to memorize C features. This is how it is:

## 1.32. ARM-SPECIFIC DETAILS

C term	ARM term	C statement	how it works
Post-increment	post-indexed addressing	*ptr++	use *ptr value, then increment ptr pointer
Post-decrement	post-indexed addressing	*ptr--	use *ptr value, then decrement ptr pointer
Pre-increment	pre-indexed addressing	*++ptr	increment ptr pointer, then use *ptr value
Pre-decrement	pre-indexed addressing	*--ptr	decrement ptr pointer, then use *ptr value

Pre-indexing is marked with an exclamation mark in the ARM assembly language. For example, see line 2 in listing 1.28.

Dennis Ritchie (one of the creators of the C language) mentioned that it presumably was invented by Ken Thompson (another C creator) because this processor feature was present in PDP-7 <sup>191</sup>, [Dennis M. Ritchie, *The development of the C language*, (1993)]<sup>192</sup>.

Thus, C language compilers may use it, if it is present on the target processor.

That's very convenient for array processing.

### 1.32.3 Loading a constant into a register

#### 32-bit ARM

As we already know, all instructions have a length of 4 bytes in ARM mode and 2 bytes in Thumb mode.

Then how can we load a 32-bit value into a register, if it's not possible to encode it in one instruction?

Let's try:

```
unsigned int f()
{
    return 0x12345678;
};
```

Listing 1.405: GCC 4.6.3 -O3 ARM mode

```
f:
    ldr    r0, .L2
    bx    lr
.L2:
    .word 305419896 ; 0x12345678
```

So, the `0x12345678` value is just stored aside in memory and loaded if needed.

But it's possible to get rid of the additional memory access.

Listing 1.406: GCC 4.6.3 -O3 -march=armv7-a (ARM mode)

```
movw    r0, #22136      ; 0x5678
movt    r0, #4660       ; 0x1234
bx    lr
```

We see that the value is loaded into the register by parts, the lower part first (using `MOVW`), then the higher (using `MOVT`).

This implies that 2 instructions are necessary in ARM mode for loading a 32-bit value into a register.

It's not a real problem, because in fact there are not many constants in real code (except of 0 and 1).

<sup>191</sup>[http://yurichev.com/mirrors/C/c\\_dmr\\_postincrement.txt](http://yurichev.com/mirrors/C/c_dmr_postincrement.txt)

<sup>192</sup>Also available as <http://go.yurichev.com/17264>

### 1.32. ARM-SPECIFIC DETAILS

Does it mean that the two-instruction version is slower than one-instruction version?

Doubtfully. Most likely, modern ARM processors are able to detect such sequences and execute them fast.

On the other hand, [IDA](#) is able to detect such patterns in the code and disassembles this function as:

```
MOV    R0, 0x12345678  
BX    LR
```

## ARM64

```
uint64_t f()  
{  
    return 0x12345678ABCDEF01;  
};
```

Listing 1.407: GCC 4.9.1 -O3

```
mov    x0, 61185 ; 0xef01  
movk   x0, 0xabcd, lsl 16  
movk   x0, 0x5678, lsl 32  
movk   x0, 0x1234, lsl 48  
ret
```

[MOVK](#) stands for “MOV Keep”, i.e., it writes a 16-bit value into the register, not touching the rest of the bits. The [LSL](#) suffix shifts left the value by 16, 32 and 48 bits at each step. The shifting is done before loading.

This implies that 4 instructions are necessary to load a 64-bit value into a register.

## Storing floating-point number into register

It's possible to store a floating-point number into a D-register using only one instruction.

For example:

```
double a()  
{  
    return 1.5;  
};
```

Listing 1.408: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:  
0: 1e6f1000      fmov    d0, #1.5000000000000000e+000  
4: d65f03c0      ret
```

The number 1.5 was indeed encoded in a 32-bit instruction. But how?

In ARM64, there are 8 bits in the [FMOV](#) instruction for encoding some floating-point numbers.

The algorithm is called [VFPExpandImm\(\)](#) in [\[ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, \(2013\)\]](#)<sup>193</sup>. This is also called [minifloat](#)<sup>194</sup>.

We can try different values: the compiler is able to encode 30.0 and 31.0, but it couldn't encode 32.0, as 8 bytes have to be allocated for this number in the IEEE 754 format:

```
double a()  
{  
    return 32;  
};
```

<sup>193</sup>Also available as [http://yurichev.com/mirrors/ARMv8-A\\_Architecture\\_Reference\\_Manual\\_\(Issue\\_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

<sup>194</sup>wikipedia

```
a:
    ldr      d0, .LC0
    ret
.LC0:
    .word    0
    .word    1077936128
```

### 1.32.4 Relocs in ARM64

As we know, there are 4-byte instructions in ARM64, so it is impossible to write a large number into a register using a single instruction.

Nevertheless, an executable image can be loaded at any random address in memory, so that's why relocations exist. Read more about them (in relation to Win32 PE): [6.5.2 on page 759](#).

The address is formed using the `ADRP` and `ADD` instruction pair in ARM64.

The first loads a 4KiB-page address and the second one adds the remainder. Let's compile the example from "Hello, world!" (listing.1.8) in GCC (Linaro) 4.9 under win32:

Listing 1.410: GCC (Linaro) 4.9 and objdump of object file

```
...>aarch64-linux-gnu-gcc.exe hw.c -c
...>aarch64-linux-gnu-objdump.exe -d hw.o
...
0000000000000000 <main>:
 0:  a9bf7bfd      stp      x29, x30, [sp,#-16]!
 4:  910003fd      mov      x29, sp
 8:  90000000      adrp     x0, 0 <main>
 c:  91000000      add      x0, x0, #0x0
10:  94000000      bl       0 <printf>
14:  52800000      mov      w0, #0x0          // #0
18:  a8c17bfd      ldp      x29, x30, [sp],#16
1c:  d65f03c0      ret
...>aarch64-linux-gnu-objdump.exe -r hw.o
...
RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000008 R_AARCH64_ADR_PREL_PG_HI21  .rodata
000000000000000c R_AARCH64_ADD_ABS_L012_NC   .rodata
0000000000000010 R_AARCH64_CALL26   printf
```

So there are 3 relocations in this object file.

- The first one takes the page address, cuts the lowest 12 bits and writes the remaining high 21 bits to the `ADRP` instruction's bit fields. This is because we don't need to encode the low 12 bits, and the `ADRP` instruction has space only for 21 bits.
- The second one puts the 12 bits of the address relative to the page start into the `ADD` instruction's bit fields.
- The last, 26-bit one, is applied to the instruction at address `0x10` where the jump to the `printf()` function is.

All ARM64 (and in ARM in ARM mode) instruction addresses have zeros in the two lowest bits (because all instructions have a size of 4 bytes), so one has to encode only the highest 26 bits of 28-bit address space ( $\pm 128\text{MB}$ ).

There are no such relocations in the executable file: because it's known where the "Hello!" string is located, in which page, and the address of `puts()` is also known.

### 1.33. MIPS-SPECIFIC DETAILS

So there are values set already in the `ADRP`, `ADD` and `BL` instructions (the linker has written them while linking):

Listing 1.411: objdump of executable file

```
0000000000400590 <main>:
400590: a9bf7bfd      stp    x29, x30, [sp,#-16]!
400594: 910003fd      mov    x29, sp
400598: 90000000      adrp   x0, 400000 <_init-0x3b8>
40059c: 91192000      add    x0, x0, #0x648
4005a0: 97fffffa0      bl    400420 <puts@plt>
4005a4: 52800000      mov    w0, #0x0          // #0
4005a8: a8c17bfd      ldp    x29, x30, [sp],#16
4005ac: d65f03c0      ret

...
Contents of section .rodata:
400640 01000200 00000000 48656c6c 6f210000 .....Hello!...
```

As an example, let's try to disassemble the `BL` instruction manually.

`0x97fffffa0` is `0b100101111111111111111110100000`. According to [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)C5.6.26], `imm26` is the last 26 bits:

$imm26 = 0b1111111111111111111110100000$ . It is `0x3FFFFA0`, but the MSB is 1, so the number is negative, and we can convert it manually to convenient form for us. By the rules of negation ( 2.2 on page 455), just invert all bits: (it is `0b1011111=0x5F`), and add 1 (`0x5F+1=0x60`). So the number in signed form is `-0x60`. Let's multiply `-0x60` by 4 (because address stored in opcode is divided by 4): it is `-0x180`. Now let's calculate destination address: `0x4005a0 + (-0x180) = 0x400420` (please note: we consider the address of the `BL` instruction, not the current value of `PC`, which may be different!). So the destination address is `0x400420`.

More about ARM64-related relocations: [ELF for the ARM 64-bit Architecture (AArch64), (2013)]<sup>195</sup>.

## 1.33 MIPS-specific details

### 1.33.1 Loading a 32-bit constant into register

```
unsigned int f()
{
    return 0x12345678;
};
```

All instructions in MIPS, just like ARM, have a size of 32-bit, so it's not possible to embed a 32-bit constant into one instruction.

So one have to use at least two instructions: the first loads the high part of the 32-bit number and the second one applies an OR operation, which effectively sets the low 16-bit part of the target register:

Listing 1.412: GCC 4.4.5 -O3 (assembly output)

```
li      $2,305397760 # 0x12340000
j      $31
ori   $2,$2,0x5678 ; branch delay slot
```

IDA is fully aware of such frequently encountered code patterns, so, for convenience it shows the last `ORI` instruction as the `LI` pseudo instruction, which allegedly loads a full 32-bit number into the `$V0` register.

Listing 1.413: GCC 4.4.5 -O3 (IDA)

```
lui   $v0, 0x1234
jr   $ra
li   $v0, 0x12345678 ; branch delay slot
```

<sup>195</sup>Also available as <http://go.yurichev.com/17288>

### 1.33. MIPS-SPECIFIC DETAILS

The GCC assembly output has the LI pseudo instruction, but in fact, LUI ("Load Upper Immediate") is there, which stores a 16-bit value into the high part of the register.

Let's see in *objdump* output:

Listing 1.414: objdump

```
00000000 <f>:  
 0: 3c021234      lui    v0,0x1234  
 4: 03e00008      jr     ra  
 8: 34425678      ori    v0,v0,0x5678
```

### Loading a 32-bit global variable into register

```
unsigned int global_var=0x12345678;  
  
unsigned int f2()  
{  
    return global_var;  
}
```

This is slightly different: LUI loads upper 16-bit from *global\_var* into \$2 (or \$V0) and then LW loads lower 16-bits summing it with the contents of \$2:

Listing 1.415: GCC 4.4.5 -O3 (assembly output)

```
f2:  
    lui    $2,%hi(global_var)  
    lw     $2,%lo(global_var)($2)  
    j     $31  
    nop    ; branch delay slot  
  
    ...  
  
global_var:  
    .word 305419896
```

IDA is fully aware of often used LUI / LW instruction pair, so it coalesces both into a single LW instruction:

Listing 1.416: GCC 4.4.5 -O3 (IDA)

```
_f2:  
    lw     $v0, global_var  
    jr     $ra  
    or     $at, $zero      ; branch delay slot  
  
    ...  
  
.data  
.globl global_var  
global_var: .word 0x12345678      # DATA XREF: _f2
```

*objdump*'s output is the same as GCC's assembly output. Let's also dump relocs of the object file:

Listing 1.417: objdump

```
objdump -D filename.o  
  
...  
  
0000000c <f2>:  
  c: 3c020000      lui    v0,0x0  
 10: 8c420000      lw     v0,0(v0)  
 14: 03e00008      jr     ra  
 18: 00200825      move   at,at    ; branch delay slot  
 1c: 00200825      move   at,at  
  
Disassembly of section .data:
```

### 1.33. MIPS-SPECIFIC DETAILS

```
00000000 <global_var>:  
 0: 12345678      beq    s1,s4,159e4 <f2+0x159d8>  
...  
objdump -r filename.o  
...  
RELOCATION RECORDS FOR [.text]:  
OFFSET  TYPE           VALUE  
0000000c R_MIPS_HI16  global_var  
00000010 R_MIPS_L016  global_var  
...
```

We can see that address of *global\_var* is to be written right into `LUI` and `LW` instructions during executable file loading: high 16-bit part of *global\_var* goes into the first one (`LUI`), lower 16-bit part goes into the second one (`LW`).

#### 1.33.2 Further reading about MIPS

Dominic Sweetman, See *MIPS Run, Second Edition*, (2010).

## Chapter 2

# Important fundamentals



## 2.1 Integral datatypes

Integral datatype is a type for a value which can be converted to number. These are numbers, enumerations, booleans.

### 2.1.1 Bit

Obvious usage for bits are boolean values: 0 for *false* and 1 for *true*.

Set of booleans can be packed into [word](#): there will be 32 booleans in 32-bit word, etc. This way is called *bitmap* or *bitfield*.

But it has obvious overhead: a bit jiggling, isolating, etc. While using [word](#) (or *int* type) for boolean variable is not economic, but highly efficient.

In C/C++ environment, 0 is for *false* and any non-zero value is for *true*. For example:

```
if (1234)
    printf ("this will always be executed\n");
else
    printf ("this will never\n");
```

This is popular way of enumerating characters in a C-string:

```
char *input=...;

while(*input) // execute body if *input character is non-zero
{
    // do something with *input
    input++;
};
```

### 2.1.2 Nibble AKA nybble

[AKA](#) half-byte, tetrade. Equals to 4 bits.

All these terms are still in use today.

#### Binary-coded decimal ([BCD](#)<sup>1</sup>)

4-bit nibbles were used in 4-bit CPUs like legendary Intel 4004 (used in calculators).

It's interesting to know that there was *binary-coded decimal* ([BCD](#)) way of representing decimal digit using 4 bits. Decimal 0 is represented as 0b0000, decimal 9 as 0b1001 and higher values are not used. Decimal 1234 is represented as 0x1234. Of course, this way is not economical.

Nevertheless, it has one advantage: decimal to [BCD](#)-packed number conversion and back is extremely easy. BCD-numbers can be added, subtracted, etc., but an additional correction is needed. x86 CPUs has rare instructions for that: [AAA / DAA](#) (adjust after addition), [AAS / DAS](#) (adjust after subtraction), [AAM](#) (after multiplication), [AAD](#) (after division).

The need for CPUs to support [BCD](#) numbers is a reason why *half-carry flag* (on 8080/Z80) and *auxiliary flag* ([AF](#) on x86) are exist: this is carry-flag generated after proceeding of lower 4 bits. The flag is then used for adjustment instructions.

The fact of easy conversion had led to popularity of [Peter Abel, *IBM PC assembly language and programming* (1987)] book. But aside of this book, the author of these notes never seen [BCD](#) numbers in practice, except for *magic numbers* ([5.6.1 on page 718](#)), like when someone's birthday is encoded like 0x19861115—this is indeed packed [BCD](#) number.

[BCD](#) instructions in x86 were often used for other purposes, especially in undocumented ways, for example:

---

<sup>1</sup>Binary-coded decimal

## 2.1. INTEGRAL DATATYPES

```
cmp al,10  
sbb al,69h  
das
```

This obscure code converts number in 0..15 range into [ASCII](#)<sup>2</sup> character '0'..'9', 'A'..'F'.

### Z80

Z80 was clone of 8-bit Intel 8080 CPU, and because of space constraints, it has 4-bit [ALU](#), i.e., each operation over two 8-bit numbers had to be proceeded in two steps. One side-effect of this was easy and natural generation of *half-carry flag*.

#### 2.1.3 Byte

Byte is primarily used for character storage. 8-bit bytes were not common as today. Punched tapes for teletypes had 5 and 6 possible holes, this is 5 or 6 bits for byte.

To emphasize the fact the byte has 8 bits, byte is sometimes called *octet*: at least *fetchmail* uses this terminology.

9-bit bytes were exist in 36-bit architectures: 4 9-bit bytes were fit in single [word](#). Probably because of this fact, C/C++ standard tells that *char* has to have a room for *at least* 8 bits, but more bits is allowable.

#### Standard ASCII table

7-bit ASCII table is standard, which has only 128 possible characters. Early E-Mail transport software were operating only on 7-bit ASCII codes, so a [MIME](#)<sup>3</sup> standard needed to encode messages in non-Latin writing systems. 7-bit ASCII code was augmented by parity bit, resulting in 8 bits.

*Data Encryption Standard* ([DES](#)<sup>4</sup>) has a 56 bits key, this is 8 7-bit bytes, leaving a space to parity bit for each character.

There is no need to memorize whole [ASCII](#) table, but rather ranges. [0..0x1F] are not control characters (non-printable). [0x20..0x7E] are printable ones. Codes starting at 0x80 are usually used for non-Latin writing systems and/or pseudographics.

Significant codes which will be easily memorized are: 0 (end of C-string, `'\0'` in C/C++); 0xA or 10 (line feed, `'\n'` in C/C++); 0xD or 13 (carriage return, `'\r'` in C/C++).

0x20 (space) is also often memorized.

#### 8-bit CPUs

x86 has capability to work with byte(s) on register level (because they are descendants of 8-bit 8080 CPU), RISC CPUs like ARM and MIPS—not.

#### 2.1.4 Wide char

This is attempt to support multi-lingual environment by extending byte to 16-bit. Most well-known example is Windows NT kernel and win32 functions with *W* suffix. This is why each Latin character in plain English text string is interleaved with zero byte. This encoding is called UCS-2 or UTF-16

Usually, *wchar\_t* is synonym to 16-bit *short* data type.

<sup>2</sup>American Standard Code for Information Interchange

<sup>3</sup>Multipurpose Internet Mail Extensions

<sup>4</sup>Data Encryption Standard

## 2.1.5 Signed integer vs unsigned

Some may argue, why unsigned data types exist at first place, since any unsigned number can be represented as signed. Yes, but absence of sign bit in a value extends its range twice. Hence, signed byte has range of -128..127, and unsigned one: 0..255. Another benefit of using unsigned data types is self-documenting: you define a variable which can't be assigned to negative values.

Unsigned data types absent in Java, for which it's criticized. It's hard to implement cryptographical algorithms using boolean operations over signed data types.

Values like 0xFFFFFFFF (-1) are used often, mostly as error codes.

## 2.1.6 Word

**Word** word is somewhat ambiguous term and usually denotes a data type fitting in **GPR**. Bytes are practical for characters, but impractical for other arithmetical calculations.

Hence, many **CPU**s has **GPRs** with width of 16, 32 or 64 bits. Even 8-bit CPUs like 8080 and Z80 offers to work with 8-bit register pairs, each pair forms 16-bit *pseudoregister* (*BC*, *DE*, *HL*, etc.). Z80 has some capability to work with register pairs, and this is, in a sense, some kind of 16-bit CPU emulation.

In general, if a CPU marketed as "n-bit CPU", this usually means it has n-bit **GPRs**.

There was a time when hard disks and **RAM** modules were marketed as having *n* kilo-words instead of *b* kilobytes/megabytes.

For example, *Apollo Guidance Computer*<sup>5</sup> has 2048 words of **RAM**. This was a 16-bit computer, so there was 4096 bytes of **RAM**.

*TX-0*<sup>6</sup> had 64K of 18-bit words of magnetic core memory, i.e., 64 kilo-words.

*DECSYSTEM-2060*<sup>7</sup> could have up to 4096 kilowords of *solid state memory* (i.e., hard disks, tapes, etc). This was 36-bit computer, so this is 18432 kilobytes or 18 megabytes.

---

*int* in C/C++ is almost always mapped to **word**. (Except of AMD64 architecture where *int* is still 32-bit one, perhaps, for the reason of better portability.)

*int* is 16-bit on PDP-11 and old MS-DOS compilers. *int* is 32-bit on VAX, on x86 starting at 80386, etc.

Even more than that, if type declaration for a variable is omitted in C/C++ program, *int* is used silently by default. Perhaps, this is inheritance of B programming language<sup>8</sup>.

---

**GPR** is usually fastest container for variable, faster than packed bit, and sometimes even faster than byte (because there is no need to isolate a single bit/byte from **GPR**). Even if you use it as a container for loop counter in 0..99 range.

---

**Word** in assembly language is still 16-bit for x86, because it was so for 16-bit 8086. *Double word* is 32-bit, *quad word* is 64-bit. That's why 16-bit words are declared using **DW** in x86 assembly, 32-bit ones using **DD** and 64-bit ones using **DQ**.

**Word** is 32-bit for ARM, MIPS, etc., 16-bit data types are called *half-word* there. Hence, *double word* on 32-bit RISC is 64-bit data type.

*GDB* has the following terminology: *halfword* for 16-bit, **word** for 32-bit and *giant word* for 64-bit.

16-bit C/C++ environment on PDP-11 and MS-DOS has *long* data type with width of 32 bits, perhaps, they meant *long word* or *long int*?

32-bit C/C++ environment has *long long* data type with width of 64 bits.

Now you see why the **word** word is ambiguous.

---

<sup>5</sup>[https://en.wikipedia.org/wiki/Apollo\\_Guidance\\_Computer](https://en.wikipedia.org/wiki/Apollo_Guidance_Computer)

<sup>6</sup><https://en.wikipedia.org/wiki/TX-0>

<sup>7</sup><https://en.wikipedia.org/wiki/DECSYSTEM-20>

<sup>8</sup><http://yurichev.com/blog/typeless/>

**Should I use *int*?**

Some people argue that *int* shouldn't be used at all, because it ambiguity can lead to bugs. For example, well-known *lzhuf* library uses *int* at one point and everything works fine on 16-bit architecture. But if ported to architecture with 32-bit *int*, it can crash: <http://yurichev.com/blog/lzhuf/>.

Less ambiguous types are defined in *stdint.h* file: *uint8\_t*, *uint16\_t*, *uint32\_t*, *uint64\_t*, etc.

Some people like Donald E. Knuth proposed<sup>9</sup> more sonorous words for these types: *byte/wyde/tetra-byte/octabyte*. But these names are less popular than clear terms with inclusion of *u* (*unsigned*) character and number right into the type name.

**Word-oriented computers**

Despite the ambiguity of the *word* term, modern computers are still word-oriented: **RAM** and all levels of cache are still organized by words, not by bytes. However, size in bytes is used in marketing.

Access to RAM/cache by address aligned by word boundary is often cheaper than non-aligned.

During data structures development, which are supposed to be fast and efficient, one should always take into consideration length of the *word* on the CPU to be executed on. Sometimes compiler do this for programmer, sometimes not.

**2.1.7 Address register**

For those who fostered on 32-bit and/or 64-bit x86, and/or RISC of 90s like ARM, MIPS, PowerPC, it's natural that address bus has the same width as **GPR** or *word*. Nevertheless, width of address bus can be different on other architectures.

8-bit Z80 can address  $2^{16}$  bytes, using 8-bit registers pairs or dedicated registers (*IX*, *IY*). *SP* and *PC* registers are also 16-bit ones.

Cray-1 supercomputer has 64-bit GPRs, but 24-bit address registers, so it can address  $2^{24}$  (16 megawords or 128 megabytes). RAM was very expensive in 1970s, and even in supercomputing environment it cannot be expected it could have more. So why to allocate 64-bit register for address or pointer?

8086/8088 CPUs has really weird addressing scheme: values of two 16-bit registers were summed in weird manner resulting 20-bit address. Perhaps, this was some kind of toy-level virtualization ([10.6 on page 992](#))? 8086 could run several programs (not simultaneously, though).

Early ARM1 has interesting artifact:

Another interesting thing about the register file is the PC register is missing a few bits. Since the ARM1 uses 26-bit addresses, the top 6 bits are not used. Because all instructions are aligned on a 32-bit boundary, the bottom two address bits in the PC are always zero. These 8 bits are not only unused, they are omitted from the chip entirely.

( <http://www.righto.com/2015/12/reverse-engineering-arm1-ancestor-of.html> )

Hence, it's physically not possible to push a value with one of two last bits set into PC register. Nor it's possible to set any bits in high 6 bits of PC.

x86-64 architecture has virtual 64-bit pointers/addresses, but internally, width of address bus is 48 bits (seems enough to address 256TB of **RAM**).

**2.1.8 Numbers**

What numbers are used for?

When you see some number(s) altering in CPU register, you may be interesting, what this number means. It's an important skill of reverse engineer to determine possible data type from a set of changing numbers.

<sup>9</sup><http://www-cs-faculty.stanford.edu/~uno/news98.html>

**Boolean**

If the number is switching from 0 to 1 and back, most chances that this value has boolean data type.

**Loop counter, array index**

Variable increasing from 0, like: 0, 1, 2, 3...—a good chance this is loop counter and/or array index.

**Signed numbers**

If you see a variable which holds very low numbers and sometimes very high numbers, like 0, 1, 2, 3, and 0xFFFFFFFF, 0xFFFFFFF, 0xFFFFFD, it's a good chance this is a signed variable in *two's complement* form ([2.2 on the following page](#)), and last 3 numbers are -1, -2, -3.

**32-bit numbers**

There are numbers so large<sup>10</sup>, that there is even special notation for it exist (Knuth's up-arrow notation<sup>11</sup>). These numbers are so large so these are not practical for engineering, science and mathematics.

Almost all engineers and scientists are happy with IEEE 754 double precision floating point, which has maximal value around  $1.8 \cdot 10^{308}$ . (As a comparison, the number of atoms in the observable universe, is estimated to be between  $4 \cdot 10^{79}$  and  $4 \cdot 10^{81}$ .)

In fact, upper bound in practical computing is much, much lower. If you get source code of UNIX v6 for PDP-11<sup>12</sup>, 16-bit *int* is used everywhere while 32-bit *long* type is not used at all.

Same story was in MS-DOS era: 16-bit *int* was used almost for everything (array indices, loop counters), while 32-bit *long* was used rarely.

During advent of x86-64, it was decided for *int* to stay as 32 bit size integer, because, probably, usage of 64-bit *int* is even rarer.

I would say, 16-bit numbers in range 0..65535 are probably most used numbers in computing.

Given that, if you see unusually large 32-bit value like 0x87654321, this is a good chance this can be:

- this can be still 16-bit number, but signed, between 0xFFFF8000 (-32768) and 0xFFFFFFF (-1);
- address of memory cell (can be checked using memory map feature of debugger);
- packed bytes (can be checked visually);
- bit flags;
- something related to (amateur) cryptography;
- magic number ([5.6.1 on page 718](#));
- IEEE 754 floating point number (can also be checked).

Almost same story for 64-bit values.

**...so 16-bit *int* is enough for almost everything?**

It's interesting to note: in [Michael Abrash, *Graphics Programming Black Book*, 1997 chapter 13] we can find that there are plenty cases in which 16-bit variables are just enough. In a meantime, Michael Abrash has a pity that 80386 and 80486 CPUs has so little available registers, so he offers to put two 16-bit values into one 32-bit register and then to rotate it using `ROR reg, 16` (on 80386 and later) (`ROL reg, 16` will also work) or `BSWAP` (on 80486 and later) instruction.

That reminds us Z80 with alternate pack of registers (suffixed with apostrophe), to which CPU can switch (and then switch back) using `EXX` instruction.

<sup>10</sup>[https://en.wikipedia.org/wiki/Large\\_numbers](https://en.wikipedia.org/wiki/Large_numbers)

<sup>11</sup>[https://en.wikipedia.org/wiki/Knuth%27s\\_up-arrow\\_notation](https://en.wikipedia.org/wiki/Knuth%27s_up-arrow_notation)

<sup>12</sup>[http://minnie.tuhs.org/Archive/PDP-11/Distributions/research/Dennis\\_v6/](http://minnie.tuhs.org/Archive/PDP-11/Distributions/research/Dennis_v6/)

**Size of buffer**

When a programmer need to declare a size of some buffer, values in form of  $2^x$  are usually used (512 bytes, 1024, etc.). Values in  $2^x$  form are easily recognizable ([1.22.5 on page 322](#)) in decimal, hexadecimal and binary base.

But needless to say, programmers are still humans with their decimal culture. And somehow, in **DBMS** area, size of textual database fields is often choosen as  $10^x$  number, like 100, 200. They just think “Okay, 100 is enough, wait, 200 will be better”. And they are right, of course.

Maximum width of **VARCHAR2** data type in Oracle RDBMS is 4000 characters, not 4096.

There is nothing wrong with this, this is just a place where numbers like  $10^x$  can be encountered.

**Address**

It's always good idea to keep in mind approximate memory map of the process you currently debug. For example, many win32 executables started at 0x00401000, so address like 0x00451230 is probably located inside executable section. You'll see addresses like these in **EIP** register.

Stack is usually located somewhere below.

Many debuggers is able to show the memory map of the debugger, for example: [1.9.3 on page 79](#).

If a value is increasing by step 4 on 32-bit architecture or by step 8 on 64-bit one, this probably sliding address of some elements of array.

It's important to know that win32 doesn't use addresses below 0x10000, so if you see some number below this constant, this cannot be an address (see also: <https://msdn.microsoft.com/en-us/library/ms810627.aspx>).

Anyway, many debuggers can show you if the value in a register can be an address to something. OllyDbg can also show an ASCII string if the value is an address of it.

**Bit field**

If you see a value where one (or more) bit(s) are flipping from time to time like 0xABCD1234 → 0xABCD1434 and back, this is probably bit field (or bitmap).

**Packed bytes**

When *strcmp()* or *memcmp()* copies a buffer, it loads/stores 4 (or 8) bytes simultaneously, so if a string containing “4321”, and it would be copied to another place, at one point you'll see 0x31323334 value in some register. This is 4 packed bytes into 32-bit value.

## 2.2 Signed number representations

There are several methods for representing signed numbers<sup>13</sup>, but “two's complement” is the most popular one in computers.

Here is a table for some byte values:

<sup>13</sup>[wikipedia](#)

## 2.2. SIGNED NUMBER REPRESENTATIONS

binary	hexadecimal	unsigned	signed
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000110	0x6	6	6
00000101	0x5	5	5
00000100	0x4	4	4
00000011	0x3	3	3
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
11111101	0xfd	253	-3
11111100	0xfc	252	-4
11111011	0xfb	251	-5
11111010	0xfa	250	-6
...			
10000010	0x82	130	-126
10000001	0x81	129	-127
10000000	0x80	128	-128

The difference between signed and unsigned numbers is that if we represent `0xFFFFFFFF` and `0x00000002` as unsigned, then the first number (4294967294) is bigger than the second one (2). If we represent them both as signed, the first one becomes -2, and it is smaller than the second (2). That is the reason why conditional jumps ([1.14 on page 125](#)) are present both for signed (e.g. `JG`, `JL`) and unsigned (`JA`, `JB`) operations.

For the sake of simplicity, this is what one needs to know:

- Numbers can be signed or unsigned.
- C/C++ signed types:

- `int64_t` (-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807) (- 9.2.. 9.2 quintillions)  
or  
`0x8000000000000000..0x7FFFFFFFFFFFFF`,
- `int` (-2,147,483,648..2,147,483,647 (- 2.15.. 2.15Gb) or  
`0x80000000..0x7FFFFFFF`),
- `char` (-128..127 or `0x80..0x7F`),
- `ssize_t`.

Unsigned:

- `uint64_t` (0..18,446,744,073,709,551,615 ( 18 quintillions) or `0..0xFFFFFFFFFFFFFF`),
- `unsigned int` (0..4,294,967,295 ( 4.3Gb) or `0..0xFFFFFFFF`),
- `unsigned char` (0..255 or `0..0xFF`),
- `size_t`.

- Signed types have the sign in the **MSB**: 1 means “minus”, 0 means “plus”.
- Promoting to a larger data types is simple: [1.28.5 on page 406](#).
- Negation is simple: just invert all bits and add 1.

We can keep in mind that a number of inverse sign is located on the opposite side at the same proximity from zero. The addition of one is needed because zero is present in the middle.

- The addition and subtraction operations work well for both signed and unsigned values. But for multiplication and division operations, x86 has different instructions: `IDIV / IMUL` for signed and `DIV / MUL` for unsigned.
- Here are some more instructions that work with signed numbers:  
`CBW/CWD/CWDE/CDQ/CDQE` ([.1.6 on page 1014](#)), `MOVSX` ([1.17.1 on page 201](#)), `SAR` ([.1.6 on page 1018](#)).

## 2.2. SIGNED NUMBER REPRESENTATIONS

A table of some negative and positive values ([??](#)) looks like thermometer with Celsius scale. This is why addition and subtraction works equally well for both signed and unsigned numbers: if the first addend is represented as mark on thermometer, and one need to add a second addend, and it's positive, we just shift mark up on thermometer by the value of second addend. If the second addend is negative, then we shift mark down to absolute value of the second addend.

Addition of two negative numbers works as follows. For example, we need to add -2 and -3 using 16-bit registers. -2 and -3 is 0xffffe and 0xffffd respectively. If we add these numbers as unsigned, we will get  $0xffffe + 0xffffd = 0x1fffb$ . But we work on 16-bit registers, so the result is *cut off*, the first 1 is dropped, 0xffffb is left, and this is -5. This works because -2 (or 0xffffe) can be represented using plain English like this: “2 lacks in this value up to maximal value in 16-bit register + 1”. -3 can be represented as “...3 lacks in this value up to ...”. Maximal value of 16-bit register + 1 is 0x10000. During addition of two numbers and *cutting off* by  $2^{16}$  modulo,  $2 + 3 = 5$  will be lacking.

### 2.2.1 Using IMUL over MUL

Example like listing.[3.22.2](#) where two unsigned values are multiplied compiles into listing.[3.22.2](#) where `IMUL` is used instead of `MUL`.

This is important property of both `MUL` and `IMUL` instructions. First of all, they both produce 64-bit value if two 32-bit values are multiplied, or 128-bit value if two 64-bit values are multiplied (biggest possible `product` in 32-bit environment is

`0xffffffff*0xffffffff=0xffffffffe00000001`). But C/C++ standards have no way to access higher half of result, and a `product` always has the same size as multiplicands. And both `MUL` and `IMUL` instructions works in the same way if higher half is ignored, i.e., they both generate the same lower half. This is important property of “two's complement” way of representing signed numbers.

So C/C++ compiler can use any of these instructions.

But `IMUL` is more versatile than `MUL` because it can take any register(s) as source, while `MUL` requires one of multiplicands stored in `AX / EAX / RAX` register. Even more than that: `MUL` stores result in `EDX:EAX` pair in 32-bit environment, or `RDX:RAX` in 64-bit one, so it always calculates the whole result. On contrary, it's possible to set a single destination register while using `IMUL` instead of pair, and then `CPU` will calculate only lower half, which works faster [see Torborn Granlund, *Instruction latencies and throughput for AMD and Intel x86 processors*<sup>14</sup>].

Given than, C/C++ compilers may generate `IMUL` instruction more often than `MUL`.

Nevertheless, using compiler intrinsic, it's still possible to do unsigned multiplication and get *full* result. This is sometimes called *extended multiplication*. MSVC has intrinsic for this called `_emu`<sup>15</sup> and another one: `_umul128`<sup>16</sup>. GCC offer `_int128` data type, and if 64-bit multiplicands are first promoted to 128-bit ones, then a `product` is stored into another `_int128` value, then result is shifted by 64 bits right, you'll get higher half of result<sup>17</sup>.

### MulDiv() function in Windows

Windows has `MulDiv()` function <sup>18</sup>, fused multiply/divide function, it multiplies two 32-bit integers into intermediate 64-bit value and then divides it by a third 32-bit integer. It is easier than to use two compiler intrinsic, so Microsoft developers made a special function for it. And it seems, this is busy function, judging by its usage.

### 2.2.2 Couple of additions about two's complement form

#### Getting maximum number of some word

Maximum unsigned number is just a number where all bits are set: `0xFF...FF` (this is -1 if the `word` is treated as signed integer). So you take a `word`, set all bits and get the value:

<sup>14</sup><http://yurichev.com/mirrors/x86-timing.pdf>

<sup>15</sup>[https://msdn.microsoft.com/en-us/library/d2s81xt0\(v=vs.80\).aspx](https://msdn.microsoft.com/en-us/library/d2s81xt0(v=vs.80).aspx)

<sup>16</sup><https://msdn.microsoft.com/library/3dayytw9%28v=vs.100%29.aspx>

<sup>17</sup> Example: <http://stackoverflow.com/a/13187798>

<sup>18</sup>[https://msdn.microsoft.com/en-us/library/windows/desktop/aa383718\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa383718(v=vs.85).aspx)

## 2.3. AND

```
#include <stdio.h>

int main()
{
    unsigned int val=~0; // change to "unsigned char" to get maximal value for the unsigned✓
    ↵ 8-bit byte
    ↵ // 0-1 will also work, or just -1
    ↵ printf ("%u\n", val); // %u for unsigned
};
```

This is 4294967295 for 32-bit integer.

### Getting minimum number for some signed word

Minimum signed number is encoded as *0x80....00*, i.e., most significant bit is set, while others are cleared. Maximum signed number is encoded in the same way, but all bits are inverted: *0x7F....FF*.

Let's shift a lone bit left until it disappears:

```
#include <stdio.h>

int main()
{
    signed int val=1; // change to "signed char" to find values for signed byte
    while (val!=0)
    {
        printf ("%d %d\n", val, ~val);
        val=val<<1;
    };
}
```

Output is:

```
...
536870912 -536870913
1073741824 -1073741825
-2147483648 2147483647
```

Two last numbers are minimum and maximum signed 32-bit *int* respectively.

## 2.3 AND

### 2.3.1 Checking if a value is on $2^n$ boundary

If you need to check if your value is divisible by  $2^n$  number (like 1024, 4096, etc.) without remainder, you can use a `%` operator in C/C++, but there is a simpler way. 4096 is *0x1000*, so it always has  $4 * 3 = 12$  lower bits cleared.

What you need is just:

```
if (value&0FFF)
{
    printf ("value is not divisible by 0x1000 (or 4096)\n");
    printf ("by the way, remainder is %d\n", value&0FFF);
}
else
    printf ("value is divisible by 0x1000 (or 4096)\n");
```

In other words, this code checks if there are any bit set among lower 12 bits. As a side effect, lower 12 bits is always a remainder from division a value by 4096 (because division by  $2^n$  is merely a right shift, and shifted (and dropped) bits are bits of remainder).

Same story if you want to check if the number is odd or even:

## 2.3. AND

```
if (value&1)
    // odd
else
    // even
```

This is merely the same as if to divide by 2 and get 1-bit remainder.

### 2.3.2 KOI-8R Cyrillic encoding

It was a time when 8-bit [ASCII](#) table wasn't supported by some Internet services, including email. Some supported, some others—not.

It was also a time, when non-Latin writing systems used second half of 8-bit ASCII table to accommodate non-Latin characters. There were several popular Cyrillic encodings, but KOI-8R (devised by Andrey "ache" Chernov) is somewhat unique in comparison with others.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0I																
1I																
2I	!	"	#	\$	%	&	'	(	)	*	,	-	.	/		
3I	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4I	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5I	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	^	_	
6I	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7I	р	q	г	с	т	у	в	ш	х	ъ	з	{	}	~		
8I	-		Г	Л	У	Т	І	Т	І	+	■	■	■	■	■	
9I	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
AI	=	Ї	ї	Ї	ї	Ї	ї	Ї	ї	Ї	ї	Ї	ї	Ї	ї	Ї
BI	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї	Ї
CI	ю	а	б	ц	д	е	ф	г	х	и	ў	к	л	м	н	о
DI	п	я	р	с	т	у	ж	в	ъ	ы	з	ш	э	щ	ч	ъ
EI	ю	а	б	ц	д	е	ф	г	х	и	й	к	л	м	н	о
FI	П	Я	Р	С	Т	У	Ж	В	Ъ	Ы	З	Ш	Э	Щ	Ч	Ъ

Figure 2.1: KOI8-R table

Someone may notice that Cyrillic characters are allocated almost in the same sequence as Latin ones. This leads to one important property: if all 8th bits in Cyrillic text encoded in KOI-8R are to be reset, a text transforms into transliterated text with Latin characters in place of Cyrillic. For example, Russian sentence:

Мой дядя самых честных правил, Когда не в шутку занемог, Он уважать себя заставил, И лучше выдумать не мог.

...if encoded in KOI-8R and then 8th bit stripped, transforms into:

mOJ DQQD SAMYH ^ESTNYH PRAWIL, kOGDA NE W [UTKU ZANEMOG, oN UWAVATX SEBQ ZASTAWIL, i LU^E WYDUMATX NE MOG.

...perhaps this is not very appealing aesthetically, but this text is still readable to Russian language natives.

Hence, Cyrillic text encoded in KOI-8R, passed through an old 7-bit service will survive into transliterated, but still readable text.

Stripping 8th bit is automatically transposes any character from the second half of the (any) 8-bit [ASCII](#) table to the first one, into the same place (take a look at red arrow right of table). If the character has already been placed in the first half (i.e., it has been in standard 7-bit [ASCII](#) table), it's not transposed.

Perhaps, transliterated text is still recoverable, if you'll add 8th bit to the characters which were seems transliterated.

Drawback is obvious: Cyrillic characters allocated in KOI-8R table are not in the same sequence as in Russian/Bulgarian/Ukrainian/etc. alphabet, and this isn't suitable for sorting, for example.

## 2.4 AND and OR as subtraction and addition

### 2.4.1 ZX Spectrum ROM text strings

Those who once investigated ZX Spectrum ROM internals, probably noticed that the last symbol of each text string is seemingly absent.

```
~(.....6.....i ..0.NEXT
without F0.Variable not
foun.Subscript wron.Out
of memor.Out of scree.N
umber too bi.RETURN with
out GOSU.End of fil.STOP
statemen.Invalid argume
n.Integer out of rang.No
nsense in BASI.BREAK - C
ONT repeat.Out of DAT.In
valid file nam.No room f
or lin.STOP in INPU.FOR
without NEX.Invalid I/O
devic.Invalid colou.BREA
K into progra.RAMTOP no
goo.Statement los.Invalid
strea.FN without DE.Pa
rameter erro.Tape loadin
g erro.... 1982 Sinclair
Research Lt.>.....CI
```

Figure 2.2: Part of ZX Spectrum ROM

There are present, in fact.

Here is excerpt of ZX Spectrum 128K ROM disassembled:

L048C:	DEFM "MERGE erro"	; Report 'a'.
	DEFB 'r'+\$80	
L0497:	DEFM "Wrong file typ"	; Report 'b'.
	DEFB 'e'+\$80	
L04A6:	DEFM "CODE erro"	; Report 'c'.
	DEFB 'r'+\$80	
L04B0:	DEFM "Too many bracket"	; Report 'd'.
	DEFB 's'+\$80	
L04C1:	DEFM "File already exist"	; Report 'e'.
	DEFB 's'+\$80	

( [http://www.matthew-wilson.net/spectrum/rom/128\\_ROM0.html](http://www.matthew-wilson.net/spectrum/rom/128_ROM0.html) )

Last character has most significant bit set, which marks string end. Presumably, it was done to save some space? Old 8-bit computers has very tight environment.

Characters of all messages are always in standard 7-bit ASCII table, so it's guaranteed 8th bit is never used for characters.

To print such string, we must check MSB of each byte, and if it's set, we must clear it, then print character, and then stop. Here is a C example:

```
unsigned char hw[]=
{
    'H',
    'e',
    'l',
    'l',
    'o'|0x80
};

void print_string()
{
```

## 2.4. AND AND OR AS SUBTRACTION AND ADDITION

```
for (int i=0; ;i++)
{
    if (hw[i]&0x80) // check MSB
    {
        // clear MSB
        // (in other words, clear all, but leave 7 lower bits intact)
        printf ("%c", hw[i] & 0x7F);
        // stop
        break;
    };
    printf ("%c", hw[i]);
};

};
```

Now what is interesting, since 8th bit is the most significant bit (in byte), we can check it, set it and remove it using arithmetical operations instead of logical.

I can rewrite my C example:

```
unsigned char hw[]=
{
    'H',
    'e',
    'l',
    'l',
    'o'+0x80
};

void print()
{
    for (int i=0; ;i++)
    {
        // hw[] must have 'unsigned char' type
        if (hw[i] >= 0x80) // check for MSB
        {
            printf ("%c", hw[i]-0x80); // clear MSB
            // stop
            break;
        };
        printf ("%c", hw[i]);
    };
}
```

By default, *char* is signed type in C/C++, so to compare it with variable like 0x80 (which is negative (-128) if treated as signed), we must treat each character in text message as unsigned.

Now if 8th bit is set, the number is always larger or equal to 0x80. If 8th bit is clear, the number is always smaller than 0x80.

Even more than that: if 8th bit is set, it can be cleared by subtracting 0x80, nothing else. If it's not set beforehand, however, subtracting will destruct other bits.

Likewise, if 8th bit is clear, it's possible to set it by adding 0x80. But if it's set beforehand, addition operation will destruct some other bits.

In fact, this is valid for any bit. If the 4th bit is clear, you can set it just by adding 0x10:  $0x100+0x10 = 0x110$ . If the 4th bit is set, you can clear it by subtracting 0x10:  $0x1234-0x10 = 0x1224$ .

It works, because carry isn't happened during addition/subtraction. It will, however, happen, if the bit is already set there before addition, or absent before subtraction.

Likewise, addition/subtraction can be replaced using OR/AND operation if two conditions are met: 1) you want to add/subtract by a number in form of  $2^n$ ; 2) this bit in source value is clear/set.

For example, addition of 0x20 is the same as ORing value with 0x20 under condition that this bit is clear before:  $0x1204|0x20 = 0x1204+0x20 = 0x1224$ .

Subtraction of 0x20 is the same as ANDing value with 0x20 (0x....FFDF), but if this bit is set before:  $0x1234&(~0x20) = 0x1234\&0xFFDF = 0x1234-0x20 = 0x1214$ .

Again, it works because carry not happened when you add  $2^n$  number and this bit isn't set before.

This property of boolean algebra is important, worth understanding and keeping it in mind.

## 2.5 XOR (exclusive OR)

XOR is widely used when one needs just to flip specific bit(s). Indeed, the XOR operation applied with 1 effectively inverts a bit:

input A	input B	output
0	0	0
0	1	1
1	0	1
1	1	0

And vice-versa, the XOR operation applied with 0 does nothing, i.e., it's an idle operation. This is a very important property of the XOR operation and it's highly recommended to memorize it.

### 2.5.1 Everyday speech

XOR operation present in common everyday speech. When someone asks “please buy apples or bananas”, this usually means “buy the first object or the second, but not both”—this is exactly exclusive OR, because logical OR would mean “both objects are also fine”.

Some people suggest “and/or” should be used in everyday speech to make emphasis that logical OR is used instead of exclusive OR: <https://en.wikipedia.org/wiki/And/or>.

### 2.5.2 Encryption

XOR is heavily used in both amateur (9.1) and *real* encryption (at least in *Feistel network*).

XOR is very useful here because:  $\text{cipher\_text} = \text{plain\_text} \oplus \text{key}$  and then:  $(\text{plain\_text} \oplus \text{key}) \oplus \text{key} = \text{plain\_text}$ .

### 2.5.3 RAID4

RAID4 offers a very simple method to protect hard disks. For example, there are several disks ( $D_1$ ,  $D_2$ ,  $D_3$ , etc.) and one parity disk ( $P$ ). Each bit/byte written to parity disk is calculated and written on-fly:

$$P = D_1 \oplus D_2 \oplus D_3 \quad (2.1)$$

If any of disks is failed, for example,  $D_2$ , it's restored using the very same way:

$$D_2 = D_1 \oplus P \oplus D_3 \quad (2.2)$$

If parity disk failed, it is restored using 2.1 way. If two of any disks are failed, then it wouldn't be possible to restore both.

RAID5 is more advanced, but this XOR property is still exploited there.

That's why RAID controllers has hardware “XOR accelerators” helping to XOR large chunks of written data on-fly. When computers get faster and faster, it now can be done at software level, using SIMD.

### 2.5.4 XOR swap algorithm

Hard to believe, but this code swaps values in EAX and EBX without aid of any other additional register or memory cell:

```
xor eax, ebx
xor ebx, eax
xor eax, ebx
```

Let's find out, how it works. First, we will rewrite it to step aside from x86 assembly language:

## 2.5. XOR (EXCLUSIVE OR)

```
X = X XOR Y  
Y = Y XOR X  
X = X XOR Y
```

What X and Y has at each step? Just keep in mind the simple rule:  $(X \oplus Y) \oplus Y = X$  for any values of X and Y.

Let's see, X after 1st step has  $X \oplus Y$ ; Y after 2nd step has  $Y \oplus (X \oplus Y) = X$ ; X after 3rd step has  $(X \oplus Y) \oplus X = Y$ .

Hard to say if anyone should use this trick, but it servers as a good demonstration example of XOR properties.

Wikipedia article ([https://en.wikipedia.org/wiki/XOR\\_swap\\_algorithm](https://en.wikipedia.org/wiki/XOR_swap_algorithm)) has also yet another explanation: addition and subtraction operations can be used instead of XOR:

```
X = X + Y  
Y = X - Y  
X = X - Y
```

Let's see: X after 1st step has  $X + Y$ ; Y after 2nd step has  $X + Y - Y = X$ ; X after 3rd step has  $X + Y - X = Y$ .

### 2.5.5 XOR linked list

Doubly linked list is a list in which each element has link to the previous element and to the next one. Hence, it's very easy to traverse list backwards or forward. `std::list` in C++ implements doubly linked list which also is examined in this book: [3.19.4](#).

So each element has two pointers. Is it possible, perhaps in environment of small **RAM** footprint, to preserve all functionality with one pointer instead of two? Yes, if it a value of  $prev \oplus next$  will be stored in this memory cell, which is usually called "link".

Maybe, we could say that address to the previous element is "encrypted" using address of next element and otherwise: next element address is "encrypted" using previous element address.

When we traverse this list forward, we always know address of the previous element, so we can "decrypt" this field and get address of the next element. Likewise, it's possible to traverse this list backwards, "decrypting" this field using next element's address.

But it's not possible to find address of previous or next element of some specific element without knowing address of the first one.

Couple of things to complete this solution: first element will have address of next element without any XOR-ing, last element will have address of previous element without any XOR-ing.

Now let's sum it up. This is example of doubly linked list of 5 elements.  $A_x$  is address of element.

address	link field contents
$A_0$	$A_1$
$A_1$	$A_0 \oplus A_2$
$A_2$	$A_1 \oplus A_3$
$A_3$	$A_2 \oplus A_4$
$A_4$	$A_3$

And again, hard to say if anyone should use this tricky hacks, but this is also a good demonstration of XOR properties. As with XOR swap algorithm, Wikipedia article about it also offers way to use addition or subtraction instead of XOR: [https://en.wikipedia.org/wiki/XOR\\_linked\\_list](https://en.wikipedia.org/wiki/XOR_linked_list).

### 2.5.6 By the way

The usual **OR** also sometimes called *inclusive OR* (or even *IOR*), as opposed to *exclusive OR*.

## 2.5.7 AND/OR/XOR as MOV

`OR reg, 0xFFFFFFFF` sets all bits to 1, hence, no matter what has been in register before, it will be set to -1. `OR reg, -1` is shorter than `MOV reg, -1`, so MSVC uses OR instead the latter, for example: [3.15.1 on page 526](#).

Likewise, `AND reg, 0` always resets all bits, hence, it acts like `MOV reg, 0`.

`XOR reg, reg`, no matter what has been in register beforehand, resets all bits, and also acts like `MOV reg, 0`.

## 2.6 Population count

`POPCNT` instruction is population count (AKA Hamming weight). It just counts number of bits set in an input value.

As a side effect, `POPCNT` instruction (or operation) can be used to determine, if the value has  $2^n$  form. Since,  $2^n$  number always has just one single bit, `POPCNT`'s result will always be just 1.

For example, I once wrote a base64 strings scanner for hunting something interesting in binary files<sup>19</sup>. And there is a lot of garbage and false positives, so I add an option to filter out data blocks which has size of  $2^n$  bytes (i.e., 256 bytes, 512, 1024, etc.). The size of block is checked just like this:

```
if (popcnt(size)==1)
    // OK
...
...
```

The instruction is also known as “[NSA<sup>20</sup>](#) instruction” due to rumors:

This branch of cryptography is fast-paced and very politically charged. Most designs are secret; a majority of military encryptions systems in use today are based on LFSRs. In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as “population count.” It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR. I’ve heard this called the canonical NSA instruction, demanded by almost all computer contracts.

[Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)]

## 2.7 Endianness

The endianness is a way of representing values in memory.

### 2.7.1 Big-endian

The `0x12345678` value is represented in memory as:

address in memory	byte value
+0	0x12
+1	0x34
+2	0x56
+3	0x78

Big-endian CPUs include Motorola 68k, IBM POWER.

<sup>19</sup><https://github.com/dennis714/base64scanner>

<sup>20</sup>National Security Agency

## 2.7.2 Little-endian

The `0x12345678` value is represented in memory as:

address in memory	byte value
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Little-endian CPUs include Intel x86.

### 2.7.3 Example

Let's take big-endian MIPS Linux installed and ready in QEMU <sup>21</sup>.

And let's compile this simple example:

```
#include <stdio.h>

int main()
{
    int v, i;

    v=123;

    printf ("%02X %02X %02X %02X\n",
            *(char*)&v,
            *(((char*)&v)+1),
            *(((char*)&v)+2),
            *(((char*)&v)+3));
}
```

After running it we get:

```
root@debian-mips:~# ./a.out
00 00 00 7B
```

That is it. `0x7B` is 123 in decimal. In little-endian architectures, `7B` is the first byte (you can check on x86 or x86-64), but here it is the last one, because the highest byte goes first.

That's why there are separate Linux distributions for MIPS ("mips" (big-endian) and "mipsel" (little-endian)). It is impossible for a binary compiled for one endianness to work on an [OS](#) with different endianness.

There is another example of MIPS big-endiannes in this book: [1.24.4 on page 366](#).

### 2.7.4 Bi-endian

CPUs that may switch between endianness are ARM, PowerPC, SPARC, MIPS, [IA64](#)<sup>22</sup>, etc.

### 2.7.5 Converting data

The `BSWAP` instruction can be used for conversion.

TCP/IP network data packets use the big-endian conventions, so that is why a program working on a little-endian architecture has to convert the values. The `htonl()` and `htons()` functions are usually used.

In TCP/IP, big-endian is also called "network byte order", while byte order on the computer "host byte order". "host byte order" is little-endian on Intel x86 and other little-endian architectures, but it is big-endian on IBM POWER, so `htonl()` and `htons()` don't shuffle any bytes on the latter.

<sup>21</sup>Available for download here: <http://go.yurichev.com/17008>

<sup>22</sup>Intel Architecture 64 (Itanium): [10.5 on page 990](#)

## 2.8 Memory

There are 3 main types of memory:

- Global memory **AKA** “static memory allocation”. No need to allocate explicitly, the allocation is performed just by declaring variables/arrays globally. These are global variables, residing in the data or constant segments. They are available globally (hence, considered as an **anti-pattern**). Not convenient for buffers/arrays, because they must have a fixed size. Buffer overflows that occur here usually overwrite variables or buffers residing next to them in memory. There’s an example in this book: [1.9.3 on page 76](#).
- Stack **AKA** “allocate on stack”. The allocation is performed just by declaring variables/arrays locally in the function. These are usually local variables for the function. Sometimes these local variable are also available to descending functions (to **callee** functions, if caller passes a pointer to a variable to the **callee** to be executed). Allocation and deallocation are very fast, it just **SP** needs to be shifted.

But they’re also not convenient for buffers/arrays, because the buffer size has to be fixed, unless `alloca()` ([1.7.2 on page 35](#)) (or a variable-length array) is used. Buffer overflows usually overwrite important stack structures: [1.20.2 on page 275](#).

- Heap **AKA** “dynamic memory allocation”. Allocation/deallocation is performed by calling `malloc()/free()` or `new/delete` in C++. This is the most convenient method: the block size may be set at runtime.

Resizing is possible (using `realloc()`), but can be slow. This is the slowest way to allocate memory: the memory allocator must support and update all control structures while allocating and deallocating. Buffer overflows usually overwrite these structures. Heap allocations are also source of memory leak problems: each memory block has to be deallocated explicitly, but one may forget about it, or do it incorrectly.

Another problem is the “use after free”—using a memory block after `free()` has been called on it, which is very dangerous.

Example in this book: [1.24.2 on page 349](#).

## 2.9 CPU

### 2.9.1 Branch predictors

Some latest compilers try to get rid of conditional jump instructions. Examples in this book are: [1.14.1 on page 136](#), [1.14.3 on page 144](#), [1.22.5 on page 331](#).

This is because the branch predictor is not always perfect, so the compilers try to do without conditional jumps, if possible.

Conditional instructions in ARM (like ADRcc) are one way, another one is the CMOVcc x86 instruction.

### 2.9.2 Data dependencies

Modern CPUs are able to execute instructions simultaneously (**OOE**<sup>23</sup>), but in order to do so, the results of one instruction in a group must not influence the execution of others. Hence, the compiler endeavors to use instructions with minimal influence on the CPU state.

That’s why the `LEA` instruction is so popular, because it does not modify CPU flags, while other arithmetic instructions does.

## 2.10 Hash functions

A very simple example is CRC32, an algorithm that provides “stronger” checksum for integrity checking purposes. It is impossible to restore the original text from the hash value, it has much less information:

<sup>23</sup>Out-of-order execution

## 2.10. HASH FUNCTIONS

---

But CRC32 is not cryptographically secure: it is known how to alter a text in a way that the resulting CRC32 hash value will be the one we need. Cryptographic hash functions are protected from this.

MD5, SHA1, etc. are such functions and they are widely used to hash user passwords in order to store them in a database. Indeed: an Internet forum database may not contain user passwords (a stolen database can compromise all users' passwords) but only hashes (so a cracker can't reveal the passwords). Besides, an Internet forum engine does not need to know your password exactly, it needs only to check if its hash is the same as the one in the database, and give you access if they match. One of the simplest password cracking methods is just to try hashing all possible passwords in order to see which matches the resulting value that we need. Other methods are much more complex.

### 2.10.1 How do one-way functions work?

A one-way function is a function which is able to transform one value into another, while it is impossible (or very hard) to reverse it. Some people have difficulties while understanding how this is possible at all. Here is a simple demonstration.

We have a vector of 10 numbers in range 0..9, each is present only once, for example:

```
4 6 0 1 3 5 7 8 9 2
```

The algorithm for the simplest possible one-way function is:

- take the number at zeroth position (4 in our case);
- take the number at first position (6 in our case);
- swap numbers at positions of 4 and 6.

Let's mark the numbers at positions 4 and 6:

```
4 6 0 1 3 5 7 8 9 2  
  ^   ^
```

Let's swap them and we get this result:

```
4 6 0 1 7 5 3 8 9 2
```

While looking at the result, and even if we know the algorithm, we can't know unambiguously the initial state, because the first two numbers could be 0 and/or 1, and then they could participate in the swapping procedure.

This is an utterly simplified example for demonstration. Real one-way functions are much more complex.

# Chapter 3

## Slightly more advanced examples

### 3.1 Double negation

A popular way<sup>1</sup> to convert non-zero value into 1 (or boolean *true*) and zero value into 0 (or boolean *false*) is `!!statement`:

```
int convert_to_bool(int a)
{
    return !!a;
};
```

Optimizing GCC 5.4 x86:

```
convert_to_bool:
    mov    edx, DWORD PTR [esp+4]
    xor    eax, eax
    test   edx, edx
    setne al
    ret
```

`XOR` always clears return value in `EAX`, even in case if `SETNE` will not trigger. I.e., `XOR` sets default return value to zero.

If the input value is not equal to zero (`-NE` suffix in `SET` instruction), 1 is set to `AL`, otherwise `AL` isn't touched.

Why `SETNE` operates on low 8-bit part of `EAX` register? Because the matter is just in the last bit (0 or 1), while other bits are cleared by `XOR`.

Therefore, that C/C++ code could be rewritten like this:

```
int convert_to_bool(int a)
{
    if (a!=0)
        return 1;
    else
        return 0;
};
```

...or even:

```
int convert_to_bool(int a)
{
    if (a)
        return 1;
    else
        return 0;
};
```

Compilers targeting CPUs lacking instruction similar to `SET`, in this case, generates branching instructions, etc.

<sup>1</sup>This way is also controversial, because it leads to hard-to-read code

## 3.2 `strstr()` example

Let's back to the fact that GCC sometimes can use part of string: [1.5.3 on page 18](#).

The `strstr()` C/C++ standard library function is used to find any occurrence in a string. This is what we will do:

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

int main()
{
    char *s="Hello, world!";
    char *w=strstr(s, "world");

    printf ("%p, [%s]\n", s, s);
    printf ("%p, [%s]\n", w, w);
}
```

The output is:

```
0x8048530, [Hello, world!]
0x8048537, [world!]
```

The difference between the address of the original string and the address of the substring that `strstr()` has returned is 7. Indeed, "Hello, " string has length of 7 characters.

The `printf()` function during second call has no idea there are some other characters before the passed string and it prints characters from the middle of original string till the end (marked by zero byte).

## 3.3 Temperature converting

Another very popular example in programming books for beginners is a small program that converts Fahrenheit temperature to Celsius or back.

$$C = \frac{5 \cdot (F - 32)}{9}$$

We can also add simple error handling: 1) we must check if the user has entered a correct number; 2) we must check if the Celsius temperature is not below -273 (which is below absolute zero, as we may recall from school physics lessons).

The `exit()` function terminates the program instantly, without returning to the `caller` function.

### 3.3.1 Integer values

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%d", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    }

    celsius = 5 * (fahr-32) / 9;

    if (celsius<-273)
    {
        printf ("Error: incorrect temperature!\n");
    }
}
```

### 3.3. TEMPERATURE CONVERTING

```
        exit(0);
    };
    printf ("Celsius: %d\n", celsius);
};
```

## Optimizing MSVC 2012 x86

Listing 3.1: Optimizing MSVC 2012 x86

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%d', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %d', 0aH, 00H

_fahr$ = -4      ; size = 4
main   PROC
    push  ecx
    push  esi
    mov   esi, DWORD PTR __imp__printf
    push  OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
    call  esi                   ; call printf()
    lea   eax, DWORD PTR _fahr$[esp+12]
    push  eax
    push  OFFSET $SG4230          ; '%d'
    call  DWORD PTR __imp__scanf
    add   esp, 12
    cmp   eax, 1
    je    SHORT $LN2@main
    push  OFFSET $SG4231          ; 'Error while parsing your input'
    call  esi                   ; call printf()
    add   esp, 4
    push  0
    call  DWORD PTR __imp__exit

$LN9@main:
$LN2@main:
    mov   eax, DWORD PTR _fahr$[esp+8]
    add   eax, -32              ; ffffffe0H
    lea   ecx, DWORD PTR [eax+eax*4]
    mov   eax, 954437177        ; 38e38e39H
    imul  ecx
    sar   edx, 1
    mov   eax, edx
    shr   eax, 31              ; 00000001fH
    add   eax, edx
    cmp   eax, -273             ; ffffffeefH
    jge   SHORT $LN1@main
    push  OFFSET $SG4233          ; 'Error: incorrect temperature!'
    call  esi                   ; call printf()
    add   esp, 4
    push  0
    call  DWORD PTR __imp__exit

$LN10@main:
$LN1@main:
    push  eax
    push  OFFSET $SG4234          ; 'Celsius: %d'
    call  esi                   ; call printf()
    add   esp, 8
    ; return 0 - by C99 standard
    xor   eax, eax
    pop   esi
    pop   ecx
    ret   0

$LN8@main:
_main   ENDP
```

What we can say about it:

### 3.3. TEMPERATURE CONVERTING

- The address of `printf()` is first loaded in the `ESI` register, so the subsequent `printf()` calls are done just by the `CALL ESI` instruction. It's a very popular compiler technique, possible if several consequent calls to the same function are present in the code, and/or if there is a free register which can be used for this.
- We see the `ADD EAX, -32` instruction at the place where 32 has to be subtracted from the value.  $EAX = EAX + (-32)$  is equivalent to  $EAX = EAX - 32$  and somehow, the compiler decided to use `ADD` instead of `SUB`. Maybe it's worth it, it's hard to be sure.
- The `LEA` instruction is used when the value is to be multiplied by 5: `lea ecx, DWORD PTR [eax+eax*4]`. Yes,  $i + i * 4$  is equivalent to  $i * 5$  and `LEA` works faster than `IMUL`. By the way, the `SHL EAX, 2 / ADD EAX, EAX` instruction pair could be also used here instead—some compilers do it like.
- The division by multiplication trick ([3.9 on page 496](#)) is also used here.
- `main()` returns 0 if we don't have `return 0` at its end. The C99 standard tells us [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)5.1.2.2.3] that `main()` will return 0 in case the `return` statement is missing. This rule works only for the `main()` function.

Though, MSVC doesn't officially support C99, but maybe it supports it partially?

### Optimizing MSVC 2012 x64

The code is almost the same, but we can find `INT 3` instructions after each `exit()` call.

```
xor    ecx, ecx
call   QWORD PTR __imp_exit
int    3
```

`INT 3` is a debugger breakpoint.

It is known that `exit()` is one of the functions which can never return <sup>2</sup>, so if it does, something really odd has happened and it's time to load the debugger.

### 3.3.2 Floating-point values

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%lf", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    };

    celsius = 5 * (fahr-32) / 9;

    if (celsius<-273)
    {
        printf ("Error: incorrect temperature!\n");
        exit(0);
    };
    printf ("Celsius: %lf\n", celsius);
}
```

MSVC 2010 x86 uses [FPU](#) instructions...

<sup>2</sup>another popular one is `longjmp()`

### 3.3. TEMPERATURE CONVERTING

Listing 3.2: Optimizing MSVC 2010 x86

```
$SG4038 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB      '%lf', 00H
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius: %lf', 0aH, 00H

_real@c0711000000000000 DQ 0c071100000000000r ; -273
_real@4022000000000000 DQ 0402200000000000r ; 9
_real@4014000000000000 DQ 0401400000000000r ; 5
_real@4040000000000000 DQ 0404000000000000r ; 32

_fahr$ = -8      ; size = 8
main PROC
    sub esp, 8
    push esi
    mov esi, DWORD PTR __imp__printf
    push OFFSET $SG4038      ; 'Enter temperature in Fahrenheit:'
    call esi                 ; call printf()
    lea eax, DWORD PTR _fahr$[esp+16]
    push eax
    push OFFSET $SG4040      ; '%lf'
    call DWORD PTR __imp__scanf
    add esp, 12
    cmp eax, 1
    je SHORT $LN2@main
    push OFFSET $SG4041      ; 'Error while parsing your input'
    call esi                 ; call printf()
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
$LN2@main:
    fld QWORD PTR _fahr$[esp+12]
    fsub QWORD PTR __real@4040000000000000 ; 32
    fmul QWORD PTR __real@4014000000000000 ; 5
    fdiv QWORD PTR __real@4022000000000000 ; 9
    fld QWORD PTR __real@c071100000000000 ; -273
    fcomp ST(1)
    fnstsw ax
    test ah, 65 ; 00000041H
    jne SHORT $LN1@main
    push OFFSET $SG4043      ; 'Error: incorrect temperature!'
    fstp ST(0)
    call esi                 ; call printf()
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
$LN1@main:
    sub esp, 8
    fstp QWORD PTR [esp]
    push OFFSET $SG4044      ; 'Celsius: %lf'
    call esi
    add esp, 12
    ; return 0 - by C99 standard
    xor eax, eax
    pop esi
    add esp, 8
    ret 0
$LN10@main:
_main ENDP
```

...but MSVC 2012 uses [SIMD](#) instructions instead:

Listing 3.3: Optimizing MSVC 2010 x86

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%lf', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %lf', 0aH, 00H
```

### 3.4. FIBONACCI NUMBERS

```
_real@0c0711000000000000 DQ 0c071100000000000r ; -273
_real@4040000000000000 DQ 0404000000000000r ; 32
_real@4022000000000000 DQ 0402200000000000r ; 9
_real@4014000000000000 DQ 0401400000000000r ; 5

_fahr$ = -8      ; size = 8
_main    PROC
    sub     esp, 8
    push    esi
    mov     esi, DWORD PTR __imp__printf
    push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
    call    esi                  ; call printf()
    lea    eax, DWORD PTR _fahr$[esp+16]
    push    eax
    push    OFFSET $SG4230          ; '%lf'
    call    DWORD PTR __imp__scanf
    add     esp, 12
    cmp     eax, 1
    je      SHORT $LN2@main
    push    OFFSET $SG4231          ; 'Error while parsing your input'
    call    esi                  ; call printf()
    add     esp, 4
    push    0
    call    DWORD PTR __imp__exit

$LN9@main:
$LN2@main:
    movsd  xmm1, QWORD PTR _fahr$[esp+12]
    subsd  xmm1, QWORD PTR __real@4040000000000000 ; 32
    movsd  xmm0, QWORD PTR __real@c071100000000000 ; -273
    mulsd  xmm1, QWORD PTR __real@4014000000000000 ; 5
    divsd  xmm1, QWORD PTR __real@4022000000000000 ; 9
    comisd xmm0, xmm1
    jbe    SHORT $LN1@main
    push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
    call    esi                  ; call printf()
    add     esp, 4
    push    0
    call    DWORD PTR __imp__exit

$LN10@main:
$LN1@main:
    sub     esp, 8
    movsd  QWORD PTR [esp], xmm1
    push    OFFSET $SG4234          ; 'Celsius: %lf'
    call    esi                  ; call printf()
    add     esp, 12
    ; return 0 - by C99 standard
    xor     eax, eax
    pop     esi
    add     esp, 8
    ret     0

$LN8@main:
_main    ENDP
```

Of course, SIMD instructions are available in x86 mode, including those working with floating point numbers.

It's somewhat easier to use them for calculations, so the new Microsoft compiler uses them.

We can also see that the -273 value is loaded into `XMM0` register too early. And that's OK, because the compiler may emit instructions not in the order they are in the source code.

## 3.4 Fibonacci numbers

Another widespread example used in programming textbooks is a recursive function that generates the Fibonacci numbers<sup>3</sup>. The sequence is very simple: each consecutive number is the sum of the previous two. The first two numbers are 1's or 0, 1 and 1.

<sup>3</sup><http://go.yurichev.com/17332>

### 3.4. FIBONACCI NUMBERS

The sequence starts like this:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...

#### 3.4.1 Example #1

The implementation is simple. This program generates the sequence until 21.

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
    printf ("%d\n", a+b);
    if (a+b > limit)
        return;
    fib (b, a+b, limit);
}

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
}
```

Listing 3.4: MSVC 2010 x86

```
_a$ = 8          ; size = 4
_b$ = 12         ; size = 4
_limit$ = 16     ; size = 4
_fib    PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    push    eax
    push    OFFSET $SG2643
    call    DWORD PTR __imp__printf
    add     esp, 8
    mov     ecx, DWORD PTR _a$[ebp]
    add     ecx, DWORD PTR _b$[ebp]
    cmp     ecx, DWORD PTR _limit$[ebp]
    jle     SHORT $LN1@fib
    jmp     SHORT $LN2@fib
$LN1@fib:
    mov     edx, DWORD PTR _limit$[ebp]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    call    _fib
    add     esp, 12
$LN2@fib:
    pop    ebp
    ret    0
_fib    ENDP

_main   PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2647 ; "0\n1\n1\n"
    call    DWORD PTR __imp__printf
    add     esp, 4
    push    20
    push    1
    push    1
    call    _fib
    add     esp, 12
```

### 3.4. FIBONACCI NUMBERS

```
    xor    eax, eax
    pop    ebp
    ret    0
_main  ENDP
```

We will illustrate the stack frames with this.

### 3.4. FIBONACCI NUMBERS

Let's load the example in OllyDbg and trace to the last call of `f()`:

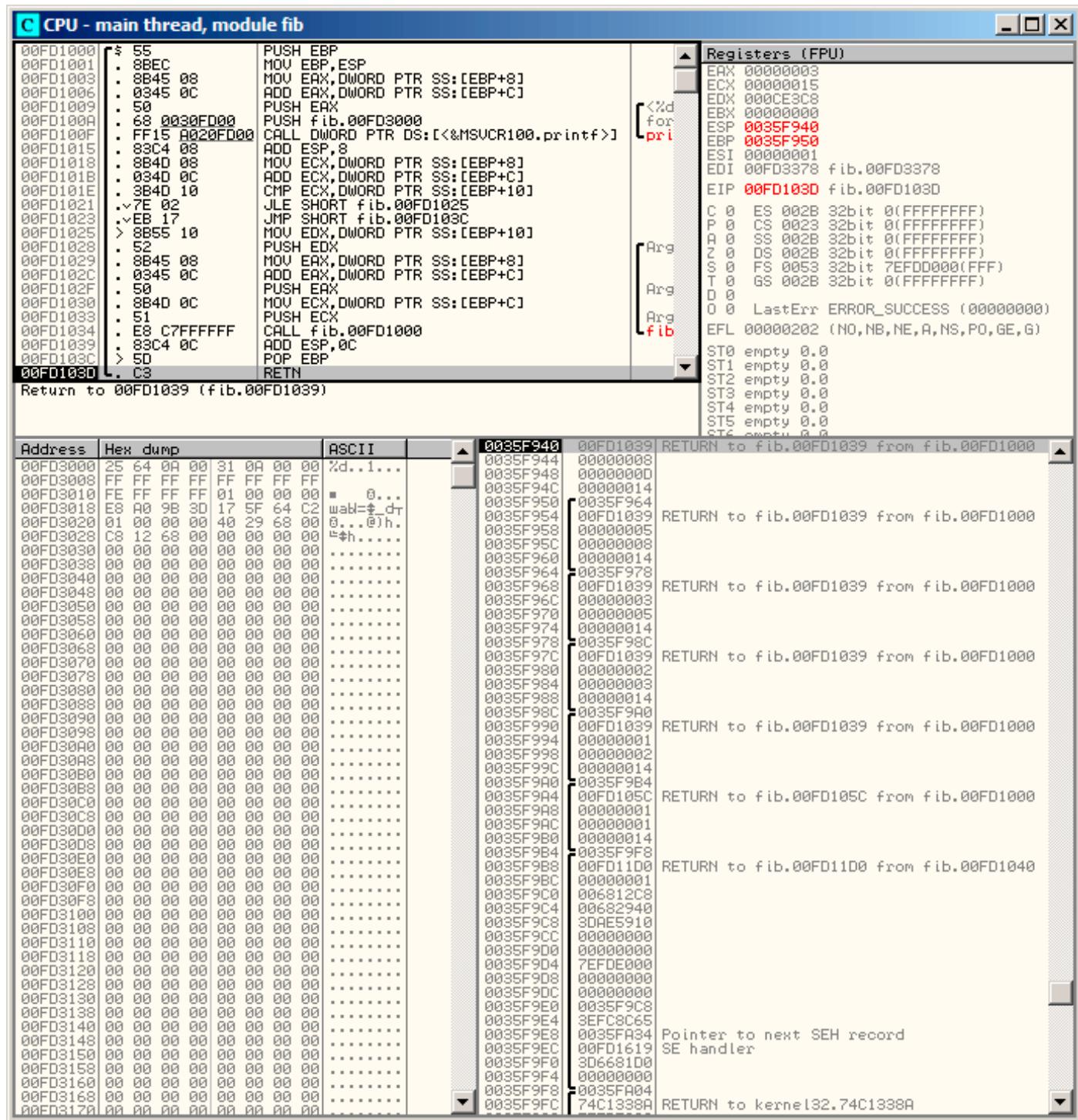


Figure 3.1: OllyDbg: last call of `f()`

### 3.4. FIBONACCI NUMBERS

Let's investigate the stack more closely. Comments were added by the author of this book <sup>4</sup>:

```
0035F940 00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F944 00000008 1st argument: a
0035F948 0000000D 2nd argument b
0035F94C 00000014 3rd argument: limit
0035F950 /0035F964 saved EBP register
0035F954 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F958 |00000005 1st argument: a
0035F95C |00000008 2nd argument: b
0035F960 |00000014 3rd argument: limit
0035F964 ]0035F978 saved EBP register
0035F968 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F96C |00000003 1st argument: a
0035F970 |00000005 2nd argument: b
0035F974 |00000014 3rd argument: limit
0035F978 ]0035F98C saved EBP register
0035F97C |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F980 |00000002 1st argument: a
0035F984 |00000003 2nd argument: b
0035F988 |00000014 3rd argument: limit
0035F98C ]0035F9A0 saved EBP register
0035F990 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F994 |00000001 1st argument: a
0035F998 |00000002 2nd argument: b
0035F99C |00000014 3rd argument: limit
0035F9A0 ]0035F9B4 saved EBP register
0035F9A4 |00FD105C RETURN to fib.00FD105C from fib.00FD1000
0035F9A8 |00000001 1st argument: a \
0035F9AC |00000001 2nd argument: b | prepared in main() for f1()
0035F9B0 |00000014 3rd argument: limit /
0035F9B4 ]0035F9F8 saved EBP register
0035F9B8 |00FD11D0 RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC |00000001 main() 1st argument: argc \
0035F9C0 |006812C8 main() 2nd argument: argv | prepared in CRT for main()
0035F9C4 |00682940 main() 3rd argument: envp /
```

The function is recursive <sup>5</sup>, hence stack looks like a “sandwich”.

We see that the *limit* argument is always the same (`0x14` or 20), but the *a* and *b* arguments are different for each call.

There are also the **RA**-s and the saved **EBP** values. OllyDbg is able to determine the EBP-based frames, so it draws these brackets. The values inside each bracket make the **stack frame**, in other words, the stack area which each function incarnation uses as scratch space.

We can also say that each function incarnation must not access stack elements beyond the boundaries of its frame (excluding function arguments), although it's technically possible.

It's usually true, unless the function has bugs.

Each saved **EBP** value is the address of the previous **stack frame**: this is the reason why some debuggers can easily divide the stack in frames and dump each function's arguments.

As we see here, each function incarnation prepares the arguments for the next function call.

At the end we see the 3 arguments for `main()`. `argc` is 1 (yes, indeed, we have ran the program without command-line arguments).

This easily lead to a stack overflow: just remove (or comment out) the limit check and it will crash with exception `0xC00000FD` (stack overflow.)

#### 3.4.2 Example #2

My function has some redundancy, so let's add a new local variable *next* and replace all “*a+b*” with it:

<sup>4</sup>By the way, it's possible to select several entries in OllyDbg and copy them to the clipboard (Ctrl-C). That's what was done by author for this example.

<sup>5</sup>i.e., it calls itself

### 3.4. FIBONACCI NUMBERS

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
    int next=a+b;
    printf ("%d\n", next);
    if (next > limit)
        return;
    fib (b, next, limit);
};

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
}
```

This is the output of non-optimizing MSVC, so the *next* variable is actually allocated in the local stack:

Listing 3.5: MSVC 2010 x86

```
_next$ = -4      ; size = 4
_a$ = 8          ; size = 4
_b$ = 12         ; size = 4
_limit$ = 16     ; size = 4
_fib PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _next$[ebp], eax
    mov     ecx, DWORD PTR _next$[ebp]
    push    ecx
    push    OFFSET $SG2751 ; '%d'
    call    DWORD PTR __imp__printf
    add     esp, 8
    mov     edx, DWORD PTR _next$[ebp]
    cmp     edx, DWORD PTR _limit$[ebp]
    jle     SHORT $LN1@fib
    jmp     SHORT $LN2@fib
$LN1@fib:
    mov     eax, DWORD PTR _limit$[ebp]
    push    eax
    mov     ecx, DWORD PTR _next$[ebp]
    push    ecx
    mov     edx, DWORD PTR _b$[ebp]
    push    edx
    call    _fib
    add     esp, 12
$LN2@fib:
    mov     esp, ebp
    pop     ebp
    ret     0
_fib ENDP

_main PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2753 ; "0\n1\n1\n"
    call    DWORD PTR __imp__printf
    add     esp, 4
    push    20
    push    1
    push    1
    call    _fib
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
```

### *3.4. FIBONACCI NUMBERS*

```
main    ENDP
```

### 3.4. FIBONACCI NUMBERS

Let's load it in OllyDbg once again:

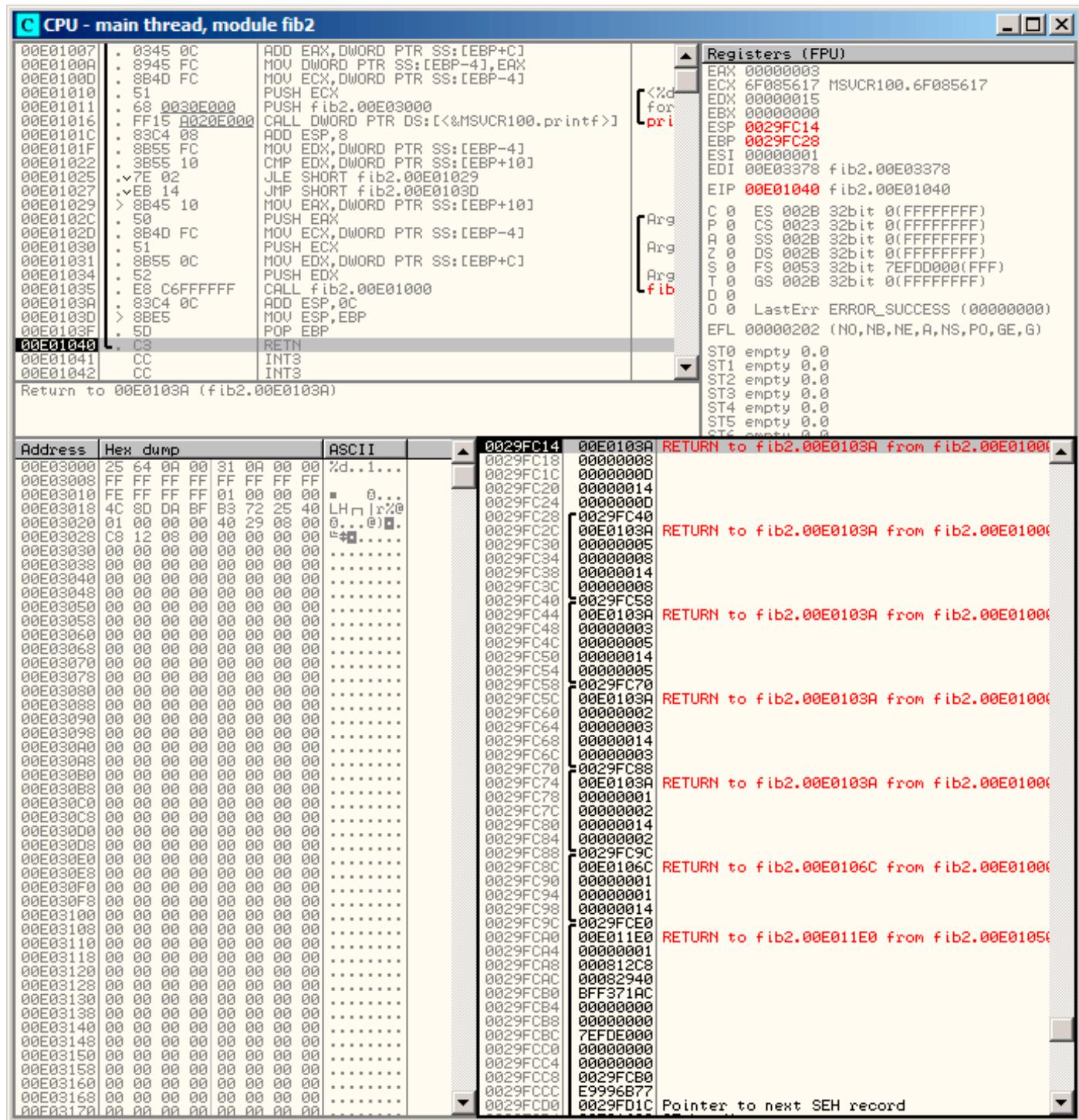


Figure 3.2: OllyDbg: last call of `f()`

Now the *next* variable is present in each frame.

### 3.4. FIBONACCI NUMBERS

Let's investigate the stack more closely. The author has again added his comments:

```
0029FC14 00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC18 00000008 1st argument: a
0029FC1C 0000000D 2nd argument: b
0029FC20 00000014 3rd argument: limit
0029FC24 0000000D "next" variable
0029FC28 /0029FC40 saved EBP register
0029FC2C |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC30 |00000005 1st argument: a
0029FC34 |00000008 2nd argument: b
0029FC38 |00000014 3rd argument: limit
0029FC3C |00000008 "next" variable
0029FC40 |0029FC58 saved EBP register
0029FC44 |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC48 |00000003 1st argument: a
0029FC4C |00000005 2nd argument: b
0029FC50 |00000014 3rd argument: limit
0029FC54 |00000005 "next" variable
0029FC58 |0029FC70 saved EBP register
0029FC5C |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC60 |00000002 1st argument: a
0029FC64 |00000003 2nd argument: b
0029FC68 |00000014 3rd argument: limit
0029FC6C |00000003 "next" variable
0029FC70 |0029FC88 saved EBP register
0029FC74 |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC78 |00000001 1st argument: a \
0029FC7C |00000002 2nd argument: b | prepared in f1() for next f1() call
0029FC80 |00000014 3rd argument: limit /
0029FC84 |00000002 "next" variable
0029FC88 |0029FC9C saved EBP register
0029FC8C |00E0106C RETURN to fib2.00E0106C from fib2.00E01000
0029FC90 |00000001 1st argument: a \
0029FC94 |00000001 2nd argument: b | prepared in main() for f1()
0029FC98 |00000014 3rd argument: limit /
0029FC9C |0029FCE0 saved EBP register
0029FCA0 |00E011E0 RETURN to fib2.00E011E0 from fib2.00E01050
0029FCA4 |00000001 main() 1st argument: argc \
0029FCA8 |000812C8 main() 2nd argument: argv | prepared in CRT for main()
0029FCAC |00082940 main() 3rd argument: envp /
```

Here we see it: the *next* value is calculated in each function incarnation, then passed as argument *b* to the next incarnation.

#### 3.4.3 Summary

Recursive functions are aesthetically nice, but technically may degrade performance because of their heavy stack usage. Everyone who writes performance critical code probably should avoid recursion.

For example, the author of this book once wrote a function to seek a particular node in a binary tree. As a recursive function it looked quite stylish but since additional time was spent at each function call for the prologue/epilogue, it was working a couple of times slower than an iterative (recursion-free) implementation.

By the way, that is the reason that some functional [PL<sup>6</sup>](#) compilers (where recursion is used heavily) use [tail call](#). We talk about tail call when a function has only one single call to itself located at the end of it, like:

Listing 3.6: Scheme, example is copypasted from Wikipedia

```
;; factorial : number -> number
;; to calculate the product of all positive
;; integers less than or equal to n.
(define (factorial n)
  (if (= n 1)
    1
```

<sup>6</sup>LISP, Python, Lua, etc.

### 3.5. CRC32 CALCULATION EXAMPLE

```
(* n (factorial (- n 1))))
```

Tail call is important because compiler can rework this code easily into iterative one, to get rid of recursion.

## 3.5 CRC32 calculation example

This is a very popular table-based CRC32 hash calculation technique<sup>7</sup>.

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef unsigned long ub4;
typedef unsigned char ub1;

static const ub4 crctab[256] = {
    0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419,
    0x706af48f, 0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4,
    0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07,
    0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
    0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856,
    0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
    0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4,
    0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
    0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3,
    0x45df5c75, 0xdc60dcf, 0xabd13d59, 0x26d930ac, 0x51de003a,
    0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599,
    0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
    0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190,
    0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f,
    0x9fbfe4a5, 0xe8b8d433, 0x7807c9a2, 0x0f00f934, 0x9609a88e,
    0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
    0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed,
    0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
    0x8bbeb8ea, 0xfc9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3,
    0xfb44c65, 0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
    0x4adfa51, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a,
    0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5,
    0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa, 0xbe0b1010,
    0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
    0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17,
    0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6,
    0x03b6e20c, 0x74b1d29a, 0xeadd54739, 0x9dd277af, 0x04db2615,
    0x73dc1683, 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
    0xe40ecf0b, 0x9309ff9d, 0xa00ae27, 0x7d079eb1, 0xf00f9344,
    0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
    0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a,
    0x67dd4acc, 0xf9b9df6f, 0x8ebeeef9, 0x17b7be43, 0x60b08ed5,
    0xd6d6a3e8, 0x1ald1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1,
    0xa6bc5767, 0x3fb506dd, 0x48b2364b, 0xd80d2bda, 0xaf0a1b4c,
    0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef,
    0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
    0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe,
    0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31,
    0x2cd99e8b, 0x5bdea1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c,
    0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
    0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b,
    0xe5d5be0d, 0x7cdcef7, 0xbdbdf21, 0x86d3d2d4, 0xf1d4e242,
    0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1,
    0x18b74777, 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
    0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45, 0xa00ae278,
    0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7,
    0x4969474d, 0x3e6e77db, 0xaea16a4a, 0xd9d65adc, 0x40df0b66,
    0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
```

<sup>7</sup>The source code has been taken from here: <http://go.yurichev.com/17327>

### 3.5. CRC32 CALCULATION EXAMPLE

```

0xbdbdf21c, 0xabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605,
0xcdd70693, 0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8,
0xd681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b,
0x2d02ef8d
};

/* how to derive the values in crctab[] from polynomial 0xedb88320 */
void build_table()
{
    ub4 i, j;
    for (i=0; i<256; ++i) {
        j = i;
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        printf("0x%.8lx, ", j);
        if (i%6 == 5) printf("\n");
    }
}

/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
    ub4 i;
    const ub1 *k = key;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
    return hash;
}

/* To use, try "gcc -O0 crc.c -o crc; crc < crc.c" */
int main()
{
    char s[1000];
    while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
    return 0;
}

```

We are interested in the `crc()` function only. By the way, pay attention to the two loop initializers in the `for()` statement: `hash=len, i=0`. The C/C++ standard allows this, of course. The emitted code will contain two operations in the loop initialization part instead of one.

Let's compile it in MSVC with optimization (`/Ox`). For the sake of brevity, only the `crc()` function is listed here, with my comments.

```

_key$ = 8          ; size = 4
_len$ = 12         ; size = 4
_hash$ = 16         ; size = 4
_crc    PROC
    mov     edx, DWORD PTR _len$[esp-4]
    xor     ecx, ecx ; i will be stored in ECX
    mov     eax, edx
    test    edx, edx
    jbe    SHORT $LN1@crc
    push   ebx
    push   esi
    mov     esi, DWORD PTR _key$[esp+4] ; ESI = key
    push   edi
$L1@crc:
    ; work with bytes using only 32-bit registers. byte from address key+i we store into EDI
    movzx  edi, BYTE PTR [ecx+esi]

```

### 3.5. CRC32 CALCULATION EXAMPLE

```

mov    ebx, eax ; EBX = (hash = len)
and    ebx, 255 ; EBX = hash & 0xff

; XOR EDI, EBX (EDI=EDI^EBX) - this operation uses all 32 bits of each register
; but other bits (8-31) are cleared all time, so its OK'
; these are cleared because, as for EDI, it was done by MOVZX instruction above
; high bits of EBX was cleared by AND EBX, 255 instruction above (255 = 0xff)

xor    edi, ebx

; EAX=EAX>>8; bits 24-31 taken from nowhere will be cleared
shr    eax, 8

; EAX=EAX^crctab[EDI*4] - choose EDI-th element from crctab[] table
xor    eax, DWORD PTR _crctab[edi*4]
inc    ecx          ; i++
cmp    ecx, edx      ; i<len ?
jb     SHORT $LL3@crc ; yes
pop    edi
pop    esi
pop    ebx
$LN1@crc:
ret    0
_crc   ENDP

```

Let's try the same in GCC 4.4.1 with `-O3` option:

```

public crc
proc near

key        = dword ptr  8
hash       = dword ptr  0Ch

push    ebp
xor    edx, edx
mov    ebp, esp
push    esi
mov    esi, [ebp+key]
push    ebx
mov    ebx, [ebp+hash]
test   ebx, ebx
mov    eax, ebx
jz     short loc_80484D3
nop          ; padding
lea     esi, [esi+0]    ; padding; works as NOP (ESI does not change here)

loc_80484B8:
mov    ecx, eax      ; save previous state of hash to ECX
xor    al, [esi+edx]  ; AL=*(key+i)
add    edx, 1         ; i++
shr    ecx, 8         ; ECX=hash>>8
movzx  eax, al        ; EAX=*(key+i)
mov    eax, dword ptr ds:crctab[eax*4] ; EAX=crctab[EAX]
xor    eax, ecx        ; hash=EAX^ECX
cmp    ebx, edx
ja    short loc_80484B8

loc_80484D3:
pop    ebx
pop    esi
pop    ebp
ret
crc
\
```

GCC has aligned the loop start on a 8-byte boundary by adding `NOP` and `lea esi, [esi+0]` (that is an *idle operation* too). Read more about it in npad section ([.1.7 on page 1021](#)).

## 3.6 Network address calculation example

As we know, a TCP/IP address (IPv4) consists of four numbers in the 0...255 range, i.e., four bytes.

Four bytes can be fit in a 32-bit variable easily, so an IPv4 host address, network mask or network address can all be 32-bit integers.

From the user's point of view, the network mask is defined as four numbers and is formatted like 255.255.255.0 or so, but network engineers (sysadmins) use a more compact notation (CIDR<sup>8</sup>), like "/8", "/16", etc.

This notation just defines the number of bits the mask has, starting at the MSB.

Mask	Hosts	Usable	Netmask	Hex mask	
/30	4	2	255.255.255.252	0xffffffffc	
/29	8	6	255.255.255.248	0xfffffff8	
/28	16	14	255.255.255.240	0xfffffff0	
/27	32	30	255.255.255.224	0xffffffe0	
/26	64	62	255.255.255.192	0xffffffc0	
/24	256	254	255.255.255.0	0xfffffff00	class C network
/23	512	510	255.255.254.0	0xfffffe00	
/22	1024	1022	255.255.252.0	0xfffffc00	
/21	2048	2046	255.255.248.0	0xfffff800	
/20	4096	4094	255.255.240.0	0xfffff000	
/19	8192	8190	255.255.224.0	0xffffe000	
/18	16384	16382	255.255.192.0	0xffffc000	
/17	32768	32766	255.255.128.0	0xffff8000	
/16	65536	65534	255.255.0.0	0xffff0000	class B network
/8	16777216	16777214	255.0.0.0	0xff000000	class A network

Here is a small example, which calculates the network address by applying the network mask to the host address.

```
#include <stdio.h>
#include <stdint.h>

uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4)
{
    return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
};

void print_as_IP (uint32_t a)
{
    printf ("%d.%d.%d.%d\n",
            (a>>24)&0xFF,
            (a>>16)&0xFF,
            (a>>8)&0xFF,
            (a)&0xFF);
};

// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
    return input=input|(1<<bit);
};

uint32_t form_netmask (uint8_t netmask_bits)
{
    uint32_t netmask=0;
    uint8_t i;

    for (i=0; i<netmask_bits; i++)
        netmask=set_bit(netmask, 31-i);

    return netmask;
};

void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4, uint8_t ↴
    ↴ netmask_bits)
{
```

<sup>8</sup>Classless Inter-Domain Routing

### 3.6. NETWORK ADDRESS CALCULATION EXAMPLE

```
uint32_t netmask=form_netmask(netmask_bits);
uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
uint32_t netw_addr;

printf ("netmask=");
print_as_IP (netmask);

netw_addr=ip&netmask;

printf ("network address=");
print_as_IP (netw_addr);
};

int main()
{
    calc_network_address (10, 1, 2, 4, 24);      // 10.1.2.4, /24
    calc_network_address (10, 1, 2, 4, 8);       // 10.1.2.4, /8
    calc_network_address (10, 1, 2, 4, 25);      // 10.1.2.4, /25
    calc_network_address (10, 1, 2, 64, 26);     // 10.1.2.4, /26
};
```

#### 3.6.1 calc\_network\_address()

`calc_network_address()` function is simplest one: it just ANDs the host address with the network mask, resulting in the network address.

Listing 3.7: Optimizing MSVC 2012 /Ob0

```
1 _ip1$ = 8          ; size = 1
2 _ip2$ = 12         ; size = 1
3 _ip3$ = 16         ; size = 1
4 _ip4$ = 20         ; size = 1
5 _netmask_bits$ = 24 ; size = 1
6 _calc_network_address PROC
7     push    edi
8     push    DWORD PTR _netmask_bits$[esp]
9     call    _form_netmask
10    push   OFFSET $SG3045 ; 'netmask='
11    mov     edi, eax
12    call   DWORD PTR __imp__printf
13    push   edi
14    call   _print_as_IP
15    push   OFFSET $SG3046 ; 'network address='
16    call   DWORD PTR __imp__printf
17    push   DWORD PTR _ip4$[esp+16]
18    push   DWORD PTR _ip3$[esp+20]
19    push   DWORD PTR _ip2$[esp+24]
20    push   DWORD PTR _ip1$[esp+28]
21    call   _form_IP
22    and    eax, edi      ; network address = host address & netmask
23    push   eax
24    call   _print_as_IP
25    add    esp, 36
26    pop    edi
27    ret    0
28 _calc_network_address ENDP
```

At line 22 we see the most important `AND`—here the network address is calculated.

#### 3.6.2 form\_IP()

The `form_IP()` function just puts all 4 bytes into a 32-bit value.

Here is how it is usually done:

- Allocate a variable for the return value. Set it to 0.

### 3.6. NETWORK ADDRESS CALCULATION EXAMPLE

- Take the fourth (lowest) byte, apply OR operation to this byte and return the value. The return value contain the 4th byte now.
- Take the third byte, shift it left by 8 bits. You'll get a value like `0x0000bb00` where `bb` is your third byte. Apply the OR operation to the resulting value and it. The return value has contained `0x000000aa` so far, so ORing the values will produce a value like `0x0000bbaa`.
- Take the second byte, shift it left by 16 bits. You'll get a value like `0x00cc0000`, where `cc` is your second byte. Apply the OR operation to the resulting value and return it. The return value has contained `0x0000bbaa` so far, so ORing the values will produce a value like `0x00ccbbaa`.
- Take the first byte, shift it left by 24 bits. You'll get a value like `0xdd000000`, where `dd` is your first byte. Apply the OR operation to the resulting value and return it. The return value contain `0x00ccbbaa` so far, so ORing the values will produce a value like `0xddccbbaa`.

And this is how it's done by non-optimizing MSVC 2012:

Listing 3.8: Non-optimizing MSVC 2012

```
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".  
_ip1$ = 8          ; size = 1  
_ip2$ = 12         ; size = 1  
_ip3$ = 16         ; size = 1  
_ip4$ = 20         ; size = 1  
_form_IP PROC  
    push    ebp  
    mov     ebp, esp  
    movzx   eax, BYTE PTR _ip1$[ebp]  
    ; EAX=000000dd  
    shl     eax, 24  
    ; EAX=dd000000  
    movzx   ecx, BYTE PTR _ip2$[ebp]  
    ; ECX=000000cc  
    shl     ecx, 16  
    ; ECX=00cc0000  
    or      eax, ecx  
    ; EAX=ddcc0000  
    movzx   edx, BYTE PTR _ip3$[ebp]  
    ; EDX=000000bb  
    shl     edx, 8  
    ; EDX=0000bb00  
    or      eax, edx  
    ; EAX=ddccb000  
    movzx   ecx, BYTE PTR _ip4$[ebp]  
    ; ECX=000000aa  
    or      eax, ecx  
    ; EAX=ddccbbaa  
    pop    ebp  
    ret    0  
_form_IP ENDP
```

Well, the order is different, but, of course, the order of the operations doesn't matter.

Optimizing MSVC 2012 does essentially the same, but in a different way:

Listing 3.9: Optimizing MSVC 2012 /Ob0

```
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".  
_ip1$ = 8          ; size = 1  
_ip2$ = 12         ; size = 1  
_ip3$ = 16         ; size = 1  
_ip4$ = 20         ; size = 1  
_form_IP PROC  
    movzx   eax, BYTE PTR _ip1$[esp-4]  
    ; EAX=000000dd  
    movzx   ecx, BYTE PTR _ip2$[esp-4]  
    ; ECX=000000cc  
    shl     eax, 8  
    ; EAX=0000dd00  
    or      eax, ecx  
    ; EAX=0000ddcc
```

### 3.6. NETWORK ADDRESS CALCULATION EXAMPLE

```
movzx  ecx, BYTE PTR _ip3$[esp-4]
; ECX=000000bb
shl    eax, 8
; EAX=00ddcc00
or     eax, ecx
; EAX=00ddccbb
movzx  ecx, BYTE PTR _ip4$[esp-4]
; ECX=000000aa
shl    eax, 8
; EAX=ddccbb00
or     eax, ecx
; EAX=ddccbbaa
ret    0
_form_IP ENDP
```

We could say that each byte is written to the lowest 8 bits of the return value, and then the return value is shifted left by one byte at each step.

Repeat 4 times for each input byte.

That's it! Unfortunately, there are probably no other ways to do it.

There are no popular CPUs or ISAs which has instruction for composing a value from bits or bytes.

It's all usually done by bit shifting and ORing.

#### 3.6.3 print\_as\_IP()

`print_as_IP()` does the inverse: splitting a 32-bit value into 4 bytes.

Slicing works somewhat simpler: just shift input value by 24, 16, 8 or 0 bits, take the bits from zeroth to seventh (lowest byte), and that's it:

Listing 3.10: Non-optimizing MSVC 2012

```
_a$ = 8           ; size = 4
_print_as_IP PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
; EAX=ddccbbaa
    and    eax, 255
; EAX=000000aa
    push    eax
    mov     ecx, DWORD PTR _a$[ebp]
; ECX=ddccbbaa
    shr    ecx, 8
; ECX=00ddccbb
    and    ecx, 255
; ECX=000000bb
    push    ecx
    mov     edx, DWORD PTR _a$[ebp]
; EDX=ddccbbaa
    shr    edx, 16
; EDX=0000ddcc
    and    edx, 255
; EDX=000000cc
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
; EAX=ddccbbaa
    shr    eax, 24
; EAX=000000dd
    and    eax, 255 ; probably redundant instruction
; EAX=000000dd
    push    eax
    push    OFFSET $SG2973 ; '%d.%d.%d.%d'
    call    DWORD PTR __imp__printf
    add    esp, 20
    pop    ebp
    ret    0
```

### 3.6. NETWORK ADDRESS CALCULATION EXAMPLE

```
_print_as_IP ENDP
```

Optimizing MSVC 2012 does almost the same, but without unnecessary reloading of the input value:

Listing 3.11: Optimizing MSVC 2012 /Ob0

```
_a$ = 8          ; size = 4
_print_as_IP PROC
    mov     ecx, DWORD PTR _a$[esp-4]
    ; ECX=ddccbbaa
    movzx  eax, cl
    ; EAX=000000aa
    push   eax
    mov     eax, ecx
    ; EAX=ddccbbaa
    shr    eax, 8
    ; EAX=00ddccbb
    and    eax, 255
    ; EAX=000000bb
    push   eax
    mov     eax, ecx
    ; EAX=ddccbbaa
    shr    eax, 16
    ; EAX=0000ddcc
    and    eax, 255
    ; EAX=000000cc
    push   eax
    ; ECX=ddccbbaa
    shr    ecx, 24
    ; ECX=000000dd
    push   ecx
    push   OFFSET $SG3020 ; '%d.%d.%d.%d'
    call   DWORD PTR __imp__printf
    add    esp, 20
    ret    0
_print_as_IP ENDP
```

#### 3.6.4 form\_netmask() and set\_bit()

`form_netmask()` makes a network mask value from [CIDR](#) notation. Of course, it would be much effective to use here some kind of a precalculated table, but we consider it in this way intentionally, to demonstrate bit shifts.

We will also write a separate function `set_bit()`. It's a not very good idea to create a function for such primitive operation, but it would be easy to understand how it all works.

Listing 3.12: Optimizing MSVC 2012 /Ob0

```
_input$ = 8          ; size = 4
_bit$ = 12         ; size = 4
_set_bit PROC
    mov     ecx, DWORD PTR _bit$[esp-4]
    mov     eax, 1
    shl    eax, cl
    or     eax, DWORD PTR _input$[esp-4]
    ret    0
_set_bit ENDP

_netmask_bits$ = 8      ; size = 1
_form_netmask PROC
    push   ebx
    push   esi
    movzx  esi, BYTE PTR _netmask_bits$[esp+4]
    xor    ecx, ecx
    xor    bl, bl
    test   esi, esi
    jle   SHORT $LN9@form_netma
    xor    edx, edx
```

### 3.7. LOOPS: SEVERAL ITERATORS

```
$LL3@form_netma:  
    mov     eax, 31  
    sub     eax, edx  
    push    eax  
    push    ecx  
    call    _set_bit  
    inc     bl  
    movzx   edx, bl  
    add     esp, 8  
    mov     ecx, eax  
    cmp     edx, esi  
    jl      SHORT $LL3@form_netma  
$LN9@form_netma:  
    pop     esi  
    mov     eax, ecx  
    pop     ebx  
    ret     0  
_form_netmask ENDP
```

`set_bit()` is primitive: it just shift left 1 to number of bits we need and then ORs it with the “input” value. `form_netmask()` has a loop: it will set as many bits (starting from the [MSB](#)) as passed in the `netmask_bits` argument

#### 3.6.5 Summary

That's it! We run it and getting:

```
netmask=255.255.255.0  
network address=10.1.2.0  
netmask=255.0.0.0  
network address=10.0.0.0  
netmask=255.255.255.128  
network address=10.1.2.0  
netmask=255.255.255.192  
network address=10.1.2.64
```

## 3.7 Loops: several iterators

In most cases loops have only one iterator, but there could be several in the resulting code.

Here is a very simple example:

```
#include <stdio.h>  
  
void f(int *a1, int *a2, size_t cnt)  
{  
    size_t i;  
  
    // copy from one array to another in some weird scheme  
    for (i=0; i<cnt; i++)  
        a1[i*3]=a2[i*7];  
}
```

There are two multiplications at each iteration and they are costly operations. Can we optimize it somehow?

Yes, if we notice that both array indices are jumping on values that we can easily calculate without multiplication.

### 3.7.1 Three iterators

### 3.7. LOOPS: SEVERAL ITERATORS

Listing 3.13: Optimizing MSVC 2013 x64

```
f      PROC
; RCX=a1
; RDX=a2
; R8=cnt
    test    r8, r8      ; cnt==0? exit then
    je     SHORT $LN1@f
    npad   11
$LL3@f:
    mov     eax, DWORD PTR [rdx]
    lea     rcx, QWORD PTR [rcx+12]
    lea     rdx, QWORD PTR [rdx+28]
    mov     DWORD PTR [rcx-12], eax
    dec     r8
    jne     SHORT $LL3@f
$LN1@f:
    ret     0
f      ENDP
```

Now there are 3 iterators: the *cnt* variable and two indices, which are increased by 12 and 28 at each iteration. We can rewrite this code in C/C++:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    size_t idx1=0; idx2=0;

    // copy from one array to another in some weird scheme
    for (i=0; i<cnt; i++)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
    };
}
```

So, at the cost of updating 3 iterators at each iteration instead of one, we can remove two multiplication operations.

#### 3.7.2 Two iterators

GCC 4.9 does even more, leaving only 2 iterators:

Listing 3.14: Optimizing GCC 4.9 x64

```
; RDI=a1
; RSI=a2
; RDX=cnt
f:
    test    rdx, rdx ; cnt==0? exit then
    je     .L1
; calculate last element address in "a2" and leave it in RDX
    lea     rax, [0+rdx*4]
; RAX=RDX*4=cnt*4
    sal     rdx, 5
; RDX=RDX<<5=cnt*32
    sub     rdx, rax
; RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
    add     rdx, rsi
; RDX=RDX+RSI=a2+cnt*28
.L3:
    mov     eax, DWORD PTR [rsi]
    add     rsi, 28
    add     rdi, 12
    mov     DWORD PTR [rdi-12], eax
    cmp     rsi, rdx
```

### 3.7. LOOPS: SEVERAL ITERATORS

```
jne      .L3
.L1:
    rep ret
```

There is no *counter* variable any more: GCC concluded that it is not needed.

The last element of the *a2* array is calculated before the loop begins (which is easy:  $cnt * 7$ ) and that's how the loop is to be stopped: just iterate until the second index reaches this precalculated value.

You can read more about multiplication using shifts/additions/subtractions here: [1.18.1 on page 213](#).

This code can be rewritten into C/C++ like that:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t idx1=0; idx2=0;
    size_t last_idx2=cnt*7;

    // copy from one array to another in some weird scheme
    for (;;)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
        if (idx2==last_idx2)
            break;
    };
}
```

GCC (Linaro) 4.9 for ARM64 does the same, but it precalculates the last index of *a1* instead of *a2*, which, of course has the same effect:

Listing 3.15: Optimizing GCC (Linaro) 4.9 ARM64

```
; X0=a1
; X1=a2
; X2=cnt
f:
    cbz      x2, .L1          ; cnt==0? exit then
; calculate last element of "a1" array
    add      x2, x2, x2, lsl 1
; X2=X2+X2<<1=X2+X2*2=X2*3
    mov      x3, 0
    lsl      x2, x2, 2
; X2=X2<<2=X2*4=X2*3*4=X2*12
.L3:
    ldr      w4, [x1],28       ; load at X1, add 28 to X1 (post-increment)
    str      w4, [x0,x3]       ; store at X0+X3=a1+X3
    add      x3, x3, 12        ; shift X3
    cmp      x3, x2            ; end?
    bne      .L3
.L1:
    ret
```

GCC 4.4.5 for MIPS does the same:

Listing 3.16: Optimizing GCC 4.4.5 for MIPS (IDA)

```
; $a0=a1
; $a1=a2
; $a2=cnt
f:
; jump to loop check code:
    beqz    $a2, locret_24
; initialize counter (i) at 0:
    move    $v0, $zero ; branch delay slot, NOP

locret_24:
; load 32-bit word at $a1
```

### 3.7. LOOPS: SEVERAL ITERATORS

```

        lw      $a3, 0($a1)
; increment counter (i):
        addiu   $v0, 1
; check for finish (compare "i" in $v0 and "cnt" in $a2):
        sltu    $v1, $v0, $a2
; store 32-bit word at $a0:
        sw      $a3, 0($a0)
; add 0x1C (28) to $a1 at each iteration:
        addiu   $a1, 0x1C
; jump to loop body if i<cnt:
        bnez    $v1, loc_8
; add 0xC (12) to $a0 at each iteration:
        addiu   $a0, 0xC ; branch delay slot

locret_24:
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP

```

### 3.7.3 Intel C++ 2011 case

Compiler optimizations can also be weird, but nevertheless, still correct. Here is what the Intel C++ compiler 2011 does:

Listing 3.17: Optimizing Intel C++ 2011 x64

```

f      PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8  = cnt
.B1.1::
    test    r8, r8
    jbe     exit

.B1.2::
    cmp     r8, 6
    jbe     just_copy

.B1.3::
    cmp     rcx, rdx
    jbe     .B1.5

.B1.4::
    mov     r10, r8
    mov     r9, rcx
    shl     r10, 5
    lea     rax, QWORD PTR [r8*4]
    sub     r9, rdx
    sub     r10, rax
    cmp     r9, r10
    jge     just_copy2

.B1.5::
    cmp     rdx, rcx
    jbe     just_copy

.B1.6::
    mov     r9, rdx
    lea     rax, QWORD PTR [r8*8]
    sub     r9, rcx
    lea     r10, QWORD PTR [rax+r8*4]
    cmp     r9, r10
    jl      just_copy

just_copy2::
; R8 = cnt
; RDX = a2
; RCX = a1
        xor     r10d, r10d

```

### 3.8. DUFF'S DEVICE

```
xor      r9d, r9d
xor      eax, eax

.B1.8::
    mov      r11d, DWORD PTR [rax+rdx]
    inc
    mov      DWORD PTR [r9+rcx], r11d
    add      r9, 12
    add      rax, 28
    cmp      r10, r8
    jb       .B1.8
    jmp      exit

just_copy::
; R8 = cnt
; RDX = a2
; RCX = a1
    xor      r10d, r10d
    xor      r9d, r9d
    xor      eax, eax

.B1.11::
    mov      r11d, DWORD PTR [rax+rdx]
    inc
    mov      r10
    mov      DWORD PTR [r9+rcx], r11d
    add      r9, 12
    add      rax, 28
    cmp      r10, r8
    jb       .B1.11

exit::
    ret
```

First, there are some decisions taken, then one of the routines is executed.

Looks like it is a check if arrays intersect.

This is very well known way of optimizing memory block copy routines. But copy routines are the same!

This is has to be an error of the Intel C++ optimizer, which still produces workable code, though.

We intentionally considering such example code in this book so the reader would understand that compiler output is weird at times, but still correct, because when the compiler was tested, it passed the tests.

## 3.8 Duff's device

Duff's device <sup>9</sup> is an unrolled loop with the possibility to jump to the middle of it. The unrolled loop is implemented using a fallthrough switch() statement. We would use here a slightly simplified version of Tom Duff's original code. Let's say, we have to write a function that clears a region in memory. One can come with a simple loop, clearing byte by byte. It's obviously slow, since all modern computers have much wider memory bus. So the better way is to clear the memory region using 4 or 8 byte blocks. Since we are going to work with a 64-bit example here, we are going to clear the memory in 8 byte blocks. So far so good. But what about the tail? Memory clearing routine can also be called for regions of size that's not a multiple of 8. So here is the algorithm:

- calculate the number of 8-byte blocks, clear them using 8-byte (64-bit) memory accesses;
- calculate the size of the tail, clear it using 1-byte memory accesses.

The second step can be implemented using a simple loop. But let's implement it as an unrolled loop:

```
#include <stdint.h>
#include <stdio.h>

void bzero(uint8_t* dst, size_t count)
{
    int i;
```

<sup>9</sup>[wikipedia](#)

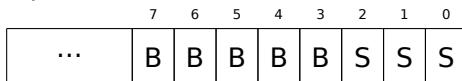
```

if (count&(~7))
    // work out 8-byte blocks
    for (i=0; i<count>>3; i++)
    {
        *(uint64_t*)dst=0;
        dst=dst+8;
    };

    // work out the tail
    switch(count & 7)
    {
    case 7: *dst++ = 0;
    case 6: *dst++ = 0;
    case 5: *dst++ = 0;
    case 4: *dst++ = 0;
    case 3: *dst++ = 0;
    case 2: *dst++ = 0;
    case 1: *dst++ = 0;
    case 0: // do nothing
            break;
    }
}

```

Let's first understand how the calculation is performed. The memory region size comes as a 64-bit value. And this value can be divided in two parts:



( "B" is number of 8-byte blocks and "S" is length of the tail in bytes ).

When we divide the input memory region size by 8, the value is just shifted right by 3 bits. But to calculate the remainder, we can just isolate the lowest 3 bits! So the number of 8-byte blocks is calculated as `count >> 3` and remainder as `count&7`. We also have to find out if we are going to execute the 8-byte procedure at all, so we need to check if the value of `count` is greater than 7. We do this by clearing the 3 lowest bits and comparing the resulting number with zero, because all we need here is to answer the question, is the high part of `count` non-zero. Of course, this works because 8 is  $2^3$  and division by numbers that are  $2^n$  is easy. It's not possible for other numbers. It's actually hard to say if these hacks are worth using, because they lead to hard-to-read code. However, these tricks are very popular and a practicing programmer, even if he/she is not using them, nevertheless has to understand them.

So the first part is simple: get the number of 8-byte blocks and write 64-bit zero values to memory. The second part is an unrolled loop implemented as fallthrough switch() statement.

First, let's express in plain English what we have to do here.

We have to "write as many zero bytes in memory, as `count&7` value tells us". If it's 0, jump to the end, there is no work to do. If it's 1, jump to the place inside switch() statement where only one storage operation is to be executed. If it's 2, jump to another place, where two storage operation are to be executed, etc. 7 as input value leads to the execution of all 7 operations. There is no 8, because a memory region of 8 bytes is to be processed by the first part of our function. So we wrote an unrolled loop. It was definitely faster on older computers than normal loops (and conversely, latest CPUs works better for short loops than for unrolled ones). Maybe this is still meaningful on modern low-cost embedded MCUs<sup>10</sup>.

Let's see what the optimizing MSVC 2012 does:

```

dst$ = 8
count$ = 16
bzero PROC
    test    rdx, -8
    je     SHORT $LN11@bzero
; work out 8-byte blocks
    xor     r10d, r10d
    mov     r9, rdx
    shr     r9, 3
    mov     r8d, r10d
    test    r9, r9
    je     SHORT $LN11@bzero

```

<sup>10</sup>Microcontroller unit

### 3.8. DUFF'S DEVICE

```

npad    5
$LL19@bzero:
    inc    r8d
    mov    QWORD PTR [rcx], r10
    add    rcx, 8
    movsxd  rax, r8d
    cmp    rax, r9
    jb     SHORT $LL19@bzero
$LN11@bzero:
; work out the tail
    and   edx, 7
    dec    rdx
    cmp    rdx, 6
    ja    SHORT $LN9@bzero
    lea    r8, OFFSET FLAT:__ImageBase
    mov    eax, DWORD PTR $LN22@bzero[r8+rdx*4]
    add    rax, r8
    jmp    rax
$LN8@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN7@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN6@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN5@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN4@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN3@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN2@bzero:
    mov    BYTE PTR [rcx], 0
$LN9@bzero:
    fatret 0
    npad    1
$LN22@bzero:
    DD    $LN2@bzero
    DD    $LN3@bzero
    DD    $LN4@bzero
    DD    $LN5@bzero
    DD    $LN6@bzero
    DD    $LN7@bzero
    DD    $LN8@bzero
bzero  ENDP

```

The first part of the function is predictable. The second part is just an unrolled loop and a jump passing control flow to the correct instruction inside it. There is no other code between the `MOV / INC` instruction pairs, so the execution is to fall until the very end, executing as many pairs as needed. By the way, we can observe that the `MOV / INC` pair consumes a fixed number of bytes (3+3). So the pair consumes 6 bytes. Knowing that, we can get rid of the `switch()` jumptable, we can just multiple the input value by 6 and jump to `current_RIP + input_value * 6`.

This can also be faster because we are not in need to fetch a value from the jumptable.

It's possible that 6 probably is not a very good constant for fast multiplication and maybe it's not worth it, but you get the idea<sup>11</sup>.

That is what old-school demomakers did in the past with unrolled loops.

<sup>11</sup>As an exercise, you can try to rework the code to get rid of the jumptable. The instruction pair can be rewritten in a way that it will consume 4 bytes or maybe 8. 1 byte is also possible (using `STOSB` instruction).

### 3.8.1 Should one use unrolled loops?

Unrolled loops can have benefits if there is no fast cache memory between **RAM** and **CPU**, and the **CPU**, in order to get the code of the next instruction, must load it from **RAM** each time. This is a case of modern low-cost **MCU** and old **CPU**s.

Unrolled loops are slower than short loops if there is a fast cache between **RAM** and **CPU** and the body of loop can fit into cache, and **CPU** will load the code from it not touching the **RAM**. Fast loops are the loops which body's size can fit into L1 cache, but even faster loops are those small ones which can fit into micro-operation cache.

## 3.9 Division using multiplication

A very simple function:

```
int f(int a)
{
    return a/9;
};
```

### 3.9.1 x86

...is compiled in a very predictable way:

Listing 3.18: MSVC

```
_a$ = 8          ; size = 4
_f    PROC
    push  ebp
    mov   ebp, esp
    mov   eax, DWORD PTR _a$[ebp]
    cdq           ; sign extend EAX to EDX:EAX
    mov   ecx, 9
    idiv  ecx
    pop   ebp
    ret   0
_f    ENDP
```

**IDIV** divides the 64-bit number stored in the **EDX:EAX** register pair by the value in the **ECX**. As a result, **EAX** will contain the **quotient**, and **EDX** — the remainder. The result is returned from the **f()** function in the **EAX** register, so the value is not moved after the division operation, it is in right place already.

Since **IDIV** uses the value in the **EDX:EAX** register pair, the **CDQ** instruction (before **IDIV**) extends the value in **EAX** to a 64-bit value taking its sign into account, just as **MOVVSX** does.

If we turn optimization on (**/Ox**), we get:

Listing 3.19: Optimizing MSVC

```
_a$ = 8          ; size = 4
_f    PROC
    mov   ecx, DWORD PTR _a$[esp-4]
    mov   eax, 954437177 ; 38e38e39H
    imul  ecx
    sar   edx, 1
    mov   eax, edx
    shr   eax, 31        ; 00000001fH
    add   eax, edx
    ret   0
_f    ENDP
```

This is division by multiplication. Multiplication operations work much faster. And it is possible to use this trick <sup>12</sup> to produce code which is effectively equivalent and faster.

<sup>12</sup>Read more about division by multiplication in [Henry S. Warren, *Hacker's Delight*, (2002)10-3]

### 3.9. DIVISION USING MULTIPLICATION

This is also called “strength reduction” in compiler optimizations.

GCC 4.4.1 generates almost the same code even without additional optimization flags, just like MSVC with optimization turned on:

Listing 3.20: Non-optimizing GCC 4.4.1

```
public f
f proc near

arg_0 = dword ptr 8

    push    ebp
    mov     ebp, esp
    mov     ecx, [ebp+arg_0]
    mov     edx, 954437177 ; 38E38E39h
    mov     eax, ecx
    imul   edx
    sar     edx, 1
    mov     eax, ecx
    sar     eax, 1Fh
    mov     ecx, edx
    sub     ecx, eax
    mov     eax, ecx
    pop     ebp
    retn
f endp
```

## 3.9.2 How it works

From school-level mathematics, we can remember that division by 9 can be replaced by multiplication by  $\frac{1}{9}$ . In fact, sometimes compilers do so for floating-point arithmetics, for example, FDIV instruction in x86 code can be replaced by FMUL. At least MSVC 6.0 will replace division by 9 by multiplication by 0.11111... and sometimes it's hard to be sure, what operation was in original source code.

But when we operate over integer values and integer CPU registers, we can't use fractions. However, we can rework fraction like that:

$$\text{result} = \frac{x}{9} = x \cdot \frac{1}{9} = x \cdot \frac{1 \cdot \text{MagicNumber}}{9 \cdot \text{MagicNumber}}$$

Given the fact that division by  $2^n$  is very fast (using shifts), we now should find that *MagicNumber*, for which the following equation will be true:  $2^n = 9 \cdot \text{MagicNumber}$ .

Division by  $2^{32}$  is somewhat hidden: lower 32-bit of product in EAX is not used (dropped), only higher 32-bit of product (in EDX) is used and then shifted by additional 1 bit.

In other words, the assembly code we have just seen multiplies by  $\frac{954437177}{2^{32+1}}$ , or divides by  $\frac{2^{32+1}}{954437177}$ . To find divisor we just have to divide numerator by denominator. Using Wolfram Alpha, we can get 8.9999999.... as result (which is close to 9).

Read more about it in [Henry S. Warren, *Hacker's Delight*, (2002)10-3].

Couple of words about better understanding. Many people miss “hidden” division by  $2^{32}$  or  $2^{64}$ , when lower 32-bit part (or 64-bit part) of product is not used. Also, there is misconception that modulo inverse is used here. This is close, but not the same thing. Extended Euclidean algorithm is usually used to find *magic coefficient*, but in fact, this algorithm is rather used to solve the equation. You can solve it using any other method. For example, the author used Z3 SMT solver to find *coefficient*<sup>13</sup>. This is overkill, but it was used it for demonstration. Anyway, Extended Euclidean algorithm is probably the most efficient way to solve it. Also, needless to mention, the equation is unsolvable for some divisors, because this is diophantine equation (i.e., equation allowing result to be only integer), since we work on integer CPU registers, after all.

<sup>13</sup>[https://yurichev.com/tmp/SAT\\_SMT\\_DRAFT.pdf](https://yurichev.com/tmp/SAT_SMT_DRAFT.pdf)

### 3.9.3 ARM

The ARM processor, just like in any other “pure” RISC processor lacks an instruction for division. It also lacks a single instruction for multiplication by a 32-bit constant (recall that a 32-bit constant cannot fit into a 32-bit opcode).

By taking advantage of this clever trick (or *hack*), it is possible to do division using only three instructions: addition, subtraction and bit shifts ([1.22 on page 304](#)).

Here is an example that divides a 32-bit number by 10, from [Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)3.3 Division by a Constant]. The output consists of the quotient and the remainder.

```
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10          ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1          ; fix-up quotient
    ADDMI  a2, a2, #10         ; fix-up remainder
    MOV    pc, lr
```

### Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
text:00002C58 39 1E 08 E3 E3 18 43 E3  MOV    R1, 0x38E38E39
text:00002C60 10 F1 50 E7  SMMUL R0, R0, R1
text:00002C64 C0 10 A0 E1  MOV    R1, R0, ASR#1
text:00002C68 A0 0F 81 E0  ADD    R0, R1, R0, LSR#31
text:00002C6C 1E FF 2F E1  BX     LR
```

This code is almost the same as the one generated by the optimizing MSVC and GCC.

Apparently, LLVM uses the same algorithm for generating constants.

The observant reader may ask, how does `MOV` writes a 32-bit value in a register, when this is not possible in ARM mode.

It is impossible indeed, but, as we see, there are 8 bytes per instruction instead of the standard 4, in fact, there are two instructions.

The first instruction loads `0x8E39` into the low 16 bits of register and the second instruction is `MOV`, it loads `0x383E` into the high 16 bits of the register. [IDA](#) is fully aware of such sequences, and for the sake of compactness reduces them to one single “pseudo-instruction”.

The `SMMUL` (*Signed Most Significant Word Multiply*) instruction two multiplies numbers, treating them as signed numbers and leaving the high 32-bit part of result in the `R0` register, dropping the low 32-bit part of the result.

The “`MOV R1, R0, ASR#1`” instruction is an arithmetic shift right by one bit.

“`ADD R0, R1, R0, LSR#31`” is  $R0 = R1 + R0 \gg 31$

There is no separate shifting instruction in ARM mode. Instead, an instructions like (`MOV`, `ADD`, `SUB`, `RSB`)<sup>14</sup> can have a suffix added, that says if the second operand must be shifted, and if yes, by what value and how. `ASR` stands for *Arithmetic Shift Right*, `LSR`—*Logical Shift Right*.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

<sup>14</sup>These instructions are also called “data processing instructions”

### 3.10. STRING TO NUMBER CONVERSION (ATOI())

```
MOV          R1, 0x38E38E39
SMMUL.W    R0, R0, R1
ASRS        R1, R0, #1
ADD.W       R0, R1, R0, LSR#31
BX          LR
```

There are separate instructions for shifting in Thumb mode, and one of them is used here— ASRS (arithmetic shift right).

#### Non-optimizing Xcode 4.6.3 (LLVM) and Keil 6/2013

Non-optimizing LLVM does not generate the code we saw before in this section, but instead inserts a call to the library function `_divsi3`.

What about Keil: it inserts a call to the library function `_aeabi_idivmod` in all cases.

#### 3.9.4 MIPS

For some reason, optimizing GCC 4.4.5 generate just a division instruction:

Listing 3.21: Optimizing GCC 4.4.5 (IDA)

```
f:
    li      $v0, 9
    bnez   $v0, loc_10
    div    $a0, $v0 ; branch delay slot
    break  0x1C00 ; "break 7" in assembly output and objdump

loc_10:
    mflo   $v0
    jr    $ra
    or    $at, $zero ; branch delay slot, NOP
```

Here we see here a new instruction: BREAK. It just raises an exception.

In this case, an exception is raised if the divisor is zero (it's not possible to divide by zero in conventional math).

But GCC probably did not do very well the optimization job and did not see that \$V0 is never zero.

So the check is left here. So if \$V0 is zero somehow, BREAK is to be executed, signaling to the OS about the exception.

Otherwise, MFLO executes, which takes the result of the division from the LO register and copies it in \$V0.

By the way, as we may know, the MUL instruction leaves the high 32 bits of the result in register HI and the low 32 bits in register LO.

DIV leaves the result in the LO register, and remainder in the HI register.

If we alter the statement to “a % 9”, the MFHI instruction is to be used here instead of MFLO.

#### 3.9.5 Exercise

- <http://challenges.re/27>

### 3.10 String to number conversion (atoi())

Let's try to reimplement the standard atoi() C function.

### 3.10.1 Simple example

Here is the simplest possible way to read a number represented in [ASCII](#) encoding.

It's not error-prone: a character other than a digit leads to incorrect result.

```
#include <stdio.h>

int my_atoi (char *s)
{
    int rt=0;

    while (*s)
    {
        rt=rt*10 + (*s-'0');
        s++;
    };

    return rt;
};

int main()
{
    printf ("%d\n", my_atoi ("1234"));
    printf ("%d\n", my_atoi ("1234567890"));
};
```

So what the algorithm does is just reading digits from left to right.

The zero [ASCII](#) character is subtracted from each digit.

The digits from "0" to "9" are consecutive in the [ASCII](#) table, so we do not even need to know the exact value of the "0" character.

All we have to know is that "0" minus "0" is 0, "9" minus "0" is 9 and so on.

Subtracting "0" from each character results in a number from 0 to 9 inclusive.

Any other character leads to an incorrect result, of course!

Each digit has to be added to the final result (in variable "rt"), but the final result is also multiplied by 10 at each digit.

In other words, the result is shifted left by one position in decimal form on each iteration.

The last digit is added, but there is no shift.

### Optimizing MSVC 2013 x64

Listing 3.22: Optimizing MSVC 2013 x64

```
s$ = 8
my_atoi PROC
; load first character
    movzx r8d, BYTE PTR [rcx]
; EAX is allocated for "rt" variable
; its 0 at start
    xor eax, eax
; first character is zero-byte, i.e., string terminator?
; exit then.
    test r8b, r8b
    je SHORT $LN9@my_atoi
$LL2@my_atoi:
    lea edx, DWORD PTR [rax+rax*4]
; EDX=RAX+RAX*4=rt+rt*4=rt*5
    movsx eax, r8b
; EAX=input character
; load next character to R8D
    movzx r8d, BYTE PTR [rcx+1]
; shift pointer in RCX to the next character:
    lea rcx, QWORD PTR [rcx+1]
```

### 3.10. STRING TO NUMBER CONVERSION (ATOI())

```
    lea      eax, DWORD PTR [rax+rdx*2]
; EAX=RAX+RDX*2=input character + rt*5*2=input character + rt*10
; correct digit by subtracting 48 (0x30 or '0')
        add     eax, -48                                ; ffffffffffffffd0H
; was last character zero?
        test    r8b, r8b
; jump to loop begin, if not
        jne     SHORT $LL2@my_atoi
$LN9@my_atoi:
        ret     0
my_atoi ENDP
```

A character can be loaded in two places: the first character and all subsequent characters. This is arranged so for loop regrouping.

There is no instruction for multiplication by 10, two LEA instruction do this instead.

MSVC sometimes uses the ADD instruction with a negative constant instead of SUB. This is the case.

It's very hard to say why this is better than SUB. But MSVC does this often.

### Optimizing GCC 4.9.1 x64

Optimizing GCC 4.9.1 is more concise, but there is one redundant RET instruction at the end. One would be enough.

Listing 3.23: Optimizing GCC 4.9.1 x64

```
my_atoi:
; load input character into EDX
    movsx  edx, BYTE PTR [rdi]
; EAX is allocated for "rt" variable
    xor    eax, eax
; exit, if loaded character is null byte
    test   dl, dl
    je     .L4
.L3:
    lea    eax, [rax+rax*4]
; EAX=RAX*5=rt*5
; shift pointer to the next character:
    add    rdi, 1
    lea    eax, [rdx-48+rax*2]
; EAX=input character - 48 + RAX*2 = input character - '0' + rt*10
; load next character:
    movsx  edx, BYTE PTR [rdi]
; goto loop begin, if loaded character is not null byte
    test   dl, dl
    jne   .L3
    rep    ret
.L4:
    rep    ret
```

### Optimizing Keil 6/2013 (ARM mode)

Listing 3.24: Optimizing Keil 6/2013 (ARM mode)

```
my_atoi PROC
; R1 will contain pointer to character
    MOV    r1,r0
; R0 will contain "rt" variable
    MOV    r0,#0
    B     |L0.28|
|L0.12|
    ADD    r0,r0,r0,LSL #2
; R0=R0+R0<<2=rt*5
    ADD    r0,r2,r0,LSL #1
; R0=input character + rt*5<<1 = input character + rt*10
; correct whole thing by subtracting '0' from rt:
```

### 3.10. STRING TO NUMBER CONVERSION (ATOI())

```
SUB      r0, r0, #0x30
; shift pointer to the next character:
ADD      r1, r1, #1
|L0.28|
; load input character to R2
LDRB    r2, [r1, #0]
; is it null byte? if no, jump to loop body.
CMP     r2, #0
BNE    |L0.12|
; exit if null byte.
; "rt" variable is still in R0 register, ready to be used in caller function
BX     lr
ENDP
```

### Optimizing Keil 6/2013 (Thumb mode)

Listing 3.25: Optimizing Keil 6/2013 (Thumb mode)

```
my_atoi PROC
; R1 will be pointer to the input character
MOVS   r1, r0
; R0 is allocated to "rt" variable
MOVS   r0, #0
B      |L0.16|
|L0.6|
MOVS   r3, #0xa
; R3=10
MULS   r0, r3, r0
; R0=R3*R0=rt*10
; shift pointer to the next character:
ADDS   r1, r1, #1
; correct whole thing by subtracting '0' character from it':
SUBS   r0, r0, #0x30
ADDS   r0, r2, r0
; rt=R2+R0=input character + (rt*10 - '0')
|L0.16|
; load input character to R2
LDRB   r2, [r1, #0]
; is it zero?
CMP    r2, #0
; jump to loop body if it is not
BNE    |L0.6|
; rt variable in R0 now, ready to be used in caller function
BX     lr
ENDP
```

Interestingly, from school mathematics we may recall that the order of addition and subtraction operations doesn't matter.

That's our case: first, the  $rt * 10 - '0'$  expression is computed, then the input character value is added to it. Indeed, the result is the same, but the compiler did some regrouping.

### Optimizing GCC 4.9.1 ARM64

The ARM64 compiler can use the pre-increment instruction suffix:

Listing 3.26: Optimizing GCC 4.9.1 ARM64

```
my_atoi:
; load input character into W1
    ldrb   w1, [x0]
    mov    x2, x0
; X2=address of input string
; is loaded character zero?
; jump to exit if its so
; W1 will contain 0 in this case.
```

### 3.10. STRING TO NUMBER CONVERSION (ATOI())

```
; it will be reloaded into W0 at L4.
    cbz      w1, .L4
; W0 will contain "rt" variable
; initialize it at zero:
    mov      w0, 0
.L3:
; subtract 48 or '0' from input variable and put result into W3:
    sub      w3, w1, #48
; load next character at address X2+1 into W1 with pre-increment:
    ldrb    w1, [x2,1]!
    add     w0, w0, w0, lsl 2
; W0=W0+W0<<2=W0+W0*4=rt*5
    add     w0, w3, w0, lsl 1
; W0=input digit + W0<<1 = input digit + rt*5*2 = input digit + rt*10
; if the character we just loaded is not null byte, jump to the loop begin
    cbnz    w1, .L3
; variable to be returned (rt) is in W0, ready to be used in caller function
    ret
.L4:
    mov      w0, w1
    ret
```

### 3.10.2 A slightly advanced example

My new code snippet is more advanced, now it checks for the “minus” sign at the first character and reports an error if a non-digit has been found in the input string:

```
#include <stdio.h>

int my_atoi (char *s)
{
    int negative=0;
    int rt=0;

    if (*s=='-')
    {
        negative=1;
        s++;
    };

    while (*s)
    {
        if (*s<'0' || *s>'9')
        {
            printf ("Error! Unexpected char: '%c'\n", *s);
            exit(0);
        };
        rt=rt*10 + (*s-'0');
        s++;
    };

    if (negative)
        return -rt;
    return rt;
};

int main()
{
    printf ("%d\n", my_atoi ("1234"));
    printf ("%d\n", my_atoi ("1234567890"));
    printf ("%d\n", my_atoi ("-1234"));
    printf ("%d\n", my_atoi ("-1234567890"));
    printf ("%d\n", my_atoi ("-a1234567890")); // error
};
```

Listing 3.27: Optimizing GCC 4.9.1 x64

```

.LC0:
    .string "Error! Unexpected char: '%c'\n"

my_atoi:
    sub    rsp, 8
    movsx  edx, BYTE PTR [rdi]
; check for minus sign
    cmp    dl, 45 ; '-'
    je     .L22
    xor    esi, esi
    test   dl, dl
    je     .L20
.L10:
; ESI=0 here if there was no minus sign and 1 if it was
    lea    eax, [rdx-48]
; any character other than digit will result in unsigned number greater than 9 after ↴ subtraction
; so if it is not digit, jump to L4, where error will be reported:
    cmp    al, 9
    ja    .L4
    xor    eax, eax
    jmp    .L6
.L7:
    lea    ecx, [rdx-48]
    cmp    cl, 9
    ja    .L4
.L6:
    lea    eax, [rax+rax*4]
    add    rdi, 1
    lea    eax, [rdx-48+rax*2]
    movsx  edx, BYTE PTR [rdi]
    test   dl, dl
    jne    .L7
; if there was no minus sign, skip NEG instruction
; if it was, execute it.
    test   esi, esi
    je     .L18
    neg    eax
.L18:
    add    rsp, 8
    ret
.L22:
    movsx  edx, BYTE PTR [rdi+1]
    lea    rax, [rdi+1]
    test   dl, dl
    je     .L20
    mov    rdi, rax
    mov    esi, 1
    jmp    .L10
.L20:
    xor    eax, eax
    jmp    .L18
.L4:
; report error. character is in EDX
    mov    edi, 1
    mov    esi, OFFSET FLAT:.LC0 ; "Error! Unexpected char: '%c'\n"
    xor    eax, eax
    call   __printf_chk
    xor    edi, edi
    call   exit

```

If the “minus” sign has been encountered at the string start, the `NEG` instruction is to be executed at the end. It just negates the number.

There is one more thing that needs mentioning.

### 3.10. STRING TO NUMBER CONVERSION (ATOI())

How would a common programmer check if the character is not a digit? Just how we have it in the source code:

```
if (*s<'0' || *s>'9')
...
```

There are two comparison operations.

What is interesting is that we can replace both operations by single one: just subtract “0” from character value,

treat result as unsigned value (this is important) and check if it's greater than 9.

For example, let's say that the user input contains the dot character (“.”) which has **ASCII** code 46.  $46 - 48 = -2$  if we treat the result as a signed number.

Indeed, the dot character is located two places earlier than the “0” character in the **ASCII** table. But it is **0xFFFFFFF**E (4294967294) if we treat the result as an unsigned value, and that's definitely bigger than 9!

The compilers do this often, so it's important to recognize these tricks.

Another example of it in this book: [3.16.1 on page 535](#).

Optimizing MSVC 2013 x64 does the same tricks.

### Optimizing Keil 6/2013 (ARM mode)

Listing 3.28: Optimizing Keil 6/2013 (ARM mode)

```
1 my_atoi PROC
2     PUSH    {r4-r6,lr}
3     MOV     r4,r0
4     LDRB   r0,[r0,#0]
5     MOV     r6,#0
6     MOV     r5,r6
7     CMP     r0,#0x2d '-'
8 ; R6 will contain 1 if minus was encountered, 0 if otherwise
9     MOVEQ   r6,#1
10    ADDEQ   r4,r4,#1
11    B      |L0.80|
12 |L0.36|
13    SUB    r0,r1,#0x30
14    CMP    r0,#0xa
15    BCC   |L0.64|
16    ADR    r0,|L0.220|
17    BL     __2printf
18    MOV    r0,#0
19    BL     exit
20 |L0.64|
21    LDRB   r0,[r4],#1
22    ADD    r1,r5,r5,LSL #2
23    ADD    r0,r0,r1,LSL #1
24    SUB    r5,r0,#0x30
25 |L0.80|
26    LDRB   r1,[r4,#0]
27    CMP    r1,#0
28    BNE   |L0.36|
29    CMP    r6,#0
30 ; negate result
31    RSBNE r0,r5,#0
32    MOVEQ r0,r5
33    POP    {r4-r6,pc}
34    ENDP
35
36 |L0.220|
37    DCB    "Error! Unexpected char: '%c'\n",0
```

There is no NEG instruction in 32-bit ARM, so the “Reverse Subtraction” operation (line 31) is used here.

It is triggered if the result of the **CMP** instruction (at line 29) has been “Not Equal” (hence **-NE** suffix).

### 3.11. INLINE FUNCTIONS

So what RSBNE does is to subtract the resulting value from 0.

It works just like the regular subtraction operation, but swaps operands.

Subtracting any number from 0 results in negation:  $0 - x = -x$ .

Thumb mode code is mostly the same.

GCC 4.9 for ARM64 can use the NEG instruction, which is available in ARM64.

### 3.10.3 Exercise

Oh, by the way, security researchers deals often with unpredictable behavior of program while handling of incorrect data.

For example, while fuzzing. As an exercise, you may try to enter non-digit characters and see what happens.

Try to explain, what happened and why.

## 3.11 Inline functions

Inlined code is when the compiler, instead of placing a call instruction to a small or tiny function, just places its body right in-place.

Listing 3.29: A simple example

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
    return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
    int celsius=atol(argv[1]);
    printf ("%d\n", celsius_to_fahrenheit (celsius));
}
```

...is compiled in very predictable way, however, if we turn on GCC optimizations (-O3), we'll see:

Listing 3.30: Optimizing GCC 4.8.1

```
_main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    call    __main
    mov     eax, DWORD PTR [ebp+12]
    mov     eax, DWORD PTR [eax+4]
    mov     DWORD PTR [esp], eax
    call    _atol
    mov     edx, 1717986919
    mov     DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\12\0"
    lea     ecx, [eax+eax*8]
    mov     eax, ecx
    imul   edx
    sar     ecx, 31
    sar     edx
    sub     edx, ecx
    add     edx, 32
    mov     DWORD PTR [esp+4], edx
    call    _printf
    leave
    ret
```

### 3.11. INLINE FUNCTIONS

(Here the division is performed by multiplication( [3.9 on page 496](#)).)

Yes, our small function `celsius_to_fahrenheit()` has just been placed before the `printf()` call.

Why? It can be faster than executing this function's code plus the overhead of calling/returning.

Modern optimizing compilers are choosing small functions for inlining automatically. But it's possible to force compiler additionally to inline some function, if to mark it with the "inline" keyword in its declaration.

#### 3.11.1 Strings and memory functions

Another very common automatic optimization tactic is the inlining of string functions like `strcpy()`, `strcmp()`, `strlen()`, `memset()`, `memcmp()`, `memcpy()`, etc..

Sometimes it's faster than to call a separate function.

These are very frequent patterns and it is highly advisable for reverse engineers to learn to detect automatically.

##### strcmp()

Listing 3.31: strcmp() example

```
bool is_bool (char *s)
{
    if (strcmp (s, "true") == 0)
        return true;
    if (strcmp (s, "false") == 0)
        return false;

    assert(0);
}
```

Listing 3.32: Optimizing GCC 4.8.1

```
.LC0:
    .string "true"
.LC1:
    .string "false"
is_bool:
.LFB0:
    push    edi
    mov     ecx, 5
    push    esi
    mov     edi, OFFSET FLAT:.LC0
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    repz   cmpsb
    je      .L3
    mov     esi, DWORD PTR [esp+32]
    mov     ecx, 6
    mov     edi, OFFSET FLAT:.LC1
    repz   cmpsb
    seta   cl
    setb   dl
    xor    eax, eax
    cmp    cl, dl
    jne   .L8
    add    esp, 20
    pop    esi
    pop    edi
    ret
.L8:
    mov    DWORD PTR [esp], 0
    call   assert
    add    esp, 20
    pop    esi
    pop    edi
```

### 3.11. INLINE FUNCTIONS

```

    ret
.L3:
    add    esp, 20
    mov    eax, 1
    pop    esi
    pop    edi
    ret

```

Listing 3.33: Optimizing MSVC 2010

```

$SG3454 DB      'true', 00H
$SG3456 DB      'false', 00H

_s$ = 8          ; size = 4
?is_bool@@YA_NPAD@Z PROC ; is_bool
    push    esi
    mov     esi, DWORD PTR _s$[esp]
    mov     ecx, OFFSET $SG3454 ; 'true'
    mov     eax, esi
    npad   4 ; align next label
$LL6@is_bool:
    mov     dl, BYTE PTR [eax]
    cmp     dl, BYTE PTR [ecx]
    jne    SHORT $LN7@is_bool
    test   dl, dl
    je     SHORT $LN8@is_bool
    mov     dl, BYTE PTR [eax+1]
    cmp     dl, BYTE PTR [ecx+1]
    jne    SHORT $LN7@is_bool
    add    eax, 2
    add    ecx, 2
    test   dl, dl
    jne    SHORT $LL6@is_bool
$LN8@is_bool:
    xor    eax, eax
    jmp    SHORT $LN9@is_bool
$LN7@is_bool:
    sbb    eax, eax
    sbb    eax, -1
$LN9@is_bool:
    test   eax, eax
    jne    SHORT $LN2@is_bool

    mov    al, 1
    pop    esi

    ret    0
$LN2@is_bool:

    mov    ecx, OFFSET $SG3456 ; 'false'
    mov    eax, esi
$LL10@is_bool:
    mov    dl, BYTE PTR [eax]
    cmp    dl, BYTE PTR [ecx]
    jne    SHORT $LN11@is_bool
    test   dl, dl
    je     SHORT $LN12@is_bool
    mov    dl, BYTE PTR [eax+1]
    cmp    dl, BYTE PTR [ecx+1]
    jne    SHORT $LN11@is_bool
    add    eax, 2
    add    ecx, 2
    test   dl, dl
    jne    SHORT $LL10@is_bool
$LN12@is_bool:
    xor    eax, eax
    jmp    SHORT $LN13@is_bool
$LN11@is_bool:
    sbb    eax, eax
    sbb    eax, -1

```

### 3.11. INLINE FUNCTIONS

```
$LN1@is_bool:
    test    eax, eax
    jne     SHORT $LN1@is_bool

    xor    al, al
    pop    esi

    ret    0
$LN1@is_bool:

    push   11
    push   OFFSET $SG3458
    push   OFFSET $SG3459
    call   DWORD PTR __imp__wassert
    add    esp, 12
    pop    esi

    ret    0
?is_bool@@YA_NPAD@Z ENDP ; is_bool
```

### strlen()

Listing 3.34: strlen() example

```
int strlen_test(char *s1)
{
    return strlen(s1);
};
```

Listing 3.35: Optimizing MSVC 2010

```
_s1$ = 8 ; size = 4
_strlen_test PROC
    mov    eax, DWORD PTR _s1$[esp-4]
    lea    edx, DWORD PTR [eax+1]
$LL3@strlen_tes:
    mov    cl, BYTE PTR [eax]
    inc    eax
    test   cl, cl
    jne    SHORT $LL3@strlen_tes
    sub    eax, edx
    ret    0
_strlen_test ENDP
```

### strcpy()

Listing 3.36: strcpy() example

```
void strcpy_test(char *s1, char *outbuf)
{
    strcpy(outbuf, s1);
};
```

Listing 3.37: Optimizing MSVC 2010

```
_s1$ = 8      ; size = 4
_outbuf$ = 12 ; size = 4
_strcpy_test PROC
    mov    eax, DWORD PTR _s1$[esp-4]
    mov    edx, DWORD PTR _outbuf$[esp-4]
    sub    edx, eax
    npad   6 ; align next label
$LL3@strcpy_tes:
    mov    cl, BYTE PTR [eax]
    mov    BYTE PTR [edx+eax], cl
    inc    eax
```

### 3.11. INLINE FUNCTIONS

```
    test    cl, cl
    jne     SHORT $LL3@strcpy_tes
    ret     0
 strcpy_test ENDP
```

## memset()

### Example#1

Listing 3.38: 32 bytes

```
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 32);
}
```

Many compilers don't generate a call to `memset()` for short blocks, but rather insert a pack of `MOV`s:

Listing 3.39: Optimizing GCC 4.9.1 x64

```
f:
    mov    QWORD PTR [rdi], 0
    mov    QWORD PTR [rdi+8], 0
    mov    QWORD PTR [rdi+16], 0
    mov    QWORD PTR [rdi+24], 0
    ret
```

By the way, that remind us of unrolled loops: [1.16.1 on page 193](#).

### Example#2

Listing 3.40: 67 bytes

```
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 67);
}
```

When the block size is not a multiple of 4 or 8, the compilers can behave differently.

For instance, MSVC 2012 continues to insert `MOV`s:

Listing 3.41: Optimizing MSVC 2012 x64

```
out$ = 8
f      PROC
    xor    eax, eax
    mov    QWORD PTR [rcx], rax
    mov    QWORD PTR [rcx+8], rax
    mov    QWORD PTR [rcx+16], rax
    mov    QWORD PTR [rcx+24], rax
    mov    QWORD PTR [rcx+32], rax
    mov    QWORD PTR [rcx+40], rax
    mov    QWORD PTR [rcx+48], rax
    mov    QWORD PTR [rcx+56], rax
    mov    WORD PTR [rcx+64], ax
    mov    BYTE PTR [rcx+66], al
    ret    0
f      ENDP
```

...while GCC uses `REP STOSQ`, concluding that this would be shorter than a pack of `MOV`s:

### 3.11. INLINE FUNCTIONS

Listing 3.42: Optimizing GCC 4.9.1 x64

```
f:  
    mov    QWORD PTR [rdi], 0  
    mov    QWORD PTR [rdi+59], 0  
    mov    rcx, rdi  
    lea    rdi, [rdi+8]  
    xor    eax, eax  
    and    rdi, -8  
    sub    rcx, rdi  
    add    ecx, 67  
    shr    ecx, 3  
    rep    stosq  
    ret
```

## memcpy()

### Short blocks

The routine to copy short blocks is often implemented as a sequence of `MOV` instructions.

Listing 3.43: memcpy() example

```
void memcpy_7(char *inbuf, char *outbuf)  
{  
    memcpy(outbuf+10, inbuf, 7);  
}
```

Listing 3.44: Optimizing MSVC 2010

```
_inbuf$ = 8      ; size = 4  
_outbuf$ = 12    ; size = 4  
_memcpy_7 PROC  
    mov    ecx, DWORD PTR _inbuf$[esp-4]  
    mov    edx, DWORD PTR [ecx]  
    mov    eax, DWORD PTR _outbuf$[esp-4]  
    mov    DWORD PTR [eax+10], edx  
    mov    dx, WORD PTR [ecx+4]  
    mov    WORD PTR [eax+14], dx  
    mov    cl, BYTE PTR [ecx+6]  
    mov    BYTE PTR [eax+16], cl  
    ret    0  
_memcpy_7 ENDP
```

Listing 3.45: Optimizing GCC 4.8.1

```
memcpy_7:  
    push   ebx  
    mov    eax, DWORD PTR [esp+8]  
    mov    ecx, DWORD PTR [esp+12]  
    mov    ebx, DWORD PTR [eax]  
    lea    edx, [ecx+10]  
    mov    DWORD PTR [ecx+10], ebx  
    movzx  ecx, WORD PTR [eax+4]  
    mov    WORD PTR [edx+4], cx  
    movzx  eax, BYTE PTR [eax+6]  
    mov    BYTE PTR [edx+6], al  
    pop    ebx  
    ret
```

That's usually done as follows: 4-byte blocks are copied first, then a 16-bit word (if needed), then the last byte (if needed).

Structures are also copied using `MOV`: [1.24.4 on page 362](#).

### 3.11. INLINE FUNCTIONS

#### Long blocks

The compilers behave differently in this case.

Listing 3.46: memcpy() example

```
void memcpy_128(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 128);
}

void memcpy_123(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 123);
}
```

For copying 128 bytes, MSVC uses a single `MOVSD` instruction (because 128 divides evenly by 4):

Listing 3.47: Optimizing MSVC 2010

```
_inbuf$ = 8          ; size = 4
_outbuf$ = 12        ; size = 4
_memcpy_128 PROC
    push    esi
    mov     esi, DWORD PTR _inbuf$[esp]
    push    edi
    mov     edi, DWORD PTR _outbuf$[esp+4]
    add    edi, 10
    mov    ecx, 32
    rep    movsd
    pop    edi
    pop    esi
    ret    0
_memcpy_128 ENDP
```

When copying 123 bytes, 30 32-byte words are copied first using `MOVSD` (that's 120 bytes), then 2 bytes are copied using `MOVSW`, then one more byte using `MOVSB`.

Listing 3.48: Optimizing MSVC 2010

```
_inbuf$ = 8          ; size = 4
_outbuf$ = 12        ; size = 4
_memcpy_123 PROC
    push    esi
    mov     esi, DWORD PTR _inbuf$[esp]
    push    edi
    mov     edi, DWORD PTR _outbuf$[esp+4]
    add    edi, 10
    mov    ecx, 30
    rep    movsd
    movsw
    movsb
    pop    edi
    pop    esi
    ret    0
_memcpy_123 ENDP
```

GCC uses one big universal functions, that works for any block size:

Listing 3.49: Optimizing GCC 4.8.1

```
memcpy_123:
.LFB3:
    push    edi
    mov     eax, 123
    push    esi
    mov     edx, DWORD PTR [esp+16]
    mov     esi, DWORD PTR [esp+12]
    lea    edi, [edx+10]
    test   edi, 1
```

### 3.11. INLINE FUNCTIONS

```
jne      .L24
test    edi, 2
jne      .L25
.L7:
    mov     ecx, eax
    xor     edx, edx
    shr     ecx, 2
    test    al, 2
    rep    movsd
    je     .L8
    movzx  edx, WORD PTR [esi]
    mov    WORD PTR [edi], dx
    mov     edx, 2
.L8:
    test    al, 1
    je     .L5
    movzx  eax, BYTE PTR [esi+edx]
    mov    BYTE PTR [edi+edx], al
.L5:
    pop    esi
    pop    edi
    ret
.L24:
    movzx  eax, BYTE PTR [esi]
    lea    edi, [edx+11]
    add    esi, 1
    test   edi, 2
    mov    BYTE PTR [edx+10], al
    mov    eax, 122
    je     .L7
.L25:
    movzx  edx, WORD PTR [esi]
    add    edi, 2
    add    esi, 2
    sub    eax, 2
    mov    WORD PTR [edi-2], dx
    jmp    .L7
.LFE3:
```

Universal memory copy functions usually work as follows: calculate how many 32-bit words can be copied, then copy them using `MOVSD`, then copy the remaining bytes.

More advanced and complex copy functions use `SIMD` instructions and also take the memory alignment in consideration.

As an example of SIMD `strlen()` function: [1.29.2 on page 417](#).

### memcmp()

Listing 3.50: memcmp() example

```
int memcmp_1235(char *buf1, char *buf2)
{
    return memcmp(buf1, buf2, 1235);
};
```

For any block size, MSVC 2013 inserts the same universal function:

Listing 3.51: Optimizing MSVC 2010

```
_buf1$ = 8      ; size = 4
_buf2$ = 12      ; size = 4
_memcmp_1235 PROC
    mov     ecx, DWORD PTR _buf1$[esp-4]
    mov     edx, DWORD PTR _buf2$[esp-4]
    push    esi
    mov     esi, 1231
    npad   2
$LL5@memcmp_123:
```

### 3.11. INLINE FUNCTIONS

```
mov    eax, DWORD PTR [ecx]
cmp    eax, DWORD PTR [edx]
jne    SHORT $LN4@memcmp_123
add    ecx, 4
add    edx, 4
sub    esi, 4
jae    SHORT $LL5@memcmp_123
$LN4@memcmp_123:
    mov    al, BYTE PTR [ecx]
    cmp    al, BYTE PTR [edx]
    jne    SHORT $LN6@memcmp_123
    mov    al, BYTE PTR [ecx+1]
    cmp    al, BYTE PTR [edx+1]
    jne    SHORT $LN6@memcmp_123
    mov    al, BYTE PTR [ecx+2]
    cmp    al, BYTE PTR [edx+2]
    jne    SHORT $LN6@memcmp_123
    cmp    esi, -1
    je     SHORT $LN3@memcmp_123
    mov    al, BYTE PTR [ecx+3]
    cmp    al, BYTE PTR [edx+3]
    jne    SHORT $LN6@memcmp_123
$LN3@memcmp_123:
    xor    eax, eax
    pop    esi
    ret    0
$LN6@memcmp_123:
    sbb    eax, eax
    or     eax, 1
    pop    esi
    ret    0
 memcmp_1235 ENDP
```

### strcat()

This is inlined strcat() as it has been generated by MSVC 6.0. There are 3 parts visible: 1) getting source string length (first `scasb`); 2) getting destination string length (second `scasb`); 3) copying source string into the end of destination string (`movsd / movsb` pair).

Listing 3.52: strcat()

```
lea    edi, [src]
or    ecx, 0FFFFFFFh
repne scasb
not   ecx
sub    edi, ecx
mov    esi, edi
mov    edi, [dst]
mov    edx, ecx
or    ecx, 0FFFFFFFh
repne scasb
mov    ecx, edx
dec    edi
shr    ecx, 2
rep    movsd
mov    ecx, edx
and    ecx, 3
rep    movsb
```

### IDA script

There is also a small [IDA](#) script for searching and folding such very frequently seen pieces of inline code: [GitHub](#).

## 3.12 C99 restrict

Here is a reason why Fortran programs, in some cases, work faster than C/C++ ones.

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
}
```

That's very simple example with one specific thing in it: the pointer to the `update_me` array could be a pointer to the `sum` array, `product` array, or even the `sum_product` array—nothing forbids that, right?

The compiler is fully aware of this, so it generates code with four stages in the loop body:

- calculate next `sum[i]`
- calculate next `product[i]`
- calculate next `update_me[i]`
- calculate next `sum_product[i]` —on this stage, we need to load from memory the already calculated `sum[i]` and `product[i]`

Is it possible to optimize the last stage? Since we have already calculated `sum[i]` and `product[i]` it is not necessary to load them again from memory.

Yes, but compiler is not sure that nothing has been overwritten at the 3rd stage! This is called “pointer aliasing”, a situation when the compiler cannot be sure that a memory to which a pointer is pointing hasn't been changed.

`restrict` in the C99 standard [ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.3/1] is a promise, given by programmer to the compiler that the function arguments marked by this keyword always points to different memory locations and never intersects.

To be more precise and describe this formally, `restrict` shows that only this pointer is to be used to access an object, and no other pointer will be used for it.

It can be even said the object will be accessed only via one single pointer, if it is marked as `restrict`.

Let's add this keyword to each pointer argument:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* ↴
         ↴ restrict sum_product,
         int* restrict update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
}
```

Let's see results:

Listing 3.53: GCC x64: f1()

```
f1:
    push    r15 r14 r13 r12 rbp rdi rsi rbx
    mov     r13, QWORD PTR 120[rsp]
    mov     rbp, QWORD PTR 104[rsp]
    mov     r12, QWORD PTR 112[rsp]
    test    r13, r13
```

### 3.12. C99 RESTRICT

```

je      .L1
add    r13, 1
xor    ebx, ebx
mov    edi, 1
xor    r11d, r11d
jmp    .L4
.L6:
    mov    r11, rdi
    mov    rdi, rax
.L4:
    lea    rax, 0[0+r11*4]
    lea    r10, [rcx+rax]
    lea    r14, [rdx+rax]
    lea    rsi, [r8+rax]
    add    rax, r9
    mov    r15d, DWORD PTR [r10]
    add    r15d, DWORD PTR [r14]
    mov    DWORD PTR [rsi], r15d      ; store to sum[]
    mov    r10d, DWORD PTR [r10]
    imul   r10d, DWORD PTR [r14]
    mov    DWORD PTR [rax], r10d      ; store to product[]
    mov    DWORD PTR [r12+r11*4], ebx  ; store to update_me[]
    add    ebx, 123
    mov    r10d, DWORD PTR [rsi]      ; reload sum[i]
    add    r10d, DWORD PTR [rax]      ; reload product[i]
    lea    rax, 1[rdi]
    cmp    rax, r13
    mov    DWORD PTR 0[rbp+r11*4], r10d ; store to sum_product[]
    jne    .L6
.L1:
    pop    rbx rsi rdi rbp r12 r13 r14 r15
    ret

```

Listing 3.54: GCC x64: f2()

```

f2:
    push   r13 r12 rbp rdi rsi rbx
    mov    r13, QWORD PTR 104[rsp]
    mov    rbp, QWORD PTR 88[rsp]
    mov    r12, QWORD PTR 96[rsp]
    test   r13, r13
    je     .L7
    add    r13, 1
    xor    r10d, r10d
    mov    edi, 1
    xor    eax, eax
    jmp    .L10
.L11:
    mov    rax, rdi
    mov    rdi, r11
.L10:
    mov    esi, DWORD PTR [rcx+rax*4]
    mov    r11d, DWORD PTR [rdx+rax*4]
    mov    DWORD PTR [r12+rax*4], r10d ; store to update_me[]
    add    r10d, 123
    lea    ebx, [rsi+r11]
    imul   r11d, esi
    mov    DWORD PTR [r8+rax*4], ebx    ; store to sum[]
    mov    DWORD PTR [r9+rax*4], r11d    ; store to product[]
    add    r11d, ebx
    mov    DWORD PTR 0[rbp+rax*4], r11d ; store to sum_product[]
    lea    r11, 1[rdi]
    cmp    r11, r13
    jne    .L11
.L7:
    pop    rbx rsi rdi rbp r12 r13
    ret

```

The difference between the compiled `f1()` and `f2()` functions is as follows: in `f1()`, `sum[i]` and

### 3.13. BRANCHLESS ABS() FUNCTION

`product[i]` are reloaded in the middle of the loop, and in `f2()` there is no such thing, the already calculated values are used, since we “promised” the compiler that no one and nothing will change the values in `sum[i]` and `product[i]` during the execution of the loop’s body, so it is “sure” that there is no need to load the value from memory again.

Obviously, the second example works faster.

But what if the pointers in the function’s arguments intersect somehow?

This is on the programmer’s conscience, and the results will be incorrect.

Let’s go back to Fortran.

Compilers of this programming language treats all pointers as such, so when it was not possible to set `restrict` in C, Fortran could generate faster code in these cases.

How practical is it?

In the cases when the function works with several big blocks in memory.

There are a lot of such in linear algebra, for instance.

Supercomputers/[HPC<sup>15</sup>](#) are very busy with linear algebra, so probably that is why, traditionally, Fortran is still used there [Eugene Loh, *The Ideal HPC Programming Language*, (2010)].

But when the number of iterations is not very big, certainly, the speed boost may not to be significant.

## 3.13 Branchless `abs()` function

Let’s revisit an example we considered earlier [1.14.2 on page 142](#) and ask ourselves, is it possible to make a branchless version of the function in x86 code?

```
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
};
```

And the answer is yes.

### 3.13.1 Optimizing GCC 4.9.1 x64

We could see it if we compile it using optimizing GCC 4.9:

Listing 3.55: Optimizing GCC 4.9 x64

```
my_abs:
    mov    edx, edi
    mov    eax, edi
    sar    edx, 31
; EDX is 0xFFFFFFFF here if sign of input value is minus
; EDX is 0 if sign of input value is plus (including 0)
; the following two instructions have effect only if EDX is 0xFFFFFFFF
; or idle if EDX is 0
    xor    eax, edx
    sub    eax, edx
    ret
```

This is how it works:

Arithmetically shift the input value right by 31.

Arithmetical shift implies sign extension, so if the **MSB** is 1, all 32 bits are to be filled with 1, or with 0 if otherwise.

<sup>15</sup>High-Performance Computing

### 3.14. VARIADIC FUNCTIONS

In other words, the `SAR REG, 31` instruction makes `0xFFFFFFFF` if the sign has been negative or 0 if positive.

After the execution of `SAR`, we have this value in `EDX`.

Then, if the value is `0xFFFFFFFF` (i.e., the sign is negative), the input value is inverted (because `XOR REG, 0xFFFFFFFF` is effectively an inverse all bits operation).

Then, again, if the value is `0xFFFFFFFF` (i.e., the sign is negative), 1 is added to the final result (because subtracting  $-1$  from some value resulting in incrementing it).

Inversion of all bits and incrementing is exactly how two's complement value is negated: [2.2 on page 454](#).

We may observe that the last two instruction do something if the sign of the input value is negative.

Otherwise (if the sign is positive) they do nothing at all, leaving the input value untouched.

The algorithm is explained in [Henry S. Warren, *Hacker's Delight*, (2002)2-4].

It's hard to say, how GCC did it, deduced it by itself or found a suitable pattern among known ones?

## 3.13.2 Optimizing GCC 4.9 ARM64

GCC 4.9 for ARM64 generates mostly the same, just decides to use the full 64-bit registers.

There are less instructions, because the input value can be shifted using a suffixed instruction ("asr") instead of using a separate instruction.

Listing 3.56: Optimizing GCC 4.9 ARM64

```
my_abs:  
; sign-extend input 32-bit value to X0 64-bit register:  
    sxtw    x0, w0  
    eor     x1, x0, x0, asr 63  
; X1=X0^(X0>>63) (shift is arithmetical)  
    sub     x0, x1, x0, asr 63  
; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63) (all shifts are arithmetical)  
    ret
```

## 3.14 Variadic functions

Functions like `printf()` and `scanf()` can have a variable number of arguments. How are these arguments accessed?

### 3.14.1 Computing arithmetic mean

Let's imagine that we want to calculate [arithmetic mean](#), and for some weird reason we want to specify all the values as function arguments.

But it's impossible to get the number of arguments in a variadic function in C/C++, so let's denote the value of  $-1$  as a terminator.

#### Using va\_arg macro

There is the standard stdarg.h header file which define macros for dealing with such arguments.

The `printf()` and `scanf()` functions use them as well.

```
#include <stdio.h>  
#include <stdarg.h>  
  
int arith_mean(int v, ...)  
{  
    va_list args;
```

### 3.14. VARIADIC FUNCTIONS

```
int sum=v, count=1, i;
va_start(args, v);

while(1)
{
    i=va_arg(args, int);
    if (i==-1) // terminator
        break;
    sum=sum+i;
    count++;
}

va_end(args);
return sum/count;
};

int main()
{
    printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
};
```

The first argument has to be treated just like a normal argument.

All other arguments are loaded using the `va_arg` macro and then summed.

So what is inside?

### cdecl calling conventions

Listing 3.57: Optimizing MSVC 6.0

```
v$ = 8
_arith_mean PROC NEAR
    mov     eax, DWORD PTR _v$[esp-4] ; load 1st argument into sum
    push    esi
    mov     esi, 1                   ; count=1
    lea     edx, DWORD PTR _v$[esp]   ; address of the 1st argument
$L838:
    mov     ecx, DWORD PTR [edx+4]   ; load next argument
    add     edx, 4                  ; shift pointer to the next argument
    cmp     ecx, -1                 ; is it -1?
    je      SHORT $L856            ; exit if so
    add     eax, ecx                ; sum = sum + loaded argument
    inc     esi                    ; count++
    jmp     SHORT $L838

$L856:
; calculate quotient

    cdq
    idiv    esi
    pop     esi
    ret     0
_arith_mean ENDP

$SG851 DB      '%d', 0aH, 00H

_main  PROC NEAR
    push    -1
    push    15
    push    10
    push    7
    push    2
    push    1
    call    _arith_mean
    push    eax
    push    OFFSET FLAT:$SG851 ; '%d'
    call    _printf
    add    esp, 32
    ret    0
```

```
_main    ENDP
```

The arguments, as we may see, are passed to `main()` one-by-one.

The first argument is pushed into the local stack as first.

The terminating value (-1) is pushed last.

The `arith_mean()` function takes the value of the first argument and stores it in the `sum` variable.

Then, it sets the `EDX` register to the address of the second argument, takes the value from it, adds it to `sum`, and does this in an infinite loop, until -1 is found.

When it's found, the sum is divided by the number of all values (excluding -1) and the `quotient` is returned.

So, in other words, the function treats the stack fragment as an array of integer values of infinite length.

Now we can understand why the `cdecl` calling convention forces us to push the first argument into the stack as last.

Because otherwise, it would not be possible to find the first argument, or, for printf-like functions, it would not be possible to find the address of the format-string.

## Register-based calling conventions

The observant reader may ask, what about calling conventions where the first few arguments are passed in registers? Let's see:

Listing 3.58: Optimizing MSVC 2012 x64

```
$SG3013 DB      '%d', 0aH, 00H

v$ = 8
arith_mean PROC
    mov    DWORD PTR [rsp+8], ecx ; 1st argument
    mov    QWORD PTR [rsp+16], rdx ; 2nd argument
    mov    QWORD PTR [rsp+24], r8  ; 3rd argument
    mov    eax, ecx              ; sum = 1st argument
    lea    rcx, QWORD PTR v$[rsp+8] ; pointer to the 2nd argument
    mov    QWORD PTR [rsp+32], r9  ; 4th argument
    mov    edx, DWORD PTR [rcx]   ; load 2nd argument
    mov    r8d, 1                 ; count=1
    cmp    edx, -1                ; 2nd argument is -1?
    je     SHORT $LN8@arith_mean ; exit if so
$LL3@arith_mean:
    add    eax, edx              ; sum = sum + loaded argument
    mov    edx, DWORD PTR [rcx+8] ; load next argument
    lea    rcx, QWORD PTR [rcx+8] ; shift pointer to point to the argument after next
    inc    r8d                   ; count++
    cmp    edx, -1                ; is loaded argument -1?
    jne    SHORT $LL3@arith_mean ; go to loop begin if its not
$LN8@arith_mean:
; calculate quotient
    cdq
    idiv   r8d
    ret    0
arith_mean ENDP

main  PROC
    sub    rsp, 56
    mov    edx, 2
    mov    DWORD PTR [rsp+40], -1
    mov    DWORD PTR [rsp+32], 15
    lea    r9d, QWORD PTR [rdx+8]
    lea    r8d, QWORD PTR [rdx+5]
    lea    ecx, QWORD PTR [rdx-1]
    call   arith_mean
    lea    rcx, OFFSET FLAT:$SG3013
    mov    edx, eax
    call   printf
```

### 3.14. VARIADIC FUNCTIONS

```

xor    eax, eax
add    rsp, 56
ret    0
main  ENDP

```

We see that the first 4 arguments are passed in the registers and two more—in the stack.

The `arith_mean()` function first places these 4 arguments into the *Shadow Space* and then treats the *Shadow Space* and stack behind it as a single continuous array!

What about GCC? Things are slightly clumsier here, because now the function is divided in two parts: the first part saves the registers into the “red zone”, processes that space, and the second part of the function processes the stack:

Listing 3.59: Optimizing GCC 4.9.1 x64

```

arith_mean:
    lea    rax, [rsp+8]
    ; save 6 input registers in
    ; red zone in the local stack
    mov    QWORD PTR [rsp-40], rsi
    mov    QWORD PTR [rsp-32], rdx
    mov    QWORD PTR [rsp-16], r8
    mov    QWORD PTR [rsp-24], rcx
    mov    esi, 8
    mov    QWORD PTR [rsp-64], rax
    lea    rax, [rsp-48]
    mov    QWORD PTR [rsp-8], r9
    mov    DWORD PTR [rsp-72], 8
    lea    rdx, [rsp+8]
    mov    r8d, 1
    mov    QWORD PTR [rsp-56], rax
    jmp    .L5

.L7:
    ; work out saved arguments
    lea    rax, [rsp-48]
    mov    ecx, esi
    add    esi, 8
    add    rcx, rax
    mov    ecx, DWORD PTR [rcx]
    cmp    ecx, -1
    je     .L4

.L8:
    add    edi, ecx
    add    r8d, 1

.L5:
    ; decide, which part we will work out now.
    ; is current argument number less or equal 6?
    cmp    esi, 47
    jbe    .L7          ; no, process saved arguments then
    ; work out arguments from stack
    mov    rcx, rdx
    add    rdx, 8
    mov    ecx, DWORD PTR [rcx]
    cmp    ecx, -1
    jne    .L8

.L4:
    mov    eax, edi
    cdq
    idiv   r8d
    ret

.LC1:
    .string "%d\n"
main:
    sub    rsp, 8
    mov    edx, 7
    mov    esi, 2
    mov    edi, 1
    mov    r9d, -1
    mov    r8d, 15

```

### 3.14. VARIADIC FUNCTIONS

```
mov    ecx, 10
xor    eax, eax
call   arith_mean
mov    esi, OFFSET FLAT:.LC1
mov    edx, eax
mov    edi, 1
xor    eax, eax
add    rsp, 8
jmp    __printf_chk
```

By the way, a similar usage of the *Shadow Space* is also considered here: [6.1.8 on page 740](#).

#### Using pointer to the first function argument

The example can be rewritten without `va_arg` macro:

```
#include <stdio.h>

int arith_mean(int v, ...)
{
    int *i=&v;
    int sum=*i, count=1;
    i++;

    while(1)
    {
        if ((*i)==-1) // terminator
            break;
        sum=sum+(*i);
        count++;
        i++;
    }

    return sum/count;
};

int main()
{
    printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
    // test: https://www.wolframalpha.com/input/?i=mean(1,2,7,10,15)
};
```

In other words, if an argument set is array of words (32-bit or 64-bit), we just enumerate array elements starting at first one.

#### 3.14.2 `vprintf()` function case

Many programmers define their own logging functions which take a `printf`-like format string + a variable number of arguments.

Another popular example is the `die()` function, which prints some message and exits.

We need some way to pack input arguments of unknown number and pass them to the `printf()` function. But how?

That's why there are functions with "v" in name.

One of them is `vprintf()`: it takes a format-string and a pointer to a variable of type `va_list`:

```
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);
```

### 3.14. VARIADIC FUNCTIONS

```
vprintf (fmt, va);
exit(0);
};
```

By closer examination, we can see that `va_list` is a pointer to an array. Let's compile:

Listing 3.60: Optimizing MSVC 2010

```
_fmt$ = 8
_die PROC
    ; load 1st argument (format-string)
    mov    ecx, DWORD PTR _fmt$[esp-4]
    ; get pointer to the 2nd argument
    lea    eax, DWORD PTR _fmt$[esp]
    push   eax           ; pass a pointer
    push   ecx
    call   _vprintf
    add    esp, 8
    push   0
    call   _exit
$LN3@die:
    int    3
_die ENDP
```

We see that all our function does is just taking a pointer to the arguments and passing it to `vprintf()`, and that function is treating it like an infinite array of arguments!

Listing 3.61: Optimizing MSVC 2012 x64

```
fmt$ = 48
die  PROC
    ; save first 4 arguments in Shadow Space
    mov    QWORD PTR [rsp+8], rcx
    mov    QWORD PTR [rsp+16], rdx
    mov    QWORD PTR [rsp+24], r8
    mov    QWORD PTR [rsp+32], r9
    sub    rsp, 40
    lea    rdx, QWORD PTR fmt$[rsp+8] ; pass pointer to the 1st argument
    ; RCX here is still points to the 1st argument (format-string) of die()
    ; so vprintf() will take it right from RCX
    call   vprintf
    xor    ecx, ecx
    call   exit
    int    3
die  ENDP
```

### 3.14.3 Pin case

It's interesting to note how some functions from Pin DBI<sup>16</sup> framework takes number of arguments:

```
INS_InsertPredicatedCall(
    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
    IARG_INST_PTR,
    IARG_MEMORYOP_EA, memOp,
    IARG_END);
```

( `pinatrace.cpp` )

And this is how `INS_InsertPredicatedCall()` function is declared:

```
extern VOID INS_InsertPredicatedCall(INS ins, IPOINT ipoint, AFUNPTR funptr, ...);
```

( `pin_client.PH` )

<sup>16</sup>Dynamic Binary Instrumentation

### 3.14. VARIADIC FUNCTIONS

Hence, constants with names starting with `IARG_` are some kinds of arguments to the function, which are handled inside of `INS_InsertPredicatedCall()`. You can pass as many arguments, as you need. Some commands has additional argument(s), some are not. Full list of arguments: [https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group\\_\\_INST\\_\\_ARGS.html](https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group__INST__ARGS.html). And it has to be a way to detect an end of arguments list, so the list must be terminated with `IARG_END` constant, without which, the function will (try to) handle random noise in the local stack, treating it as additional arguments.

Also, in [Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)] we can find a nice example of C/C++ routines very similar to `pack/unpack`<sup>17</sup> in Python.

#### 3.14.4 Format string exploit

It's a popular mistake, to write `printf(string)` instead of `puts(string)` or `printf("%s", string)`. If the attacker can put his/her own text into `string`, he/she can crash process, or get insight into variables in the the local stack.

Take a look at this:

```
#include <stdio.h>

int main()
{
    char *s1="hello";
    char *s2="world";
    char buf[128];

    // do something mundane here
    strcpy (buf, s1);
    strcpy (buf, " ");
    strcpy (buf, s2);

    printf ("%s");
}
```

Please note, that `printf()` has no additional arguments besides single format string.

Now let's imagine, that was the attacker who put `%s` string into the last `printf()` first arguments. I compile this example using GCC 5.4.0 on x86 Ubuntu, and the resulting executable prints "world" string if it gets executed!

If I turn optimization on, `printf()` outputs some garbage, though—probably, `strcpy()` calls has been optimized and/or local variables as well. Also, result will be different for x64 code, different compiler, OS, etc.

Now, let's say, attacker could pass the following string to `printf()` call: `%x %x %x %x %x`. In may case, output is: "80485c6 b7751b48 1 0 80485c0" (these are just values from local stack). You see, there are 1 and 0 values, and some pointers (first is probably pointer to "world" string). So if the attacker passes `%s %s %s %s %s` string, the process will crash, because `printf()` treats 1 and/or 0 as pointer to string, tries to read characters from there and fails.

Even worse, there could be `sprintf(buf, string)` in code, where `buf` is a buffer in the local stack with size of 1024 bytes or so, attacker can craft `string` in such a way that `buf` will be overflowed, maybe in even such a way, that would lead to code execution.

Many popular and well-known software was (or even still) vulnerable:

QuakeWorld went up, got to around 4000 users, then the master server exploded.

Disrupter and cohorts are working on more robust code now.

If anyone did it on purpose, how about letting us know... (It wasn't all the people that tried `%s` as a name)

<sup>17</sup><https://docs.python.org/3/library/struct.html>

### 3.15. STRINGS TRIMMING

( John Carmack's .plan file, 17-Dec-1996<sup>18</sup> )

Nowadays, almost all decent compilers warns about this.

Another problem is lesser known `%n` `printf()` argument: whenever `printf()` reaches it in format string, it writes number of characters printed so far into corresponding argument: <http://stackoverflow.com/questions/3401156/what-is-the-use-of-the-n-format-specifier-in-c>. Thus, attacker could zap local variables passing many `%n` commands in format string.

## 3.15 Strings trimming

A very common string processing task is to remove some characters at the start and/or at the end.

In this example, we are going to work with a function which removes all newline characters (`CR19/LF20`) from the end of the input string:

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

char* str_trim (char *s)
{
    char c;
    size_t str_len;

    // work as long as \r or \n is at the end of string
    // stop if some other character there or its an empty string'
    // (at start or due to our operation)
    for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
    {
        if (c=='\r' || c=='\n')
            s[str_len-1]=0;
        else
            break;
    };
    return s;
};

int main()
{
    // test

    // strdup() is used to copy text string into data segment,
    // because it will crash on Linux otherwise,
    // where text strings are allocated in constant data segment,
    // and not modifiable.

    printf ("%s\n", str_trim (strdup(""))));
    printf ("%s\n", str_trim (strdup("\n")));
    printf ("%s\n", str_trim (strdup("\r")));
    printf ("%s\n", str_trim (strdup("\n\r")));
    printf ("%s\n", str_trim (strdup("\r\n")));
    printf ("%s\n", str_trim (strdup("test1\r\n")));
    printf ("%s\n", str_trim (strdup("test2\n\r")));
    printf ("%s\n", str_trim (strdup("test3\n\r\n\r")));
    printf ("%s\n", str_trim (strdup("test4\n")));
    printf ("%s\n", str_trim (strdup("test5\r")));
    printf ("%s\n", str_trim (strdup("test6\r\r\r")));
};
```

The input argument is always returned on exit, this is convenient when you want to chain string processing functions, like it has done here in the `main()` function.

<sup>18</sup>[https://github.com/ESWAT/john-carmack-plan-archive/blob/33ae52fdb46aa0d1abfed6fc7598233748541c0/by\\_day/johnc\\_plan\\_19961217.txt](https://github.com/ESWAT/john-carmack-plan-archive/blob/33ae52fdb46aa0d1abfed6fc7598233748541c0/by_day/johnc_plan_19961217.txt)

<sup>19</sup>Carriage return (13 or '\r' in C/C++)

<sup>20</sup>Line feed (10 or '\n' in C/C++)

### 3.15. STRINGS TRIMMING

The second part of `for() ( str_len>0 && (c=s[str_len-1]) )` is the so called “short-circuit” in C/C++ and is very convenient [Dennis Yurichev, *C/C++ programming language notes* 1.3.8].

The C/C++ compilers guarantee an evaluation sequence from left to right.

So if the first clause is false after evaluation, the second one is never to be evaluated.

#### 3.15.1 x64: Optimizing MSVC 2013

Listing 3.62: Optimizing MSVC 2013 x64

```
s$ = 8
str_trim PROC

; RCX is the first function argument and it always holds pointer to the string
    mov     rdx, rcx
; this is strlen() function inlined right here:
; set RAX to 0xFFFFFFFFFFFFFF ( -1 )
    or      rax, -1
$LL14@str_trim:
    inc     rax
    cmp     BYTE PTR [rcx+rax], 0
    jne     SHORT $LL14@str_trim
; is the input string length zero? exit then:
    test    rax, rax
    je      SHORT $LN15@str_trim
; RAX holds string length
    dec     rcx
; RCX = s-1
    mov     r8d, 1
    add     rcx, rax
; RCX = s-1+strlen(s), i.e., this is the address of the last character in the string
    sub     r8, rdx
; R8 = 1-s
$LL6@str_trim:
; load the last character of the string:
; jump, if its code is 13 or 10:
    movzx   eax, BYTE PTR [rcx]
    cmp     al, 13
    je      SHORT $LN2@str_trim
    cmp     al, 10
    jne    SHORT $LN15@str_trim
$LN2@str_trim:
; the last character has a 13 or 10 code
; write zero at this place:
    mov     BYTE PTR [rcx], 0
; decrement address of the last character,
; so it will point to the character before the one which has just been erased:
    dec     rcx
    lea     rax, QWORD PTR [r8+rcx]
; RAX = 1 - s + address of the current last character
; thus we can determine if we reached the first character and we need to stop, if it is so
    test    rax, rax
    jne    SHORT $LL6@str_trim
$LN15@str_trim:
    mov     rax, rdx
    ret     0
str_trim ENDP
```

First, MSVC inlined the `strlen()` function code, because it concluded this is to be faster than the usual `strlen()` work + the cost of calling it and returning from it. This is called inlining: [3.11 on page 506](#).

The first instruction of the inlined `strlen()` is

`OR RAX, 0xFFFFFFFFFFFFFF`.

MSVC often uses `OR` instead of `MOV RAX, 0xFFFFFFFFFFFFFF`, because resulting opcode is shorter.

### 3.15. STRINGS TRIMMING

And of course, it is equivalent: all bits are set, and a number with all bits set is  $-1$  in two's complement arithmetic: [2.2 on page 454](#).

Why would the  $-1$  number be used in `strlen()`, one might ask. Due to optimizations, of course. Here is the code that MSVC generated:

Listing 3.63: Inlined `strlen()` by MSVC 2013 x64

```
; RCX = pointer to the input string
; RAX = current string length
    or      rax, -1
label:
    inc      rax
    cmp      BYTE PTR [rcx+rax], 0
    jne      SHORT label
; RAX = string length
```

Try to write shorter if you want to initialize the counter at 0! OK, let's try:

Listing 3.64: Our version of `strlen()`

```
; RCX = pointer to the input string
; RAX = current string length
    xor      rax, rax
label:
    cmp      byte ptr [rcx+rax], 0
    jz      exit
    inc      rax
    jmp      label
exit:
; RAX = string length
```

We failed. We have to use additional `JMP` instruction!

So what the MSVC 2013 compiler did is to move the `INC` instruction to the place before the actual character loading.

If the first character is 0, that's OK, `RAX` is 0 at this moment, so the resulting string length is 0.

The rest in this function seems easy to understand.

## 3.15.2 x64: Non-optimizing GCC 4.9.1

```
str_trim:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    mov     QWORD PTR [rbp-24], rdi
; for() first part begins here
    mov     rax, QWORD PTR [rbp-24]
    mov     rdi, rax
    call    strlen
    mov     QWORD PTR [rbp-8], rax ; str_len
; for() first part ends here
    jmp    .L2
; for() body begins here
.L5:
    cmp    BYTE PTR [rbp-9], 13 ; c=='\r'?
    je     .L3
    cmp    BYTE PTR [rbp-9], 10 ; c=='\n'?
    jne    .L4
.L3:
    mov    rax, QWORD PTR [rbp-8] ; str_len
    lea    rdx, [rax-1] ; EDX=str_len-1
    mov    rax, QWORD PTR [rbp-24] ; s
    add    rax, rdx ; RAX=s+str_len-1
    mov    BYTE PTR [rax], 0 ; s[str_len-1]=0
; for() body ends here
; for() third part begins here
```

### 3.15. STRINGS TRIMMING

```
sub    QWORD PTR [rbp-8], 1      ; str_len--
; for() third part ends here
.L2:
; for() second part begins here
    cmp    QWORD PTR [rbp-8], 0      ; str_len==0?
    je     .L4                      ; exit then
; check second clause, and load "c"
    mov    rax, QWORD PTR [rbp-8]   ; RAX=str_len
    lea    rdx, [rax-1]            ; RDX=str_len-1
    mov    rax, QWORD PTR [rbp-24]  ; RAX=s
    add    rax, rdx                ; RAX=s+str_len-1
    movzx eax, BYTE PTR [rax]     ; AL=s[str_len-1]
    mov    BYTE PTR [rbp-9], al    ; store loaded char into "c"
    cmp    BYTE PTR [rbp-9], 0     ; is it zero?
    jne    .L5                      ; yes? exit then
; for() second part ends here
.L4:
; return "s"
    mov    rax, QWORD PTR [rbp-24]
    leave
    ret
```

Comments are added by the author of the book.

After the execution of `strlen()`, the control is passed to the L2 label, and there two clauses are checked, one after another.

The second will never be checked, if the first one (`str_len==0`) is false (this is “short-circuit”).

Now let's see this function in short form:

- First for() part (call to `strlen()`)
- goto L2
- L5: for() body. goto exit, if needed
- for() third part (decrement of `str_len`)
- L2: for() second part: check first clause, then second. goto loop body begin or exit.
- L4: // exit
- return s

### 3.15.3 x64: Optimizing GCC 4.9.1

```
str_trim:
    push    rbx
    mov     rbx, rdi
; RBX will always be "s"
    call    strlen
; check for str_len==0 and exit if its so
    test   rax, rax
    je     .L9
    lea    rdx, [rax-1]
; RDX will always contain str_len-1 value, not str_len
; so RDX is more like buffer index variable
    lea    rsi, [rbx+rdx]       ; RSI=s+str_len-1
    movzx  ecx, BYTE PTR [rsi]  ; load character
    test   cl, cl
    je     .L9                  ; exit if its zero
    cmp    cl, 10
    je     .L4
    cmp    cl, 13                  ; exit if its not '\n' and not '\r'
    jne    .L9
.L4:
; this is weird instruction. we need RSI=s-1 here.
; its possible to get it by MOV RSI, EBX / DEC RSI
; but this is two instructions instead of one
    sub    rsi, rax
```

### 3.15. STRINGS TRIMMING

```
; RSI = s+str_len-1-str_len = s-1
; main loop begin
.L12:
    test    rdx, rdx
; store zero at address s-1+str_len-1+1 = s-1+str_len = s+str_len-1
    mov     BYTE PTR [rsi+1+rdx], 0
; check for str_len-1==0. exit if so.
    je     .L9
    sub    rdx, 1           ; equivalent to str_len--
; load next character at address s+str_len-1
    movzx  ecx, BYTE PTR [rbx+rdx]
    test    cl, cl          ; is it zero? exit then
    je     .L9
    cmp    cl, 10            ; is it '\n'?
    je     .L12
    cmp    cl, 13            ; is it '\r'?
    je     .L12
.L9:
; return "s"
    mov    rax, rbx
    pop    rbx
    ret
```

Now this is more complex.

The code before the loop's body start is executed only once, but it has the [CR/LF](#) characters check too! What is this code duplication for?

The common way to implement the main loop is probably this:

- (loop start) check for [CR/LF](#) characters, make decisions
- store zero character

But GCC has decided to reverse these two steps.

Of course, *store zero character* cannot be first step, so another check is needed:

- workout first character. match it to [CR/LF](#), exit if character is not [CR/LF](#)
- (loop begin) store zero character
- check for [CR/LF](#) characters, make decisions

Now the main loop is very short, which is good for latest [CPUs](#).

The code doesn't use the `str_len` variable, but `str_len-1`. So this is more like an index in a buffer.

Apparently, GCC notices that the `str_len-1` statement is used twice.

So it's better to allocate a variable which always holds a value that's smaller than the current string length by one, and decrement it (this is the same effect as decrementing the `str_len` variable).

#### 3.15.4 ARM64: Non-optimizing GCC (Linaro) 4.9

This implementation is straightforward:

Listing 3.65: Non-optimizing GCC (Linaro) 4.9

```
str_trim:
    stp    x29, x30, [sp, -48]!
    add    x29, sp, 0
    str    x0, [x29,24] ; copy input argument into local stack
    ldr    x0, [x29,24] ; s
    bl    strlen
    str    x0, [x29,40] ; str_len variable in local stack
    b     .L2
; main loop begin
.L5:
    ldrb   w0, [x29,39]
; W0=c
    cmp    w0, 13        ; is it '\r'?
    beq    .L3
```

### 3.15. STRINGS TRIMMING

```
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 10      ; is it '\n'?
        bne     .L4       ; goto exit if it is not
.L3:
        ldr     x0, [x29,40]
; X0=str_len
        sub    x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add    x0, x1, x0
; X0=s+str_len-1
        strb   wzr, [x0]   ; write byte at s+str_len-1
; decrement str_len:
        ldr     x0, [x29,40]
; X0=str_len
        sub    x0, x0, #1
; X0=str_len-1
        str    x0, [x29,40]
; save X0 (or str_len-1) to local stack
.L2:
        ldr     x0, [x29,40]
; str_len==0?
        cmp     x0, xzr
; goto exit then
        beq     .L4
        ldr     x0, [x29,40]
; X0=str_len
        sub    x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add    x0, x1, x0
; X0=s+str_len-1
; load byte at address s+str_len-1 to W0
        ldrb   w0, [x0]
        strb   w0, [x29,39] ; store loaded byte to "c"
        ldrb   w0, [x29,39] ; reload it
; is it zero byte?
        cmp     w0, wzr
; goto exit, if its zero or to L5 if its not
        bne     .L5
.L4:
; return s
        ldr     x0, [x29,24]
        ldp     x29, x30, [sp], 48
        ret
```

### 3.15.5 ARM64: Optimizing GCC (Linaro) 4.9

This is a more advanced optimization.

The first character is loaded at the beginning, and compared against 10 (the LF character).

Characters are also loaded in the main loop, for the characters after first one.

This is somewhat similar to the [3.15.3 on page 528](#) example.

Listing 3.66: Optimizing GCC (Linaro) 4.9

```
str_trim:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str    x19, [sp,16]
        mov    x19, x0
; X19 will always hold value of "s"
        bl     strlen
; X0=str_len
```

### 3.15. STRINGS TRIMMING

```

        cbz      x0, .L9          ; goto L9 (exit) if str_len==0
        sub      x1, x0, #1
; X1=X0-1=str_len-1
        add      x3, x19, x1
; X3=X19+X1=s+str_len-1
        ldrb    w2, [x19,x1]   ; load byte at address X19+X1=s+str_len-1
; W2=loaded character
        cbz      w2, .L9          ; is it zero? jump to exit then
        cmp      w2, 10           ; is it '\n'?
        bne      .L15
.L12:
; main loop body. loaded character is always 10 or 13 at this moment!
        sub      x2, x1, x0
; X2=X1-X0=str_len-1-str_len=-1
        add      x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
        strb    w2r, [x2,1]     ; store zero byte at address s+str_len-2+1=s+str_len-1
        cbz      x1, .L9          ; str_len-1==0? goto exit, if so
        sub      x1, x1, #1       ; str_len--
        ldrb    w2, [x19,x1]   ; load next character at address X19+X1=s+str_len-1
        cmp      w2, 10           ; is it '\n'?
        cbz      w2, .L9          ; jump to exit, if its zero
        beq      .L12            ; jump to begin loop, if its '\n'
.L15:
        cmp      w2, 13           ; is it '\r'?
        beq      .L12            ; yes, jump to the loop body begin
.L9:
; return "s"
        mov      x0, x19
        ldr      x19, [sp,16]
        ldp      x29, x30, [sp], 32
        ret

```

### 3.15.6 ARM: Optimizing Keil 6/2013 (ARM mode)

And again, the compiler took advantage of ARM mode's conditional instructions, so the code is much more compact.

Listing 3.67: Optimizing Keil 6/2013 (ARM mode)

```

str_trim PROC
    PUSH    {r4,lr}
; R0=s
    MOV     r4,r0
; R4=s
    BL     strlen      ; strlen() takes "s" value from R0
; R0=str_len
    MOV     r3,#0
; R3 will always hold 0
|L0.16|
    CMP     r0,#0        ; str_len==0?
    ADDNE  r2,r4,r0      ; (if str_len!=0) R2=R4+R0=s+str_len
    LDRBNE r1,[r2,#-1]    ; (if str_len!=0) R1=load byte at address R2-1=s+str_len-1
    CMPNE  r1,#0          ; (if str_len!=0) compare loaded byte against 0
    BEQ    |L0.56|         ; jump to exit if str_len==0 or loaded byte is 0
    CMP     r1,#0xd        ; is loaded byte '\r'?
    CMPNE  r1,#0xa        ; (if loaded byte is not '\r') is loaded byte '\r'?
    SUBEQ  r0,r0,#1       ; (if loaded byte is '\r' or '\n') R0-- or str_len--
    STRBEQ r3,[r2,#-1]    ; (if loaded byte is '\r' or '\n') store R3 (zero) at address R2-
    \ -1=s+str_len-1
    BEQ    |L0.16|         ; jump to loop begin if loaded byte was '\r' or '\n'
|L0.56|
; return "s"
    MOV     r0,r4
    POP    {r4,pc}
ENDP

```

### 3.15.7 ARM: Optimizing Keil 6/2013 (Thumb mode)

There are less conditional instructions in Thumb mode, so the code is simpler.

But there are is really weird thing with the 0x20 and 0x1F offsets (lines 22 and 23). Why did the Keil compiler do so? Honestly, it's hard to say.

It has to be a quirk of Keil's optimization process. Nevertheless, the code works correctly.

Listing 3.68: Optimizing Keil 6/2013 (Thumb mode)

```

1 str_trim PROC
2     PUSH    {r4,lr}
3     MOVS    r4,r0
4 ; R4=s
5     BL      strlen      ; strlen() takes "s" value from R0
6 ; R0=str_len
7     MOVS    r3,#0
8 ; R3 will always hold 0
9     B      |L0.24|
10 |L0.12|
11     CMP    r1,#0xd      ; is loaded byte '\r'?
12     BEQ    |L0.20|
13     CMP    r1,#0xa      ; is loaded byte '\n'?
14     BNE    |L0.38|      ; jump to exit, if no
15 |L0.20|
16     SUBS   r0,r0,#1      ; R0-- or str_len--
17     STRB   r3,[r2,#0x1f] ; store 0 at address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1
18 |L0.24|
19     CMP    r0,#0          ; str_len==0?
20     BEQ    |L0.38|      ; yes? jump to exit
21     ADDS   r2,r4,r0      ; R2=R4+R0=s+str_len
22     SUBS   r2,r2,#0x20    ; R2=R2-0x20=s+str_len-0x20
23     LDRB   r1,[r2,#0x1f] ; load byte at address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 ↵
24     ↳ to R1
25     CMP    r1,#0          ; is loaded byte 0?
26     BNE    |L0.12|      ; jump to loop begin, if its not 0
27 |L0.38|
28 ; return "s"
29     MOVS   r0,r4
30     POP    {r4,pc}
31     ENDP

```

### 3.15.8 MIPS

Listing 3.69: Optimizing GCC 4.4.5 (IDA)

```

str_trim:
; IDA is not aware of local variable names, we gave them manually:
saved_GP      = -0x10
saved_S0      = -8
saved_RA      = -4

        lui    $gp, (_gnu_local_gp >> 16)
        addiu $sp, -0x20
        la    $gp, (_gnu_local_gp & 0xFFFF)
        sw    $ra, 0x20+saved_RA($sp)
        sw    $s0, 0x20+saved_S0($sp)
        sw    $gp, 0x20+saved_GP($sp)
; call strlen(). input string address is still in $a0, strlen() will take it from there:
        lw    $t9, (strlen & 0xFFFF)($gp)
        or    $at, $zero ; load delay slot, NOP
        jalr $t9
; input string address is still in $a0, put it to $s0:
        move $s0, $a0 ; branch delay slot
; result of strlen() (i.e, length of string) is in $v0 now
; jump to exit if $v0==0 (i.e., if length of string is 0):
        beqz $v0, exit
        or    $at, $zero ; branch delay slot, NOP

```

### 3.16. TOUPPER() FUNCTION

```
        addiu    $a1, $v0, -1
; $a1 = $v0-1 = str_len-1
        addu    $a1, $s0, $a1
; $a1 = input string address + $a1 = s+strlen-1
; load byte at address $a1:
        lb      $a0, 0($a1)
        or      $at, $zero ; load delay slot, NOP
; loaded byte is zero? jump to exit if its so:
        beqz   $a0, exit
        or      $at, $zero ; branch delay slot, NOP
        addiu   $v1, $v0, -2
; $v1 = str_len-2
        addu    $v1, $s0, $v1
; $v1 = $s0+$v1 = s+str_len-2
        li      $a2, 0xD
; skip loop body:
        b      loc_6C
        li      $a3, 0xA    ; branch delay slot
loc_5C:
; load next byte from memory to $a0:
        lb      $a0, 0($v1)
        move   $a1, $v1
; $a1=s+str_len-2
; jump to exit if loaded byte is zero:
        beqz   $a0, exit
; decrement str_len:
        addiu   $v1, -1    ; branch delay slot
loc_6C:
; at this moment, $a0=loaded byte, $a2=0xD (CR symbol) and $a3=0xA (LF symbol)
; loaded byte is CR? jump to loc_7C then:
        beq    $a0, $a2, loc_7C
        addiu   $v0, -1    ; branch delay slot
; loaded byte is LF? jump to exit if its not LF:
        bne    $a0, $a3, exit
        or     $at, $zero ; branch delay slot, NOP
loc_7C:
; loaded byte is CR at this moment
; jump to loc_5c (loop body begin) if str_len (in $v0) is not zero:
        bnez   $v0, loc_5C
; simultaneously, store zero at that place in memory:
        sb      $zero, 0($a1) ; branch delay slot
; "exit" label was named by me manually:
exit:
        lw      $ra, 0x20+saved_RA($sp)
        move   $v0, $s0
        lw      $s0, 0x20+saved_S0($sp)
        jr      $ra
        addiu   $sp, 0x20    ; branch delay slot
```

Registers prefixed with S- are also called “saved temporaries”, so \$S0 value is saved in the local stack and restored upon finish.

## 3.16 toupper() function

Another very popular function transforms a symbol from lower case to upper case, if needed:

```
char toupper (char c)
{
    if(c>='a' && c<='z')
        return c-'a'+'A';
    else
        return c;
}
```

The 'a'+'A' expression is left in the source code for better readability, it will be optimized by compiler,

### 3.16. TOUPPER() FUNCTION

of course <sup>21</sup>.

The ASCII code of “a” is 97 (or 0x61), and 65 (or 0x41) for “A”.

The difference (or distance) between them in the ASCII table is 32 (or 0x20).

For better understanding, the reader may take a look at the 7-bit standard ASCII table:

Characters in the coded character set ascii.															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x C-@	C-a	C-b	C-c	C-d	C-e	C-f	C-g	C-h	TAB	C-j	C-k	C-l	RET	C-n	C-o
1x C-p	C-q	C-r	C-s	C-t	C-u	C-v	C-w	C-x	C-y	C-z	ESC	C-\	C-]	C-^	C-_
2x	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3x 0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x @	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	
6x ^	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x p	q	r	s	t	u	v	w	x	y	z	{	}	~		DEL

Figure 3.3: 7-bit ASCII table in Emacs

#### 3.16.1 x64

##### Two comparison operations

Non-optimizing MSVC is straightforward: the code checks if the input symbol is in [97..122] range (or in ['a'..'z'] range) and subtracts 32 if it's true.

There are also some minor compiler artifact:

Listing 3.70: Non-optimizing MSVC 2013 (x64)

```
1 c$ = 8
2 toupper PROC
3     mov    BYTE PTR [rsp+8], cl
4     movsx  eax, BYTE PTR c$[rsp]
5     cmp    eax, 97
6     jl    SHORT $LN2@toupper
7     movsx  eax, BYTE PTR c$[rsp]
8     cmp    eax, 122
9     jg    SHORT $LN2@toupper
10    movsx  eax, BYTE PTR c$[rsp]
11    sub    eax, 32
12    jmp    SHORT $LN3@toupper
13    jmp    SHORT $LN1@toupper ; compiler artefact
14 $LN2@toupper:
15    movzx  eax, BYTE PTR c$[rsp] ; unnecessary casting
16 $LN1@toupper:
17 $LN3@toupper: ; compiler artefact
18    ret    0
19 toupper ENDP
```

It's important to notice that the input byte is loaded into a 64-bit local stack slot at line 3.

All the remaining bits ([8..63]) are untouched, i.e., contain some random noise (you'll see it in debugger).

All instructions operate only on byte-level, so it's fine.

The last MOVZX instruction at line 15 takes the byte from the local stack slot and zero-extends it to a *int* 32-bit data type.

Non-optimizing GCC does mostly the same:

Listing 3.71: Non-optimizing GCC 4.9 (x64)

```
toupper:
    push    rbp
```

<sup>21</sup>However, to be meticulous, there still could be compilers which can't optimize such expressions and will leave them right in the code.

### 3.16. TOUPPER() FUNCTION

```
mov    rbp, rsp
mov    eax, edi
mov    BYTE PTR [rbp-4], al
cmp    BYTE PTR [rbp-4], 96
jle    .L2
cmp    BYTE PTR [rbp-4], 122
jg     .L2
movzx eax, BYTE PTR [rbp-4]
sub    eax, 32
jmp    .L3
.L2:
    movzx eax, BYTE PTR [rbp-4]
.L3:
    pop    rbp
    ret
```

### One comparison operation

Optimizing MSVC does a better job, it generates only one comparison operation:

Listing 3.72: Optimizing MSVC 2013 (x64)

```
toupper PROC
    lea    eax, DWORD PTR [rcx-97]
    cmp    al, 25
    ja    SHORT $LN2@toupper
    movsx eax, cl
    sub    eax, 32
    ret    0
$LN2@toupper:
    movzx eax, cl
    ret    0
toupper ENDP
```

It was explained earlier how to replace the two comparison operations with a single one: [3.10.2 on page 504](#).

We will now rewrite this in C/C++:

```
int tmp=c-97;
if (tmp>25)
    return c;
else
    return c-32;
```

The *tmp* variable must be signed.

This makes two subtraction operations in case of a transformation plus one comparison.

In contrast the original algorithm uses two comparison operations plus one subtracting.

Optimizing GCC is even better, it gets rid of the jumps (which is good: [2.9.1 on page 465](#)) by using the CMOVcc instruction:

Listing 3.73: Optimizing GCC 4.9 (x64)

```
1 toupper:
2     lea    edx, [rdi-97] ; 0x61
3     lea    eax, [rdi-32] ; 0x20
4     cmp    dl, 25
5     cmova eax, edi
6     ret
```

At line 3 the code prepares the subtracted value in advance, as if the conversion will always happen.

At line 5 the subtracted value in EAX is replaced by the untouched input value if a conversion is not needed. And then this value (of course incorrect) is dropped.

Advance subtracting is a price the compiler pays for the absence of conditional jumps.

**3.16.2 ARM**

Optimizing Keil for ARM mode also generates only one comparison:

Listing 3.74: Optimizing Keil 6/2013 (ARM mode)

```
toupper PROC
    SUB    r1,r0,#0x61
    CMP    r1,#0x19
    SUBLS  r0,r0,#0x20
    ANDLS  r0,r0,#0xff
    BX     lr
    ENDP
```

The SUBLS and ANDLS instructions are executed only if the value in R1 is less than 0x19 (or equal). They also do the actual conversion.

Optimizing Keil for Thumb mode generates only one comparison operation as well:

Listing 3.75: Optimizing Keil 6/2013 (Thumb mode)

```
toupper PROC
    MOVS   r1,r0
    SUBS   r1,r1,#0x61
    CMP    r1,#0x19
    BHI    |L0.14|
    SUBS   r0,r0,#0x20
    LSLS   r0,r0,#24
    LSRS   r0,r0,#24
|L0.14|
    BX     lr
    ENDP
```

The last two LSLS and LSRS instructions work like `AND reg, 0xFF`: they are equivalent to the C/C++ expression `(i << 24) >> 24`.

Seems like that Keil for Thumb mode deduced that two 2-byte instructions are shorter than the code that loads the 0xFF constant into a register plus an AND instruction.

**GCC for ARM64**

Listing 3.76: Non-optimizing GCC 4.9 (ARM64)

```
toupper:
    sub    sp, sp, #16
    strb  w0, [sp,15]
    ldrb  w0, [sp,15]
    cmp   w0, 96
    bls   .L2
    ldrb  w0, [sp,15]
    cmp   w0, 122
    bhi   .L2
    ldrb  w0, [sp,15]
    sub   w0, w0, #32
    uxtb  w0, w0
    b     .L3
.L2:
    ldrb  w0, [sp,15]
.L3:
    add   sp, sp, 16
    ret
```

Listing 3.77: Optimizing GCC 4.9 (ARM64)

```
toupper:
    uxtb  w0, w0
    sub   w1, w0, #97
    uxtb  w1, w1
    cmp   w1, 25
```

### 3.17. INCORRECTLY DISASSEMBLED CODE

```
bhi      .L2
sub     w0, w0, #32
uxtb    w0, w0
.L2:
        ret
```

### 3.16.3 Summary

All these compiler optimizations are very popular nowadays and a practicing reverse engineer usually sees such code patterns often.

## 3.17 Incorrectly disassembled code

Practicing reverse engineers often have to deal with incorrectly disassembled code.

### 3.17.1 Disassembling from an incorrect start (x86)

Unlike ARM and MIPS (where any instruction has a length of 2 or 4 bytes), x86 instructions have variable size, so any disassembler that starts in the middle of a x86 instruction may produce incorrect results.

As an example:

```
add    [ebp-31F7Bh], cl
dec    dword ptr [ecx-3277Bh]
dec    dword ptr [ebp-2CF7Bh]
inc    dword ptr [ebx-7A76F33Ch]
fdiv   st(4), st
db 0FFh
dec    dword ptr [ecx-21F7Bh]
dec    dword ptr [ecx-22373h]
dec    dword ptr [ecx-2276Bh]
dec    dword ptr [ecx-22B63h]
dec    dword ptr [ecx-22F4Bh]
dec    dword ptr [ecx-23343h]
jmp    dword ptr [esi-74h]
xchg   eax, ebp
clc
std
db 0FFh
db 0FFh
mov    word ptr [ebp-214h], cs ; <- disassembler finally found right track here
mov    word ptr [ebp-238h], ds
mov    word ptr [ebp-23Ch], es
mov    word ptr [ebp-240h], fs
mov    word ptr [ebp-244h], gs
pushf
pop    dword ptr [ebp-210h]
mov    eax, [ebp+4]
mov    [ebp-218h], eax
lea    eax, [ebp+4]
mov    [ebp-20Ch], eax
mov    dword ptr [ebp-2D0h], 10001h
mov    eax, [eax-4]
mov    [ebp-21Ch], eax
mov    eax, [ebp+0Ch]
mov    [ebp-320h], eax
mov    eax, [ebp+10h]
mov    [ebp-31Ch], eax
mov    eax, [ebp+4]
mov    [ebp-314h], eax
call   ds:IsDebuggerPresent
mov    edi, eax
lea    eax, [ebp-328h]
```

### 3.17. INCORRECTLY DISASSEMBLED CODE

```
push    eax
call    sub_407663
pop     ecx
test    eax, eax
jnz     short loc_402D7B
```

There are incorrectly disassembled instructions at the beginning, but eventually the disassembler gets on the right track.

#### 3.17.2 How does random noise looks disassembled?

Common properties that can be spotted easily are:

- Unusually big instruction dispersion. The most frequent x86 instructions are `PUSH`, `MOV`, `CALL`, but here we see instructions from all instruction groups: `FPU` instructions, `IN / OUT` instructions, rare and system instructions, everything mixed up in one single place.
- Big and random values, offsets and immediates.
- Jumps having incorrect offsets, often jumping in the middle of another instructions.

Listing 3.78: random noise (x86)

```
mov    bl, 0Ch
mov    ecx, 0D38558Dh
mov    eax, ds:2C869A86h
db     67h
mov    dl, 0CCh
insb
movsb
push   eax
xor   [edx-53h], ah
fcom  qword ptr [edi-45A0EF72h]
pop    esp
pop    ss
in    eax, dx
dec   ebx
push   esp
lds   esp, [esi-41h]
retf
rcl   dword ptr [eax], cl
mov    cl, 9Ch
mov    ch, 0DFh
push   cs
insb
mov    esi, 0D9C65E4Dh
imul  ebp, [ecx], 66h
pushf
sal   dword ptr [ebp-64h], cl
sub   eax, 0AC433D64h
out   8Ch, eax
pop    ss
sbb   [eax], ebx
aas
xchg  cl, [ebx+ebx*4+14B31Eh]
jecxz short near ptr loc_58+1
xor   al, 0C6h
inc   edx
db    36h
pusha
stosb
test  [ebx], ebx
sub   al, 0D3h ; 'L'
pop   eax
stosb

loc_58: ; CODE XREF: seg000:0000004A
test  [esi], eax
inc   ebp
```

### 3.17. INCORRECTLY DISASSEMBLED CODE

```
das
db    64h
pop   ecx
das
hlt

pop   edx
out   0B0h, al
lodsb
push  ebx
cdq
out   dx, al
sub   al, 0Ah
sti
outsd
add   dword ptr [edx], 96FCBE4Bh
and   eax, 0E537EE4Fh
inc   esp
stosd
cdq
push  ecx
in    al, 0CBh
mov   ds:0D114C45Ch, al
mov   esi, 659D1985h
```

Listing 3.79: random noise (x86-64)

```
lea   esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:000000000000000B46
rcl   byte ptr [rsi+rax*8+29BB423Ah], 1
lea   ecx, cs:0FFFFFFFB2A6780Fh
mov   al, 96h
mov   ah, 0CEh
push  rsp
lodsd byte ptr [esi]

db  2Fh ; /

pop   rsp
db   64h
retf  0E993h

cmp   ah, [rax+4Ah]
movzx rsi, dword ptr [rbp-25h]
push  4Ah
movzx rdi, dword ptr [rdi+rdx*8]

db  9Ah

rcr   byte ptr [rax+1Dh], cl
lodsd
xor   [rbp+6CF20173h], edx
xor   [rbp+66F8B593h], edx
push  rbx
sbb   ch, [rbx-0Fh]
stosd
int   87h
db   46h, 4Ch
out   33h, rax
xchg  eax, ebp
test  ecx, ebp
movsd
leave
push  rsp

db  16h

xchg  eax, esi
pop   rdi
```

### 3.17. INCORRECTLY DISASSEMBLED CODE

```

loc_B3D: ; CODE XREF: seg000:00000000000000B5F
    mov     ds:93CA685DF98A90F9h, eax
    jnz     short near ptr loc_AF3+6
    out    dx, eax
    cwde
    mov     bh, 5Dh ; ']'
    movsb
    pop    rbp

```

Listing 3.80: random noise (ARM (ARM mode))

BLNE	0xFE16A9D8
BGE	0x1634D0C
SVCCS	0x450685
STRNVT	R5, [PC],#-0x964
LDCGE	p6, c14, [R0],#0x168
STCCSL	p9, c9, [LR],#0x14C
CMNHIP	PC, R10,LSL#22
FLDMIADNV	LR!, {D4}
MCR	p5, 2, R2,c15,c6, 4
BLGE	0x1139558
BLGT	0xFF9146E4
STRNEB	R5, [R4],#0xCA2
STMNEIB	R5, {R0,R4,R6,R7,R9-SP,PC}
STMIA	R8, {R0,R2-R4,R7,R8,R10,SP,LR}^
STRB	SP, [R8],PC,ROR#18
LDCCS	p9, c13, [R6,#0x1BC]
LDRGE	R8, [R9,#0x66E]
STRNEB	R5, [R8],#-0x8C3
STCCSL	p15, c9, [R7,#-0x84]
RSBLS	LR, R2, R11,ASR LR
SVCGT	0x9B0362
SVCGT	0xA73173
STMNEDB	R11!, {R0,R1,R4-R6,R8,R10,R11,SP}
STR	R0, [R3],#-0xCE4
LDCGT	p15, c8, [R1,#0x2CC]
LDRCCB	R1, [R11],-R7,ROR#30
BLLT	0xFED9D58C
BL	0x13E60F4
LDMVSIB	R3!, {R1,R4-R7}^
USATNE	R10, #7, SP,LSL#11
LDRGEB	LR, [R1],#0xE56
STRPLT	R9, [LR],#0x567
LDRLT	R11, [R1],#-0x29B
SVCNV	0x12DB29
MVNNVS	R5, SP,LSL#25
LDCL	p8, c14, [R12,#-0x288]
STCNEL	p2, c6, [R6,#-0xBC]!
SVCNV	0x2E5A2F
BLX	0x1A8C97E
TEQGE	R3, #0x1100000
STMLSIA	R6, {R3,R6,R10,R11,SP}
BICPLS	R12, R2, #0x5800
BNE	0x7CC408
TEQGE	R2, R4,LSL#20
SUBS	R1, R11, #0x28C
BICVS	R3, R12, R7,ASR R0
LDRMI	R7, [LR],R3,LSL#21
BLMI	0x1A79234
STMVCDB	R6, {R0-R3,R6,R7,R10,R11}
EORMI	R12, R6, #0xC5
MCRRCS	p1, 0xF, R1,R3,c2

Listing 3.81: random noise (ARM (Thumb mode))

LSRS	R3, R6, #0x12
LDRH	R1, [R7,#0x2C]
SUBS	R0, #0x55 ; 'U'
ADR	R1, loc_3C

### 3.17. INCORRECTLY DISASSEMBLED CODE

```

LDR    R2, [SP,#0x218]
CMP    R4, #0x86
SXTB   R7, R4
LDR    R4, [R1,#0x4C]
STR    R4, [R4,R2]
STR    R0, [R6,#0x20]
BGT    0xFFFFFFF72
LDRH   R7, [R2,#0x34]
LDRSH  R0, [R2,R4]
LDRB   R2, [R7,R2]

DCB 0x17
DCB 0xED

STRB   R3, [R1,R1]
STR    R5, [R0,#0x6C]
LDMIA  R3, {R0-R5,R7}
ASRS   R3, R2, #3
LDR    R4, [SP,#0x2C4]
SVC    0xB5
LDR    R6, [R1,#0x40]
LDR    R5, =0xB2C5CA32
STMIA  R6, {R1-R4,R6}
LDR    R1, [R3,#0x3C]
STR    R1, [R5,#0x60]
BCC    0xFFFFFFF70
LDR    R4, [SP,#0x1D4]
STR    R5, [R5,#0x40]
ORRS   R5, R7

loc_3C ; DATA XREF: ROM:00000006
B      0xFFFFFFF98

```

Listing 3.82: random noise (MIPS little endian)

```

lw      $t9, 0xCB3($t5)
sb      $t5, 0x3855($t0)
sltiu  $a2, $a0, -0x657A
ldr     $t4, -0x4D99($a2)
daddi  $s0, $s1, 0x50A4
lw      $s7, -0x2353($s4)
bgtzl $a1, 0x17C5C

.byte 0x17
.byte 0xED
.byte 0x4B # K
.byte 0x54 # T

lwc2   $31, 0x66C5($sp)
lwu    $s1, 0x10D3($a1)
ldr     $t6, -0x204B($zero)
lwc1   $f30, 0x4DBE($s2)
daddiu $t1, $s1, 0x6BD9
lwu    $s5, -0x2C64($v1)
cop0   0x13D642D
bne   $gp, $t4, 0xFFFF9EF0
lh    $ra, 0x1819($s1)
sdl   $fp, -0x6474($t8)
jal   0x78C0050
ori   $v0, $s2, 0xC634
blez  $gp, 0xFFFFEA9D4
swl   $t8, -0x2CD4($s2)
sltiu $a1, $k0, 0x685
sdc1   $f15, 0x5964($at)
sw    $s0, -0x19A6($a1)
sltiu $t6, $a3, -0x66AD
lb    $t7, -0x4F6($t3)
sd    $fp, 0x4B02($a1)

```

It is also important to keep in mind that cleverly constructed unpacking and decryption code (including

### 3.18. OBFUSCATION

self-modifying) may look like noise as well, but still execute correctly.

## 3.18 Obfuscation

The obfuscation is an attempt to hide the code (or its meaning) from reverse engineers.

### 3.18.1 Text strings

As we saw in ([5.4 on page 710](#)), text strings may be really helpful.

Programmers who are aware of this try to hide them, making it impossible to find the string in [IDA](#) or any hex editor.

Here is the simplest method.

This is how the string can be constructed:

```
mov    byte ptr [ebx], 'h'
mov    byte ptr [ebx+1], 'e'
mov    byte ptr [ebx+2], 'l'
mov    byte ptr [ebx+3], 'l'
mov    byte ptr [ebx+4], 'o'
mov    byte ptr [ebx+5], ' '
mov    byte ptr [ebx+6], 'w'
mov    byte ptr [ebx+7], 'o'
mov    byte ptr [ebx+8], 'r'
mov    byte ptr [ebx+9], 'l'
mov    byte ptr [ebx+10], 'd'
```

The string can also be compared with another one like this:

```
mov    ebx, offset username
cmp    byte ptr [ebx], 'j'
jnz    fail
cmp    byte ptr [ebx+1], 'o'
jnz    fail
cmp    byte ptr [ebx+2], 'h'
jnz    fail
cmp    byte ptr [ebx+3], 'n'
jnz    fail
jz     it_is_john
```

In both cases, it is impossible to find these strings straightforwardly in a hex editor.

By the way, this is a way to work with the strings when it is impossible to allocate space for them in the data segment, for example in a [PIC](#) or in shellcode.

Another method is to use `sprintf()` for the construction:

```
sprintf(buf, "%s%c%s%c%s", "hel",'l',"o w",'o',"rld");
```

The code looks weird, but as a simple anti-reversing measure, it may be helpful.

Text strings may also be present in encrypted form, then every string usage is to be preceded by a string decrypting routine. For example: [8.5.2 on page 821](#).

### 3.18.2 Executable code

#### Inserting garbage

Executable code obfuscation implies inserting random garbage code between real one, which executes but does nothing useful.

A simple example:

Listing 3.83: original code

```
add    eax, ebx
mul    ecx
```

Listing 3.84: obfuscated code

```
xor    esi, 011223344h ; garbage
add    esi, eax        ; garbage
add    eax, ebx
mov    edx, eax        ; garbage
shl    edx, 4          ; garbage
mul    ecx
xor    esi, ecx        ; garbage
```

Here the garbage code uses registers which are not used in the real code (`ESI` and `EDX`). However, the intermediate results produced by the real code may be used by the garbage instructions for some extra mess—why not?

### Replacing instructions with bloated equivalents

- `MOV op1, op2` can be replaced by the `PUSH op2 / POP op1` pair.
- `JMP label` can be replaced by the `PUSH label / RET` pair. [IDA](#) will not show the references to the label.
- `CALL label` can be replaced by the following instructions triplet:  
`PUSH label_after_CALL_instruction / PUSH label / RET`.
- `PUSH op` can also be replaced with the following instructions pair:  
`SUB ESP, 4 (or 8) / MOV [ESP], op`.

### Always executed/never executed code

If the developer is sure that `ESI` at always 0 at that point:

```
mov    esi, 1
...
; some code not touching ESI
dec    esi
...
; some code not touching ESI
cmp    esi, 0
jz     real_code
; fake luggage
real_code:
```

The reverse engineer needs some time to get into it.

This is also called an *opaque predicate*.

Another example (and again, the developer is sure that `ESI` is always zero):

```
add    eax, ebx      ; real code
mul    ecx          ; real code
add    eax, esi      ; opaque predicate. XOR, AND or SHL, etc, can be here instead of ADD.
```

### Making a lot of mess

```
instruction 1
instruction 2
instruction 3
```

Can be replaced with:

### 3.18. OBFUSCATION

```
begin:      jmp     ins1_label
ins2_label: instruction 2
            jmp     ins3_label
ins3_label: instruction 3
            jmp     exit:
ins1_label: instruction 1
            jmp     ins2_label
exit:
```

### Using indirect pointers

```
dummy_data1    db      100h dup (0)
message1       db      'hello world',0
dummy_data2    db      200h dup (0)
message2       db      'another message',0
func           proc
...
    mov     eax, offset dummy_data1 ; PE or ELF reloc here
    add     eax, 100h
    push    eax
    call   dump_string
...
    mov     eax, offset dummy_data2 ; PE or ELF reloc here
    add     eax, 200h
    push    eax
    call   dump_string
...
func           endp
```

IDA will show references only to `dummy_data1` and `dummy_data2`, but not to the text strings.

Global variables and even functions may be accessed like that.

### 3.18.3 Virtual machine / pseudo-code

A programmer can construct his/her own PL or ISA and interpreter for it.

(Like the pre-5.0 Visual Basic, .NET or Java machines). The reverse engineer will have to spend some time to understand the meaning and details of all of the ISA's instructions.

He/she will also have to write a disassembler/decompiler of some sort.

### 3.18.4 Other things to mention

My own (yet weak) attempt to patch the Tiny C compiler to produce obfuscated code: <http://go.yurichev.com/17220>.

Using the `MOV` instruction for really complicated things: [Stephen Dolan, *mov is Turing-complete*, (2013)]<sup>22</sup>.

### 3.18.5 Exercise

- <http://challenges.re/29>

<sup>22</sup>Also available as <http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf>

### 3.19.1 Classes

#### A simple example

Internally, the representation of C++ classes is almost the same as the structures.

Let's try an example with two variables, two constructors and one method:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // default ctor
    {
        v1=667;
        v2=999;
    }

    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    }

    void dump()
    {
        printf ("%d; %d\n", v1, v2);
    }
};

int main()
{
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
}
```

#### MSVC: x86

Here is how the `main()` function looks like, translated into assembly language:

Listing 3.85: MSVC

```
_c2$ = -16 ; size = 8
_c1$ = -8 ; size = 8
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 16
    lea ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
```

### 3.19. C++

```
lea  ecx, DWORD PTR _c2$[ebp]
call ?dump@c@@QAEXXZ ; c::dump
xor  eax, eax
mov  esp, ebp
pop  ebp
ret  0
_main ENDP
```

Here's what's going on. For each object (instance of class *c*) 8 bytes are allocated, exactly the size needed to store the 2 variables.

For *c1* a default argumentless constructor `??0c@@QAE@XZ` is called. For *c2* another constructor `??0c@@QAE@HH@Z` is called and two numbers are passed as arguments.

A pointer to the object (*this* in C++ terminology) is passed in the `ECX` register. This is called `this-call` ([3.19.1](#))—the method for passing a pointer to the object.

MSVC does it using the `ECX` register. Needless to say, it is not a standardized method, other compilers can do it differently, e.g., via the first function argument (like GCC).

Why do these functions have such odd names? That's [name mangling](#).

A C++ class may contain several methods sharing the same name but having different arguments—that is polymorphism. And of course, different classes may have their own methods with the same name.

[Name mangling](#) enable us to encode the class name + method name + all method argument types in one ASCII string, which is then used as an internal function name. That's all because neither the linker, nor the DLL [OS](#) loader (mangled names may be among the DLL exports as well) knows anything about C++ or [OOP<sup>23</sup>](#).

The `dump()` function is called two times.

Now let's see the constructors' code:

Listing 3.86: MSVC

```
_this$ = -4      ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  DWORD PTR [eax], 667
    mov  ecx, DWORD PTR _this$[ebp]
    mov  DWORD PTR [ecx+4], 999
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8      ; size = 4
_b$ = 12     ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _a$[ebp]
    mov  DWORD PTR [eax], ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR _b$[ebp]
    mov  DWORD PTR [edx+4], eax
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
```

<sup>23</sup>Object-Oriented Programming

### 3.19. C++

```
pop    ebp  
ret    8  
??0c@@QAE@HH@Z ENDP ; c::c
```

The constructors are just functions, they use a pointer to the structure in `ECX`, copying the pointer into their own local variable, however, it is not necessary.

From the C++ standard (C++11 12.1) we know that constructors are not required to return any values. In fact, internally, the constructors return a pointer to the newly created object, i.e., `this`.

Now the `dump()` method:

Listing 3.87: MSVC

```
this$ = -4      ; size = 4  
?dump@c@@QAEXXZ PROC ; c::dump, COMDAT  
; _this$ = ecx  
    push ebp  
    mov  ebp, esp  
    push ecx  
    mov  DWORD PTR _this$[ebp], ecx  
    mov  eax, DWORD PTR _this$[ebp]  
    mov  ecx, DWORD PTR [eax+4]  
    push ecx  
    mov  edx, DWORD PTR _this$[ebp]  
    mov  eax, DWORD PTR [edx]  
    push eax  
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@  
    call _printf  
    add  esp, 12  
    mov  esp, ebp  
    pop  ebp  
    ret  0  
?dump@c@@QAEXXZ ENDP ; c::dump
```

Simple enough: `dump()` takes a pointer to the structure that contains the two `int`'s from `ECX`, takes both values from it and passes them to `printf()`.

The code is much shorter if compiled with optimizations (`/Ox`):

Listing 3.88: MSVC

```
??0c@@QAE@XZ PROC ; c::c, COMDAT  
; _this$ = ecx  
    mov  eax, ecx  
    mov  DWORD PTR [eax], 667  
    mov  DWORD PTR [eax+4], 999  
    ret  0  
??0c@@QAE@XZ ENDP ; c::c  
  
_a$ = 8  ; size = 4  
_b$ = 12 ; size = 4  
??0c@@QAE@HH@Z PROC ; c::c, COMDAT  
; _this$ = ecx  
    mov  edx, DWORD PTR _b$[esp-4]  
    mov  eax, ecx  
    mov  ecx, DWORD PTR _a$[esp-4]  
    mov  DWORD PTR [eax], ecx  
    mov  DWORD PTR [eax+4], edx  
    ret  8  
??0c@@QAE@HH@Z ENDP ; c::c  
  
?dump@c@@QAEXXZ PROC ; c::dump, COMDAT  
; _this$ = ecx  
    mov  eax, DWORD PTR [ecx+4]  
    mov  ecx, DWORD PTR [ecx]  
    push eax  
    push ecx  
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
```

### 3.19. C++

```
call _printf
add esp, 12
ret 0
?dump@c@@QAEXXZ ENDP ; c::dump
```

That's all. The other thing we must note is that the [stack pointer](#) hasn't been corrected with `add esp, X` after the constructor has been called. At the same time, the constructor has `ret 8` instead of `RET` at the end.

This is all because the `thiscall` ([3.19.1 on page 546](#)) calling convention is used here, which together with the `stdcall` ([6.1.2 on page 733](#)) method offers the [callee](#) to correct the stack instead of the [caller](#). The `ret X` instruction adds `X` to the value in `ESP`, then passes the control to the [caller](#) function.

See also the section about calling conventions ([6.1 on page 733](#)).

It also has to be noted that the compiler decides when to call the constructor and destructor—but we already know that from the C++ language basics.

### MSVC: x86-64

As we already know, the first 4 function arguments in x86-64 are passed in `RCX`, `RDX`, `R8` and `R9` registers, all the rest—via the stack.

Nevertheless, the *this* pointer to the object is passed in `RCX`, the first argument of the method in `RDX`, etc. We can see this in the `c(int a, int b)` method internals:

Listing 3.89: Optimizing MSVC 2012 x64

```
; void dump()

?dump@c@@QEAXXZ PROC ; c::dump
    mov     r8d, DWORD PTR [rcx+4]
    mov     edx, DWORD PTR [rcx]
    lea     rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@ ; '%d; %d'
    jmp     printf
?dump@c@@QEAXXZ ENDP ; c::dump

; c(int a, int b)

??0c@@QEAA@HH@Z PROC ; c::c
    mov     DWORD PTR [rcx], edx ; 1st argument: a
    mov     DWORD PTR [rcx+4], r8d ; 2nd argument: b
    mov     rax, rcx
    ret     0
??0c@@QEAA@HH@Z ENDP ; c::c

; default ctor

??0c@@QEAA@XZ PROC ; c::c
    mov     DWORD PTR [rcx], 667
    mov     DWORD PTR [rcx+4], 999
    mov     rax, rcx
    ret     0
??0c@@QEAA@XZ ENDP ; c::c
```

The `int` data type is still 32-bit in x64 <sup>24</sup>, so that is why 32-bit register parts are used here.

We also see `JMP printf` instead of `RET` in the `dump()` method, that *hack* we already saw earlier: [1.15.1 on page 155](#).

### GCC: x86

It is almost the same story in GCC 4.4.1, with a few exceptions.

<sup>24</sup>Apparently, for easier porting of 32-bit C/C++ code to x64

```

public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8  = dword ptr -8

push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
lea eax, [esp+20h+var_8]
mov [esp+20h+var_20], eax
call _ZN1cC1Ev
mov [esp+20h+var_18], 6
mov [esp+20h+var_1C], 5
lea eax, [esp+20h+var_10]
mov [esp+20h+var_20], eax
call _ZN1cC1Eii
lea eax, [esp+20h+var_8]
mov [esp+20h+var_20], eax
call _ZN1c4dumpEv
lea eax, [esp+20h+var_10]
mov [esp+20h+var_20], eax
call _ZN1c4dumpEv
mov eax, 0
leave
retn
main endp

```

Here we see another *name mangling* style, specific to GNU<sup>25</sup>. It can also be noted that the pointer to the object is passed as the first function argument—invisible to programmer, of course.

First constructor:

```

public _ZN1cC1Ev ; weak
_ZN1cC1Ev    proc near           ; CODE XREF: main+10
arg_0        = dword ptr 8

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
mov     dword ptr [eax], 667
mov     eax, [ebp+arg_0]
mov     dword ptr [eax+4], 999
pop     ebp
retn
_ZN1cC1Ev   endp

```

It just writes two numbers using the pointer passed in the first (and only) argument.

Second constructor:

```

public _ZN1cC1Eii
_ZN1cC1Eii  proc near

arg_0        = dword ptr 8
arg_4        = dword ptr 0Ch
arg_8        = dword ptr 10h

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]

```

<sup>25</sup>There is a good document about the various name mangling conventions in different compilers: [Agner Fog, *Calling conventions* (2015)].

### 3.19. C++

```
    mov    edx, [ebp+arg_4]
    mov    [eax], edx
    mov    eax, [ebp+arg_0]
    mov    edx, [ebp+arg_8]
    mov    [eax+4], edx
    pop    ebp
    retn
_ZN1cC1Eii    endp
```

This is a function, the analog of which can look like this:

```
void ZN1cC1Eii (int *obj, int a, int b)
{
    *obj=a;
    *(obj+1)=b;
};
```

...and that is completely predictable.

Now the `dump()` function:

```
public _ZN1c4dumpEv
_ZN1c4dumpEv proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8

    push    ebp
    mov     ebp, esp
    sub     esp, 18h
    mov     eax, [ebp+arg_0]
    mov     edx, [eax+4]
    mov     eax, [ebp+arg_0]
    mov     eax, [eax]
    mov     [esp+18h+var_10], edx
    mov     [esp+18h+var_14], eax
    mov     [esp+18h+var_18], offset aDD ; "%d; %d\n"
    call    _printf
    leave
    retn
_ZN1c4dumpEv endp
```

This function in its *internal representation* has only one argument, used as pointer to the object (*this*).

This function could be rewritten in C like this:

```
void ZN1c4dumpEv (int *obj)
{
    printf ("%d; %d\n", *obj, *(obj+1));
};
```

Thus, if we base our judgment on these simple examples, the difference between MSVC and GCC is the style of the encoding of function names (*name mangling*) and the method for passing a pointer to the object (via the `ECX` register or via the first argument).

### GCC: x86-64

The first 6 arguments, as we already know, are passed in the `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)*] <sup>26</sup>) registers, and the pointer to *this* via the first one (`rdi`) and that is what we see here. The `int` data type is also 32-bit here.

The `JMP` instead of `RET` hack is also used here.

<sup>26</sup>Also available as <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

```

; default ctor

_ZN1cC2Ev:
    mov  DWORD PTR [rdi], 667
    mov  DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)

_ZN1cC2Eii:
    mov  DWORD PTR [rdi], esi
    mov  DWORD PTR [rdi+4], edx
    ret

; dump()

_ZN1c4dumpEv:
    mov  edx, DWORD PTR [rdi+4]
    mov  esi, DWORD PTR [rdi]
    xor  eax, eax
    mov  edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp  printf

```

## Class inheritance

Inherited classes are similar to the simple structures we already discussed, but extended in inheritable classes.

Let's take this simple example:

```

#include <stdio.h>

class object
{
public:
    int color;
    object() { };
    object (int color) { this->color=color; };
    void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
private:
    int width, height, depth;
public:
    box(int color, int width, int height, int depth)
    {
        this->color=color;
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width,
        height, depth);
    };
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {

```

### 3.19. C++

```
    this->color=color;
    this->radius=radius;
};

void dump()
{
    printf ("this is sphere. color=%d, radius=%d\n", color, radius);
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};
```

Let's investigate the generated code of the `dump()` functions/methods and also `object::print_color()`, and see the memory layout for the structures-objects (for 32-bit code).

So, here are the `dump()` methods for several classes, generated by MSVC 2008 with `/Ox` and `/Ob0` options <sup>27</sup>

Listing 3.92: Optimizing MSVC 2008 /Ob0

```
?_C@_09GCEDOLPA@color?$_DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; `string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    push eax

; 'color=%d', 0aH, 00H
    push OFFSET ??_C@_09GCEDOLPA@color?$_DN?$CFd?6?$AA@
    call _printf
    add esp, 8
    ret 0
?print_color@object@@QAEXXZ ENDP ; object::print_color
```

Listing 3.93: Optimizing MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx+12]
    mov edx, DWORD PTR [ecx+8]
    push eax
    mov eax, DWORD PTR [ecx+4]
    mov ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx

; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H ; `string'
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$_DN?$CFd?0?5width?$_DN?$CFd?0@
    call _printf
    add esp, 20
    ret 0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Listing 3.94: Optimizing MSVC 2008 /Ob0

```
?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
```

<sup>27</sup>The `/Ob0` option stands for disabling inline expansion since function inlining can make our experiment harder

### 3.19. C++

```
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push eax
push ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CFd?0?5radius@
call _printf
add esp, 12
ret 0
?dump@sphere@@QAEXXZ ENDP ; sphere::dump
```

So, here is the memory layout:

(base class *object*)

offset	description
+0x0	int color

(inherited classes)

*box*:

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

*sphere*:

offset	description
+0x0	int color
+0x4	int radius

Let's see `main()` function body:

Listing 3.95: Optimizing MSVC 2008 /Ob0

```
PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
    sub esp, 24
    push 30
    push 20
    push 10
    push 1
    lea ecx, DWORD PTR _b$[esp+40]
    call ??0box@@QAE@HHH@Z ; box::box
    push 40
    push 2
    lea ecx, DWORD PTR _s$[esp+32]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    lea ecx, DWORD PTR _b$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea ecx, DWORD PTR _s$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea ecx, DWORD PTR _b$[esp+24]
    call ?dump@box@@QAEXXZ ; box::dump
    lea ecx, DWORD PTR _s$[esp+24]
    call ?dump@sphere@@QAEXXZ ; sphere::dump
    xor eax, eax
    add esp, 24
    ret 0
_main ENDP
```

The inherited classes must always add their fields after the base classes' fields, to make it possible for the base class methods to work with their own fields.

### 3.19. C++

When the `object::print_color()` method is called, a pointers to both the `box` and `sphere` objects are passed as `this`, and it can work with these objects easily since the `color` field in these objects is always at the pinned address (at offset `+0x0`).

It can be said that the `object::print_color()` method is agnostic in relation to the input object type as long as the fields are *pinned* at the same addresses, and this condition is always true.

And if you create inherited class of the `box` class, the compiler will add the new fields after the `depth` field, leaving the `box` class fields at the pinned addresses.

Thus, the `box::dump()` method will work fine for accessing the `color`, `width`, `height` and `depths` fields, which are always pinned at known addresses.

The code generated by GCC is almost the same, with the sole exception of passing the `this` pointer (as it has been explained above, it is passed as the first argument instead of using the `ECX` register).

## Encapsulation

Encapsulation is hiding the data in the *private* sections of the class, e.g. to allow access to them only from this class methods.

However, are there any marks in code about the fact that some field is private and some other—not?

No, there are no such marks.

Let's try this simple example:

```
#include <stdio.h>

class box
{
    private:
        int color, width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width,
                   height, depth);
        };
};
```

Let's compile it again in MSVC 2008 with `/Ox` and `/Ob0` options and see the `box::dump()` method code:

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov    eax, DWORD PTR [ecx+12]
    mov    edx, DWORD PTR [ecx+8]
    push   eax
    mov    eax, DWORD PTR [ecx+4]
    mov    ecx, DWORD PTR [ecx]
    push   edx
    push   eax
    push   ecx
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
    push  OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
    call   _printf
    add    esp, 20
    ret    0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Here is a memory layout of the class:

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

All fields are private and not allowed to be accessed from any other function, but knowing this layout, can we create code that modifies these fields?

To do this we'll add the `hack_oop_encapsulation()` function, which is not going to compile if it looked like this:

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code cant be compiled:
                // "error C2248: 'box::width' : cannot access private member declared in class ↴
                ↴ 'box'"
};
```

Nevertheless, if we cast the `box` type to a *pointer to an int array*, and we modify the array of `int`-s that we have, we can succeed.

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

This function's code is very simple—it can be said that the function takes a pointer to an array of `int`-s for input and writes 123 to the second `int`:

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
    mov DWORD PTR [eax+4], 123
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; hack_oop_encapsulation
```

Let's check how it works:

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

Let's run:

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

We see that the encapsulation is just protection of class fields only in the compilation stage.

The C++ compiler is not allowing the generation of code that modifies protected fields straightforwardly, nevertheless, it is possible with the help of *dirty hacks*.

## Multiple inheritance

Multiple inheritance is creating a class which inherits fields and methods from two or more classes.

Let's write a simple example again:

```

#include <stdio.h>

class box
{
public:
    int width, height, depth;
    box() { };
    box(int width, int height, int depth)
    {
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is box. width=%d, height=%d, depth=%d\n", width, height, depth);
    };
    int get_volume()
    {
        return width * height * depth;
    };
};

class solid_object
{
public:
    int density;
    solid_object() { };
    solid_object(int density)
    {
        this->density=density;
    };
    int get_density()
    {
        return density;
    };
    void dump()
    {
        printf ("this is solid_object. density=%d\n", density);
    };
};

class solid_box: box, solid_object
{
public:
    solid_box (int width, int height, int depth, int density)
    {
        this->width=width;
        this->height=height;
        this->depth=depth;
        this->density=density;
    };
    void dump()
    {
        printf ("this is solid_box. width=%d, height=%d, depth=%d, density=%d\n", width, ↴
            height, depth, density);
    };
    int get_weight() { return get_volume() * get_density(); };
};

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
}

```

### 3.19. C++

```

    printf ("%d\n", sb.get_weight());
    return 0;
};
```

Let's compile it in MSVC 2008 with the `/Ox` and `/Ob0` options and see the code of `box::dump()`, `solid_object::dump()` and `solid_box::dump()`:

Listing 3.96: Optimizing MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
    mov  edx, DWORD PTR [ecx+4]
    push eax
    mov  eax, DWORD PTR [ecx]
    push edx
    push eax
; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?$DN?$CFd?0?5height?$DN?$CFd@
    call _printf
    add  esp, 16
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Listing 3.97: Optimizing MSVC 2008 /Ob0

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax
; 'this is solid_object. density=%d', 0aH
    push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5density?$DN?$CFd@
    call _printf
    add  esp, 8
    ret  0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

Listing 3.98: Optimizing MSVC 2008 /Ob0

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is solid_box. width=%d, height=%d, depth=%d, density=%d', 0aH
    push OFFSET ??_C@_0D0@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?$CFd?0?5hei@
    call _printf
    add  esp, 20
    ret  0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

So, the memory layout for all three classes is:

*box* class:

offset	description
+0x0	width
+0x4	height
+0x8	depth

*solid\_object* class:

offset	description
+0x0	density

### 3.19. C++

It can be said that the `solid_box` class memory layout is *united*:

`solid_box` class:

offset	description
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

The code of the `box::get_volume()` and `solid_object::get_density()` methods is trivial:

Listing 3.99: Optimizing MSVC 2008 /Ob0

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret 0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume
```

Listing 3.100: Optimizing MSVC 2008 /Ob0

```
?get_density@solid_object@@QAEHXZ PROC ; solid_object::get_density, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    ret 0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::get_density
```

But the code of the `solid_box::get_weight()` method is much more interesting:

Listing 3.101: Optimizing MSVC 2008 /Ob0

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMDAT
; _this$ = ecx
    push esi
    mov esi, ecx
    push edi
    lea ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::get_density
    mov ecx, esi
    mov edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop edi
    pop esi
    ret 0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

`get_weight()` just calls two methods, but for `get_volume()` it just passes pointer to `this`, and for `get_density()` it passes a pointer to `this` incremented by 12 (or `0xC`) bytes, and there, in the `solid_box` class memory layout, the fields of the `solid_object` class start.

Thus, the `solid_object::get_density()` method will believe like it is dealing with the usual `solid_object` class, and the `box::get_volume()` method will work with its three fields, believing this is just the usual object of class `box`.

Thus, we can say, an object of a class, that inherits from several other classes, is representing in memory as a *united* class, that contains all inherited fields. And each inherited method is called with a pointer to the corresponding structure's part.

## Virtual methods

Yet another simple example:

```

#include <stdio.h>

class object
{
public:
    int color;
    object() { };
    object (int color) { this->color=color; };
    virtual void dump()
    {
        printf ("color=%d\n", color);
    };
};

class box : public object
{
private:
    int width, height, depth;
public:
    box(int color, int width, int height, int depth)
    {
        this->color=color;
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, ↴
        height, depth);
    };
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
}

```

Class *object* has a virtual method `dump()` that is being replaced in the inheriting *box* and *sphere* classes.

If we are in an environment where it is not known the type of an object, as in the `main()` function in example, where the virtual method `dump()` is called, the information about its type must be stored somewhere, to be able to call the relevant virtual method.

### 3.19. C++

Let's compile it in MSVC 2008 with the `/Ox` and `/Ob0` options and see the code of `main()`:

```
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub esp, 32
    push 30
    push 20
    push 10
    push 1
    lea ecx, DWORD PTR _b$[esp+48]
    call ??0box@@QAE@HHH@Z ; box::box
    push 40
    push 2
    lea ecx, DWORD PTR _s$[esp+40]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov eax, DWORD PTR _b$[esp+32]
    mov edx, DWORD PTR [eax]
    lea ecx, DWORD PTR _b$[esp+32]
    call edx
    mov eax, DWORD PTR _s$[esp+32]
    mov edx, DWORD PTR [eax]
    lea ecx, DWORD PTR _s$[esp+32]
    call edx
    xor eax, eax
    add esp, 32
    ret 0
_main ENDP
```

A pointer to the `dump()` function is taken somewhere from the object. Where could we store the address of the new method? Only somewhere in the constructors: there is no other place since nothing else is called in the `main()` function. <sup>28</sup>

Let's see the code of the constructor of the `box` class:

```
?_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box `RTTI Type Descriptor'
    DD 00H
    DB '.?AVbox@@', 00H

??_R1A@?0A@EA@box@@8 DD FLAT:??_R0?AVbox@@@8 ; box::`RTTI Base Class Descriptor at (0,-1,0,64)'
    DD 01H
    DD 00H
    DD 0xffffffffH
    DD 00H
    DD 040H
    DD FLAT:??_R3box@@8

??_R2box@@8 DD     FLAT:??_R1A@?0A@EA@box@@8 ; box::`RTTI Base Class Array'
    DD FLAT:??_R1A@?0A@EA@object@@8

??_R3box@@8 DD     00H ; box::`RTTI Class Hierarchy Descriptor'
    DD 00H
    DD 02H
    DD FLAT:??_R2box@@8

??_R4box@@6B@ DD 00H ; box::`RTTI Complete Object Locator'
    DD 00H
    DD 00H
    DD FLAT:??_R0?AVbox@@@8
    DD FLAT:??_R3box@@8

??_7box@@6B@ DD     FLAT:??_R4box@@6B@ ; box::`vftable'
    DD FLAT:?dump@box@@UAEXXZ

_color$ = 8    ; size = 4
_width$ = 12   ; size = 4
_height$ = 16   ; size = 4
_depth$ = 20   ; size = 4
```

<sup>28</sup>You can read more about pointers to functions in the relevant section: ([1.27 on page 385](#))

### 3.19. C++

```
?>0box@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
push esi
mov esi, ecx
call ??0object@@QAE@XZ ; object::object
mov eax, DWORD PTR _color$[esp]
mov ecx, DWORD PTR _width$[esp]
mov edx, DWORD PTR _height$[esp]
mov DWORD PTR [esi+4], eax
mov eax, DWORD PTR _depth$[esp]
mov DWORD PTR [esi+16], eax
mov DWORD PTR [esi], OFFSET ??_7box@@6B@_
mov DWORD PTR [esi+8], ecx
mov DWORD PTR [esi+12], edx
mov eax, esi
pop esi
ret 16
?>0box@@QAE@HHHH@Z ENDP ; box::box
```

Here we see a slightly different memory layout: the first field is a pointer to some table `box::`vftable'` (the name has been set by the MSVC compiler).

In this table we see a link to a table named `box::`RTTI Complete Object Locator'` and also a link to the `box::dump()` method.

These are called virtual methods table and [RTTI](#)<sup>29</sup>. The table of virtual methods has the addresses of methods and the [RTTI](#) table contains information about types.

By the way, the [RTTI](#) tables are used while calling `dynamic_cast` and `typeid` in C++. You can also see here the class name as a plain text string.

Thus, a method of the base `object` class may call the virtual method `object::dump()`, which in turn will call a method of an inherited class, since that information is present right in the object's structure.

Some additional CPU time is needed for doing look-ups in these tables and finding the right virtual method address, thus virtual methods are widely considered as slightly slower than common methods.

In GCC-generated code the [RTTI](#) tables are constructed slightly differently.

#### 3.19.2 ostream

Let's start again with a "hello world" example, but now we are going to use `ostream`:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Almost any C++ textbook tells us that the `<<` operation can be defined (*overloaded*) for other types. That is what is done in `ostream`. We see that `operator<<` is called for `ostream`:

Listing 3.102: MSVC 2012 (reduced listing)

```
$SG37112 DB 'Hello, world!', 0aH, 00H

_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAV?$basic_ostream@DU??
    ↴ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<std::char_traits<char> >
    add esp, 8
    xor eax, eax
    ret 0
_main ENDP
```

<sup>29</sup>Run-time type information

### 3.19. C++

Let's modify the example:

```
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

And again, from many C++ textbooks we know that the result of each `operator<<` in ostream is forwarded to the next one. Indeed:

Listing 3.103: MSVC 2012

```
$SG37112 DB 'world!', 0AH, 00H
$SG37113 DB 'Hello, ', 00H

_main PROC
    push OFFSET $SG37113 ; 'Hello, '
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?@
    ↴ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add esp, 8

    push OFFSET $SG37112 ; 'world!'
    push eax           ; result of previous function execution
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?@
    ↴ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add esp, 8

    xor eax, eax
    ret 0
_main ENDP
```

If we would rename `operator<<` method name to `f()`, that code will looks like:

```
f(f(std::cout, "Hello, "), "world!");
```

GCC generates almost the same code as MSVC.

### 3.19.3 References

In C++, references are pointers ([3.22 on page 615](#)) as well, but they are called *safe*, because it is harder to make a mistake while dealing with them (C++11 8.3.2).

For example, reference must always be pointing to an object of the corresponding type and cannot be NULL [Marshall Cline, *C++ FAQ8.6*].

Even more than that, references cannot be changed, it is impossible to point them to another object (reseat) [Marshall Cline, *C++ FAQ8.5*].

If we are going to try to change the example with pointers ([3.22 on page 615](#)) to use references instead ...

```
void f2 (int x, int y, int & sum, int & product)
{
    sum=x+y;
    product=x*y;
};
```

...then we can see that the compiled code is just the same as in the pointers example ([3.22 on page 615](#)):

Listing 3.104: Optimizing MSVC 2010

```
_x$ = 8          ; size = 4
_y$ = 12         ; size = 4
_sum$ = 16        ; size = 4
_product$ = 20      ; size = 4
?f2@YAXHAAH0@Z PROC    ; f2
```

### 3.19. C++

```
mov    ecx, DWORD PTR _y$[esp-4]
mov    eax, DWORD PTR _x$[esp-4]
lea    edx, DWORD PTR [eax+ecx]
imul  eax, ecx
mov    ecx, DWORD PTR _product$[esp-4]
push   esi
mov    esi, DWORD PTR _sum$[esp]
mov    DWORD PTR [esi], edx
mov    DWORD PTR [ecx], eax
pop    esi
ret    0
?f2@@@YAXHAAH0@Z ENDP ; f2
```

(The reason why C++ functions has such strange names is explained here: [3.19.1 on page 546](#).)

Hence, C++ references are as much efficient as usual pointers.

## 3.19.4 STL

N.B.: all examples here were checked only in 32-bit environment. x64 wasn't checked.

### std::string

#### Internals

Many string libraries [Dennis Yurichev, *C/C++ programming language notes*2.2] implement a structure that contains a pointer to a string buffer, a variable that always contains the current string length (which is very convenient for many functions: [Dennis Yurichev, *C/C++ programming language notes*2.2.1]) and a variable containing the current buffer size.

The string in the buffer is usually terminated with zero, in order to be able to pass a pointer to the buffer into the functions that take usual C **ASCII** strings.

It is not specified in the C++ standard how std::string has to be implemented, however, it is usually implemented as explained above.

The C++ string is not a class (as QString in Qt, for instance) but a template (basic\_string), this is made in order to support various character types: at least *char* and *wchar\_t*.

So, std::string is a class with *char* as its base type.

And std::wstring is a class with *wchar\_t* as its base type.

### MSVC

The MSVC implementation may store the buffer in place instead of using a pointer to a buffer (if the string is shorter than 16 symbols).

This implies that a short string is to occupy at least  $16 + 4 + 4 = 24$  bytes in 32-bit environment or at least  $16 + 8 + 8 = 32$

bytes in 64-bit one, and if the string is longer than 16 characters, we also have to add the length of the string itself.

Listing 3.105: example for MSVC

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

struct std_string
{
    union
    {
        char buf[16];
        char* ptr;
    } u;
```

### 3.19. C++

```
size_t size; // AKA 'Mysize' in MSVC
size_t capacity; // AKA 'Myres' in MSVC
};

void dump_std_string(std::string s)
{
    struct std_string *p=(struct std_string*)&s;
    printf ("[%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr : p->u.buf, p->size, p->capacity);
};

int main()
{
    std::string s1="short string";
    std::string s2="string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // that works without using c_str()
    printf ("%s\n", &s1);
    printf ("%s\n", s2);
}
```

Almost everything is clear from the source code.

A couple of notes:

If the string is shorter than 16 symbols, a buffer for the string is not to be allocated in the [heap](#).

This is convenient because in practice, a lot of strings are short indeed.

Looks like that Microsoft's developers chose 16 characters as a good balance.

One very important thing here can be seen at the end of main(): we're not using the `c_str()` method, nevertheless, if we compile and run this code, both strings will appear in the console!

This is why it works.

In the first case the string is shorter than 16 characters and the buffer with the string is located in the beginning of the `std::string` object (it can be treated as a structure). `printf()` treats the pointer as a pointer to the null-terminated array of characters, hence it works.

Printing the second string (longer than 16 characters) is even more dangerous: it is a typical programmer's mistake (or typo) to forget to write `c_str()`.

This works because at the moment a pointer to buffer is located at the start of structure.

This may stay unnoticed for a long time, until a longer string appears there at some time, then the process will crash.

## GCC

GCC's implementation of this structure has one more variable—reference count.

One interesting fact is that in GCC a pointer an instance of `std::string` instance points not to the beginning of the structure, but to the buffer pointer. In `libstdc++-v3/include/bits/basic_string.h` we can read that it was done for more convenient debugging:

```
* The reason you want _M_data pointing to the character %array and
* not the _Rep is so that the debugger can see the string
* contents. (Probably we should add a non-inline member to get
* the _Rep for the debugger to use, so users can check the actual
* string length.)
```

### [basic\\_string.h source code](#)

We consider this in our example:

Listing 3.106: example for GCC

```
#include <string>
```

### 3.19. C++

```
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // GCC type checking workaround
    struct std_string *p2=(struct std_string*)(p1-offsetof(struct std_string));
    printf ("[%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
};

int main()
{
    std::string s1="short string";
    std::string s2="string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // GCC type checking workaround:
    printf ("%s\n", *(char**)&s1);
    printf ("%s\n", *(char**)&s2);
}
```

A trickery has to be used to imitate the mistake we already have seen above because GCC has stronger type checking, nevertheless, printf() works here without c\_str() as well.

### A more advanced example

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

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf ("%s\n", s3.c_str());
}
```

Listing 3.107: MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub esp, 72

    push 7
    push OFFSET $SG39512
    lea ecx, DWORD PTR _s1$[esp+80]
    mov DWORD PTR _s1$[esp+100], 15
    mov DWORD PTR _s1$[esp+96], 0
    mov BYTE PTR _s1$[esp+80], 0
    call ?assign@$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QEAAV12@PBDI@Z ; ↴
        std::basic_string<char, std::char_traits<char>, std::allocator<char> >::assign

    push 7
```

### 3.19. C++

```
push OFFSET $SG39514
lea  ecx, DWORD PTR _s2$[esp+80]
mov  DWORD PTR _s2$[esp+100], 15
mov  DWORD PTR _s2$[esp+96], 0
mov  BYTE PTR _s2$[esp+80], 0
call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ; \
    std::basic_string<char, std::char_traits<char>, std::allocator<char>>::assign

lea  eax, DWORD PTR _s2$[esp+72]
push eax
lea  eax, DWORD PTR _s1$[esp+76]
push eax
lea  eax, DWORD PTR _s3$[esp+80]
push eax
call ??$?HDU?$char_traits@D@std@@V?$allocator@D@1@@std@@YA?AV?$basic_string@DU? \
    $char_traits@D@std@@V?$allocator@D@2@@@ABV10@0@Z ; std::operator+<char, std::char_traits< \
    char>, std::allocator<char>>

; inlined c_str() method:
cmp  DWORD PTR _s3$[esp+104], 16
lea  eax, DWORD PTR _s3$[esp+84]
cmovae eax, DWORD PTR _s3$[esp+84]

push eax
push OFFSET $SG39581
call _printf
add  esp, 20

cmp  DWORD PTR _s3$[esp+92], 16
jb   SHORT $LN119@main
push DWORD PTR _s3$[esp+72]
call ??3@YAXPAX@Z           ; operator delete
add  esp, 4
$LN119@main:
cmp  DWORD PTR _s2$[esp+92], 16
mov  DWORD PTR _s3$[esp+92], 15
mov  DWORD PTR _s3$[esp+88], 0
mov  BYTE PTR _s3$[esp+72], 0
jb   SHORT $LN151@main
push DWORD PTR _s2$[esp+72]
call ??3@YAXPAX@Z           ; operator delete
add  esp, 4
$LN151@main:
cmp  DWORD PTR _s1$[esp+92], 16
mov  DWORD PTR _s2$[esp+92], 15
mov  DWORD PTR _s2$[esp+88], 0
mov  BYTE PTR _s2$[esp+72], 0
jb   SHORT $LN195@main
push DWORD PTR _s1$[esp+72]
call ??3@YAXPAX@Z           ; operator delete
add  esp, 4
$LN195@main:
xor  eax, eax
add  esp, 72
ret  0
_main ENDP
```

The compiler does not construct strings statically: it would not be possible anyway if the buffer needs to be located in the [heap](#).

Instead, the [ASCII](#) strings are stored in the data segment, and later, at runtime, with the help of the "assign" method, the s1 and s2 strings are constructed. And with the help of [operator+](#), the s3 string is constructed.

Please note that there is no call to the c\_str() method, because its code is tiny enough so the compiler inlined it right there: if the string is shorter than 16 characters, a pointer to buffer is left in [EAX](#), otherwise the address of the string buffer located in the [heap](#) is fetched.

Next, we see calls to the 3 destructors, they are called if the string is longer than 16 characters: then the buffers in the [heap](#) have to be freed. Otherwise, since all three std::string objects are stored in the stack,

### 3.19. C++

they are freed automatically, when the function ends.

As a consequence, processing short strings is faster, because of less [heap](#) accesses.

GCC code is even simpler (because the GCC way, as we saw above, is to not store shorter strings right in the structure):

Listing 3.108: GCC 4.8.1

```
.LC0:
    .string "Hello, "
.LC1:
    .string "world!\n"
main:
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, -16
    sub  esp, 32
    lea   ebx, [esp+28]
    lea   edi, [esp+20]
    mov  DWORD PTR [esp+8], ebx
    lea   esi, [esp+24]
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], edi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+8], ebx
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC1
    mov  DWORD PTR [esp], esi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+4], edi
    mov  DWORD PTR [esp], ebx

    call _ZNSsC1ERKSs

    mov  DWORD PTR [esp+4], esi
    mov  DWORD PTR [esp], ebx

    call _ZNSs6appendERKSs

; inlined c_str():
    mov  eax, DWORD PTR [esp+28]
    mov  DWORD PTR [esp], eax

    call puts

    mov  eax, DWORD PTR [esp+28]
    lea   ebx, [esp+19]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_DisposeERKSaIcE
    mov  eax, DWORD PTR [esp+24]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_DisposeERKSaIcE
    mov  eax, DWORD PTR [esp+20]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_DisposeERKSaIcE
    lea   esp, [ebp-12]
    xor  eax, eax
    pop  ebx
    pop  esi
```

### 3.19. C++

```
pop edi  
pop ebp  
ret
```

It can be seen that it's not a pointer to the object that is passed to destructors, but rather an address 12 bytes (or 3 words) before, i.e., a pointer to the real start of the structure.

### std::string as a global variable

Experienced C++ programmers know that global variables of [STL<sup>30</sup>](#) types can be defined without problems.

Yes, indeed:

```
#include <stdio.h>  
#include <string>  
  
std::string s="a string";  
  
int main()  
{  
    printf ("%s\n", s.c_str());  
}
```

But how and where `std::string` constructor will be called?

In fact, this variable is to be initialized even before `main()` start.

**Listing 3.109: MSVC 2012:** here is how a global variable is constructed and also its destructor is registered

```
??__Es@@YAXXZ PROC  
    push 8  
    push OFFSET $SG39512 ; 'a string'  
    mov  ecx, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s  
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ; ↴  
        std::basic_string<char, std::char_traits<char>, std::allocator<char> >::assign  
    push OFFSET ??__Fs@@YAXXZ ; `dynamic atexit destructor for 's'  
    call _atexit  
    pop  ecx  
    ret  0  
??__Es@@YAXXZ ENDP
```

**Listing 3.110: MSVC 2012:** here a global variable is used in `main()`

```
$SG39512 DB 'a string', 00H  
$SG39519 DB '%s', 0aH, 00H  
  
_main PROC  
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16  
    mov  eax, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s  
    cmovae eax, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A  
    push eax  
    push OFFSET $SG39519 ; '%s'  
    call _printf  
    add  esp, 8  
    xor  eax, eax  
    ret  0  
_main ENDP
```

**Listing 3.111: MSVC 2012:** this destructor function is called before exit

```
??__Fs@@YAXXZ PROC  
    push ecx  
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16  
    jb   SHORT $LN23@dynamic  
    push esi
```

<sup>30</sup>(C++) Standard Template Library: [3.19.4 on page 563](#)

### 3.19. C++

```
mov    esi, DWORD PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
lea    ecx, DWORD PTR $T2[esp+8]
call   ??0$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
push   OFFSET ?s@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
lea    ecx, DWORD PTR $T2[esp+12]
call   ??$destroy@PAD@?$_Wrap_alloc@V?$allocator@D@std@@@std@@QEXPAPAD@Z
lea    ecx, DWORD PTR $T1[esp+8]
call   ??0$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
push   esi
call   ??3@YAXPAX@Z ; operator delete
add    esp, 4
pop    esi
$LN23@dynamic:
mov    DWORD PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 15
mov    DWORD PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+16, 0
mov    BYTE PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A, 0
pop    ecx
ret    0
??_Fs@YAXXZ ENDP
```

In fact, a special function with all constructors of global variables is called from [CRT](#), before main().

More than that: with the help of atexit() another function is registered, which contain calls to all destructors of such global variables.

GCC works likewise:

Listing 3.112: GCC 4.8.1

```
main:
push  ebp
mov   ebp, esp
and   esp, -16
sub   esp, 16
mov   eax, DWORD PTR s
mov   DWORD PTR [esp], eax
call  puts
xor   eax, eax
leave
ret
.LC0:
.string "a string"
_GLOBAL_sub_I_s:
sub   esp, 44
lea   eax, [esp+31]
mov   DWORD PTR [esp+8], eax
mov   DWORD PTR [esp+4], OFFSET FLAT:.LC0
mov   DWORD PTR [esp], OFFSET FLAT:s
call  __ZNSSc1EPKcRKSaIcE
mov   DWORD PTR [esp+8], OFFSET FLAT:_dso_handle
mov   DWORD PTR [esp+4], OFFSET FLAT:s
mov   DWORD PTR [esp], OFFSET FLAT:_ZNSSd1Ev
call  __cxa_atexit
add   esp, 44
ret
.LFE645:
.size _GLOBAL_sub_I_s, .-_GLOBAL_sub_I_s
.section .init_array, "aw"
.align 4
.long _GLOBAL_sub_I_s
.globl s
.bss
.align 4
.type  s, @object
.size  s, 4
s:
.zero 4
.hidden __dso_handle
```

But it does not create a separate function for this, each destructor is passed to atexit(), one by one.

This is the well-known doubly-linked list: each element has two pointers, to the previous and next elements.

This implies that the memory footprint is enlarged by 2 [words](#) for each element (8 bytes in 32-bit environment or 16 bytes in 64-bit).

C++ STL just adds the “next” and “previous” pointers to the existing structure of the type that you want to unite in a list.

Let's work out an example with a simple 2-variable structure that we want to store in a list.

Although the C++ standard does not say how to implement it, both MSVC's and GCC's implementations are straightforward and similar, so here is only one source code for both:

```
#include <stdio.h>
#include <list>
#include <iostream>

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
           n, n->_Next, n->_Prev, n->x, n->y);
};

void dump_List_vals (struct List_node* n)
{
    struct List_node* current=n;

    for (;;)
    {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    };
};

void dump_List_val (unsigned int *a)
{
#ifdef _MSC_VER
    // GCC implementation does not have "size" field
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
#endif
    dump_List_vals ((struct List_node*)a[0]);
};

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
```

### 3.19. C++

```
t1.y=2;
l.push_front (t1);
t1.x=3;
t1.y=4;
l.push_front (t1);
t1.x=5;
t1.y=6;
l.push_back (t1);

printf ("* 3-elements list:\n");
dump_List_val((unsigned int*)(void*)&l);

std::list<struct a>::iterator tmp;
printf ("node at .begin:\n");
tmp=l.begin();
dump_List_node ((struct List_node *)*(void**)&tmp);
printf ("node at .end:\n");
tmp=l.end();
dump_List_node ((struct List_node *)*(void**)&tmp);

printf ("* let's count from the beginning:\n");
std::list<struct a>::iterator it=l.begin();
printf ("1st element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("2nd element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("3rd element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("element at .end(): %d %d\n", (*it).x, (*it).y);

printf ("* let's count from the end:\n");
std::list<struct a>::iterator it2=l.end();
printf ("element at .end(): %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("3rd element: %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("2nd element: %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("1st element: %d %d\n", (*it2).x, (*it2).y);

printf ("removing last element...\n");
l.pop_back();
dump_List_val((unsigned int*)(void*)&l);
};
```

## GCC

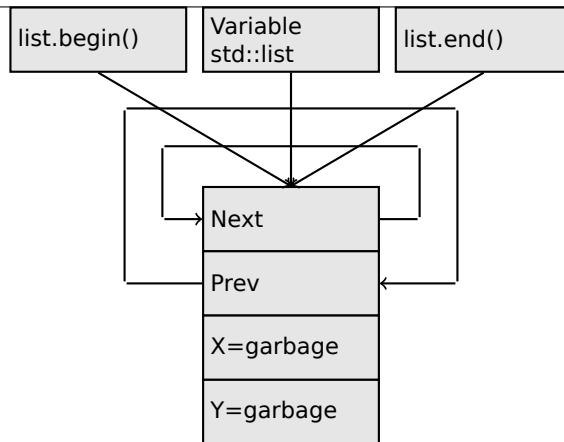
Let's start with GCC.

When we run the example, we'll see a long dump, let's work with it in pieces.

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
```

Here we see an empty list.

Despite the fact it is empty, it has one element with garbage ([AKA dummy node](#)) in *x* and *y*. Both the "next" and "prev" pointers are pointing to the self node:



At this moment, the .begin and .end iterators are equal to each other.

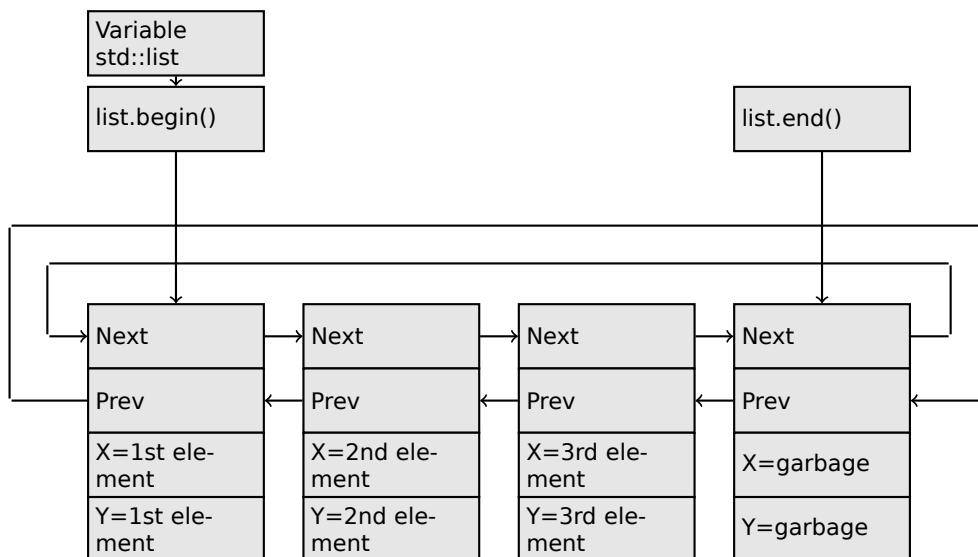
If we push 3 elements, the list internally will be:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

The last element is still at 0x0028fe90, it not to be moved until the list's disposal.

It still contain random garbage in *x* and *y* (5 and 6). By coincidence, these values are the same as in the last element, but it doesn't mean that they are meaningful.

Here is how these 3 elements are stored in memory:



The *l* variable always points to the first node.

The .begin() and .end() iterators are not variables, but functions, which when called return pointers to the corresponding nodes.

Having a dummy element (AKA *sentinel node*) is a very popular practice in implementing doubly-linked lists.

Without it, a lot of operations may become slightly more complex and, hence, slower.

The iterator is in fact just a pointer to a node. list.begin() and list.end() just return pointers.

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

The fact that the last element has a pointer to the first and the first element has a pointer to the last one remind us of circular lists.

### 3.19. C++

This is very helpful here: having a pointer to the first list element, i.e., that is in the *l* variable, it is easy to get a pointer to the last one quickly, without the necessity to traverse the whole list.

Inserting an element at the end of the list is also quick, thanks to this feature.

`operator--` and `operator++` just set the current iterator's value to the `current_node->prev` or `current_node->next` values.

The reverse iterators (`.rbegin`, `.rend`) work just as the same, but in reverse.

`operator*` just returns a pointer to the point in the node structure, where the user's structure starts, i.e., a pointer to the first element of the structure (*x*).

The list insertion and deletion are trivial: just allocate a new node (or deallocate) and update all pointers to be valid.

That's why an iterator may become invalid after element deletion: it may still point to the node that has been already deallocated. This is also called a *dangling pointer*.

And of course, the information from the freed node (to which iterator still points) cannot be used anymore.

The GCC implementation (as of 4.8.1) doesn't store the current size of the list: this implies a slow `.size()` method: it has to traverse the whole list to count the elements, because it doesn't have any other way to get the information.

This means that this operation is  $O(n)$ , i.e., it steadily gets slower as the list grows.

Listing 3.113: Optimizing GCC 4.8.1 -fno-inline-small-functions

```
main proc near
    push ebp
    mov ebp, esp
    push esi
    push ebx
    and esp, 0FFFFFFF0h
    sub esp, 20h
    lea ebx, [esp+10h]
    mov dword ptr [esp], offset s ; /* empty list:*/
    mov [esp+10h], ebx
    mov [esp+14h], ebx
    call puts
    mov [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    lea esi, [esp+18h]
    mov [esp+4], esi
    mov [esp], ebx
    mov dword ptr [esp+18h], 1 ; X for new element
    mov dword ptr [esp+1Ch], 2 ; Y for new element
    call _ZNSt4listI1aSais0_EE10push_frontERKS0_ ; std::list<a, std::allocator<a>>::push_front(a&
    ↴ const&)
    mov [esp+4], esi
    mov [esp], ebx
    mov dword ptr [esp+18h], 3 ; X for new element
    mov dword ptr [esp+1Ch], 4 ; Y for new element
    call _ZNSt4listI1aSais0_EE10push_frontERKS0_ ; std::list<a, std::allocator<a>>::push_front(a&
    ↴ const&)
    mov dword ptr [esp], 10h
    mov dword ptr [esp+18h], 5 ; X for new element
    mov dword ptr [esp+1Ch], 6 ; Y for new element
    call _Znwj ; operator new(uint)
    cmp eax, 0FFFFFFF8h
    jz short loc_80002A6
    mov ecx, [esp+1Ch]
    mov edx, [esp+18h]
    mov [eax+0Ch], ecx
    mov [eax+8], edx

loc_80002A6: ; CODE XREF: main+86
    mov [esp+4], ebx
    mov [esp], eax
    call _ZNSt8__detail15_List_node_base7_M_hookEPS0_ ; std::__detail::_List_node_base::M_hook(
    ↴ (std::__detail::_List_node_base*)
```

### 3.19. C++

```
mov dword ptr [esp], offset a3ElementsList ; "* 3-elements list:"
call puts
mov [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
mov dword ptr [esp], offset aNodeAt_begin ; "node at .begin:"
call puts
mov eax, [esp+10h]
mov [esp], eax
call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
mov dword ptr [esp], offset aNodeAt_end ; "node at .end:"
call puts
mov [esp], ebx
call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
mov dword ptr [esp], offset aLetSCountFromT ; "* let's count from the beginning:"
call puts
mov esi, [esp+10h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi] ; operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi] ; operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov eax, [esi] ; operator++: get ->next pointer
mov edx, [eax+0Ch]
mov [esp+0Ch], edx
mov eax, [eax+8]
mov dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov dword ptr [esp], offset aLetSCountFro_0 ; "* let's count from the end:"
call puts
mov eax, [esp+1Ch]
mov dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
mov dword ptr [esp], 1
mov [esp+0Ch], eax
mov eax, [esp+18h]
mov [esp+8], eax
call __printf_chk
mov esi, [esp+14h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi+4] ; operator--: get ->prev pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov dword ptr [esp], 1
```

### 3.19. C++

```
mov [esp+8], eax
call __printf_chk
mov eax, [esi+4] ; operator--: get ->prev pointer
mov edx, [eax+0Ch]
mov [esp+0Ch], edx
mov eax, [eax+8]
mov dword ptr [esp+4], offset alstElementDD ; "1st element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov dword ptr [esp], offset aRemovingLastEl ; "removing last element..."
call puts
mov esi, [esp+14h]
mov [esp], esi
call _ZNSt8_detail15_List_node_base9_M_unhookEv ; std::__detail::_List_node_base::~
    ↴ _M_unhook(void)
mov [esp], esi ; void *
call _ZdlPv ; operator delete(void *)
mov [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
mov [esp], ebx
call _ZNSt10_List_base11aSaIS0_EE8_M_clearEv ; std::List_base<a, std::allocator<a>>::~
    ↴ _M_clear(void)
lea esp, [ebp-8]
xor eax, eax
pop ebx
pop esi
pop ebp
retn
main endp
```

Listing 3.114: The whole output

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
* let's count from the beginning:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 5 6
* let's count from the end:
element at .end(): 5 6
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0028fe90 _Prev=0x000349a0 x=1 y=2
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034988 x=5 y=6
```

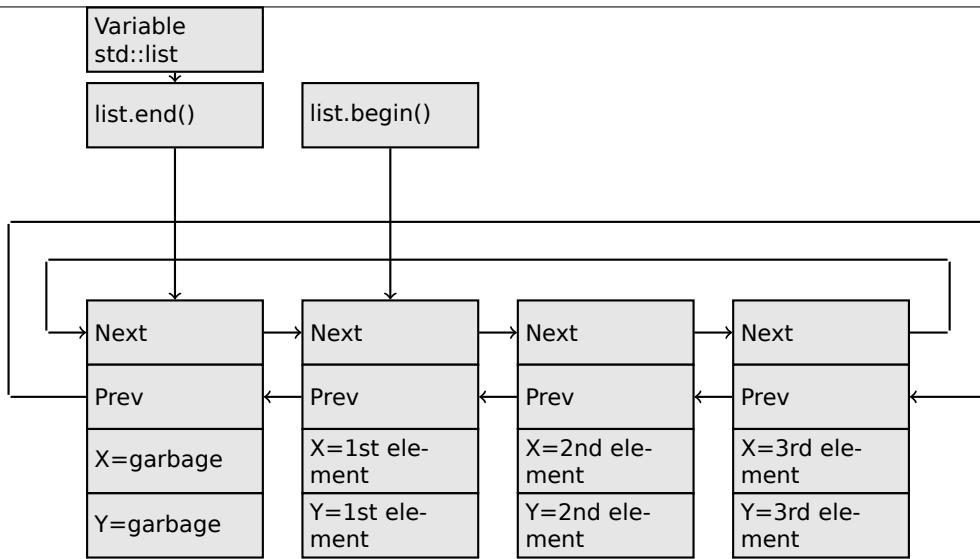
### MSVC

MSVC's implementation (2012) is just the same, but it also stores the current size of the list.

This implies that the `.size()` method is very fast ( $O(1)$ ): it just reads one value from memory.

On the other hand, the size variable must be updated at each insertion/deletion.

MSVC's implementation is also slightly different in the way it arranges the nodes:



GCC has its dummy element at the end of the list, while MSVC's is at the beginning.

Listing 3.115: Optimizing MSVC 2012 /Fa2.asm /GS- /Ob1

```

_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8
_main PROC
    sub esp, 16
    push ebx
    push esi
    push edi
    push 0
    push 0
    lea ecx, DWORD PTR _l$[esp+36]
    mov DWORD PTR _l$[esp+40], 0
    ; allocate first garbage element
    call ?_Buynode0@?$_List_alloc@U@?$_List_base_types@Ua@@V?_
    ↳ $allocator@Ua@@@std@@@std@@@std@@@QAEPAU?$_List_node@Ua@@PAX@2@PAU32@0@Z ; std::
    ↳ _List_alloc<0,std::_List_base_types<a,std::allocator<a>>::_Buynode0
    mov edi, DWORD PTR __imp__printf
    mov ebx, eax
    push OFFSET $SG40685 ; '* empty list:'
    mov DWORD PTR _l$[esp+32], ebx
    call edi ; printf
    lea eax, DWORD PTR _l$[esp+32]
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    mov esi, DWORD PTR [ebx]
    add esp, 8
    lea eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [esi+4]
    lea ecx, DWORD PTR _l$[esp+36]
    push esi
    mov DWORD PTR _t1$[esp+40], 1 ; data for a new node
    mov DWORD PTR _t1$[esp+44], 2 ; data for a new node
    ; allocate new node
    call ??$_Buynode@ABuA@@@?$_List_buy@Ua@@V?$_allocator@Ua@@@std@@@std@@@QAEPAU?_
    ↳ $List_node@Ua@@PAX@1@PAU21@0ABuA@@@Z ; std::_List_buy<a,std::allocator<a>>::_Buynode<a>
    ↳ const &gt;
    mov DWORD PTR [esi+4], eax
    mov ecx, DWORD PTR [eax+4]
    mov DWORD PTR _t1$[esp+28], 3 ; data for a new node
    mov DWORD PTR [ecx], eax
    mov esi, DWORD PTR [ebx]
    lea eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [esi+4]
    lea ecx, DWORD PTR _l$[esp+36]
    push esi
    mov DWORD PTR _t1$[esp+44], 4 ; data for a new node

```

### 3.19. C++

```
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU?_
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a, std::allocator<a>>::_Buynode<a>
↳ const &gt;
mov  DWORD PTR [esi+4], eax
mov  ecx, DWORD PTR [eax+4]
mov  DWORD PTR _t1$[esp+28], 5 ; data for a new node
mov  DWORD PTR [ecx], eax
lea   eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [ebx+4]
lea   ecx, DWORD PTR _l$[esp+36]
push ebx
mov  DWORD PTR _t1$[esp+44], 6 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU?_
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a, std::allocator<a>>::_Buynode<a>
↳ const &gt;
mov  DWORD PTR [ebx+4], eax
mov  ecx, DWORD PTR [eax+4]
push OFFSET $SG40689 ; '* 3-elements list:'
mov  DWORD PTR _l$[esp+36], 3
mov  DWORD PTR [ecx], eax
call edi ; printf
lea   eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
push OFFSET $SG40831 ; 'node at .begin:'
call edi ; printf
push DWORD PTR [ebx] ; get next field of node $l$ variable points to
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40835 ; 'node at .end:'
call edi ; printf
push ebx ; pointer to the node "l" variable points to!
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40839 ; '* let''s count from the begin:'
call edi ; printf
mov  esi, DWORD PTR [ebx] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40846 ; '1st element: %d %d'
call edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40848 ; '2nd element: %d %d'
call edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40850 ; '3rd element: %d %d'
call edi ; printf
mov  eax, DWORD PTR [esi] ; operator++: get ->next pointer
add esp, 64
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40852 ; 'element at .end(): %d %d'
call edi ; printf
push OFFSET $SG40853 ; '* let''s count from the end:'
call edi ; printf
push DWORD PTR [ebx+12] ; use x and y fields from the node "l" variable points to
push DWORD PTR [ebx+8]
push OFFSET $SG40860 ; 'element at .end(): %d %d'
call edi ; printf
mov  esi, DWORD PTR [ebx+4] ; operator--: get ->prev pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40862 ; '3rd element: %d %d'
call edi ; printf
mov  esi, DWORD PTR [esi+4] ; operator--: get ->prev pointer
```

### 3.19. C++

```

push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40864 ; '2nd element: %d %d'
call edi ; printf
mov eax, DWORD PTR [esi+4] ; operator--: get ->prev pointer
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40866 ; '1st element: %d %d'
call edi ; printf
add esp, 64
push OFFSET $SG40867 ; 'removing last element...'
call edi ; printf
mov edx, DWORD PTR [ebx+4]
add esp, 4

; prev=next?
; it is the only element, garbage one?
; if yes, do not delete it!
cmp edx, ebx
je SHORT $LN349@main
mov ecx, DWORD PTR [edx+4]
mov eax, DWORD PTR [edx]
mov DWORD PTR [ecx], eax
mov ecx, DWORD PTR [edx]
mov eax, DWORD PTR [edx+4]
push edx
mov DWORD PTR [ecx+4], eax
call ??3@YAXPAX@Z ; operator delete
add esp, 4
mov DWORD PTR _l$[esp+32], 2
$LN349@main:
lea eax, DWORD PTR _l$[esp+28]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
mov eax, DWORD PTR [ebx]
add esp, 4
mov DWORD PTR [ebx], ebx
mov DWORD PTR [ebx+4], ebx
cmp eax, ebx
je SHORT $LN412@main
$LL414@main:
mov esi, DWORD PTR [eax]
push eax
call ??3@YAXPAX@Z ; operator delete
add esp, 4
mov eax, esi
cmp esi, ebx
jne SHORT $LL414@main
$LN412@main:
push ebx
call ??3@YAXPAX@Z ; operator delete
add esp, 4
xor eax, eax
pop edi
pop esi
pop ebx
add esp, 16
ret 0
_main ENDP

```

Unlike GCC, MSVC's code allocates the dummy element at the start of the function with the help of the "Buynode" function, it is also used to allocate the rest of the nodes ( GCC's code allocates the first element in the local stack).

Listing 3.116: The whole output

```

* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072
* 3-elements list:

```

### 3.19. C++

```
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
* let's count from the beginning:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072
* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2
```

### C++11 std::forward\_list

The same thing as std::list, but singly-linked one, i.e., having only the “next” field at each node. It has a smaller memory footprint, but also don’t offer the ability to traverse list backwards.

### std::vector

We would call `std::vector` a safe wrapper of the [PODT<sup>31</sup>](#) C array. Internally it is somewhat similar to `std::string` ([3.19.4 on page 563](#)): it has a pointer to the allocated buffer, a pointer to the end of the array, and a pointer to the end of the allocated buffer.

The array’s elements lie in memory adjacently to each other, just like in a normal array ([1.20 on page 268](#)). In C++11 there is a new method called `.data()`, that returns a pointer to the buffer, like `.c_str()` in `std::string`.

The buffer allocated in the [heap](#) can be larger than the array itself.

Both MSVC’s and GCC’s implementations are similar, just the names of the structure’s fields are slightly different<sup>32</sup>, so here is one source code that works for both compilers. Here is again the C-like code for dumping the structure of `std::vector`:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // GCC structure is the same, but names are: _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
```

<sup>31</sup>(C++) Plain Old Data Type

<sup>32</sup>GCC internals: <http://go.yurichev.com/17086>

### 3.19. C++

```
{  
    printf ("_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst, in->Mylast, in->Myend);  
    size_t size=(in->Mylast-in->Myfirst);  
    size_t capacity=(in->Myend-in->Myfirst);  
    printf ("size=%d, capacity=%d\n", size, capacity);  
    for (size_t i=0; i<size; i++)  
        printf ("element %d: %d\n", i, in->Myfirst[i]);  
};  
  
int main()  
{  
    std::vector<int> c;  
    dump ((struct vector_of_ints*)(void*)&c);  
    c.push_back(1);  
    dump ((struct vector_of_ints*)(void*)&c);  
    c.push_back(2);  
    dump ((struct vector_of_ints*)(void*)&c);  
    c.push_back(3);  
    dump ((struct vector_of_ints*)(void*)&c);  
    c.push_back(4);  
    dump ((struct vector_of_ints*)(void*)&c);  
    c.reserve (6);  
    dump ((struct vector_of_ints*)(void*)&c);  
    c.push_back(5);  
    dump ((struct vector_of_ints*)(void*)&c);  
    c.push_back(6);  
    dump ((struct vector_of_ints*)(void*)&c);  
    printf ("%d\n", c.at(5)); // with bounds checking  
    printf ("%d\n", c[8]); // operator[], without bounds checking  
};
```

Here is the output of this program when compiled in MSVC:

```
_Myfirst=00000000, _Mylast=00000000, _Myend=00000000  
size=0, capacity=0  
_Myfirst=0051CF48, _Mylast=0051CF4C, _Myend=0051CF4C  
size=1, capacity=1  
element 0: 1  
_Myfirst=0051CF58, _Mylast=0051CF60, _Myend=0051CF60  
size=2, capacity=2  
element 0: 1  
element 1: 2  
_Myfirst=0051C278, _Mylast=0051C284, _Myend=0051C284  
size=3, capacity=3  
element 0: 1  
element 1: 2  
element 2: 3  
_Myfirst=0051C290, _Mylast=0051C2A0, _Myend=0051C2A0  
size=4, capacity=4  
element 0: 1  
element 1: 2  
element 2: 3  
element 3: 4  
_Myfirst=0051B180, _Mylast=0051B190, _Myend=0051B198  
size=4, capacity=6  
element 0: 1  
element 1: 2  
element 2: 3  
element 3: 4  
_Myfirst=0051B180, _Mylast=0051B194, _Myend=0051B198  
size=5, capacity=6  
element 0: 1  
element 1: 2  
element 2: 3  
element 3: 4  
element 4: 5  
_Myfirst=0051B180, _Mylast=0051B198, _Myend=0051B198  
size=6, capacity=6  
element 0: 1  
element 1: 2
```

### 3.19. C++

```
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158
```

As it can be seen, there is no allocated buffer when `main()` starts. After the first `push_back()` call, a buffer is allocated. And then, after each `push_back()` call, both array size and buffer size (*capacity*) are increased. But the buffer address changes as well, because `push_back()` reallocates the buffer in the **heap** each time. It is costly operation, that's why it is very important to predict the size of the array in the future and reserve enough space for it with the `.reserve()` method.

The last number is garbage: there are no array elements at this point, so a random number is printed. This illustrates the fact that `operator[]` of `std::vector` does not check if the index is in the array's bounds. The slower `.at()` method, however, does this checking and throws an `std::out_of_range` exception in case of error.

Let's see the code:

Listing 3.117: MSVC 2012 /GS- /Ob1

```
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
_Pos$ = 8 ; size = 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z PROC ; std::vector<int, std::allocator<int> >::at, COMDAT
; _this$ = ecx
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _this$[ebp], ecx
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR _this$[ebp]
    mov edx, DWORD PTR [eax+4]
    sub edx, DWORD PTR [ecx]
    sar edx, 2
    cmp edx, DWORD PTR __Pos$[ebp]
    ja SHORT $LN1@at
    push OFFSET ??_C@_0BM@NMJKDPP0@invalid?5vector?$DMT?$D0?5subscript?$$AA@
    call DWORD PTR __imp__Xout_of_range@std@@YAXPBD@Z
$LN1@at:
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR [eax]
    mov edx, DWORD PTR __Pos$[ebp]
    lea eax, DWORD PTR [ecx+edx*4]
$LN3@at:
    mov esp, ebp
    pop ebp
    ret 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ; std::vector<int, std::allocator<int> >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8 ; size = 4
$T6 = -4 ; size = 4
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 36
    mov DWORD PTR _c$[ebp], 0      ; Myfirst
    mov DWORD PTR _c$[ebp+4], 0    ; Mylast
```

### 3.19. C++

```
mov  DWORD PTR _c$[ebp+8], 0      ; Myend
lea   eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T6[ebp], 1
lea   ecx, DWORD PTR $T6[ebp]
push ecx
lea   ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@@QAEX$$QAH@Z ; std::vector<int, std::allocat
\ allocator<int> >::push_back
lea   edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T5[ebp], 2
lea   eax, DWORD PTR $T5[ebp]
push eax
lea   ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@@QAEX$$QAH@Z ; std::vector<int, std::allocat
\ allocator<int> >::push_back
lea   ecx, DWORD PTR _c$[ebp]
push ecx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T4[ebp], 3
lea   edx, DWORD PTR $T4[ebp]
push edx
lea   ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@@QAEX$$QAH@Z ; std::vector<int, std::allocat
\ allocator<int> >::push_back
lea   eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T3[ebp], 4
lea   ecx, DWORD PTR $T3[ebp]
push ecx
lea   ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@@QAEX$$QAH@Z ; std::vector<int, std::allocat
\ allocator<int> >::push_back
lea   edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
push 6
lea   ecx, DWORD PTR _c$[ebp]
call ?reserve@$vector@HV?$allocator@H@std@@@std@@@QAEXI@Z ; std::vector<int, std::allocator<
\ int> >::reserve
lea   eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T2[ebp], 5
lea   ecx, DWORD PTR $T2[ebp]
push ecx
lea   ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@@QAEX$$QAH@Z ; std::vector<int, std::allocat
\ allocator<int> >::push_back
lea   edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T1[ebp], 6
lea   eax, DWORD PTR $T1[ebp]
push eax
lea   ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@@QAEX$$QAH@Z ; std::vector<int, std::allocat
\ allocator<int> >::push_back
lea   ecx, DWORD PTR _c$[ebp]
```

### 3.19. C++

```
push ecx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
push 5
lea ecx, DWORD PTR _c$[ebp]
call ?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ; std::vector<int, std::allocator<int>>
↳ > ::at
mov edx, DWORD PTR [eax]
push edx
push OFFSET $SG52650 ; '%d'
call DWORD PTR __imp__printf
add esp, 8
mov eax, 8
shl eax, 2
mov ecx, DWORD PTR _c$[ebp]
mov edx, DWORD PTR [ecx+eax]
push edx
push OFFSET $SG52651 ; '%d'
call DWORD PTR __imp__printf
add esp, 8
lea ecx, DWORD PTR _c$[ebp]
call ?_Tidy@?$vector@HV?$allocator@H@std@@@std@@IAEXXZ ; std::vector<int, std::allocator<int>>
↳ > ::_Tidy
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP
```

We see how the `.at()` method checks the bounds and throws an exception in case of error. The number that the last `printf()` call prints is just taken from the memory, without any checks.

One may ask, why not use the variables like “size” and “capacity”, like it was done in `std::string`. Supposedly, this was done for faster bounds checking.

The code GCC generates is in general almost the same, but the `.at()` method is inlined:

Listing 3.118: GCC 4.8.1 -fno-inline-small-functions -O1

```
main proc near
push ebp
mov ebp, esp
push edi
push esi
push ebx
and esp, 0FFFFFFF0h
sub esp, 20h
mov dword ptr [esp+14h], 0
mov dword ptr [esp+18h], 0
mov dword ptr [esp+1Ch], 0
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 1
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(
↳ int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 2
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(
↳ int const&)
```

### 3.19. C++

```
lea    eax, [esp+14h]
mov    [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov    dword ptr [esp+10h], 3
lea    eax, [esp+10h]
mov    [esp+4], eax
lea    eax, [esp+14h]
mov    [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(<
    int const&)
lea    eax, [esp+14h]
mov    [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov    dword ptr [esp+10h], 4
lea    eax, [esp+10h]
mov    [esp+4], eax
lea    eax, [esp+14h]
mov    [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(<
    int const&)
lea    eax, [esp+14h]
mov    [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov    ebx, [esp+14h]
mov    eax, [esp+1Ch]
sub    eax, ebx
cmp    eax, 17h
ja    short loc_80001CF
mov    edi, [esp+18h]
sub    edi, ebx
sar    edi, 2
mov    dword ptr [esp], 18h
call _Znwj           ; operator new(uint)
mov    esi, eax
test   edi, edi
jz    short loc_80001AD
lea    eax, ds:0[edi*4]
mov    [esp+8], eax    ; n
mov    [esp+4], ebx    ; src
mov    [esp], esi      ; dest
call memmove

loc_80001AD: ; CODE XREF: main+F8
    mov    eax, [esp+14h]
    test   eax, eax
    jz    short loc_80001BD
    mov    [esp], eax      ; void *
    call _ZdlPv          ; operator delete(void *)

loc_80001BD: ; CODE XREF: main+117
    mov    [esp+14h], esi
    lea    eax, [esi+edi*4]
    mov    [esp+18h], eax
    add    esi, 18h
    mov    [esp+1Ch], esi

loc_80001CF: ; CODE XREF: main+DD
    lea    eax, [esp+14h]
    mov    [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov    dword ptr [esp+10h], 5
    lea    eax, [esp+10h]
    mov    [esp+4], eax
    lea    eax, [esp+14h]
    mov    [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(<
        int const&)
    lea    eax, [esp+14h]
    mov    [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
```

### 3.19. C++

```

    mov  dword ptr [esp+10h], 6
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(
        ↴ int const&
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  eax, [esp+14h]
    mov  edx, [esp+18h]
    sub  edx, eax
    cmp  edx, 17h
    ja   short loc_8000246
    mov  dword ptr [esp], offset aVector_m_range ; "vector::_M_range_check"
    call _ZSt20__throw_out_of_rangePKc ; std::__throw_out_of_range(char const*)

loc_8000246: ; CODE XREF: main+19C
    mov  eax, [eax+14h]
    mov  [esp+8], eax
    mov  dword ptr [esp+4], offset aD ; "%d\n"
    mov  dword ptr [esp], 1
    call __printf_chk
    mov  eax, [esp+14h]
    mov  eax, [eax+20h]
    mov  [esp+8], eax
    mov  dword ptr [esp+4], offset aD ; "%d\n"
    mov  dword ptr [esp], 1
    call __printf_chk
    mov  eax, [esp+14h]
    test eax, eax
    jz   short loc_80002AC
    mov  [esp], eax      ; void *
    call _ZdlPv          ; operator delete(void *)
    jmp short loc_80002AC

    mov  ebx, eax
    mov  edx, [esp+14h]
    test edx, edx
    jz   short loc_80002A4
    mov  [esp], edx      ; void *
    call _ZdlPv          ; operator delete(void *)

loc_80002A4: ; CODE XREF: main+1FE
    mov  [esp], ebx
    call _Unwind_Resume

loc_80002AC: ; CODE XREF: main+1EA
    ; main+1F4
    mov  eax, 0
    lea  esp, [ebp-0Ch]
    pop  ebx
    pop  esi
    pop  edi
    pop  ebp

locret_80002B8: ; DATA XREF: .eh_frame:08000510
    ; .eh_frame:080005BC
    retn
main endp

```

`.reserve()` is inlined as well. It calls `new()` if the buffer is too small for the new size, calls `memmove()` to copy the contents of the buffer, and calls `delete()` to free the old buffer.

Let's also see what the compiled program outputs if compiled with GCC:

```
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
```

### 3.19. C++

```
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0x8257028, _Mylast=0x8257034, _Myend=0x8257038
size=3, capacity=4
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0x8257028, _Mylast=0x8257038, _Myend=0x8257038
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257050, _Myend=0x8257058
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0x8257040, _Mylast=0x8257054, _Myend=0x8257058
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
0
```

We can spot that the buffer size grows in a different way than in MSVC.

Simple experimentation shows that in MSVC's implementation the buffer grows by ~50% each time it needs to be enlarged, while GCC's code enlarges it by 100% each time, i.e., doubles it.

## **std::map and std::set**

The binary tree is another fundamental data structure.

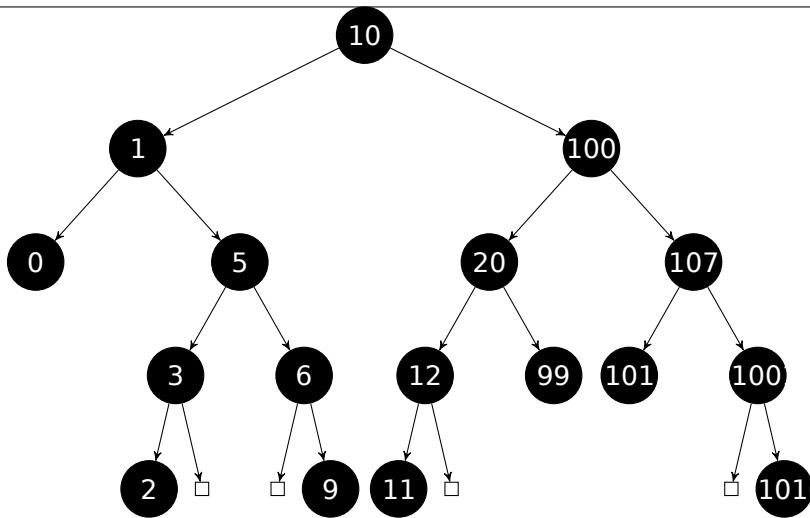
As its name states, this is a tree where each node has at most 2 links to other nodes. Each node has key and/or value: `std::set` provides only key at each node, `std::map` provides both key and value at each node.

Binary trees are usually the structure used in the implementation of "dictionaries" of key-values (AKA "associative arrays").

There are at least three important properties that a binary tree has:

- All keys are always stored in sorted form.
- Keys of any types can be stored easily. Binary tree algorithms are unaware of the key's type, only a key comparison function is required.
- Finding a specific key is relatively fast in comparison with lists and arrays.

Here is a very simple example: let's store these numbers in a binary tree: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.



All keys that are smaller than the node key's value are stored on the left side.

All keys that are bigger than the node key's value are stored on the right side.

Hence, the lookup algorithm is straightforward: if the value that you are looking for is smaller than the current node's key value: move left, if it is bigger: move right, stop if the value required is equal to the node key's value.

That is why the searching algorithm may search for numbers, text strings, etc., as long as a key comparison function is provided.

All keys have unique values.

Having that, one needs  $\approx \log_2 n$  steps in order to find a key in a balanced binary tree with  $n$  keys. This implies that  $\approx 10$  steps are needed  $\approx 1000$  keys, or  $\approx 13$  steps for  $\approx 10000$  keys.

Not bad, but the tree has always to be balanced for this: i.e., the keys has to be distributed evenly on all levels. The insertion and removal operations do some maintenance to keep the tree in a balanced state.

There are several popular balancing algorithms available, including the AVL tree and the red-black tree.

The latter extends each node with a “color” value to simplify the balancing process, hence, each node may be “red” or “black”.

Both GCC’s and MSVC’s `std::map` and `std::set` template implementations use red-black trees.

`std::set` has only keys. `std::map` is the “extended” version of `std::set`: it also has a value at each node.

## MSVC

```

#include <map>
#include <set>
#include <string>
#include <iostream>

// Structure is not packed! Each field occupies 4 bytes.
struct tree_node
{
    struct tree_node *Left;
    struct tree_node *Parent;
    struct tree_node *Right;
    char Color; // 0 - Red, 1 - Black
    char Isnil;
    //std::pair Myval;
    unsigned int first; // called Myval in std::set
    const char *second; // not present in std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
}
  
```

### 3.19. C++

### 3.19. C++

```
m[107]="one hundred seven";
m[0]="zero";
m[1]="one";
m[6]="six";
m[99]="ninety-nine";
m[5]="five";
m[11]="eleven";
m[1001]="one thousand one";
m[1010]="one thousand ten";
m[2]="two";
m[9]="nine";
printf ("dumping m as map:\n");
dump_map_and_set ((struct tree_struct *)(void*)&m, false);

std::map<int, const char*>::iterator it1=m.begin();
printf ("m.begin():\n");
dump_tree_node ((struct tree_node **)(void**)&it1, false, false);
it1=m.end();
printf ("m.end():\n");
dump_tree_node ((struct tree_node **)(void**)&it1, false, false);

// set

std::set<int> s;
s.insert(123);
s.insert(456);
s.insert(11);
s.insert(12);
s.insert(100);
s.insert(1001);
printf ("dumping s as set:\n");
dump_map_and_set ((struct tree_struct *)(void*)&s, true);
std::set<int>::iterator it2=s.begin();
printf ("s.begin():\n");
dump_tree_node ((struct tree_node **)(void**)&it2, true, false);
it2=s.end();
printf ("s.end():\n");
dump_tree_node ((struct tree_node **)(void**)&it2, true, false);
};
```

Listing 3.119: MSVC 2012

```
dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0x005BB440 Color=1 Isnil=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB520 Color=1 Isnil=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0x005BB4E0 Color=0 Isnil=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0x005BB3A0 Color=1 Isnil=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0x005BB3A0 Color=0 Isnil=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0x005BB5C0 Color=1 Isnil=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0x005BB3A0 Color=0 Isnil=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0x005BB480 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0x005BB500 Color=0 Isnil=0
first=20 second=[twenty]
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0x005BB3A0 Color=0 Isnil=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
```

### 3.19. C++

```
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0x005BB560 Color=0 Isnil=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB3A0 Color=1 Isnil=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB580 Color=1 Isnil=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0x005BB3A0 Color=0 Isnil=0
first=1010 second=[one thousand ten]
As a tree:
root----10 [ten]
    L-----1 [one]
        L-----0 [zero]
        R-----5 [five]
            L-----3 [three]
                L-----2 [two]
            R-----6 [six]
                R-----9 [nine]
    R-----100 [one hundred]
        L-----20 [twenty]
            L-----12 [twelve]
                L-----11 [eleven]
            R-----99 [ninety-nine]
        R-----107 [one hundred seven]
            L-----101 [one hundred one]
            R-----1001 [one thousand one]
                R-----1010 [one thousand ten]
m.begin():
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
m.end():
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1

dumping s as set:
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0x005BB620 Color=1 Isnil=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB680 Color=1 Isnil=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0x005BB5E0 Color=0 Isnil=0
first=1001
As a tree:
root----123
    L-----12
        L-----11
        R-----100
    R-----456
        R-----1001
s.begin():
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=1
```

The structure is not packed, so both *char* values occupy 4 bytes each.

As for `std::map`, `first` and `second` can be viewed as a single value of type `std::pair`. `std::set` has only one value at this address in the structure instead.

The current size of the tree is always present, as in the case of the implementation of `std::list` in MSVC ([3.19.4 on page 575](#)).

As in the case of `std::list`, the iterators are just pointers to nodes. The `.begin()` iterator points to

### 3.19. C++

the minimal key.

That pointer is not stored anywhere (as in lists), the minimal key of the tree is looked up every time. `operator--` and `operator++` move the current node pointer to the predecessor or successor respectively, i.e., the nodes which have the previous or next key.

The algorithms for all these operations are explained in [Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, *Introduction to Algorithms, Third Edition*, (2009)].

The `.end()` iterator points to the dummy node, it has 1 in `Isnil`, which implies that the node has no key and/or value. It can be viewed as a “landing zone” in HDD<sup>33</sup> and often called *sentinel* [see N. Wirth, *Algorithms and Data Structures*, 1985] <sup>34</sup>.

The “parent” field of the dummy node points to the root node, which serves as a vertex of the tree and contains information.

## GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse, bool dump_keys_and_values)
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p M_color=%d\n",
           n, n->M_left, n->M_parent, n->M_right, n->M_color);

    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (dump_keys_and_values)
    {
        if (is_set)
            printf ("key=%d\n", *(int*)point_after_struct);
        else
        {
            struct map_pair *p=(struct map_pair *)point_after_struct;
            printf ("key=%d value=[%s]\n", p->key, p->value);
        };
    };

    if (traverse==false)
        return;
}
```

<sup>33</sup>Hard disk drive

<sup>34</sup><http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf>



### 3.19. C++

```

std::map<int, const char*>::iterator it1=m.begin();
printf ("m.begin():\n");
dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, true);
it1=m.end();
printf ("m.end():\n");
dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, false);

// set

std::set<int> s;
s.insert(123);
s.insert(456);
s.insert(11);
s.insert(12);
s.insert(100);
s.insert(1001);
printf ("dumping s as set:\n");
dump_map_and_set ((struct tree_struct *)(&s), true);
std::set<int>::iterator it2=s.begin();
printf ("s.begin():\n");
dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, true);
it2=s.end();
printf ("s.end():\n");
dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, false);
};

```

Listing 3.120: GCC 4.8.1

```

dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, M_node_count=17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0x007A4B80 M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4C60 M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0x007A4C20 M_color=0
key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0x00000000 M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0x00000000 M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0x007A4D00 M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0x00000000 M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0x007A4BC0 M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0x007A4C40 M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0x00000000 M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0x007A4CA0 M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x00000000 M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x007A4CC0 M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0x00000000 M_color=0
key=1010 value=[one thousand ten]
As a tree:
root----10 [ten]
    L-----1 [one]
        L-----0 [zero]
        R-----5 [five]
            L-----3 [three]

```

```

L-----2 [two]
R-----6 [six]
R-----9 [nine]
R-----100 [one hundred]
L-----20 [twenty]
L-----12 [twelve]
L-----11 [eleven]
R-----99 [ninety-nine]
R-----107 [one hundred seven]
L-----101 [one hundred one]
R-----1001 [one thousand one]
R-----1010 [one thousand ten]

m.begin():
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4CC0 M_color=0

dumping s as set:
ptr=0x0028FE20, M_key_compare=0x8, M_header=0x0028FE24, M_node_count=6
ptr=0x007A1E80 M_left=0x01D5D890 M_parent=0x0028FE24 M_right=0x01D5D850 M_color=1
key=123
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8B0 M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0x00000000 M_color=0
key=1001
As a tree:
root----123
    L-----12
        L-----11
        R-----100
    R-----456
        R-----1001

s.begin():
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=0

```

GCC's implementation is very similar <sup>35</sup>. The only difference is the absence of the `Isnil` field, so the structure occupies slightly less space in memory than its implementation in MSVC.

The dummy node is also used as a place to point the `.end()` iterator also has no key and/or value.

## Rebalancing demo (GCC)

Here is also a demo showing us how a tree is rebalanced after some insertions.

Listing 3.121: GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
```

<sup>35</sup><http://go.yurichev.com/17084>



```

123, 456 has been inserted
root----123
    R-----456

11, 12 has been inserted
root----123
    L-----11
        R-----12
    R-----456

100, 1001 has been inserted
root----123
    L-----12
        L-----11
        R-----100
    R-----456
        R-----1001

667, 1, 4, 7 has been inserted
root----12
    L-----4
        L-----1
        R-----11
            L-----7
    R-----123
        L-----100
        R-----667
            L-----456
            R-----1001

```

### 3.19.5 Memory

Sometimes you may hear from C++ programmers “allocate memory on stack” and/or “allocate memory on [heap](#)”.

Allocating object *on stack*:

```

void f()
{
    ...
    Class o=Class(...);

    ...
}

```

The memory for object (or structure) is allocated in stack, using simple [SP](#) shift. The memory is deallocated upon function exit, or, more precisely, at the end of *scope*—[SP](#) is returning to its state (same as at the start of function) and destructor of *Class* is called. In the same manner, memory for allocated structure in C is deallocated upon function exit.

Allocating object *on heap*:

```

void f1()
{
    ...
    Class *o=new Class(...);

    ...
};

void f2()
{
    ...
    delete o;
}

```

```
}; ...
```

This is the same as allocating memory for a structure using `malloc()` call. In fact, `new` in C++ is wrapper for `malloc()`, and `delete` is wrapper for `free()`. Since memory block has been allocated in `heap`, it must be deallocated explicitly, using `delete`. Class destructor will be automatically called right before that moment.

Which method is better? Allocating *on stack* is very fast, and good for small, short-lived object, which will be used only in the current function.

Allocating *on heap* is slower, and better for long-lived object, which will be used across many functions. Also, objects allocated in `heap` are prone to memory leakage, because they must to be freed explicitly, but one can forget about it.

Anyway, this is matter of taste.

## 3.20 Negative array indices

It's possible to address the space *before* an array by supplying a negative index, e.g., `array[-1]`.

### 3.20.1 Addressing string from the end

Python `PL` allows to address arrays and strings from the end. For example, `string[-1]` returns the last character, `string[-2]` returns penultimate, etc. Hard to believe, but this is also possible in C/C++:

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

int main()
{
    char *s="Hello, world!";
    char *s_end=s+strlen(s);

    printf ("last character: %c\n", s_end[-1]);
    printf ("penultimate character: %c\n", s_end[-2]);
}
```

It works, but `s_end` must always has an address of terminating zero byte at the end of `s` string. If `s` string's size get changed, `s_end` must be updated.

The trick is dubious, but again, this is a demonstration of negative indices.

### 3.20.2 Addressing some kind of block from the end

Let's first recall why stack grows backwards ( [1.7.1 on page 31](#)). There is some kind of block in memory and you want to store both heap and stack there, and you are not sure, how big they both can grow during runtime.

You can set a `heap` pointer to the beginning of the block, then you can set a `stack` pointer to the end of the block (`heap + size_of_block`), and then you can address *n*th element of stack like `stack[-n]`. For example, `stack[-1]` for 1st element, `stack[-2]` for 2nd, etc.

This will work in the same fashion, as our trick of addressing string from the end.

You can easily check if the structures has not begun to overlap each other: just be sure that address of the last element in `heap` is below the address of the last element of `stack`.

Unfortunately, `-0` as index will not work, since two's complement way of representing negative numbers ( [2.2 on page 454](#)) don't allow negative zero, so it cannot be distinguished from positive zero.

This method is also mentioned in "Transaction processing", Jim Gray, 1993, "The Tuple-Oriented File System" chapter, p. 755.

### 3.20.3 Arrays started at 1

Fortran and Mathematica defined first element of array as 1th, probably because this is tradition in mathematics. Other PLs like C/C++ defined it as 0th. Which is better? Edsger W. Dijkstra argued that latter is better<sup>36</sup>.

But programmers may still have a habit after Fortran, so using this little trick, it's possible to address the first element in C/C++ using index 1:

```
#include <stdio.h>

int main()
{
    int random_value=0x11223344;
    unsigned char array[10];
    int i;
    unsigned char *fakearray=&array[-1];

    for (i=0; i<10; i++)
        array[i]=i;

    printf ("first element %d\n", fakearray[1]);
    printf ("second element %d\n", fakearray[2]);
    printf ("last element %d\n", fakearray[10]);

    printf ("array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X\n",
            array[-1],
            array[-2],
            array[-3],
            array[-4]);
}
```

Listing 3.123: Non-optimizing MSVC 2010

```

1 $SG2751 DB      'first element %d', 0aH, 00H
2 $SG2752 DB      'second element %d', 0aH, 00H
3 $SG2753 DB      'last element %d', 0aH, 00H
4 $SG2754 DB      'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]' 
5     DB      ']=%02X', 0aH, 00H
6
7 _fakearray$ = -24           ; size = 4
8 _random_value$ = -20       ; size = 4
9 _array$ = -16              ; size = 10
10 _i$ = -4                 ; size = 4
11 _main    PROC
12     push    ebp
13     mov     ebp, esp
14     sub     esp, 24
15     mov     DWORD PTR _random_value$[ebp], 287454020 ; 11223344H
16     ; set fakearray[] one byte earlier before array[]
17     lea     eax, DWORD PTR _array$[ebp]
18     add     eax, -1 ; eax=eax-1
19     mov     DWORD PTR _fakearray$[ebp], eax
20     mov     DWORD PTR _i$[ebp], 0
21     jmp     SHORT $LN3@main
22     ; fill array[] with 0..9
23 $LN2@main:
24     mov     ecx, DWORD PTR _i$[ebp]
25     add     ecx, 1
26     mov     DWORD PTR _i$[ebp], ecx
27 $LN3@main:
28     cmp     DWORD PTR _i$[ebp], 10
29     jge     SHORT $LN1@main
30     mov     edx, DWORD PTR _i$[ebp]
31     mov     al, BYTE PTR _i$[ebp]
32     mov     BYTE PTR _array$[ebp+edx], al
33     jmp     SHORT $LN2@main
34 $LN1@main:
```

<sup>36</sup>See <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html>

### 3.20. NEGATIVE ARRAY INDICES

```

35      mov     ecx, DWORD PTR _fakearray$[ebp]
36      ; ecx=address of fakearray[0], ecx+1 is fakearray[1] or array[0]
37      movzx   edx, BYTE PTR [ecx+1]
38      push    edx
39      push    OFFSET $SG2751 ; 'first element %d'
40      call    _printf
41      add    esp, 8
42      mov     eax, DWORD PTR _fakearray$[ebp]
43      ; eax=address of fakearray[0], eax+2 is fakearray[2] or array[1]
44      movzx   ecx, BYTE PTR [eax+2]
45      push    ecx
46      push    OFFSET $SG2752 ; 'second element %d'
47      call    _printf
48      add    esp, 8
49      mov     edx, DWORD PTR _fakearray$[ebp]
50      ; edx=address of fakearray[0], edx+10 is fakearray[10] or array[9]
51      movzx   eax, BYTE PTR [edx+10]
52      push    eax
53      push    OFFSET $SG2753 ; 'last element %d'
54      call    _printf
55      add    esp, 8
56      ; subtract 4, 3, 2 and 1 from pointer to array[0] in order to find values before array
57      ↵ []
58      lea     ecx, DWORD PTR _array$[ebp]
59      movzx   edx, BYTE PTR [ecx-4]
60      push    edx
61      lea     eax, DWORD PTR _array$[ebp]
62      movzx   ecx, BYTE PTR [eax-3]
63      push    ecx
64      lea     edx, DWORD PTR _array$[ebp]
65      movzx   eax, BYTE PTR [edx-2]
66      push    eax
67      lea     ecx, DWORD PTR _array$[ebp]
68      movzx   edx, BYTE PTR [ecx-1]
69      push    edx
70      push    OFFSET $SG2754 ; 'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X'
71      ↵ X'
72      call    _printf
73      add    esp, 20
74      xor    eax, eax
75      mov    esp, ebp
76      pop    ebp
    ret    0
_main  ENDP

```

So we have `array[]` of ten elements, filled with 0...9 bytes.

Then we have the `fakearray[]` pointer, which points one byte before `array[]`.

`fakearray[1]` points exactly to `array[0]`. But we are still curious, what is there before `array[]`? We have added `random_value` before `array[]` and set it to `0x11223344`. The non-optimizing compiler allocated the variables in the order they were declared, so yes, the 32-bit `random_value` is right before the array.

We ran it, and:

```

first element 0
second element 1
last element 9
array[-1]=11, array[-2]=22, array[-3]=33, array[-4]=44

```

Here is the stack fragment we will copypaste from OllyDbg's stack window (with comments added by the author):

Listing 3.124: Non-optimizing MSVC 2010

CPU Stack
Address Value
001DFBCC /001DFBD3 ; fakearray pointer

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
001DFBD0 | 11223344 ; random_value
001DFBD4 | 03020100 ; 4 bytes of array[]
001DFBD8 | 07060504 ; 4 bytes of array[]
001DFBDC | 00CB0908 ; random garbage + 2 last bytes of array[]
001DFBE0 | 0000000A ; last i value after loop was finished
001DFBE4 | 001DFC2C ; saved EBP value
001DFBE8 \00CB129D ; Return Address
```

The pointer to the `fakearray[]` (`0x001DFBD3`) is indeed the address of `array[]` in the stack (`0x001DFBD4`), but minus 1 byte.

It's still very hackish and dubious trick. Doubtfully anyone should use it in production code, but as a demonstration, it fits perfectly here.

## 3.21 Packing 12-bit values into array using bit operations (x64, ARM/ARM64, MIPS)

(This part has been first appeared in my blog at 4-Sep-2015.)

### 3.21.1 Introduction

File Allocation Table (FAT) was a widely popular filesystem. Hard to believe, but it's still used on flash drives, perhaps, for the reason of simplicity and compatibility. The FAT table itself is array of elements, each of which points to the next cluster number of a file (FAT supports files scattered across the whole disk). That implies that maximum of each element is maximum number of clusters on the disk. In MS-DOS era, most hard disks has FAT16 filesystem, because cluster number could be packed into 16-bit value. Hard disks then become cheaper, and FAT32 emerged, where 32-bit value was allocated for cluster number. But there were also a times, when floppy diskettes were not that cheap and has no much space, so FAT12 were used on them, for the reason of packing all filesystem structures as tight as possible.

So the FAT table in FAT12 filesystem is an array where each two subsequent 12-bit elements are stored into 3 bytes (triplet). Here is how 6 12-bit values (AAA, BBB, CCC, DDD, EEE and FFF) are packed into 9 bytes:

```
+0 +1 +2 +3 +4 +5 +6 +7 +8
|AA|AB|BB|CC|CD|DD|EE|EF|FF|...
```

Pushing values into array and pulling them back can be good example of bit twiddling operations (in both C/C++ and low-level machine code), so that's why I'll use FAT12 as an example here.

### 3.21.2 Data structure

We can quickly observe that each byte triplet will store 2 12-bit values: the first one is located at the left side, second one is at right:

```
+0 +1 +2
|11|12|22|...
```

We will pack nibbles (4 bit chunks) in the following way (1 - highest nibble, 3 - lowest):

(Even)

```
+0 +1 +2
|12|3.|...|...
```

(Odd)

```
+0 +1 +2
|...|.1|23|...
```

### 3.21.3 The algorithm

So the algorithm can be as follows: if the element's index is even, put it at left side, if the index is odd, place it at right side. The middle byte: if the element's index is even, place part of it in high 4 bits, if it's odd, place its part in low 4 bits. But first, find the right triplet, this is easy:  $\frac{\text{index}}{2}$ . Finding the address of right byte in array of bytes is also easy:  $\frac{\text{index}}{2} \cdot 3$  or  $\text{index} \cdot \frac{3}{2}$  or just  $\text{index} \cdot 1.5$ .

Pulling values from array: if index is even, get leftmost and middle bytes and combine its parts. If index is odd, get middle and rightmost bytes and combine them. Do not forget to isolate unneeded bits in middle byte.

Pushing values is almost the same, but be careful not to overwrite some *other's* bits in the middle byte, correcting only *yours*.

### 3.21.4 The C/C++ code

```
#include <stdio.h>
#include <stdint.h>
#include <assert.h>

#define ARRAY_SIZE (0x1000/2*3)
uint8_t array[ARRAY_SIZE]; // big enough array of triplets

unsigned int get_from_array (unsigned int idx)
{
    // find right triple in array:
    int triple=(idx>>1);
    int array_idx=triple*3;
    //assert (array_idx<ARRAY_SIZE);

    if (idx&1)
    {
        // this is odd element

        // compose value using middle and rightmost bytes:
        return ((array[array_idx+1]&0xF) << 8)|array[array_idx+2];
    }
    else
    {
        // this is even element

        // compose value using rightmost and middle bytes:
        return array[array_idx]<<4 | ((array[array_idx+1]>>4)&0xF);
    };
}

void put_to_array (unsigned int idx, unsigned int val)
{
    //assert (val<=0xFFFF);

    // find right triple in array:
    int triple=(idx>>1);
    int array_idx=triple*3;
    //assert (array_idx<ARRAY_SIZE);

    if (idx&1)
    {
        // this is odd element
        // put value into middle and rightmost bytes:

        // decompose value to be stored:
        uint8_t val_lowest_byte=val&0xFF; // isolate lowest 8 bits
        uint8_t val_highest_nibble=val>>8; // no need to apply &0xF, we already know ↴
        ↴ the val<=0xFFFF

        // clear low 4 bits in the middle byte:
        array[array_idx+1]=array[array_idx+1]&0xF0;
    }
}
```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
        array[array_idx+1]=array[array_idx+1]|val_highest_nibble;
        array[array_idx+2]=val_lowest_byte;
    }
else
{
    // this is even element
    // put value into leftmost and middle bytes:

    // decompose value to be stored:
    uint8_t val_highest_byte=val>>4;
    uint8_t val_lowest_nibble=val&0xF;

    array[array_idx]=val_highest_byte;

    // clear high 4 bits in the middle byte:
    array[array_idx+1]=array[array_idx+1]&0xF;
    array[array_idx+1]=array[array_idx+1]|val_lowest_nibble<<4;
}
};

int main()
{
    int i;

    // test
    for (i=0; i<0x1000; i++)
    {
        put_to_array(i, i);
    }

    for (i=0; i<0x1000; i++)
    {
        assert(get_from_array(i)==i);
    };
    //put_to_array(0x1000, 1); // will fail due to assert()

    // print triples:
    for (int i=0;i<0x1000/2;i++)
        printf ("0x%02X%02X%02X\n",array[i*3],array[i*3+1],array[i*3+2]);
};
```

During test, all 12-bit elements are filled with values in 0..0xFF range. And here is a dump of all triplets, each line has 3 bytes:

```
0x000001
0x002003
0x004005
0x006007
0x008009
0x00A00B
0x00C00D
0x00E00F
0x010011
0x012013
0x014015

...
0xFECFED
0xFEEFEF
0xFF0FF1
0xFF2FF3
0xFF4FF5
0xFF6FF7
0xFF8FF9
0xFFFFFB
0xFFFFFD
0xFFFFFF
```

Here is also GDB byte-level output of 300 bytes (or 100 triplets) started at 512/2\*3, i.e., it's address where

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

512th element (0x200) is beginning. I added square brackets in my text editor to show triplets explicitly. Take a notice at the middle bytes, where the last element is ended and the next is started. In other words, each middle byte has lowest 4 bits of even element and highest 4 bits of odd element.

```
(gdb) x/300xb array+512/2*3
0x601380 <array+768>: [ 0x20      0x02      0x01 ][ 0x20      0x22      0x03 ][ 0x20      0x42
0x601388 <array+776>: [ 0x05 ][ 0x20      0x62      0x07 ][ 0x20      0x82      0x09 ][ 0x20
0x601390 <array+784>: [ 0xa2      0x0b ][ 0x20      0xc2      0x0d ][ 0x20      0xe2      0x0f ]
0x601398 <array+792>: [ 0x21      0x02      0x11 ][ 0x21      0x22      0x13 ][ 0x21      0x42
0x6013a0 <array+800>: [ 0x15 ][ 0x21      0x62      0x17 ][ 0x21      0x82      0x19 ][ 0x21
0x6013a8 <array+808>: [ 0xa2      0x1b ][ 0x21      0xc2      0x1d ][ 0x21      0xe2      0x1f ]
0x6013b0 <array+816>: [ 0x22      0x02      0x21 ][ 0x22      0x22      0x23 ][ 0x22      0x42
0x6013b8 <array+824>: [ 0x25 ][ 0x22      0x62      0x27 ][ 0x22      0x82      0x29 ][ 0x22
0x6013c0 <array+832>: [ 0xa2      0x2b ][ 0x22      0xc2      0x2d ][ 0x22      0xe2      0x2f ]
0x6013c8 <array+840>: [ 0x23      0x02      0x31 ][ 0x23      0x22      0x33 ][ 0x23      0x42
0x6013d0 <array+848>: [ 0x35 ][ 0x23      0x62      0x37 ][ 0x23      0x82      0x39 ][ 0x23
0x6013d8 <array+856>: [ 0xa2      0x3b ][ 0x23      0xc2      0x3d ][ 0x23      0xe2      0x3f ]
0x6013e0 <array+864>: [ 0x24      0x02      0x41 ][ 0x24      0x22      0x43 ][ 0x24      0x42
0x6013e8 <array+872>: [ 0x45 ][ 0x24      0x62      0x47 ][ 0x24      0x82      0x49 ][ 0x24
0x6013f0 <array+880>: [ 0xa2      0x4b ][ 0x24      0xc2      0x4d ][ 0x24      0xe2      0x4f ]
0x6013f8 <array+888>: [ 0x25      0x02      0x51 ][ 0x25      0x22      0x53 ][ 0x25      0x42
0x601400 <array+896>: [ 0x55 ][ 0x25      0x62      0x57 ][ 0x25      0x82      0x59 ][ 0x25
0x601408 <array+904>: [ 0xa2      0x5b ][ 0x25      0xc2      0x5d ][ 0x25      0xe2      0x5f ]
0x601410 <array+912>: [ 0x26      0x02      0x61 ][ 0x26      0x22      0x63 ][ 0x26      0x42
0x601418 <array+920>: [ 0x65 ][ 0x26      0x62      0x67 ][ 0x26      0x82      0x69 ][ 0x26
0x601420 <array+928>: [ 0xa2      0x6b ][ 0x26      0xc2      0x6d ][ 0x26      0xe2      0x6f ]
0x601428 <array+936>: [ 0x27      0x02      0x71 ][ 0x27      0x22      0x73 ][ 0x27      0x42
0x601430 <array+944>: [ 0x75 ][ 0x27      0x62      0x77 ][ 0x27      0x82      0x79 ][ 0x27
0x601438 <array+952>: [ 0xa2      0x7b ][ 0x27      0xc2      0x7d ][ 0x27      0xe2      0x7f ]
0x601440 <array+960>: [ 0x28      0x02      0x81 ][ 0x28      0x22      0x83 ][ 0x28      0x42
0x601448 <array+968>: [ 0x85 ][ 0x28      0x62      0x87 ][ 0x28      0x82      0x89 ][ 0x28
0x601450 <array+976>: [ 0xa2      0x8b ][ 0x28      0xc2      0x8d ][ 0x28      0xe2      0x8f ]
0x601458 <array+984>: [ 0x29      0x02      0x91 ][ 0x29      0x22      0x93 ][ 0x29      0x42
0x601460 <array+992>: [ 0x95 ][ 0x29      0x62      0x97 ][ 0x29      0x82      0x99 ][ 0x29
0x601468 <array+1000>: [ 0xa2      0x9b ][ 0x29      0xc2      0x9d ][ 0x29      0xe2      0x9f ]
0x601470 <array+1008>: [ 0x2a      0x02      0xa1 ][ 0x2a      0x22      0xa3 ][ 0x2a      0x42
0x601478 <array+1016>: [ 0xa5 ][ 0x2a      0x62      0xa7 ][ 0x2a      0x82      0xa9 ][ 0x2a
0x601480 <array+1024>: [ 0xa2      0xab ][ 0x2a      0xc2      0xad ][ 0x2a      0xe2      0xaf ]
0x601488 <array+1032>: [ 0x2b      0x02      0xb1 ][ 0x2b      0x22      0xb3 ][ 0x2b      0x42
0x601490 <array+1040>: [ 0xb5 ][ 0x2b      0x62      0xb7 ][ 0x2b      0x82      0xb9 ][ 0x2b
0x601498 <array+1048>: [ 0xa2      0xbb ][ 0x2b      0xc2      0xbd ][ 0x2b      0xe2      0xbf ]
0x6014a0 <array+1056>: [ 0x2c      0x02      0xc1 ][ 0x2c      0x22      0xc3 ][ 0x2c      0x42
0x6014a8 <array+1064>: [ 0xc5 ][ 0x2c      0x62      0xc7 ]
```

#### 3.21.5 How it works

Let array be a global buffer to make simpler access to it.

##### Getter

Let's first start at the function getting values from the array, because it's simpler.

The method of finding triplet's number is just division input index by 2, but we can do it just by shifting right by 1 bit. This is a very common way of dividing/multiplication by numbers in form of  $2^n$ .

I can demonstrate how it works. Let's say, you want to divide 123 by 10. Just drop last digit (3, which is remainder of division) and 12 is left. Division by 2 is just dropping least significant bit. Dropping can be done by shifting right.

Now the functions must decide if the input index even (so 12-bit value is placed at the left) or odd (at the right). Simplest way to do so is to isolate lowest bit ( $x \& 1$ ). If it's zero, our value is even, otherwise it's odd.

This fact can be illustrated easily, take a look at the lowest bit:

decimal	binary	even/odd
---------	--------	----------

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
0    0000 even
1    0001 odd
2    0010 even
3    0011 odd
4    0100 even
5    0101 odd
6    0110 even
7    0111 odd
8    1000 even
9    1001 odd
10   1010 even
11   1011 odd
12   1100 even
13   1101 odd
14   1110 even
15   1111 odd
...
```

Zero is also even number, it's so in two's complement system, where it's located between two odd numbers (-1 and 1).

For math geeks, I could also say that even or odd sign is also remainder of division by 2. Division of a number by 2 is merely dropping the last bit, which is remainder of division. Well, we do not have to do shifts here, just isolate lowest bit.

If the element is odd, we take middle and right bytes (`array[array_idx+1]` and `array[array_idx+2]`). Lowest 4 bits of the middle byte is isolated. Right byte is taken as a whole. Now we have to combine these parts into 12-bit value. To do so, shift 4 bits from the middle byte by 8 bits left, so these 4 bits will be allocated right behind highest 8th bit of byte. Then, using OR operation, we just add these parts.

If the element is even, high 8 bits of 12-bit value is located in left byte, while lowest 4 bits are located in the high 4 bits of middle byte. We isolate highest 4 bits in the middle byte by shifting it 4 bits right and then applying AND operation, just to be sure that nothing is left there. We also take left byte and shift its value 4 bits left, because it has bits from 11th to 4th (inclusive, starting at 0), Using OR operation, we combine these two parts.

#### Setter

Setter calculates triplet's address in the same way. It also operates on left/right bytes in the same way. But it's not correct just to write to the middle byte, because write operation will destroy the information related to the other element. So the common way is to load byte, drop bits where you'll write, write it there, but leave other part intact. Using AND operation (`&` in C/C++), we drop everything except our part. Using OR operation (`|` in C/C++), we then update the middle byte.

#### 3.21.6 Optimizing GCC 4.8.2 for x86-64

Let's see what optimizing GCC 4.8.2 for Linux x64 will do. I added comments. Sometimes readers are confused because instructions order is not logical. It's OK, because optimizing compilers take CPU out-of-order-execution mechanisms into considerations, and sometimes, swapped instructions performing better.

#### Getter

```
get_from_array:
; EDI=idx
; make a copy:
    mov     eax, edi
; calculate idx>>1:
    shr     eax
; determine if the element even or odd by isolation of the lowest bit:
    and     edi, 1
; calculate (idx>>1)*3.
; multiplication is slow in general, and can be replaced by one shifting and addition operation
; LEA is capable to do both:
    lea     edx, [rax+rax*2]
```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
; EDX now is (idx>>1)*3
; point EAX to the middle byte:
    lea    eax, [rdx+1]
; sign-extend EAX value to RDX:
    cdqe
; get middle byte into EAX:
    movzx  eax, BYTE PTR array[rax]
; finally check the value of the lowest bit in index.
; jump if index is odd (NE is the same as NZ (Not Zero)):
    jne    .L9

; this is even element, go on
; sign-extend EDX to RDX:
    movsx  rdx, edx
; shift middle byte 4 bits right:
    shr    al, 4
; AL now has 4 highest bits of middle byte
; load left byte into EDX:
    movzx  edx, BYTE PTR array[rdx]
; sign-extend AL (where high 4 bits of middle byte are now)
    movzx  eax, al
; EAX has 4 high bits bits of middle byte
; EDX now has left byte
; shift left byte 4 bits left:
    sal    edx, 4
; 4 lowest bits in EDX after shifting is cleared to zero
; finally merge values:
    or     eax, edx
    ret

.L9:
; this is odd element, go on
; calculate address of right byte:
    add    edx, 2
; isolate lowest 4 bits in middle byte:
    and    eax, 15 ; 15=0xF
; sign-extend EDX (where address of right byte is) to RDX:
    movsx  rdx, edx
; shift 4 bits we got from middle bytes 8 bits left:
    sal    eax, 8
; load right byte:
    movzx  edx, BYTE PTR array[rdx]
; merge values:
    or     eax, edx
    ret
```

### Setter

```
put_to_array:
; EDI=idx
; ESI=val
; copy idx to EAX:
    mov    eax, edi
; calculate idx>>1 and put it to EAX:
    shr    eax
; isolate lowest bit in idx:
    and    edi, 1
; calculate (idx>>2)*3 and put it to EAX:
    lea    eax, [rax+rax*2]
; jump if index is odd (NE is the same as NZ (Not Zero)):
    jne    .L5

; this is even element, go on
; sign-extend address of triplet in EAX to RDX:
    movsx  rdx, eax
; copy val value to ECX:
    mov    ecx, esi
; calculate address of middle byte:
    add    eax, 1
```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
; sign-extend address in EAX to RDX:  
    cdqe  
; prepare left byte in ECX by shifting it:  
    shr    ecx, 4  
; prepare 4 bits for middle byte:  
    sal    esi, 4  
; put left byte:  
    mov    BYTE PTR array[rdx], cl  
; load middle byte (its address still in RAX):  
    movzx  edx, BYTE PTR array[rax]  
; drop high 4 bits:  
    and    edx, 15 ; 15=0xF  
; merge our data and low 4 bits which were there before:  
    or     esi, edx  
; put middle byte back:  
    mov    BYTE PTR array[rax], sil  
    ret  
.L5:  
; this is odd element, go on  
; calculate address of middle byte and put it to ECX:  
    lea    ecx, [rax+1]  
; copy val value from ESI to EDI:  
    mov    edi, esi  
; calculate address of right byte:  
    add    eax, 2  
; get high 4 bits of input value by shifting it 8 bits right:  
    shr    edi, 8  
; sign-extend address in EAX into RAX:  
    cdqe  
; sign-extend address of middle byte in ECX to RCX:  
    movsx  rcx, ecx  
; load middle byte into EDX:  
    movzx  edx, BYTE PTR array[rcx]  
; drop low 4 bits in middle byte:  
    and    edx, -16 ; -16=0xF0  
; merge data from input val with what was in middle byte before:  
    or     edx, edi  
; store middle byte:  
    mov    BYTE PTR array[rcx], dl  
; store right byte. val is still in ESI and SIL is a part of ESI register which has lowest 8 ↴  
  ↴ bits:  
    mov    BYTE PTR array[rax], sil  
    ret
```

### Other comments

All addresses in Linux x64 are 64-bit ones, so during pointer arithmetic, all values should also be 64-bit. The code calculating offsets inside of array operates on 32-bit values (input *idx* argument has type of *int*, which has width of 32 bits), and so these values must be converted to 64-bit addresses before actual memory load/store. So there are a lot of sign-extending instructions (like `CDQE`, `MOVZX`) used for conversion. Why to extend sign? By C/C++ standards, pointer arithmetic can operate on negative values (it's possible to access array using negative index like `array[-123]`, see: [3.20 on page 597](#)). Since GCC compiler cannot be sure if all indices are always positive, it adds sign-extending instructions.

### 3.21.7 Optimizing Keil 5.05 (Thumb mode)

#### Getter

The following code has final OR operation in the function epilogue. Indeed, it executes at the end of both branches, so it's possible to save some space.

```
get_from_array PROC  
; R0 = idx  
    PUSH    {r4,r5,lr}  
    LSRS    r1,r0,#1
```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```

; R1 = R0>>1 = idx>>1
; R1 is now number of triplet
    LSLS      r2,r1,#1
; R2 = R1<<1 = (R0>>1)<<1 = R0&(~1) = idx&(~1)
; the operation (x>>1)<<1 looks senseless, but it's intended to clear the lowest bit in x (or ↴
    ↴ idx)
    LSLS      r5,r0,#31
; R5 = R0<<31 = idx<<31
; thus, R5 will contain 0x80000000 in case of even idx or zero if odd
    ADDS      r4,r1,r2
; R4 = R1+R2 = idx>>1 + idx&(~1) = offset of triplet begin (or offset of left byte)
; the expression looks tricky, but it's equal to multiplication by 1.5
    LSRS      r0,r0,#1
; R0 = R0>>1 = idx>>1
; load pointer to array:
    LDR      r3,[array]
; R3 = offset of array table
    LSLS      r1,r0,#1
; R1 = R0<<1 = (idx>>1)<<1 = idx&(~1)
    ADDS      r0,r0,r1
; R0 = idx>>1 + idx&(~1) = idx*1.5 = offset of triple begin
    ADDS      r1,r3,r0
; R1 = R3+R0 = offset of array + idx*1.5
; in other words, R1 now contains absolute address of triplet
; load middle byte (at address R1+1):
    LDRB     r2,[r1,#1]
; R2 = middle byte
; finally check if the idx even or odd:
    CMP      r5,#0
; jump if even:
    BEQ      |L0.92|
; idx is odd, go on:
    LSLS      r0,r2,#28
; R0 = R2<<28 = middle_byte<<28
; load right byte at R1+2:
    LDRB     r1,[r1,#2]
; R1 = right byte
    LSRS      r0,r0,#20
; R0 = R0>>20 = (R2<<28)>>20
; this is the same as R2<<8, but Keil compiler generates more complex code in order to drop all ↴
    ↴ bits behind these 4
    B       |L0.98|
|L0.92|
; idx is even, go on:
; load left byte. R3=array now and R4=address of it
    LDRB     r0,[r3,r4]
; R0 = left byte
    LSLS      r0,r0,#4
; R0 = left_byte<<4
; shift middle_byte in R2 4 bits right:
    LSRS      r1,r2,#4
; R1=middle_byte>>4
|L0.98|
; function epilogue:
; current R0 value is shifted left byte or part of middle byte
; R1 is shifted part of middle byte or right byte
; now merge values and leave merged result in R0:
    ORRS      r0,r0,r1
; R0 = R0|R1
    POP      {r4,r5,pc}
    ENDP

```

There are at least of redundancy:  $idx*1.5$  is calculated twice. As an exercise, reader may try to rework assembly function to make it shorter. Do not forget about testing!

Another thing to mention is that it's hard to generate big constants in 16-bit Thumb instructions, so Keil compiler often generates tricky code using shifting instructions to achieve the same effect. For example, it's tricky to generate `AND Rdest, Rsrc, 1` or `TST Rsrc, 1` code in Thumb mode, so Keil generates the code which shifts input `idx` by 31 bits left and then check, if the resulting value zero or not.

**Setter**

The first half of setter code is very similar to getter, address of triplet is calculated first, then the jump is occurred in order to dispatch to the right handler's code.

```

put_to_array PROC
    PUSH    {r4,r5,lr}
; R0 = idx
; R1 = val
    LSRS    r2,r0,#1
; R2 = R0>>1 = idx>>1
    LSLS    r3,r2,#1
; R3 = R2<<1 = (idx>>1)<<1 = idx&(~1)
    LSLS    r4,r0,#31
; R4 = R0<<31 = idx<<31
    ADDS    r3,r2,r3
; R3 = R2+R3 = idx>>1 + idx&(~1) = idx*1.5
    LSRS    r0,r0,#1
; R0 = R0>>1 = idx>>1
    LDR     r2,[array]
; R2 = address of array
    LSLS    r5,r0,#1
; R5 = R0<<1 = (idx>>1)<<1 = idx&(~1)
    ADDS    r0,r0,r5
; R0 = R0+R5 = idx>>1 + idx&(~1) = idx*1.5
    ADDS    r0,r2,r0
; R0 = R2+R0 = array + idx*1.5, in other words, this is address of triplet
; finally test shifted lowest bit in idx:
    CMP     r4,#0
; jump if idx is even:
    BEQ     |L0.40|
; idx is odd, go on:
; load middle byte at R0+1:
    LDRB   r3,[r0,#1]
; R3 = middle_byte
    LSRS    r2,r1,#8
; R2 = R1>>8 = val>>8
    LSRS    r3,r3,#4
; R3 = R3>>4 = middle_byte>>4
    LSLS    r3,r3,#4
; R3 = R3<<4 = (middle_byte>>4)<<4
; this two shift operations are used to drop low 4 bits in middle_byte
; merge high 4 bits in middle byte (in R3) with val>>8 (in R2):
    ORRS    r3,r3,r2
; R3 = updated middle byte
; store it at R0+1:
    STRB   r3,[r0,#1]
; store low 8 bits of val (val&0xFF) at R0+2:
    STRB   r1,[r0,#2]
    POP    {r4,r5,pc}

|L0.40|
; idx is even, go on:
    LSRS    r4,r1,#4
; R4 = R1>>4 = val>>4
; store val>>4 at R2+R3 (address of left byte or beginning of triplet):
    STRB   r4,[r2,r3]
; load middle byte at R0+1:
    LDRB   r3,[r0,#1]
; R3 = middle byte
    LSLS    r2,r1,#4
; R2 = R1<<4 = val<<4
    LSLS    r1,r3,#28
; R1 = R3<<28 = middle_byte<<28
    LSRS    r1,r1,#28
; R1 = R1>>28 = (middle_byte<<28)>>28
; these two shifting operation are used to drop all bits in register except lowest 4
; merge lowest 4 bits (in R1) and val<<4 (in R2):
    ORRS    r1,r1,r2
; store it at R0+1:
    STRB   r1,[r0,#1]

```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
POP      {r4,r5,pc}
ENDP
```

## 3.21.8 Optimizing Keil 5.05 (ARM mode)

### Getter

Getter function for ARM mode has no conditional branches at all! Thanks to the suffixes (like `-EQ`, `-NE`), which can be supplied to many instructions in ARM mode, so the instruction will be only executed if the corresponding flag(s) are set.

Many arithmetical instructions in ARM mode can have shifting suffix like `LSL #1` (it means, the last operand is shifted left by 1 bit).

```
get_from_array PROC
; R0 = idx
    LSR      r1,r0,#1
; R1 = R0>>1 = idx>>1
; check lowest bit in idx and set flags:
    TST      r0,#1
    ADD      r2,r1,r1,LSL #1
; R2 = R1+R1<<1 = R1+R1*2 = R1*3
; thanks to shifting suffix, a single instruction in ARM mode can multiplicate by 3
    LDR      r1,[array]
; R1 = address of array
    LSR      r0,r0,#1
; R0 = R0>>1 = idx>>1
    ADD      r0,r0,r0,LSL #1
; R0 = R0+R0<<1 = R0+R0*2 = R0*3 = (idx>>1)*3 = idx*1.5
    ADD      r0,r0,r1
; R0 = R0+R1 = array + idx*1.5, this is absolute address of triplet
; load middle byte at R0+1:
    LDRB    r3,[r0,#1]
; R3 = middle byte

; the following 3 instructions executed if index is odd, otherwise all of them are skipped:
; load right byte at R0+2:
    LDRBNE   r0,[r0,#2]
; R0 = right byte
    ANDNE   r1,r3,#0xf
; R1 = R3&0xF = middle_byte&0xF
    ORRNE   r0,r0,r1,LSL #8
; R0 = R0|(R1<<8) = right_byte | (middle_byte&0xF)<<8
; this is the result returned

; the following 3 instructions executed if index is even, otherwise all of them are skipped:
; load at R1+R2 = array + (idx>>1)*3 = array + idx*1.5
    LDRBEQ   r0,[r1,r2]
; R0 = left byte
    LSLEQ   r0,r0,#4
; R0 = R0<<4 = left_byte << 4
    ORREQ   r0,r0,r3,LSR #4
; R0 = R0 | R3>>4 = left_byte << 4 | middle_byte >> 4
; this is the result returned
    BX      lr
ENDP
```

### Setter

```
put_to_array PROC
; R0 = idx
; R1 = val
    LSR      r2,r0,#1
; R2 = R0>>1 = idx>>1
; check the lowest bit of idx and set flags:
```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
TST      r0,#1
LDR      r12,[array]
; R12 = address of array
LSR      r0,r0,#1
; R0 = R0>>1 = idx>>1
ADD      r0,r0,r0,LSL #1
; R0 = R0+R0<<1 = R0+R0*2 = R0*3 = (idx>>1)*3 = idx/2*3 = idx*1.5
ADD      r3,r2,r2,LSL #1
; R3 = R2+R2<<1 = R2+R2*2 = R2*3 = (idx>>1)*3 = idx/2*3 = idx*1.5
ADD      r0,r0,r12
; R0 = R0+R12 = idx*1.5 + array
; jump if idx is even:
BEQ      |L0.56|
; idx is odd, go on:
; load middle byte at R0+1:
LDRB    r3,[r0,#1]
; R3 = middle byte
AND     r3,r3,#0xf0
; R3 = R3&0xF0 = middle_byte&0xF0
ORR     r2,r3,r1,LSR #8
; R2 = R3 | R1>>8 = middle_byte&0xF0 | val>>8
; store middle_byte&0xF0 | val>>8 at R0+1 (at the place of middle byte):
STRB    r2,[r0,#1]
; store low 8 bits of val (or val&0xFF) at R0+2 (at the place of right byte):
STRB    r1,[r0,#2]
BX     lr
|L0.56|
; idx is even, go on:
LSR      r2,r1,#4
; R2 = R1>>4 = val>>4
; store val>>4 at R12+R3 or array + idx*1.5 (place of left byte):
STRB    r2,[r12,r3]
; load byte at R0+1 (middle byte):
LDRB    r2,[r0,#1]
; R2 = middle_byte
; drop high 4 bits of middle byte:
AND     r2,r2,#0xf
; R2 = R2&0xF = middle_byte&0xF
; update middle byte:
ORR     r1,r2,r1,LSL #4
; R1 = R2 | R1<<4 = middle_byte&0xF | val<<4
; store updated middle byte at R0+1:
STRB    r1,[r0,#1]
BX     lr
ENDP
```

Value of  $idx*1.5$  is calculated twice, this is redundancy Keil compiler produced can be eliminated. You can rework assembly function as well to make it shorter. Do not forget about tests!

### 3.21.9 (32-bit ARM) Comparison of code density in Thumb and ARM modes

Thumb mode in ARM CPUs was introduced to make instructions shorter (16-bits) instead of 32-bit instructions in ARM mode. But as we can see, it's hard to say, if it was worth it: code in ARM mode is always shorter (however, instructions are longer).

### 3.21.10 Optimizing GCC 4.9.3 for ARM64

#### Getter

```
<get_from_array>:
; W0 = idx
0:          lsr    w2, w0, #1
; W2 = W0>>1 = idx>>1
4:          lsl    w1, w2, #2
; W1 = W2<<2 = (W0>>1)<<2 = (idx&(~1))<<1
```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```

8:           sub      w1, w1, w2
; W1 = W1-W2 = (idx&(~1))<<1 - idx>>1 = idx*1.5
; now test lowest bit of idx and jump if it is present.
; (ARM64 has single instruction for these operations: TBNZ (Test and Branch Not Zero)).
c:           tbnz    w0, #0, 30 <get_from_array+0x30>

; idx is even, go on:
10:          adrp     x2, page of array
14:          add      w3, w1, #0x1
; W3 = W1+1 = idx*1.5 + 1, i.e., offset of middle byte
18:          add      x2, x2, offset of array within page
; load left byte at X2+W1 = array + idx*1.5 with sign-extension ("sxtw")
1c:          ldrb    w0, [x2,w1,sxtw]
; load middle byte at X2+W3 = array + idx*1.5 + 1 with sign-extension ("sxtw")
20:          ldrb    w1, [x2,w3,sxtw]
; W0 = left byte
; W1 = middle byte
24:          lsl      w0, w0, #4
; W0 = W0<<4 = left_byte << 4
; merge parts:
28:          orr      w0, w0, w1, lsr #4
; W0 = W0 | W1>>4 = left_byte << 4 | middle_byte >> 4
; value in W0 is returned
2c:          ret

; idx is odd, go on:
30:          adrp     x2, page of array
34:          add      w0, w1, #0x1
; W0 = W1+1 = idx*1.5+1, i.e., offset of middle byte
38:          add      x2, x2, address of array within page
3c:          add      w1, w1, #0x2
; W1 = W1+2 = idx*1.5+2, i.e., offset of right byte
; load middle byte at X2+W0 = array+idx*1.5+1 with sign-extension ("sxtw")
40:          ldrb    w0, [x2,w0,sxtw]
; load right byte at X2+W1 = array+idx*1.5+2 with sign-extension ("sxtw")
44:          ldrb    w1, [x2,w1,sxtw]
; W0 = middle byte
; W1 = right byte
48:          ubfiz   w0, w0, #8, #4
; W0 = middle_byte<<8
; now merge parts:
4c:          orr      w0, w0, w1
; W0 = W0 | W1 = middle_byte<<8 | right_byte
; value in W0 is returned
50:          ret

```

ARM64 has new cool instruction **UBFIZ** (*Unsigned bitfield insert in zero, with zeros to left and right*), which can be used to place specified number of bits from one register to another. It's alias of another instruction, **UBFM** (*Unsigned bitfield move, with zeros to left and right*). **UBFM** is the instruction used internally in ARM64 instead of **LSL/LSR** (bit shifts).

### Setter

```

<put_to_array>:
W0 = idx
W1 = val
54:          lsr      w3, w0, #1
; W3 = W0>>1 = idx>>1
58:          lsl      w2, w3, #2
; W2 = W3<<2 = (W0>>1)<<2 = (idx&(~1))<<1
5c:          sub      w2, w2, w3
; W2 = W2-W3 = (idx&(~1))<<1 - idx>>1 = idx*1.5
; jump if lowest bit in idx is 1:
60:          tbnz    w0, #0, 94 <put_to_array+0x40>

; idx is even, go on:
64:          adrp     x0, page of array

```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```
68:          add      w3, w2, #0x1
; W3 = W2+1 = idx*1.5+1, i.e., offset of middle byte
  6c:          add      x0, x0, offset of array within page
; X0 = address of array
  70:          lsr      w4, w1, #4
; W4 = W1>>4 = val>>4
  74:          sxtw    x3, w3
; X3 = sign-extended 32-bit W3 (idx*1.5+1) to 64-bit
; sign-extension is needed here because the value will be used as offset within array,
; and negative offsets are possible in standard C/C++
  78:          ubfiz   w1, w1, #4, #4
; W1 = W1<<4 = val<<4
; store W4 (val>>4) at X0+W2 = array + idx*1.5, i.e., address of left byte:
  7c:          strb    w4, [x0,w2,sxtw]
; load middle byte at X0+X3 = array+idx*1.5+1
  80:          ldrb    w2, [x0,x3]
; W2 = middle byte
  84:          and     w2, w2, #0xf
; W2 = W2&0xF = middle_byte&0xF (high 4 bits in middle byte are dropped)
; merge parts of new version of middle byte:
  88:          orr     w1, w2, w1
; W1 = W2|W1 = middle_byte&0xF | val<<4
; store W2 (new middle byte) at X0+X3 = array+idx*1.5+1
  8c:          strb    w1, [x0,x3]
  90:          ret

; idx is odd, go on:
  94:          add      w4, w2, #0x1
; W4 = W2+1 = idx*1.5+1, i.e., offset of middle byte
  98:          adrp    x0, page of array
  9c:          add      x0, x0, offset of array within page
; X0 = address of array
 a0:          add      w2, w2, #0x2
; W2 = W2+2 = idx*1.5+2, i.e., offset of right byte
 a4:          sxtw    x4, w4
; X4 = sign-extended 64-bit version of 32-bit W4
; load at X0+X4 = array+idx*1.5+1:
 a8:          ldrb    w3, [x0,x4]
; W3 = middle byte
 ac:          and     w3, w3, #0xffffffff0
; W3 = W3&0xFFFFFFFF0 = middle_byte&0xFFFFFFFF0, i.e., clear lowest 4 bits
 b0:          orr     w3, w3, w1, lsr #8
; W3 = W3|W1>>8 = middle_byte&0xFFFFFFFF0 | val>>8
; store new version of middle byte at X0+X4=array+idx*1.5+1:
 b4:          strb    w3, [x0,x4]
; now store lowest 8 bits of val (in W1) at X0+W2=array+idx*1.5+2, i.e., place of right byte
; SXTW suffix means W2 will be sign-extended to 64-bit value before summing with X0
 b8:          strb    w1, [x0,w2,sxtw]
bc:          ret
```

### 3.21.11 Optimizing GCC 4.4.5 for MIPS

Needless to keep in mind that each instruction after jump/branch instruction is executed first. It's called *branch delay slot* in RISC CPUs lingo. To make things simpler, just swap instructions (mentally) in each instruction pair which is started with branch or jump instruction.

MIPS has no flags (apparently, to simplify data dependencies), so branch instructions (like `BNE`) does both comparison and branching.

There is also GP (Global Pointer) set up code in the function prologue, which can be ignored so far.

### Getter

```
get_from_array:
; $4 = idx
      srl    $2,$4,1
```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```

; $2 = $4>>1 = idx>>1
    lui      $28,%hi(__gnu_local_gp)
    sll      $3,$2,1
; $3 = $2<<1 = (idx>>1)<<1 = idx&(~1)
    andi    $4,$4,0x1
; $4 = $4&1 = idx&1
    addiu   $28,$28,%lo(__gnu_local_gp)

; jump if $4 (idx&1) is not zero (if idx is odd):
    bne    $4,$0,$L6
; $2 = $3+$2 = idx>>1 + idx&(~1) = idx*1.5
    addu   $2,$3,$2 ; branch delay slot - this instruction executed before BNE

; idx is even, go on:
    lw     $3,%got(array)($28)
; $3 = array
    nop
    addu   $2,$3,$2
; $2 = $3+$2 = array + idx*1.5
; load byte at $2+0 = array + idx*1.5 (left byte):
    lbu   $3,0($2)
; $3 = left byte
; load byte at $2+1 = array + idx*1.5+1 (middle byte):
    lbu   $2,1($2)
; $2 = middle byte
    sll   $3,$3,4
; $3 = $3<<4 = left_byte<<4
    srl   $2,$2,4
; $2 = $2>>4 = middle_byte>>4
    j     $31
    or    $2,$2,$3 ; branch delay slot - this instruction executed before J
; $2 = $2|$3 = middle_byte>>4 | left_byte<<4
; $2=returned result

$L6:
; idx is odd, go on:
    lw     $3,%got(array)($28)
; $3 = array
    nop
    addu   $2,$3,$2
; $2 = $3+$2 = array + idx*1.5
; load byte at $2+1 = array + idx*1.5 + 1 (middle byte)
    lbu   $4,1($2)
; $4 = middle byte
; load byte at $2+1 = array + idx*1.5 + 2 (right byte)
    lbu   $3,2($2)
; $3 = right byte
    andi   $2,$4,0xf
; $2 = $4&0xF = middle_byte&0xF
    sll   $2,$2,8
; $2 = $2<<8 = middle_byte&0xF << 8
    j     $31
    or    $2,$2,$3 ; branch delay slot - this instruction executed before J
; $2 = $2|$3 = middle_byte&0xF << 8 | right byte
; $2=returned result

```

### Setter

```

put_to_array:
; $4=idx
; $5=val
    srl   $2,$4,1
; $2 = $4>>1 = idx>>1
    lui      $28,%hi(__gnu_local_gp)
    sll      $3,$2,1
; $3 = $2<<1 = (idx>>1)<<1 = idx&(~1)
    andi    $4,$4,0x1
; $4 = $4&1 = idx&1

```

### 3.21. PACKING 12-BIT VALUES INTO ARRAY

```

        addiu   $28,$28,%lo(__gnu_local_gp)
; jump if $4=idx&1 is not zero (i.e., if idx is odd):
        bne    $4,$0,$L11
        addu   $2,$3,$2 ; branch delay slot, this instruction is executed before BNE
; $2 = $3+$2 = idx&(~1) + idx>>1 = idx*1.5

; idx is even, go on
        lw     $3,%got(array)($28)
; $3 = array
        addiu $4,$2,1
; $4 = $2+1 = idx*1.5+1, i.e., offset of middle byte in array
        srl   $6,$5,4
; $6 = $5>>4 = val>>4
        addu   $2,$3,$2
; $2 = $3+$2 = array + idx*1.5 (offset of left byte)
; store $6 (val>>4) as left byte:
        sb     $6,0($2)
        addu   $2,$3,$4
; $2 = $3+$4 = array + idx*1.5 + 1 (absolute address of middle byte)
; load middle byte at $2+0 = array + idx*1.5 + 1
        lbu   $3,0($2)
; $3 = middle byte
        andi   $5,$5,0xf
; $5 = $5&0xF = val&0xF
        andi   $3,$3,0xf
; $3 = $3&0xF = middle_byte&0xF
        sll   $5,$5,4
; $5 = $5<<4 = (val&0xF)<<4
        or     $5,$3,$5
; $5 = $3|$5 = middle_byte&0xF | (val&0xF)<<4 (new version of middle byte)
        j      $31
; store $5 (new middle byte) at $2 (array + idx*1.5 + 1)
        sb     $5,0($2) ; branch delay slot, this instruction is executed before J

$L11:
; idx is odd, go on
        lw     $4,%got(array)($28)
; $4 = array
        addiu $3,$2,1
; $3 = $2+1 = idx*1.5+1 (offset of middle byte)
        addu   $3,$4,$3
; $3 = $4+$3 = array + idx*1.5+1
; load middle byte at $3 (array + idx*1.5+1)
        lbu   $6,0($3)
; $6 = middle byte
        srl   $7,$5,8
; $7 = $5>>8 = val>>8
        andi   $6,$6,0xf0
; $6 = $6&0xF0 = middle_byte&0xF0
        or     $6,$6,$7
; $6 = $6|$7 = middle_byte&0xF0 | val>>8
        addu   $2,$4,$2
; store updated middle byte at $3 (array + idx*1.5+1)
        sb     $6,0($3)
        j      $31
; store low 8 bits of val at $2+2=idx*1.5+2 (place of right byte)
        sb     $5,2($2) ; branch delay slot, this instruction is executed before J

```

### 3.21.12 Difference from the real FAT12

The real FAT12 table is slightly different: [https://en.wikipedia.org/wiki/Design\\_of\\_the\\_FAT\\_file\\_system#Cluster\\_map](https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Cluster_map).

For even elements:

+0	+1	+2
23	.1	... ..

### 3.22. MORE ABOUT POINTERS

For odd elements:

```
+0 +1 +2  
|..|3.|12|..
```

Here are FAT12-related functions in Linux Kernel:

[fat12\\_ent\\_get\(\)](#), [fat12\\_ent\\_put\(\)](#).

Nevertheless, I did as I did because values are better visible and recognizable in byte-level GDB dump, for the sake of demonstration.

#### 3.21.13 Exercise

Perhaps, there could be a way to store data in a such way, so getter/setter functions would be faster. If we would place values in this way:

(Even elements)

```
+0 +1 +2  
|23|1.|..|..
```

(Odd elements)

```
+0 +1 +2  
|..|.1|23|..
```

This schema of storing data will allow to eliminate at least one shift operation. As an exercise, you may rework my C/C++ code in that way and see what compiler will produce.

#### 3.21.14 Summary

Bit shifts (`<<` and `>>` in C/C++, `SHL`/SHR/SAL/SAR in x86, `LSL`/`LSR` in ARM, `SLL`/`SRL` in MIPS) are used to place bit(s) to specific place.

AND operation (`&` in C/C++, `AND` in x86/ARM) is used to drop unneeded bits, also during isolation.

OR operation (`|` in C/C++, `OR` in x86/ARM) is used to merge or combine several values into one. One input value must have zero space at the place where another value has its information-caring bits.

ARM64 has new instructions `UBFM`, `UFBIZ` to move specific bits from one register to another.

#### 3.21.15 Conclusion

FAT12 is hardly used somewhere nowadays, but if you have space constraints and you have to store values limited to 12 bits, you may consider tightly-packed array in the manner it's done in FAT12 table.

### 3.22 More about pointers

The way C handles pointers, for example, was a brilliant innovation; it solved a lot of problems that we had before in data structuring and made the programs look good afterwards.

Donald Knuth, interview (1993)

For those, who still have hard time understanding C/C++ pointers, here are more examples. Some of them are weird and serves only demonstration purpose: use them in production code only if you really know what you're doing.

## 3.22.1 Working with addresses instead of pointers

Pointer is just an address in memory. But why we write `char* string` instead of something like `address string`? Pointer variable is supplied with a type of the value to which pointer points. So then compiler will be able to catch data typization bugs during compilation.

To be pedantic, data typing in programming languages is all about preventing bugs and self-documentation. It's possible to use maybe two of data types like `int` (or `int64_t`) and `byte`—these are the only types which are available to assembly language programmers. But it's just very hard task to write big and practical assembly programs without nasty bugs. Any small typo can lead to hard-to-find bug.

Data type information is absent in a compiled code (and this is one of the main problems for decompilers), and I can demonstrate this.

This is what sane C/C++ programmer can write:

```
#include <stdio.h>
#include <stdint.h>

void print_string (char *s)
{
    printf("(address: 0x%llx)\n", s);
    printf("%s\n", s);
};

int main()
{
    char *s="Hello, world!";
    print_string (s);
}
```

This is what I can write:

```
#include <stdio.h>
#include <stdint.h>

void print_string (uint64_t address)
{
    printf("(address: 0x%llx)\n", address);
    puts ((char*)address);
};

int main()
{
    char *s="Hello, world!";
    print_string ((uint64_t)s);
}
```

I use `uint64_t` because I run this example on Linux x64. `int` would work for 32-bit OS-es. First, a pointer to character (the very first in the greeting string) is casted to `uint64_t`, then it's passed further. `print_string()` function casts back incoming `uint64_t` value into pointer to a character.

What is interesting is that GCC 4.8.4 produces identical assembly output for both versions:

```
gcc 1.c -S -masm=intel -O3 -fno-inline
```

```
.LC0:
.string "(address: 0x%llx)\n"
print_string:
    push    rbx
    mov     rdx, rdi
    mov     rbx, rdi
    mov     esi, OFFSET FLAT:.LC0
    mov     edi, 1
    xor     eax, eax
    call    __printf_chk
    mov     rdi, rbx
```

### 3.22. MORE ABOUT POINTERS

```
pop    rbx
jmp    puts
.LC1:
.string "Hello, world!"
main:
sub    rsp, 8
mov    edi, OFFSET FLAT:.LC1
call   print_string
add    rsp, 8
ret
```

(I've removed all insignificant GCC directives.)

I also tried UNIX *diff* utility and it shows no differences at all.

Let's continue to abuse C/C++ programming traditions heavily. Someone may write this:

```
#include <stdio.h>
#include <stdint.h>

uint8_t load_byte_at_address (uint8_t* address)
{
    return *address;
    //this is also possible: return address[0];
};

void print_string (char *s)
{
    char* current_address=s;
    while (1)
    {
        char current_char=load_byte_at_address(current_address);
        if (current_char==0)
            break;
        printf ("%c", current_char);
        current_address++;
    };
};

int main()
{
    char *s="Hello, world!";
    print_string (s);
};
```

It can be rewritten like this:

```
#include <stdio.h>
#include <stdint.h>

uint8_t load_byte_at_address (uint64_t address)
{
    return *(uint8_t*)address;
    //this is also possible: return address[0];
};

void print_string (uint64_t address)
{
    uint64_t current_address=address;
    while (1)
    {
        char current_char=load_byte_at_address(current_address);
        if (current_char==0)
            break;
        printf ("%c", current_char);
        current_address++;
    };
};

int main()
```

### 3.22. MORE ABOUT POINTERS

```
{  
    char *s="Hello, world!";  
  
    print_string ((uint64_t)s);  
};
```

Both source codes resulting in the same assembly output:

```
gcc 1.c -S -masm=intel -O3 -fno-inline
```

```
load_byte_at_address:  
    movzx   eax, BYTE PTR [rdi]  
    ret  
print_string:  
.LFB15:  
    push    rbx  
    mov     rbx, rdi  
    jmp     .L4  
.L7:  
    movsx   edi, al  
    add    rbx, 1  
    call    putchar  
.L4:  
    mov     rdi, rbx  
    call    load_byte_at_address  
    test    al, al  
    jne     .L7  
    pop    rbx  
    ret  
.LC0:  
    .string "Hello, world!"  
main:  
    sub    rsp, 8  
    mov     edi, OFFSET FLAT:.LC0  
    call    print_string  
    add    rsp, 8  
    ret
```

(I have also removed all insignificant GCC directives.)

No difference: C/C++ pointers are essentially addresses, but supplied with type information, in order to prevent possible mistakes at the time of compilation. Types are not checked during runtime—it would be huge (and unneeded) overhead.

#### 3.22.2 Passing values as pointers; tagged unions

Here is an example on how to pass values in pointers:

```
#include <stdio.h>  
#include <stdint.h>  
  
uint64_t multiply1 (uint64_t a, uint64_t b)  
{  
    return a*b;  
};  
  
uint64_t* multiply2 (uint64_t *a, uint64_t *b)  
{  
    return (uint64_t*)((uint64_t)a*(uint64_t)b);  
};  
  
int main()  
{  
    printf ("%d\n", multiply1(123, 456));  
    printf ("%d\n", (uint64_t)multiply2((uint64_t*)123, (uint64_t*)456));  
};
```

It works smoothly and GCC 4.8.4 compiles both multiply1() and multiply2() functions identically!

```

multiply1:
    mov     rax, rdi
    imul   rax, rsi
    ret

multiply2:
    mov     rax, rdi
    imul   rax, rsi
    ret

```

As long as you do not dereference pointer (in other words, you don't read any data from the address stored in pointer), everything will work fine. Pointer is a variable which can store anything, like usual variable.

Signed multiplication instruction (`IMUL`) is used here instead of unsigned one (`MUL`), read more about it here: [2.2.1](#).

By the way, it's well-known hack to abuse pointers a little called *tagged pointers*. In short, if all your pointers points to blocks of memory with size of, let's say, 16 bytes (or it is always aligned on 16-byte boundary), 4 lowest bits of pointer is always zero bits and this space can be used somehow. It's very popular in LISP compilers and interpreters. They store cell/object type in these unused bits, this can save some memory. Even more, you can judge about cell/object type using just pointer, with no additional memory access. Read more about it: [Dennis Yurichev, *C/C++ programming language notes*1.3].

### 3.22.3 Pointers abuse in Windows kernel

The resource section of PE executable file in Windows OS is a section containing pictures, icons, strings, etc. Early Windows versions allowed to address resources only by IDs, but then Microsoft added a way to address them using strings.

So then it would be possible to pass ID or string to `FindResource()` function. Which is declared like this:

```

HRSRC WINAPI FindResource(
    _In_opt_ HMODULE hModule,
    _In_     LPCTSTR lpName,
    _In_     LPCTSTR lpType
);

```

`lpName` and `lpType` has `char*` or `wchar*` types, and when someone still wants to pass ID, he/she have to use `MAKEINTRESOURCE` macro, like this:

```
result = FindResource(..., MAKEINTRESOURCE(1234), ...);
```

It's interesting fact that `MAKEINTRESOURCE` is merely casting integer to pointer. In MSVC 2013, in the file `Microsoft SDKs\Windows\v7.1A\Include\Ks.h` we can find this:

```

...
#ifndef !defined( MAKEINTRESOURCE )
#define MAKEINTRESOURCE( res ) ((ULONG_PTR) (USHORT) res)
#endif

...

```

Sounds insane. Let's peek into ancient leaked Windows NT4 source code. In `private/windows/base/client/module.c` we can find `FindResource()` source code:

```

HRSRC
FindResourceA(
    HMODULE hModule,
    LPCSTR lpName,
    LPCSTR lpType
)

...
{

    NTSTATUS Status;

```

### 3.22. MORE ABOUT POINTERS

```
ULONG IdPath[ 3 ];
PVOID p;

IdPath[ 0 ] = 0;
IdPath[ 1 ] = 0;
try {
    if ((IdPath[ 0 ] = BaseDllMapResourceIdA( lpType )) == -1) {
        Status = STATUS_INVALID_PARAMETER;
    }
    else
        if ((IdPath[ 1 ] = BaseDllMapResourceIdA( lpName )) == -1) {
            Status = STATUS_INVALID_PARAMETER;
    }
...
}
```

Let's proceed to *BaseDllMapResourceIdA()* in the same source file:

```
ULONG
BaseDllMapResourceIdA(
    LPCSTR lpId
)
{
    NTSTATUS Status;
    ULONG Id;
    UNICODE_STRING UnicodeString;
    ANSI_STRING AnsiString;
    PWSTR s;

    try {
        if (((ULONG)lpId & LDR_RESOURCE_ID_NAME_MASK) {
            if (*lpId == '#') {
                Status = RtlCharToInteger( lpId+1, 10, &Id );
                if (!NT_SUCCESS( Status ) || Id & LDR_RESOURCE_ID_NAME_MASK) {
                    if (!NT_SUCCESS( Status )) {
                        Status = STATUS_INVALID_PARAMETER;
                    }
                    BaseSetLastNTErrror( Status );
                    Id = (ULONG)-1;
                }
            }
            else {
                RtlInitAnsiString( &AnsiString, lpId );
                Status = RtlAnsiStringToUnicodeString( &UnicodeString,
                    &AnsiString,
                    TRUE
                );
                if (!NT_SUCCESS( Status )) {
                    BaseSetLastNTErrror( Status );
                    Id = (ULONG)-1;
                }
                else {
                    s = UnicodeString.Buffer;
                    while (*s != UNICODE_NULL) {
                        *s = RtlUpcaseUnicodeChar( *s );
                        s++;
                    }

                    Id = (ULONG)UnicodeString.Buffer;
                }
            }
        }
        else {
            Id = (ULONG)lpId;
        }
    }
    except (EXCEPTION_EXECUTE_HANDLER) {
        BaseSetLastNTErrror( GetExceptionCode() );
        Id = (ULONG)-1;
    }
    return Id;
}
```

### 3.22. MORE ABOUT POINTERS

*lpld* is ANDed with *LDR\_RESOURCE\_ID\_NAME\_MASK*. Which we can find in *public/sdk/inc/ntldr.h*:

```
...
#define LDR_RESOURCE_ID_NAME_MASK 0xFFFF0000
...
```

So *lpld* is ANDed with *0xFFFF0000* and if some bits beyond lowest 16 bits are still present, first half of function is executed (*lpld* is treated as an address of string). Otherwise—second half (*lpld* is treated as 16-bit value).

Still, this code can be found in Windows 7 kernel32.dll file:

```
...
.text:0000000078D24510 ; __int64 __fastcall BaseDllMapResourceIdA(PCSZ SourceString)
.text:0000000078D24510 BaseDllMapResourceIdA proc near ; CODE XREF: FindResourceExA+34
.text:0000000078D24510 ; FindResourceExA+4B
.text:0000000078D24510
.text:0000000078D24510 var_38 = qword ptr -38h
.text:0000000078D24510 var_30 = qword ptr -30h
.text:0000000078D24510 var_28 = _UNICODE_STRING ptr -28h
.text:0000000078D24510 DestinationString= _STRING ptr -18h
.text:0000000078D24510 arg_8 = dword ptr 10h
.text:0000000078D24510
.text:0000000078D24510 ; FUNCTION CHUNK AT .text:0000000078D42FB4 SIZE 000000D5 BYTES
.text:0000000078D24510
.text:0000000078D24510 push rbx
.text:0000000078D24512 sub rsp, 50h
.text:0000000078D24516 cmp rcx, 10000h
.text:0000000078D2451D jnb loc_78D42FB4
.text:0000000078D24523 mov [rsp+58h+var_38], rcx
.text:0000000078D24528 jmp short $+2
.text:0000000078D2452A ; 
    ↓ -----
.text:0000000078D2452A
.text:0000000078D2452A loc_78D2452A: ; CODE XREF: ↴
    ↴ BaseDllMapResourceIdA+18
.text:0000000078D2452A jmp short $+2 ; BaseDllMapResourceIdA+1EAD0
.text:0000000078D2452A
.text:0000000078D2452C ; 
    ↓ -----
.text:0000000078D2452C
.text:0000000078D2452C loc_78D2452C: ; CODE XREF: ↴
    ↴ BaseDllMapResourceIdA:loc_78D2452A
.text:0000000078D2452C ; BaseDllMapResourceIdA+1EB74
.text:0000000078D2452C mov rax, rcx
.text:0000000078D2452F add rsp, 50h
.text:0000000078D24533 pop rbx
.text:0000000078D24534 retn
.text:0000000078D24534 ; 
    ↓ -----
.text:0000000078D24535 align 20h
.text:0000000078D24535 BaseDllMapResourceIdA endp
...
.text:0000000078D42FB4 loc_78D42FB4: ; CODE XREF: ↴
    ↴ BaseDllMapResourceIdA+D
.text:0000000078D42FB4 cmp byte ptr [rcx], '#'
.text:0000000078D42FB7 jnz short loc_78D43005
.text:0000000078D42FB9 inc rcx
.text:0000000078D42FBC lea r8, [rsp+58h+arg_8]
.text:0000000078D42FC1 mov edx, 0Ah
.text:0000000078D42FC6 call cs:_imp_RtlCharToInteger
.text:0000000078D42FCC mov ecx, [rsp+58h+arg_8]
.text:0000000078D42FD0 mov [rsp+58h+var_38], rcx
.text:0000000078D42FD5 test eax, eax
.text:0000000078D42FD7 js short loc_78D42FE6
.text:0000000078D42FD9 test rcx, 0xFFFFFFFFFFFF0000h
```

### 3.22. MORE ABOUT POINTERS

```
.text:0000000078D42FE0          jz      loc_78D2452A
```

```
....
```

If value in input pointer is greater than 0x10000, jump to string processing is occurred. Otherwise, input value of *lpId* is returned as is. *0xFFFF0000* mask is not used here any more, because this is 64-bit code after all, but still, *0xFFFFFFFFFFFF0000* could work here.

Attentive reader may ask, what if address of input string is lower than 0x10000? This code relied on the fact that in Windows there are nothing on addresses below 0x10000, at least in Win32 realm.

Raymond Chen [writes](#) about this:

How does MAKEINTRESOURCE work? It just stashes the integer in the bottom 16 bits of a pointer, leaving the upper bits zero. This relies on the convention that the first 64KB of address space is never mapped to valid memory, a convention that is enforced starting in Windows 7.

In short words, this is dirty hack and probably one should use it only if there is a real necessity. Perhaps, *FindResource()* function in past had *SHORT* type for its arguments, and then Microsoft has added a way to pass strings there, but older code must also be supported.

Now here is my short distilled example:

```
#include <stdio.h>
#include <stdint.h>

void f(char* a)
{
    if (((uint64_t)a)>0x10000)
        printf ("Pointer to string has been passed: %s\n", a);
    else
        printf ("16-bit value has been passed: %d\n", (uint64_t)a);
}

int main()
{
    f("Hello!"); // pass string
    f((char*)1234); // pass 16-bit value
}
```

It works!

### Pointers abuse in Linux kernel

As it has been noted in [comments on Hacker News](#), Linux kernel also has something like that.

For example, this function can return both error code and pointer:

```
struct kernfs_node *kernfs_create_link(struct kernfs_node *parent,
                                       const char *name,
                                       struct kernfs_node *target)
{
    struct kernfs_node *kn;
    int error;

    kn = kernfs_new_node(parent, name, S_IFLNK|S_IRWXUGO, KERNFS_LINK);
    if (!kn)
        return ERR_PTR(-ENOMEM);

    if (kernfs_ns_enabled(parent))
        kn->ns = target->ns;
    kn->symlink.target_kn = target;
    kernfs_get(target); /* ref owned by symlink */

    error = kernfs_add_one(kn);
    if (!error)
        return kn;
```

### 3.22. MORE ABOUT POINTERS

```
    kernfs_put(kn);
    return ERR_PTR(error);
}
```

(<https://github.com/torvalds/linux/blob/fceef393a538134f03b778c5d2519e670269342f/fs/kernfs/symlink.c#L25>)

*ERR\_PTR* is a macro to cast integer to pointer:

```
static inline void * __must_check ERR_PTR(long error)
{
    return (void *) error;
}
```

(<https://github.com/torvalds/linux/blob/61d0b5a4b2777dcf5daef245e212b3c1fa8091ca/tools/virtio/linux/err.h>)

This header file also has a macro helper to distinguish error code from pointer:

```
#define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)
```

This means, error codes are the “pointers” which are very close to -1 and, hopefully, there are nothing in kernel memory on the addresses like 0xFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFE, 0xFFFFFFFFFFFFFFFD, etc.

Much more popular solution is to return *NULL* in case of error and to pass error code via additional argument. Linux kernel authors don’t do that, but everyone who use these functions must always keep in mind that returning pointer must always be checked with *IS\_ERR\_VALUE* before dereferencing.

For example:

```
fman->cam_offset = fman_muram_alloc(fman->muram, fman->cam_size);
if (IS_ERR_VALUE(fman->cam_offset)) {
    dev_err(fman->dev, "%s: MURAM alloc for DMA CAM failed\n",
            __func__);
    return -ENOMEM;
}
```

(<https://github.com/torvalds/linux/blob/aa00edc1287a693eadc7bc67a3d73555d969b35d/drivers/net/ethernet/freescale/fman/fman.c#L826>)

## Pointers abuse in UNIX userland

*mmap()* function returns -1 in case of error (or *MAP\_FAILED*, which equals to -1). Some people say, *mmap()* can map a memory at zeroth address in rare situations, so it can’t use 0 or *NULL* as error code.

### 3.22.4 Null pointers

#### “Null pointer assignment” error of MS-DOS era

Some oldschoolers may recall a weird error message of MS-DOS era: “Null pointer assignment”. What does it mean?

It’s not possible to write a memory at zero address in \*NIX and Windows OSes, but it was possible to do so in MS-DOS due to absence of memory protection whatsoever.

So I’ve pulled my ancient Turbo C++ 3.0 (later it was renamed to Borland C++) from early 1990s and tried to compile this:

```
#include <stdio.h>

int main()
{
    int *ptr=NULL;
    *ptr=1234;
    printf ("Now let's read at NULL\n");
```

### 3.22. MORE ABOUT POINTERS

```
    printf ("%d\n", *ptr);  
};
```

Hard to believe, but it works, with error upon exit, though:

Listing 3.125: Ancient Turbo C 3.0

```
C:\TC30\BIN\1  
Now let's read at NULL  
1234  
Null pointer assignment  
  
C:\TC30\BIN>_
```

Let's dig deeper into the source code of [CRT](#) of Borland C++ 3.1, file *c0.asm*:

```
;      _checknull()      check for null pointer zapping copyright message  
  
...  
;  
;      Check for null pointers before exit  
  
_checknull      PROC      DIST  
                PUBLIC     __checknull  
  
IF      LDATA  EQ  false  
IFNDEF  __TINY__  
        push    si  
        push    di  
        mov     es, cs:DGROUP@@  
        xor     ax, ax  
        mov     si, ax  
        mov     cx, lgth_CopyRight  
ComputeChecksum label  near  
        add    al, es:[si]  
        adc    ah, 0  
        inc    si  
        loop   ComputeChecksum  
        sub    ax, CheckSum  
        jz     @@SumOK  
        mov    cx, lgth_NullCheck  
        mov    dx, offset DGROUP: NullCheck  
        call   ErrorDisplay  
@@SumOK:  
        pop    di  
        pop    si  
ENDIF  
ENDIF  
  
_DATA      SEGMENT  
  
;      Magic symbol used by the debug info to locate the data segment  
;          public DATASEG@  
DATASEG@      label  byte  
  
;      The CopyRight string must NOT be moved or changed without  
;          changing the null pointer check logic  
  
CopyRight      db      4 dup(0)  
                db      'Borland C++ - Copyright 1991 Borland Intl.',0  
lgth_CopyRight equ    $ - CopyRight  
  
IF      LDATA  EQ  false  
IFNDEF  __TINY__  
CheckSum      equ    00D5Ch  
NullCheck     db      'Null pointer assignment', 13, 10  
lgth_NullCheck equ    $ - NullCheck  
ENDIF  
ENDIF  
  
...
```

### 3.22. MORE ABOUT POINTERS

The MS-DOS memory model was really weird (10.6) and probably not worth looking into it unless you're fan of retrocomputing or retrogaming. One thing we have to keep in mind is that memory segment (included data segment) in MS-DOS is a memory segment in which code or data is stored, but unlike "serious" OSes, it's started at address 0.

And in Borland C++ [CRT](#), the data segment is started with 4 zero bytes and the copyright string "Borland C++ - Copyright 1991 Borland Intl.". The integrity of the 4 zero bytes and text string is checked upon exit, and if it's corrupted, the error message is displayed.

But why? Writing at null pointer is common mistake in C/C++, and if you do so in \*NIX or Windows, your application will crash. MS-DOS has no memory protection, so [CRT](#) has to check this post-factum and warn about it upon exit. If you see this message, this means, your program at some point has written at address 0.

Our program did so. And this is why 1234 number has been read correctly: because it was written at the place of the first 4 zero bytes. Checksum is incorrect upon exit (because the number has left there), so error message has been displayed.

Am I right? I've rewritten the program to check my assumptions:

```
#include <stdio.h>

int main()
{
    int *ptr=NULL;
    *ptr=1234;
    printf ("Now let's read at NULL\n");
    printf ("%d\n", *ptr);
    *ptr=0; // psst, cover our tracks!
};
```

This program executes without error message upon exit.

Though method to warn about null pointer assignment is relevant for MS-DOS, perhaps, it can still be used today in low-cost [MCUs](#) with no memory protection and/or [MMU](#)<sup>37</sup>.

### Why would anyone write at address 0?

But why would sane programmer write a code which writes something at address 0? It can be done accidentally: for example, a pointer must be initialized to newly allocated memory block and then passed to some function which returns data through pointer.

```
int *ptr=NULL;

... we forgot to allocate memory and initialize ptr

strcpy (ptr, buf); // strcpy() terminates silently because MS-DOS has no memory protection
```

Even worse:

```
int *ptr=malloc(1000);

... we forgot to check if memory has been really allocated: this is MS-DOS after all and ↴
    ↴ computers had small amount of RAM,
... and RAM shortage was very common.
... if malloc() returned NULL, the ptr will also be NULL.

strcpy (ptr, buf); // strcpy() terminates silently because MS-DOS has no memory protection
```

### NULL in C/C++

NULL in C/C++ is just a macro which is often defined like this:

```
#define NULL ((void*)0)
```

<sup>37</sup>Memory management unit

### 3.22. MORE ABOUT POINTERS

( [libio.h file](#) )

`void*` is a data type reflecting the fact it's the pointer, but to a value of unknown data type (`void`).

`NULL` is usually used to show absence of an object. For example, you have a single-linked list, and each node has a value (or pointer to a value) and `next` pointer. To show that there are no next node, 0 is stored to `next` field. Other solutions are just worse. Perhaps, you may have some crazy environment where you need to allocate memory blocks at zero address. How would you indicate absence of the next node? Some kind of *magic number*? Maybe -1? Or maybe using additional bit?

In Wikipedia we may find this:

In fact, quite contrary to the zero page's original preferential use, some modern operating systems such as FreeBSD, Linux and Microsoft Windows[2] actually make the zero page inaccessible to trap uses of `NULL` pointers.

( [https://en.wikipedia.org/wiki/Zero\\_page](https://en.wikipedia.org/wiki/Zero_page) )

### Null pointer to function

It's possible to call function by its address. For example, I compile this by MSVC 2010 and run it in Windows 7:

```
#include <windows.h>
#include <stdio.h>

int main()
{
    printf ("0x%x\n", &MessageBoxA);
}
```

The result is `0x7578feae` and doesn't change after several times I run it, because `user32.dll` (where `MessageBoxA` function resides) is always loads at the same address. And also because [ASLR<sup>38</sup>](#) is not enabled (result would be different each time in that case).

Let's call `MessageBoxA()` by address:

```
#include <windows.h>
#include <stdio.h>

typedef int (*msgboxtype)(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);

int main()
{
    msgboxtype msgboxaddr=0x7578feae;

    // force to load DLL into process memory,
    // since our code doesn't use any function from user32.dll,
    // and DLL is not imported
    LoadLibrary ("user32.dll");

    msgboxaddr(NULL, "Hello, world!", "hello", MB_OK);
}
```

Weird, but works in Windows 7 x86.

This is commonly used in shellcodes, because it's hard to call DLL functions by name from there. And [ASLR](#) is a countermeasure.

Now what is really weird, some embedded C programmers may be familiar with a code like that:

```
int reset()
{
    void (*foo)(void) = 0;
    foo();
}
```

<sup>38</sup>Address Space Layout Randomization

### 3.22. MORE ABOUT POINTERS

Who will want to call a function at address 0? This is portable way to jump at zero address. Many low-cost cheap microcontrollers also have no memory protection or [MMU](#) and after reset, they start to execute code at address 0, where some kind of initialization code is stored. So jumping to address 0 is a way to reset itself. One could use inline assembly, but if it's not possible, this portable method can be used.

It even compiles correctly by my GCC 4.8.4 on Linux x64:

```
reset:
    sub    rsp, 8
    xor    eax, eax
    call   rax
    add    rsp, 8
    ret
```

The fact that stack pointer is shifted is not a problem: initialization code in microcontrollers usually completely ignores registers and [RAM](#) state and boots from scratch.

And of course, this code will crash on \*NIX or Windows because of memory protection and even in absence of protection, there are no code at address 0.

GCC even has non-standard extension, allowing to jump to a specific address rather than call a function there: <http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html>.

#### 3.22.5 Array as function argument

Someone may ask, what is the difference between declaring function argument type as array and as pointer?

As it seems, there are no difference at all:

```
void write_something1(int a[16])
{
    a[5]=0;
}

void write_something2(int *a)
{
    a[5]=0;
}

int f()
{
    int a[16];
    write_something1(a);
    write_something2(a);
}
```

Optimizing GCC 4.8.4:

```
write_something1:
    mov    DWORD PTR [rdi+20], 0
    ret

write_something2:
    mov    DWORD PTR [rdi+20], 0
    ret
```

But you may still declare array instead of pointer for self-documenting purposes, if the size of array is always fixed. And maybe, some static analysis tool will be able to warn you about possible buffer overflow. Or is it possible with some tools today?

Some people, including Linus Torvalds, criticizes this C/C++ feature: <https://lkml.org/lkml/2015/9/3/428>.

C99 standard also have *static* keyword [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.5.3]:

If the keyword static also appears within the [ and ] of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

### 3.22.6 Pointer to function

A function name in C/C++ without brackets, like “printf” is a pointer to function of `void (*)()` type. Let's try to read function's contents and patch it:

```
#include <memory.h>
#include <stdio.h>

void print_something ()
{
    printf ("we are in %s()\n", __FUNCTION__);
}

int main()
{
    print_something();
    printf ("first 3 bytes: %x %x %x...\n",
            *(unsigned char*)print_something,
            *((unsigned char*)print_something+1),
            *((unsigned char*)print_something+2));

    *(unsigned char*)print_something=0xC3; // opcode of RET
    printf ("going to call patched print_something():\n");
    print_something();
    printf ("it must exit at this point\n");
}
```

It tells, that the first 3 bytes of functions are `55 89 e5`. Indeed, these are opcodes of `PUSH EBP` and `MOV EBP, ESP` instructions (these are x86 opcodes). But then our program crashes, because `text` section is readonly.

We can recompile our example and make `text` section writable <sup>39</sup>:

```
gcc --static -g -Wl,--omagic -o example example.c
```

That works!

```
we are in print_something()
first 3 bytes: 55 89 e5...
going to call patched print_something():
it must exit at this point
```

### 3.22.7 Pointer as object identifier

Both assembly language and C has no `OOP` features, but it's possible to write a code in `OOP` style (just treat structure as an object).

It's interesting, that sometimes, pointer to an object (or its address) is called as ID (in sense of data hiding/encapsulation).

For example, `LoadLibrary()`, according to [MSDN<sup>40</sup>](#), returns “handle to the module” <sup>41</sup>. Then you pass this “handle” to other functions like `GetProcAddress()`. But in fact, `LoadLibrary()` returns pointer to DLL file

<sup>39</sup><http://stackoverflow.com/questions/27581279/make-text-segment-writable-elf>

<sup>40</sup>Microsoft Developer Network

<sup>41</sup>[https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684175\(v=vs.85\).aspx](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684175(v=vs.85).aspx)

### 3.23. LOOP OPTIMIZATIONS

mapped into memory<sup>42</sup>. You can read two bytes from the address LoadLibrary() returns, and that would be "MZ" (first two bytes of any .EXE/.DLL file in Windows).

Apparently, Microsoft "hides" that fact to provide better forward compatibility. Also, HMODULE and HINSTANCE data types had another meaning in 16-bit Windows.

Probably, this is reason why `printf()` has "%p" modifier, which is used for printing pointers (32-bit integers on 32-bit architectures, 64-bit on 64-bit, etc) in hexadecimal form. Address of a structure dumped into debug log may help in finding it in another place of log.

Here is also from SQLite source code:

```
...
struct Pager {
    sqlite3_vfs *pVfs;           /* OS functions to use for IO */
    u8 exclusiveMode;           /* Boolean. True if locking_mode==EXCLUSIVE */
    u8 journalMode;             /* One of the PAGER_JOURNALMODE_* values */
    u8 useJournal;              /* Use a rollback journal on this file */
    u8 noSync;                  /* Do not sync the journal if true */
...
static int pagerLockDb(Pager *pPager, int eLock){
    int rc = SQLITE_OK;

    assert( eLock==SHARED_LOCK || eLock==RESERVED_LOCK || eLock==EXCLUSIVE_LOCK );
    if( pPager->eLock<eLock || pPager->eLock==UNKNOWN_LOCK ){
        rc = sqlite3OsLock(pPager->fd, eLock);
        if( rc==SQLITE_OK && (pPager->eLock!=UNKNOWN_LOCK||eLock==EXCLUSIVE_LOCK) ){
            pPager->eLock = (u8)eLock;
            IOTRACE(("LOCK %p %d\n", pPager, eLock))
        }
    }
    return rc;
}
...
PAGER_INCR(sqlite3_pager_readdb_count);
PAGER_INCR(pPager->nRead);
IOTRACE(("PGIN %p %d\n", pPager, pgno));
PAGERTRACE(("FETCH %d page %d hash(%08x)\n",
            PAGERID(pPager), pgno, pager_pagehash(pPg)));
...
...
```

## 3.23 Loop optimizations

### 3.23.1 Weird loop optimization

This is a simplest ever `memcpy()` function implementation:

```
void memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};
```

At least MSVC 6.0 from the end of 1990s till MSVC 2013 can produce a really weird code (this listing is generated by MSVC 2013 x86):

<sup>42</sup><https://blogs.msdn.microsoft.com/oldnewthing/20041025-00/?p=37483>

### 3.23. LOOP OPTIMIZATIONS

```

_dst$ = 8          ; size = 4
_src$ = 12         ; size = 4
_cnt$ = 16         ; size = 4
_memcpy PROC
    mov     edx, DWORD PTR _cnt$[esp-4]
    test    edx, edx
    je      SHORT $LN1@f
    mov     eax, DWORD PTR _dst$[esp-4]
    push    esi
    mov     esi, DWORD PTR _src$[esp]
    sub    esi, eax
; ESI=src-dst, i.e., pointers difference
$LL8@f:
    mov     cl, BYTE PTR [esi+eax] ; load byte at "esi+dst" or at "src-dst+dst" at the ↴
    ↵ beginning or at just "src"
    lea     eax, DWORD PTR [eax+1] ; dst++
    mov     BYTE PTR [eax-1], cl   ; store the byte at "(dst++)--" or at just "dst" at the ↴
    ↵ beginning
    dec     edx                  ; decrement counter until we finished
    jne    SHORT $LL8@f
    pop    esi
$LN1@f:
    ret    0
_memcpy ENDP

```

This is weird, because how humans work with two pointers? They store two addresses in two registers or two memory cells. MSVC compiler in this case stores two pointers as one pointer (*sliding dst* in `EAX`) and difference between *src* and *dst* pointers (left unchanged over the span of loop body execution in `ESI`). When it needs to load a byte from *src*, it loads it at *diff + sliding dst* and stores byte at just *sliding dst*.

This has to be some optimization trick. But I've rewritten this function to:

```

_f2  PROC
    mov     edx, DWORD PTR _cnt$[esp-4]
    test    edx, edx
    je      SHORT $LN1@f
    mov     eax, DWORD PTR _dst$[esp-4]
    push    esi
    mov     esi, DWORD PTR _src$[esp]
    ; eax=dst; esi=src
$LL8@f:
    mov     cl, BYTE PTR [esi+edx]
    mov     BYTE PTR [eax+edx], cl
    dec     edx
    jne    SHORT $LL8@f
    pop    esi
$LN1@f:
    ret    0
_f2  ENDP

```

...and it works as efficient as the *optimized* version on my Intel Xeon E31220 @ 3.10GHz. Maybe, this optimization was targeted some older x86 CPUs of 1990s era, since this trick is used at least by ancient MS VC 6.0?

Any idea?

Hex-Rays 2.2 have a hard time recognizing patterns like that (hopefully, temporary?):

```

void __cdecl f1(char *dst, char *src, size_t size)
{
    size_t counter; // edx@1
    char *sliding_dst; // eax@2
    char tmp; // cl@3

    counter = size;
    if ( size )
    {
        sliding_dst = dst;
        do

```

### 3.23. LOOP OPTIMIZATIONS

```
{  
    tmp = (sliding_dst++)[src - dst];           // difference (src-dst) is calculated once,  
    ↴ before loop body  
    *(sliding_dst - 1) = tmp;  
    --counter;  
}  
while ( counter );  
}  
}
```

Nevertheless, this optimization trick is often used by MSVC (not just in [DIY<sup>43</sup>](#) homebrew `memcpy()` routines, but in many loops which uses two or more arrays), so it's worth for reverse engineers to keep it in mind.

#### 3.23.2 Another loop optimization

If you process all elements of some array which happens to be located in global memory, compiler can optimize it. For example, let's calculate a sum of all elements of array of 128 `int`'s:

```
#include <stdio.h>  
  
int a[128];  
  
int sum_of_a()  
{  
    int rt=0;  
  
    for (int i=0; i<128; i++)  
        rt=rt+a[i];  
  
    return rt;  
};  
  
int main()  
{  
    // initialize  
    for (int i=0; i<128; i++)  
        a[i]=i;  
  
    // calculate the sum  
    printf ("%d\n", sum_of_a());  
};
```

Optimizing GCC 5.3.1 (x86) can produce this ([IDA](#)):

```
.text:080484B0 sum_of_a          proc near  
.text:080484B0                 mov     edx, offset a  
.text:080484B5                 xor     eax, eax  
.text:080484B7                 mov     esi, esi  
.text:080484B9                 lea     edi, [edi+0]  
.text:080484C0 loc_80484C0:      ; CODE XREF: sum_of_a+1B  
.text:080484C0                 add     eax, [edx]  
.text:080484C2                 add     edx, 4  
.text:080484C5                 cmp     edx, offset __libc_start_main@@GLIBC_2_0  
.text:080484CB                 jnz     short loc_80484C0  
.text:080484CD                 rep     retn  
.text:080484CD sum_of_a         endp  
.text:080484CD  
  
...  
  
.bss:0804A040                 public a  
.bss:0804A040 a                  dd 80h dup(?) ; DATA XREF: main:loc_8048338  
.bss:0804A040                   ; main+19  
.bss:0804A040 _bss              ends  
.bss:0804A040
```

<sup>43</sup>Do It Yourself

### 3.23. LOOP OPTIMIZATIONS

```
extern:0804A240 ; =====
extern:0804A240
extern:0804A240 ; Segment type: Externs
extern:0804A240 ; extern
extern:0804A240         extrn __libc_start_main@@GLIBC_2_0:near
extern:0804A240             ; DATA XREF: main+25
extern:0804A240             ; main+5D
extern:0804A244         extrn __printf_chk@@GLIBC_2_3_4:near
extern:0804A248         extrn __libc_start_main:near
extern:0804A248             ; CODE XREF: __libc_start_main
extern:0804A248             ; DATA XREF: .got.plt:off_804A00C
```

What the heck is `__libc_start_main@@GLIBC_2_0` at `0x080484C5`? This is a label just after end of `a[]` array. The function can be rewritten like this:

```
int sum_of_a_v2()
{
    int *tmp=a;
    int rt=0;

    do
    {
        rt=rt+(*tmp);
        tmp++;
    }
    while (tmp<(a+128));

    return rt;
};
```

First version has *i* counter, and the address of each element of array is to be calculated at each iteration. The second version is more optimized: the pointer to each element of array is always ready and is sliding 4 bytes forward at each iteration. How to check if the loop is ended? Just compare the pointer with the address just behind array's end, which is, in our case, is happens to be address of imported `__libc_start_main()` function from Glibc 2.0. Sometimes code like this is confusing, and this is very popular optimizing trick, so that's why I made this example.

My second version is very close to what GCC did, and when I compile it, the code is almost the same as in first version, but two first instructions are swapped:

```
.text:080484D0          public sum_of_a_v2
.text:080484D0 sum_of_a_v2    proc near
.text:080484D0             xor    eax, eax
.text:080484D2             mov    edx, offset a
.text:080484D7             mov    esi, esi
.text:080484D9             lea    edi, [edi+0]
.text:080484E0
.text:080484E0 loc_80484E0:    ; CODE XREF: sum_of_a_v2+1B
.text:080484E0             add    eax, [edx]
.text:080484E2             add    edx, 4
.text:080484E5             cmp    edx, offset __libc_start_main@@GLIBC_2_0
.text:080484EB             jnz    short loc_80484E0
.text:080484ED             rep    retn
.text:080484ED sum_of_a_v2    endp
```

Needless to say, this optimization is possible if the compiler can calculate address of the end of array during compilation time. This happens if the array is global and its size is fixed.

However, if the address of array is unknown during compilation, but size is fixed, address of the label just behind array's end can be calculated at the beginning of the loop.

## 3.24 More about structures

### 3.24.1 Sometimes a C structure can be used instead of array

#### Arithmetic mean

```
#include <stdio.h>

int mean(int *a, int len)
{
    int sum=0;
    for (int i=0; i<len; i++)
        sum=sum+a[i];
    return sum/len;
};

struct five_ints
{
    int a0;
    int a1;
    int a2;
    int a3;
    int a4;
};

int main()
{
    struct five_ints a;
    a.a0=123;
    a.a1=456;
    a.a2=789;
    a.a3=10;
    a.a4=100;
    printf ("%d\n", mean(&a, 5));
    // test: https://www.wolframalpha.com/input/?i=mean(123,456,789,10,100)
};
```

This works: *mean()* function will never access behind the end of *five\_ints* structure, because 5 is passed, meaning, only 5 integers will be accessed.

#### Putting string into structure

```
#include <stdio.h>

struct five_chars
{
    char a0;
    char a1;
    char a2;
    char a3;
    char a4;
} __attribute__ ((aligned (1),packed));

int main()
{
    struct five_chars a;
    a.a0='h';
    a.a1='i';
    a.a2='!';
    a.a3='\n';
    a.a4=0;
    printf (&a); // prints "hi!"
};
```

*((aligned (1),packed))* attribute must be used, because otherwise, each structure field will be aligned on 4-byte or 8-byte boundary.

**Summary**

This is just another example of how structures and arrays are stored in memory. Perhaps, no sane programmer will do something like in this example, except in case of some specific hack. Or maybe in case of source code obfuscation?

**3.24.2 Unsized array in C structure**

In some win32 structures we can find ones with last field defined as an array of one element:

```
typedef struct _SYMBOL_INFO {
    ULONG    SizeOfStruct;
    ULONG    TypeIndex;

    ...
    ULONG    MaxNameLen;
    TCHAR    Name[1];
} SYMBOL_INFO, *PSYMBOL_INFO;
```

([https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686(v=vs.85).aspx))

This is a hack, meaning, the last field is array of unknown size, which is to be calculated at the time of structure allocation.

Why: *Name* field may be short, so why to define it with some kind of *MAX\_NAME* constant which can be 128, 256, or even bigger?

Why not to use pointer instead? Then you have to allocate two blocks: one for structure and the other one for string. This may be slower and may require larger memory overhead. Also, you need dereference pointer (i.e., read address of the string from the structure)—not a big deal, but some people say this is still surplus cost.

This is also known as *struct hack*: <http://c-faq.com/struct/structhack.html>.

Example:

```
#include <stdio.h>

struct st
{
    int a;
    int b;
    char s[];
};

void f (struct st *s)
{
    printf ("%d %d %s\n", s->a, s->b, s->s);
    // f() can't replace s[] with bigger string - size of allocated block is unknown at ↴
    // this point
};

int main()
{
#define STRING "Hello!"
    struct st *s=malloc(sizeof(struct st)+strlen(STRING)+1); // incl. terminating zero
    s->a=1;
    s->b=2;
    strcpy (s->s, STRING);
    f(s);
}
```

In short, it works because C has no array boundary checks. Any array is treated as having infinite size.

Problem: after allocation, the whole size of allocated block for structure is unknown (except for memory manager), so you can't just replace string with larger string. You would still be able to do so if the field would be declared as something like *s[MAX\_NAME]*.

### 3.24. MORE ABOUT STRUCTURES

In other words, you have a structure plus an array (or string) fused together in the single allocated memory block. Another problem is what you obviously can't declare two such arrays in single structure, or to declare another field after such array.

Older compilers require to declare array with at least one element: `s[1]`, newer allows to declare it as variable-sized array: `s[]`. This is also called *flexible array member*<sup>44</sup> in C99 standard.

Read more about it in GCC documentation<sup>45</sup>, MSDN documentation<sup>46</sup>.

Dennis Ritchie (one of C creators) called this trick “unwarranted chumminess with the C implementation” (perhaps, acknowledging hackish nature of the trick).

Like it or not, use it or not: it is still another demonstration on how structures are stored in memory, that's why I write about it.

#### 3.24.3 Version of C structure

Many Windows programmers have seen this is MSDN:

```
SizeOfStruct  
The size of the structure, in bytes. This member must be set to sizeof(SYMBOL_INFO).
```

([https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686(v=vs.85).aspx))

Some structures like `SYMBOL_INFO` has started with this field indeed. Why? This is some kind of structure version.

Imagine you have a function which draws circle. It takes a single argument—a pointer to a structure with only two fields: X and Y. And then color displays flooded a market, sometimes in 1980s. And you want to add `color` argument to the function. But, let's say, you cannot add another argument to it (a lot of software use your API<sup>47</sup> and cannot be recompiled). And if the old piece of software uses your API with color display, let your function draw a circle in (default) black and white colors.

Another day you add another feature: circle now can be filled, and brush type can be set.

Here is one solution to the problem:

```
#include <stdio.h>  
  
struct ver1  
{  
    size_t SizeOfStruct;  
    int coord_X;  
    int coord_Y;  
};  
  
struct ver2  
{  
    size_t SizeOfStruct;  
    int coord_X;  
    int coord_Y;  
    int color;  
};  
  
struct ver3  
{  
    size_t SizeOfStruct;  
    int coord_X;  
    int coord_Y;  
    int color;  
    int fill_brush_type; // 0 - do not fill circle  
};  
  
void draw_circle(struct ver3 *s) // latest struct version is used here  
{  
    // we presume SizeOfStruct, coord_X and coord_Y fields are always present
```

<sup>44</sup>[https://en.wikipedia.org/wiki/Flexible\\_array\\_member](https://en.wikipedia.org/wiki/Flexible_array_member)

<sup>45</sup><https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html>

<sup>46</sup><https://msdn.microsoft.com/en-us/library/b6fae073.aspx>

<sup>47</sup>Application programming interface

### 3.24. MORE ABOUT STRUCTURES

```
printf ("We are going to draw a circle at %d:%d\n", s->coord_X, s->coord_Y);

if (s->SizeOfStruct>=sizeof(int)*4)
{
    // this is at least ver2, color field is present
    printf ("We are going to set color %d\n", s->color);
}

if (s->SizeOfStruct>=sizeof(int)*5)
{
    // this is at least ver3, fill_brush_type field is present
    printf ("We are going to fill it using brush type %d\n", s->fill_brush_type);
}
};

// early software version
void call_as_ver1()
{
    struct ver1 s;
    s.SizeOfStruct=sizeof(s);
    s.coord_X=123;
    s.coord_Y=456;
    printf ("** %s()\n", __FUNCTION__);
    draw_circle(&s);
};

// next software version
void call_as_ver2()
{
    struct ver2 s;
    s.SizeOfStruct=sizeof(s);
    s.coord_X=123;
    s.coord_Y=456;
    s.color=1;
    printf ("** %s()\n", __FUNCTION__);
    draw_circle(&s);
};

// latest, most advanced version
void call_as_ver3()
{
    struct ver3 s;
    s.SizeOfStruct=sizeof(s);
    s.coord_X=123;
    s.coord_Y=456;
    s.color=1;
    s.fill_brush_type=3;
    printf ("** %s()\n", __FUNCTION__);
    draw_circle(&s);
};

int main()
{
    call_as_ver1();
    call_as_ver2();
    call_as_ver3();
}
```

In other words, *SizeOfStruct* field takes a role of *version of structure* field. It could be enumerate type (1, 2, 3, etc.), but to set *SizeOfStruct* field to *sizeof(struct...)* is less prone to mistakes/bugs.

In C++, this problem is solved using *inheritance* ([3.19.1 on page 551](#)). You just extend your base class (let's call it *Circle*), and then you will have *ColoredCircle* and then *FilledColoredCircle*, and so on. A current *version* of an object (or, more precisely, *current type*) will be determined using C++ [RTTI](#).

So when you see *SizeOfStruct* somewhere in [MSDN](#)—perhaps this structure was extended at least once in past.

### 3.24.4 High-score file in “Block out” game and primitive serialization

Many videogames has high-score file, sometimes called “Hall of fame”. Ancient “Block out”<sup>48</sup> game (3D tetris from 1989) isn’t exception, here is what we see at the end:

*** H A L L O F F A M E ***			
Pit: 7x7x18, Block Set: FLAT			
1.	Irene . . . . .	10970	(08-12-2016)
2.	Saddler . . . . .	7819	(08-12-2016)
3.	Mary . . . . .	300	(08-12-2016)
4.	James . . . . .	151	(08-12-2016)
5.	Mike . . . . .	135	(08-12-2016)
6.	A A A A A A A A A A	135	(08-12-2016)
7.	Joe . . . . .	130	(08-12-2016)
8.	John . . . . .	128	(08-12-2016)
9.	Doe . . . . .	124	(08-12-2016)
10.		122	(08-12-2016)

Figure 3.4: High score table

Now we can see that the file has changed after we added our name is *BLSCORE.DAT*. Let’s take a look on it in Midnight Commander:

/home/i/tmp/BLSCORE.DAT.10				0x00000000	0%
00000000	DA	00 49 72	65 6E 65 2E	2E 2E 2E 2E	00 DA 2A 00 .Irene.....*
00000010	00	30 38 2D	31 32 2D 32	30 31 36 00	53 61 64 64 .08-12-2016.Sadd
00000020	6C	65 72 2E	2E 2E 00 88	1E 00 00 30	38 2D 31 32 ler.....08-12
00000030	2D	32 30 31	36 00 4D 61	72 79 2E 2E	2E 2E 2E 2E -2016.Mary.....
00000040	00	2C 01 00	00 30 38 2D	31 32 2D 32	30 31 36 00 .....08-12-2016.
00000050	4A	61 6D 65	73 2E 2E 2E	2E 2E 00 97	00 00 00 30 James.....0
00000060	38	2D 31 32	2D 32 30 31	36 00 4D 69	6B 65 2E 2E 8-12-2016.Mike..
00000070	2E	2E 2E 2E	00 87 00 00	00 30 38 2D	31 32 2D 32 .....08-12-2
00000080	30	31 36 00	41 41 41 41	41 41 41 41	41 41 00 87 016.AAAAAAAA..
00000090	00	00 00 30	38 2D 31 32	2D 32 30 31	36 00 4A 6F ...08-12-2016.Jo
000000A0	65	2E 2E 2E	2E 2E 2E 2E	00 82 00 00	00 30 38 2D e.....08-
000000B0	31	32 2D 32	30 31 36 00	4A 6F 68 6E	2E 2E 2E 2E 12-2016.John....
000000C0	2E	2E 00 80	00 00 00 30	38 2D 31 32	2D 32 30 31 .....08-12-201
000000D0	36	00 44 6F	65 2E 2E 2E	2E 2E 2E 2E	00 7C 00 00 6.Doe.....!..
000000E0	00	30 38 2D	31 32 2D 32	30 31 36 00	41 6C 65 78 .08-12-2016.Alex
000000F0	2E	2E 2E 2E	2E 2E 00 78	00 00 00 30	38 2D 31 32 .....x...08-12
00000100	2D	32 30 31	36 00		-2016.

Figure 3.5: *BLSCORE.DAT* file in Midnight Commander

All entries are clearly visible. The very first byte is probably number of entries. Second is zero and, in fact, number of entries can be 16-bit value spanning over first two bytes.

Next, after “Irene” name we see 0xDA and 0x2A bytes. Irene has score 10970, and this is exactly 0x2ADA in hexadecimal radix. So a high score value is probably 16-bit integer, or maybe 32-bit integer: there are two more zero bytes after.

<sup>48</sup><http://www.bestoldgames.net/eng/old-games/blockout.php>

### 3.24. MORE ABOUT STRUCTURES

Now let's think about the fact that both array elements and structure elements are always placed in memory in adjacently to each other. That enables us to write the whole array/structure to the file using simple `write()` or `fwrite()` function, and then restore it using `read()` or `fread()`, as simple as that. This is what is called *serialization* nowadays.

#### Read

Now let's write C program to read highscore file:

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>

struct entry
{
    char name[11]; // incl. terminating zero
    uint32_t score;
    char date[11]; // incl. terminating zero
} __attribute__ ((aligned (1),packed));

struct highscore_file
{
    uint8_t count;
    uint8_t unknown;
    struct entry entries[10];
} __attribute__ ((aligned (1), packed));

struct highscore_file file;

int main(int argc, char* argv[])
{
    FILE* f=fopen(argv[1], "rb");
    assert (f!=NULL);
    size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);
    for (int i=0; i<file.count; i++)
    {
        printf ("name=%s score=%d date=%s\n",
               file.entries[i].name,
               file.entries[i].score,
               file.entries[i].date);
    };
}
```

We need GCC `((aligned (1),packed))` attribute so that all structure fields will be packed on 1-byte boundary.

Of course it works:

```
name=Irene..... score=10970 date=08-12-2016
name=Saddler... score=7819 date=08-12-2016
name=Mary..... score=300 date=08-12-2016
name=James..... score=151 date=08-12-2016
name=Mike..... score=135 date=08-12-2016
name=AAAAAAAAAA score=135 date=08-12-2016
name=Joe..... score=130 date=08-12-2016
name=John..... score=128 date=08-12-2016
name=Doe..... score=124 date=08-12-2016
name=A..... score=120 date=08-12-2016
```

(Needless to say, each name is padded with dots, both on screen and in the file, perhaps, for æsthetical reasons.)

#### Write

Let's check if we right about width of score value. Is it really has 32 bits?

### 3.24. MORE ABOUT STRUCTURES

```
int main(int argc, char* argv[])
{
    FILE* f=fopen(argv[1], "rb");
    assert (f!=NULL);
    size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);

    strcpy (file.entries[1].name, "Mallory...");
    file.entries[1].score=12345678;
    strcpy (file.entries[1].date, "08-12-2016");

    f=fopen(argv[1], "wb");
    assert (f!=NULL);
    got=fwrite(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);
}
```

Let's run Blockout:



Figure 3.6: High score table

First two digits (1 or 2) are choked. Perhaps, this is formatting issues... but the number is almost correct. Now I'm changing it to 999999 and run again:

*** H A L L   O F   F A M E ***			
Pit: 7x7x18, Block Set: FLAT			
1.	Irene . . . . .	10970	(08-12-2016)
2.	Mallory . . . . .	999999	(08-12-2016)
3.	Mary . . . . .	300	(08-12-2016)
4.	James . . . . .	151	(08-12-2016)
5.	Mike . . . . .	135	(08-12-2016)
6.	AAAAAAAAAAA	135	(08-12-2016)
7.	Joe . . . . .	130	(08-12-2016)
8.	John . . . . .	128	(08-12-2016)
9.		126	(08-12-2016)
10.	Doe . . . . .	124	(08-12-2016)

Figure 3.7: High score table

Now it's correct. Yes, high score value is 32-bit integer.

### Is it serialization?

...almost. Serialization like this is highly popular in scientific and engineering software, where efficiency and speed is much more important than converting into [XML<sup>49</sup>](#) or [JSON<sup>50</sup>](#) and back.

One important thing is that you obviously cannot serialize pointers, because each time you load the file into memory, all the structures may be allocated in different places.

But: if you work on some kind of low-cost [MCU](#) with simple [OS](#) on it and you have your structures allocated at always same places in memory, perhaps you can save and restore pointers as well.

### Random noise

When I prepared this example, I had to run "Block out" many times and played for it a bit to fill high-score table with random names.

And when there were just 3 entries in the file, I saw this:

<sup>49</sup>Extensible Markup Language

<sup>50</sup>JavaScript Object Notation

### 3.25. MEMMOVE() AND MEMCPY()

Address	Value	Content
00000000	03	00 49 72 65 6E 65 2E 2E 2E 2E 00 DA 2A 00 .Irene.....*
00000010	00	30 38 2D 31 32 2D 32 30 31 36 00 53 61 64 64 .08-12-2016.Sadd
00000020	6C	65 72 2E 2E 2E 00 0B 1E 00 00 30 38 2D 31 32 1er.....08-12
00000030	2D	32 30 31 36 00 4A 6F 68 6E 2E 2E 2E 2E 2E -2016.John.....
00000040	00	80 00 00 00 00 30 38 2D 31 32 2D 32 30 31 36 00 .....08-12-2016.
00000050	00	00 57 C8 A2 01 06 01 BA F9 47 C7 05 00 F8 4F .W8.....G....0
00000060	06	01 06 01 A6 32 00 00 00 00 00 00 00 00 00 00 00 00 .....2.....
00000070	00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....0.....
00000080	00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....0.....
00000090	00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....0.....
000000A0	00	00 00 00 00 00 00 00 93 C6 A2 01 46 72 .....0.....
000000B0	8C	F9 F6 C5 05 00 F8 4F 00 02 06 01 A6 32 06 01 .....0.....
000000C0	00	00 98 F9 F2 C0 05 00 F8 4F 00 02 A6 32 A2 F9 .....0.....
000000D0	80	C1 A6 32 A6 32 F4 4F AA F9 39 C1 A6 32 06 01 ...2.2.0..9..2..
000000E0	B4	F9 2B C5 A6 32 E1 4F C7 C8 A2 01 82 72 C6 F9 ..+T 2.0.8 ...r..
000000F0	30	C0 05 00 00 00 00 00 00 00 A6 32 D4 F9 76 2D 0.....2..v-
00000100	A6	32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .2....

1Help 2Edit 3Quit 4Ascii 5Goto 6Save 7HxSrch 8Parse 9Format 10Quit

Figure 3.8: BLSCORE.DAT file in Midnight Commander

The first byte has value of 3, meaning there are 3 entries. And there are 3 entries present. But then we see a random noise at the second half of file.

The noise is probably has its origins in uninitialized data. Perhaps, “Block out” allocated memory for 10 entries somewhere in [heap](#), where, obviously, some pseudorandom noise (left from something else) was present. Then it set first/second byte, fill 3 entries, and then it never touched 7 entries left, so they are written to the file as is.

When “Block out” loads high score file at the next run, it reads number of entries from the first/second byte (3) and then completely ignores what is after it.

This is common problem. Not a problem in strict sense: it’s not a bug, but information can be exposed outwards.

Microsoft Word versions from 1990s has been often left pieces of previously edited texts into the \*.doc\* files. It was some kind of amusement back then, to get a .doc file from someone, then open it in a hexadecimal editor and read something else, what has been edited on that computer before.

The problem can be even much more serious: Heartbleed bug<sup>51</sup> in OpenSSL.

## Homework

“Block out” has several polycubes (flat/basic/extended), size of pit can be configured, etc. And it seems, for each configuration, “Block out” has its own high score table. I’ve noticed that some information is probably stored in BLSCORE.IDX file. This can be a homework for hardcore “Block out” fans—to understand its structure as well.

The “Block out” files are here: <http://beginners.re/examples/blockout.zip> (including the binary high score files I’ve used in this example). You can use DosBox to run it.

## 3.25 memmove() and memcpy()

The difference between these standard functions is that *memcpy()* blindly copies a block to another place, while *memmove()* correctly handles overlapping blocks. For example, you want to tug a string two bytes forward:

<sup>51</sup><https://en.wikipedia.org/wiki/Heartbleed>

### 3.25. MEMMOVE() AND MEMCPY()

```
`|.|.|h|e|l|l|o|...` -> `|h|e|l|l|o|...`
```

`memcpy()` which copies 32-bit or 64-bit words at once, or even SIMD, will obviously fail here, a byte-wise copy routine must be used instead.

Now even more advanced example, insert two bytes in front of string:

```
`|h|e|l|l|o|...` -> `|.|.|h|e|l|l|o|...`
```

Now even byte-wise memory copy routine will fail, you have to copy bytes starting at the end.

That's a rare case where `DF` x86 flag is to be set before `REP MOVSB` instruction: `DF` defines direction, and now we must move backwardly.

The typical `memmove()` routine works like this: 1) if source is below destination, copy forward; 2) if source is above destination, copy backward.

This is `memmove()` from uClibc:

```
void *memmove(void *dest, const void *src, size_t n)
{
    int eax, ecx, esi, edi;
    __asm__ __volatile__(
        "    movl    %%eax, %%edi\n"
        "    cmpl    %%esi, %%eax\n"
        "    je     2f\n" /* (optional) src == dest -> NOP */
        "    jb     1f\n" /* src > dest -> simple copy */
        "    leal    -1(%%esi,%%ecx), %%esi\n"
        "    leal    -1(%%eax,%%ecx), %%edi\n"
        "    std\n"
        "1:   rep; movsb\n"
        "    cld\n"
        "2:\n"
        : "=c" (ecx), "=S" (esi), "=a" (eax), "=D" (edi)
        : "0" (n), "1" (src), "2" (dest)
        : "memory"
    );
    return (void*)eax;
}
```

In the first case, `REP MOVSB` is called with `DF` flag cleared. In the second, `DF` is set, then cleared.

More complex algorithm has the following piece in it:

"if difference between *source* and *destination* is larger than width of `word`, copy using words rather than bytes, and use byte-wise copy to copy unaligned parts".

This how it happens in Glibc 2.24 in non-optimized C part.

Given all that, `memmove()` may be slower than `memcpy()`. But some people, including Linus Torvalds, argue<sup>52</sup> that `memcpy()` should be an alias (or synonym) of `memmove()`, and the latter function must just check at start, if the buffers are overlapping or not, and then behave as `memcpy()` or `memmove()`. Nowadays, check for overlapping buffers is very cheap, after all.

#### 3.25.1 Anti-debugging trick

I've heard about anti-debugging trick where all you need is just set `DF` to crash the process: the very next `memcpy()` routine will lead to crash because it copies backwardly. But I can't check this: it seems all memory copy routines clears/sets `DF` as they want to. On the other hand, `memmove()` from uClibc I cited here, has no explicit clear of `DF` (it assumes `DF` is always clear?), so it can really crash.

<sup>52</sup>[https://bugzilla.redhat.com/show\\_bug.cgi?id=638477#c132](https://bugzilla.redhat.com/show_bug.cgi?id=638477#c132)

## 3.26 setjmp/longjmp

setjmp/longjmp is a mechanism in C which is very similar to throw/catch mechanism in C++ and other higher-level PLs. Here is an example from zlib:

```
...
/* return if bits() or decode() tries to read past available input */
if (setjmp(s.env) != 0)          /* if came back here via longjmp(), */
    err = 2;                    /* then skip decomp(), return error */
else
    err = decomp(&s); /* decompress */

...
/* load at least need bits into val */
val = s->bitbuf;
while (s->bitcnt < need) {
    if (s->left == 0) {
        s->left = s->infun(s->inhow, &(s->in));
        if (s->left == 0) longjmp(s->env, 1); /* out of input */

...
if (s->left == 0) {
    s->left = s->infun(s->inhow, &(s->in));
    if (s->left == 0) longjmp(s->env, 1); /* out of input */
...
```

( zlib/contrib/blast/blast.c )

Call to `setjmp()` saves current `PC`, `SP` and other registers into `env` structure, then it returns 0.

In case of error, `longjmp()` *teleporting* you into the point after right after `setjmp()` call, as if `setjmp()` call returned non-null value (which was passed to `longjmp()`). This reminds as `fork()` syscall in UNIX.

Now let's take a look on distilled example:

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void f2()
{
    printf ("%s() begin\n", __FUNCTION__);
    // something odd happened here
    longjmp (env, 1234);
    printf ("%s() end\n", __FUNCTION__);
};

void f1()
{
    printf ("%s() begin\n", __FUNCTION__);
    f2();
    printf ("%s() end\n", __FUNCTION__);
};

int main()
{
    int err=setjmp(env);
    if (err==0)
    {
        f1();
    }
    else
    {
        printf ("Error %d\n", err);
    };
}
```

### 3.26. SETJMP/LONGJMP

If we run it, we will see:

```
f1() begin  
f2() begin  
Error 1234
```

`jmp_buf` structure usually comes undocumented, to preserve forward compatibility.

Let's see how `setjmp()` implemented in MSVC 2013 x64:

```
...  
; RCX = address of jmp_buf  
  
mov      [rcx], rax  
mov      [rcx+8], rbx  
mov      [rcx+18h], rbp  
mov      [rcx+20h], rsi  
mov      [rcx+28h], rdi  
mov      [rcx+30h], r12  
mov      [rcx+38h], r13  
mov      [rcx+40h], r14  
mov      [rcx+48h], r15  
lea      r8, [rsp+arg_0]  
mov      [rcx+10h], r8  
mov      r8, [rsp+0]      ; get saved RA from stack  
mov      [rcx+50h], r8    ; save it  
stmxcsr dword ptr [rcx+58h]  
fnstcw  word ptr [rcx+5Ch]  
movdqa  xmmword ptr [rcx+60h], xmm6  
movdqa  xmmword ptr [rcx+70h], xmm7  
movdqa  xmmword ptr [rcx+80h], xmm8  
movdqa  xmmword ptr [rcx+90h], xmm9  
movdqa  xmmword ptr [rcx+0A0h], xmm10  
movdqa  xmmword ptr [rcx+0B0h], xmm11  
movdqa  xmmword ptr [rcx+0C0h], xmm12  
movdqa  xmmword ptr [rcx+0D0h], xmm13  
movdqa  xmmword ptr [rcx+0E0h], xmm14  
movdqa  xmmword ptr [rcx+0F0h], xmm15  
retn
```

It just populates `jmp_buf` structure with current values of almost all registers. Also, current value of [RA](#) is taken from the stack and saved in `jmp_buf`: it will be used as new value of [PC](#) in future.

Now `longjmp()`:

```
...  
; RCX = address of jmp_buf  
  
mov      rax, rdx  
mov      rbx, [rcx+8]  
mov      rsi, [rcx+20h]  
mov      rdi, [rcx+28h]  
mov      r12, [rcx+30h]  
mov      r13, [rcx+38h]  
mov      r14, [rcx+40h]  
mov      r15, [rcx+48h]  
ldmxcsr dword ptr [rcx+58h]  
fnclcx  
fldcw   word ptr [rcx+5Ch]  
movdqa  xmm6, xmmword ptr [rcx+60h]  
movdqa  xmm7, xmmword ptr [rcx+70h]  
movdqa  xmm8, xmmword ptr [rcx+80h]  
movdqa  xmm9, xmmword ptr [rcx+90h]  
movdqa  xmm10, xmmword ptr [rcx+0A0h]  
movdqa  xmm11, xmmword ptr [rcx+0B0h]  
movdqa  xmm12, xmmword ptr [rcx+0C0h]
```

### 3.27. OTHER WEIRD STACK HACKS

```
movdqa xmm13, xmmword ptr [rcx+0D0h]
movdqa xmm14, xmmword ptr [rcx+0E0h]
movdqa xmm15, xmmword ptr [rcx+0F0h]
mov rdx, [rcx+50h] ; get PC (RIP)
mov rbp, [rcx+18h]
mov rsp, [rcx+10h]
jmp rdx           ; jump to saved PC

...
```

It just restores (almost) all registers, takes RA from structure and jumps there. This effectively works as if setjmp() returned to caller. Also, RAX is set to be equal to the second argument of longjmp(). This works as if setjmp() returned non-zero value at first place.

As a side effect of SP restoration, all values in stack which has been set and used between setjmp() and longjmp() calls are just dropped. They will not be used anymore. Hence, longjmp() usually jumps backwards<sup>53</sup>.

This implies that, unlike in throw/catch mechanism in C++, no memory will be freed, no destructors will be called, etc. Hence, this technique sometimes can be dangerous. Nevertheless, it's still quite popular. It's still used in Oracle RDBMS.

It also has unexpected side-effect: if some buffer has been overflowed inside of a function (maybe due to remote attack), and a function wants to report error, and it calls longjmp(), overwritten stack part just gets unused.

As an exercise, you can try to understand, why not all registers are saved. Why XMM0-XMM5 and other registers are skipped?

## 3.27 Other weird stack hacks

### 3.27.1 Accessing arguments/local variables of caller

From C/C++ basics we know that this is impossible for a function to access arguments of caller function or its local variables.

Nevertheless, it's possible using dirty hacks. For example:

```
#include <stdio.h>

void f(char *text)
{
    // print stack
    int *tmp=&text;
    for (int i=0; i<20; i++)
    {
        printf ("0x%x\n", *tmp);
        tmp++;
    }
}

void draw_text(int X, int Y, char* text)
{
    f(text);

    printf ("We are going to draw [%s] at %d:%d\n", text, X, Y);
}

int main()
{
    printf ("address of main()=0x%x\n", &main);
    printf ("address of draw_text()=0x%x\n", &draw_text);
    draw_text(100, 200, "Hello!");
}
```

<sup>53</sup>However, there are some people who can use it for much more complicated things, imitating coroutines, etc: <https://www.embeddedrelated.com/showarticle/455.php>, <http://fanf.livejournal.com/105413.html>

### 3.27. OTHER WEIRD STACK HACKS

On 32-bit Ubuntu 16.04 and GCC 5.4.0, I got this:

```
address of main()=0x80484f8
address of draw_text()=0x80484cb
0x8048645      first argument to f()
0x8048628
0xbfd8ab98
0xb7634590
0xb779eddc
0xb77e4918
0xbfd8aba8
0x8048547      return address into the middle of main()
0x64           first argument to draw_text()
0xc8           second argument to draw_text()
0x8048645      third argument to draw_text()
0x8048581
0xb779d3dc
0xbfd8abc0
0x0
0xb7603637
0xb779d000
0xb779d000
0x0
0xb7603637
```

(Comments are mine.)

Since *f()* starting to enumerate stack elements at its first argument, the first stack element is indeed a pointer to "Hello!" string. We see its address is also used as third argument to *draw\_text()* function.

In *f()* we could read all functions arguments and local variables if we know exact stack layout, but it's always changed, from compiler to compiler. Various optimization levels affects stack layout greatly.

But if we can somehow detect information we need, we can use it and even modify it. As an example, I'll rework *f()* function:

```
void f(char *text)
{
    ...

    // find 100, 200 values pair and modify the second one
    tmp=&text;
    for (int i=0; i<20; i++)
    {
        if (*tmp==100 && *(tmp+1)==200)
        {
            printf ("found\n");
            *(tmp+1)=210; // change 200 to 210
            break;
        };
        tmp++;
    };
}
```

Holy moly, it works:

```
found
We are going to draw [Hello!] at 100:210
```

## Summary

It's extremely dirty hack, intended to demonstrate stack internals. I never ever seen or heard that anyone used this in a real code. But still, this is a good example.

## Exercise

### 3.27. OTHER WEIRD STACK HACKS

The example has been compiled without optimization on 32-bit Ubuntu using GCC 5.4.0 and it works. But when I turn on `-O3` maximum optimization, it's failed. Try to find why.

Use your favorite compiler and OS, try various optimization levels, find if it's works and if it's not, find why.

#### 3.27.2 Returning string

This is classic bug from Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999):

```
#include <stdio.h>

char* amsg(int n, char* s)
{
    char buf[100];

    sprintf (buf, "error %d: %s\n", n, s) ;

    return buf;
};

int main()
{
    amsg ("%s\n", interim (1234, "something wrong!"));
}
```

It would crash. First, let's understand, why.

This is a stack state before amsg() return:

```
[amsg(): 100 bytes][RA][two amsg arguments][something else][main() local variables]
                                         ^
                                         current SP
```

When amsg() returns control flow to `main()`, so far so good. But `printf()` is called from `main()`, which is, in turn, use stack for its own needs, zapping 100-byte buffer. A random garbage will be printed at the best.

Hard to believe, but I know how to fix this problem:

```
#include <stdio.h>

char* amsg(int n, char* s)
{
    char buf[100];

    sprintf (buf, "error %d: %s\n", n, s) ;

    return buf;
};

char* interim (int n, char* s)
{
    char large_buf[8000];
    // make use of local array.
    // it will be optimized away otherwise, as useless.
    large_buf[0]=0;
    return amsg (n, s);
};

int main()
{
    printf ("%s\n", interim (1234, "something wrong!"));
}
```

It will work if compiled by MSVC 2013 with no optimizations and with `/GS-` option<sup>54</sup>. MSVC will warn: "warning C4172: returning address of local variable or temporary", but the code will run and message will be printed. Let's see stack state at the moment when amsg() returns control to interim():

<sup>54</sup>Turn off buffer security check

### 3.28. OPENMP

```
[amsg(): 100 bytes][RA][two amsg() arguments][interim() stuff, incl. 8000 bytes][something else]
  ↴ ][main() local variables]
    ^
    current SP
```

Now the stack state at the moment when interim() returns control to `main()`:

```
[amsg(): 100 bytes][RA][two amsg() arguments][interim() stuff, incl. 8000 bytes][something else]
  ↴ ][main() local variables]
    ^
    current SP
```

So when `main()` calls `printf()`, it uses stack at the place where interim()'s buffer was allocated, and doesn't zap 100 bytes with error message inside, because 8000 bytes (or maybe much less) is just enough for everything `printf()` and other descending functions do!

It may also work if there are many functions between, like: `main() → f1() → f2() → f3() ... → amsg()`, and then the result of `amsg()` is used in `main()`. The distance between `SP` in `main()` and address of `buf[]` must be long enough,

This is why bugs like these are dangerous: sometimes your code works (and bug can be hiding unnoticed), sometimes not. Bugs like these are jokingly called heisenbugs or schrödinbugs<sup>55</sup>.

## 3.28 OpenMP

OpenMP is one of the simplest ways to parallelize simple algorithms.

As an example, let's try to build a program to compute a cryptographic *nonce*.

In my simplistic example, the *nonce* is a number added to the plain unencrypted text in order to produce a hash with some specific features.

For example, at some step, the Bitcoin protocol requires to find such *nonce* so the resulting hash contains a specific number of consecutive zeros. This is also called “proof of work”<sup>56</sup> (i.e., the system proves that it did some intensive calculations and spent some time for it).

My example is not related to Bitcoin in any way, it will try to add numbers to the “hello, world!\_” string in order to find such number that when “hello, world!\_<number>” is hashed with the SHA512 algorithm, it will contain at least 3 zero bytes.

Let's limit our brute-force to the interval in `0..INT32_MAX-1` (i.e., `0xFFFFFFFF` or 2147483646).

The algorithm is pretty straightforward:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;

#ifdef __GNUC__
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
```

<sup>55</sup><https://en.wikipedia.org/wiki/Heisenbug>

<sup>56</sup>wikipedia

```

{
    uint8_t buf[32];
    struct sha512_ctx ctx;
    uint8_t res[64];

    // update statistics
    int t=omp_get_thread_num();

    if (__min[t]==-1)
        __min[t]=nonce;
    if (__max[t]==-1)
        __max[t]=nonce;

    __min[t]=min(__min[t], nonce);
    __max[t]=max(__max[t], nonce);

    // idle if valid nonce found
    if (found)
        return;

    memset (buf, 0, sizeof(buf));
    sprintf (buf, "hello, world!_%d", nonce);

    sha512_init_ctx (&ctx);
    sha512_process_bytes (buf, strlen(buf), &ctx);
    sha512_finish_ctx (&ctx, &res);
    if (res[0]==0 && res[1]==0 && res[2]==0)
    {
        printf ("found (thread %d): [%s]. seconds spent=%d\n", t, buf, time(NULL)-start);
        found=1;
    };
    #pragma omp atomic
    checked++;

    #pragma omp critical
    if ((checked % 100000)==0)
        printf ("checked=%d\n", checked);
};

int main()
{
    int32_t i;
    int threads=omp_get_max_threads();
    printf ("threads=%d\n", threads);

    __min=(int32_t*)malloc(threads*sizeof(int32_t));
    __max=(int32_t*)malloc(threads*sizeof(int32_t));
    for (i=0; i<threads; i++)
        __min[i]=__max[i]=-1;

    start=time(NULL);

    #pragma omp parallel for
    for (i=0; i<INT32_MAX; i++)
        check_nonce (i);

    for (i=0; i<threads; i++)
        printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i, __min[i], i, __max[i]);
    free(__min); free(__max);
};

```

The `check_nonce()` function just adds a number to the string, hashes it with the SHA512 algorithm and checks for 3 zero bytes in the result.

A very important part of the code is:

```
#pragma omp parallel for
for (i=0; i<INT32_MAX; i++)
```

### 3.28. OPENMP

```
check_nonce (i);
```

Yes, that simple, without `#pragma` we just call `check_nonce()` for each number from 0 to `INT32_MAX` (`0xffffffff` or 2147483647). With `#pragma`, the compiler adds some special code which slices the loop interval into smaller ones, to run them on all CPU cores available <sup>57</sup>.

The example can be compiled <sup>58</sup> in MSVC 2012:

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

Or in GCC:

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

#### 3.28.1 MSVC

Now this is how MSVC 2012 generates the main loop:

Listing 3.126: MSVC 2012

```
push    OFFSET _main$omp$1
push    0
push    1
call    __vcomp_fork
add    esp, 16
```

All functions prefixed by `vcomp` are OpenMP-related and are stored in the `vcomp*.dll` file. So here a group of threads is started.

Let's take a look on `_main$omp$1`:

Listing 3.127: MSVC 2012

```
$T1 = -8      ; size = 4
$T2 = -4      ; size = 4
_main$omp$1 PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    ecx
    push    esi
    lea    eax, DWORD PTR $T2[ebp]
    push    eax
    lea    eax, DWORD PTR $T1[ebp]
    push    eax
    push    1
    push    1
    push    2147483646      ; 7fffffffH
    push    0
    call    __vcomp_for_static_simple_init
    mov     esi, DWORD PTR $T1[ebp]
    add    esp, 24
    jmp    SHORT $LN6@main$omp$1
$LL2@main$omp$1:
    push    esi
    call    _check_nonce
    pop    ecx
    inc    esi
$LN6@main$omp$1:
    cmp    esi, DWORD PTR $T2[ebp]
    jle    SHORT $LL2@main$omp$1
    call    __vcomp_for_static_end
    pop    esi
    leave
    ret    0
_main$omp$1 ENDP
```

<sup>57</sup>N.B.: This is intentionally simplest possible example, but in practice, the usage of OpenMP can be harder and more complex

<sup>58</sup>sha512.(c|h) and u64.h files can be taken from the OpenSSL library: <http://go.yurichev.com/17324>

### 3.28. OPENMP

This function is to be started  $n$  times in parallel, where  $n$  is the number of CPU cores.

`vcomp_for_static_simple_init()` calculates the interval for the `for()` construct for the current thread, depending on the current thread's number.

The loop's start and end values are stored in the `$T1` and `$T2` local variables. You may also notice `7fffffff` (or 2147483646) as an argument to the `vcomp_for_static_simple_init()` function—this is the number of iterations for the whole loop, to be divided evenly.

Then we see a new loop with a call to the `check_nonce()` function, which does all the work.

Let's also add some code at the beginning of the `check_nonce()` function to gather statistics about the arguments with which the function has been called.

This is what we see when we run it:

```
threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0xffffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7fffffff
```

Yes, the result is correct, the first 3 bytes are zeros:

```
C:\...\sha512sum test
000000f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403
df6e3fe6019f5764fc9975e505a7395fed780fee50eb38dd4c0279cb114672e2 *test
```

The running time is  $\approx 2.3$  seconds on 4-core Intel Xeon E3-1220 3.10 GHz. In the task manager we see 5 threads: 1 main thread + 4 more. No further optimizations are done to keep this example as small and clear as possible. But probably it can be done much faster. My CPU has 4 cores, that is why OpenMP started exactly 4 threads.

By looking at the statistics table we can clearly see how the loop has been sliced into 4 even parts. Oh well, almost even, if we don't consider the last bit.

There are also pragmas for [atomic operations](#).

Let's see how this code is compiled:

```
#pragma omp atomic
checked++;

#pragma omp critical
if ((checked % 100000)==0)
    printf ("checked=%d\n", checked);
```

Listing 3.128: MSVC 2012

```
push    edi
push    OFFSET _checked
call    __vcomp_atomic_add_i4
; Line 55
push    OFFSET _$vcomp$critsect$
call    __vcomp_enter_critsect
add    esp, 12
; Line 56
mov     ecx, DWORD PTR _checked
mov     eax, ecx
cdq
mov     esi, 100000      ; 000186a0H
idiv   esi
test   edx, edx
jne    SHORT $LN1@check_nonce
; Line 57
```

### 3.28. OPENMP

```
push    ecx
push    OFFSET ??_C@_0M@NPNHLI00@checked?$DN?$CFd?6?$AA@
call    _printf
pop    ecx
pop    ecx
$LNL1@check_nonc:
push    DWORD PTR _$vcomp$critsect$
call    __vcomp_leave_critsect
pop    ecx
```

As it turns out, the `vcomp_atomic_add_i4()` function in the `vcomp*.dll` is just a tiny function with the `LOCK XADD` instruction<sup>59</sup> in it.

`vcomp_enter_critsect()` eventually calling win32 API function `EnterCriticalSection()`<sup>60</sup>.

### 3.28.2 GCC

GCC 4.8.1 produces a program which shows exactly the same statistics table, so, GCC's implementation divides the loop in parts in the same fashion.

Listing 3.129: GCC 4.8.1

```
mov    edi, OFFSET FLAT:main._omp_fn.0
call   GOMP_parallel_start
mov    edi, 0
call   main._omp_fn.0
call   GOMP_parallel_end
```

Unlike MSVC's implementation, what GCC code does is to start 3 threads, and run the fourth in the current thread. So there are 4 threads instead of the 5 in MSVC.

Here is the `main._omp_fn.0` function:

Listing 3.130: GCC 4.8.1

```
main._omp_fn.0:
    push    rbp
    mov     rbp, rsp
    push    rbx
    sub    rsp, 40
    mov     QWORD PTR [rbp-40], rdi
    call   omp_get_num_threads
    mov     ebx, eax
    call   omp_get_thread_num
    mov     esi, eax
    mov     eax, 2147483647 ; 0x7FFFFFFF
    cdq
    idiv   ebx
    mov     ecx, eax
    mov     eax, 2147483647 ; 0x7FFFFFFF
    cdq
    idiv   ebx
    mov     eax, edx
    cmp     esi, eax
    jl     .L15
.L18:
    imul   esi, ecx
    mov     edx, esi
    add     eax, edx
    lea     ebx, [rax+rcx]
    cmp     eax, ebx
    jge    .L14
    mov     DWORD PTR [rbp-20], eax
```

<sup>59</sup>Read more about LOCK prefix: [.1.6 on page 1009](#)

<sup>60</sup>You can read more about critical sections here: [6.5.4 on page 786](#)

### 3.29. ANOTHER HEISENBUG

```
.L17:
    mov    eax, DWORD PTR [rbp-20]
    mov    edi, eax
    call   check_nonce
    add    DWORD PTR [rbp-20], 1
    cmp    DWORD PTR [rbp-20], ebx
    jl     .L17
    jmp   .L14

.L15:
    mov    eax, 0
    add    ecx, 1
    jmp   .L18

.L14:
    add    rsp, 40
    pop    rbx
    pop    rbp
    ret
```

Here we see the division clearly: by calling `omp_get_num_threads()` and `omp_get_thread_num()` we get the number of threads running, and also the current thread's number, and then determine the loop's interval. Then we run `check_nonce()`.

GCC also inserted the `LOCK ADD`

instruction right in the code, unlike MSVC, which generated a call to a separate DLL function:

Listing 3.131: GCC 4.8.1

```
lock add      DWORD PTR checked[rip], 1
call  GOMP_critical_start
mov   ecx, DWORD PTR checked[rip]
mov   edx, 351843721
mov   eax, ecx
imul  edx
sar   edx, 13
mov   eax, ecx
sar   eax, 31
sub   edx, eax
mov   eax, edx
imul  eax, eax, 100000
sub   ecx, eax
mov   eax, ecx
test  eax, eax
jne   .L7
mov   eax, DWORD PTR checked[rip]
mov   esi, eax
mov   edi, OFFSET FLAT:.LC2 ; "checked=%d\n"
mov   eax, 0
call  printf
.L7:
call  GOMP_critical_end
```

The functions prefixed with GOMP are from GNU OpenMP library. Unlike vcomp\*.dll, its source code is freely available: [GitHub](#).

## 3.29 Another heisenbug

Sometimes, array (or buffer) can overflow due to *fencepost error*:

```
#include <stdio.h>

int array1[128];
int important_var1;
int important_var2;
int important_var3;
int important_var4;
```

### 3.30. WINDOWS 16-BIT

```
int important_var5;

int main()
{
    important_var1=1;
    important_var2=2;
    important_var3=3;
    important_var4=4;
    important_var5=5;

    array1[0]=123;
    array1[128]=456; // BUG

    printf ("important_var1=%d\n", important_var1);
    printf ("important_var2=%d\n", important_var2);
    printf ("important_var3=%d\n", important_var3);
    printf ("important_var4=%d\n", important_var4);
    printf ("important_var5=%d\n", important_var5);
}
```

This is what this program printed in my case (non-optimized GCC 5.4 x86 on Linux):

```
important_var1=1
important_var2=456
important_var3=3
important_var4=4
important_var5=5
```

As it happens, `important_var2` has been placed by compiler right after `array1[]`:

Listing 3.132: objdump -x

```
0804a040 g 0 .bss 00000200           array1
...
0804a240 g 0 .bss 00000004           important_var2
0804a244 g 0 .bss 00000004           important_var4
...
0804a248 g 0 .bss 00000004           important_var1
0804a24c g 0 .bss 00000004           important_var3
0804a250 g 0 .bss 00000004           important_var5
```

Other compiler can arrange variables in another order, and another variable would be zapped. This is also *heisenbug* ([3.27.2 on page 648](#))—bug may appear or may left unnoticed depending on compiler version and optimization switches.

If all variables and arrays are allocated in local stack, stack protection may be triggered, or may not. However, Valgrind can find bugs like these.

## 3.30 Windows 16-bit

16-bit Windows programs are rare nowadays, but can be used in the cases of retrocomputing or dongle hacking ([8.5 on page 814](#)).

16-bit Windows versions were up to 3.11. 95/98/ME also support 16-bit code, as well as the 32-bit versions of the [Windows NT](#) line. The 64-bit versions of [Windows NT](#) line do not support 16-bit executable code at all.

The code resembles MS-DOS's one.

Executable files are of type NE-type (so-called “new executable”).

All examples considered here were compiled by the OpenWatcom 1.9 compiler, using these switches:

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c
```

**3.30.1 Example#1**

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    MessageBeep(MB_ICONEXCLAMATION);
    return 0;
};
```

```
WinMain    proc near
    push    bp
    mov     bp, sp
    mov     ax, 30h ; '0'    ; MB_ICONEXCLAMATION constant
    push    ax
    call    MESSAGEBEEP
    xor    ax, ax        ; return 0
    pop    bp
    retn   0Ah
WinMain    endp
```

Seems to be easy, so far.

**3.30.2 Example #2**

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
    return 0;
};
```

```
WinMain    proc near
    push    bp
    mov     bp, sp
    xor    ax, ax        ; NULL
    push    ax
    push    ds
    mov     ax, offset aHelloWorld ; 0x18. "hello, world"
    push    ax
    push    ds
    mov     ax, offset aCaption ; 0x10. "caption"
    push    ax
    mov     ax, 3          ; MB_YESNOCANCEL
    push    ax
    call    MESSAGEBOX
    xor    ax, ax        ; return 0
    pop    bp
    retn   0Ah
WinMain    endp

dseg02:0010 aCaption      db 'caption',0
dseg02:0018 aHelloWorld   db 'hello, world',0
```

Couple important things here: the **PASCAL** calling convention dictates passing the first argument first (**MB\_YESNOCANCEL**), and the last argument—last (NULL). This convention also tells the **callee** to restore the **stack pointer**: hence the **RETN** instruction has **0Ah** as argument, which implies that the pointer

### 3.30. WINDOWS 16-BIT

has to be increased by 10 bytes when the function exits. It is like stdcall ( [6.1.2 on page 733](#)), but the arguments are passed in "natural" order.

The pointers are passed in pairs: first the data segment is passed, then the pointer inside the segment. There is only one segment in this example, so `DS` always points to the data segment of the executable.

#### 3.30.3 Example #3

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);

    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);

    return 0;
};
```

```
WinMain    proc near
    push    bp
    mov     bp, sp
    xor     ax, ax      ; NULL
    push    ax
    push    ds
    mov     ax, offset aHelloWorld ; "hello, world"
    push    ax
    push    ds
    mov     ax, offset aCaption ; "caption"
    push    ax
    mov     ax, 3          ; MB_YESNOCANCEL
    push    ax
    call    MESSAGEBOX
    cmp     ax, 2          ; IDCANCEL
    jnz    short loc_2F
    xor     ax, ax
    push    ax
    push    ds
    mov     ax, offset aYouPressedCanc ; "you pressed cancel"
    jmp    short loc_49

loc_2F:
    cmp     ax, 6          ; IDYES
    jnz    short loc_3D
    xor     ax, ax
    push    ax
    push    ds
    mov     ax, offset aYouPressedYes ; "you pressed yes"
    jmp    short loc_49

loc_3D:
    cmp     ax, 7          ; IDNO
    jnz    short loc_57
    xor     ax, ax
    push    ax
    push    ds
    mov     ax, offset aYouPressedNo ; "you pressed no"

loc_49:
    push    ax
    push    ds
    mov     ax, offset aCaption ; "caption"
    push    ax
```

### 3.30. WINDOWS 16-BIT

```
xor    ax, ax
push   ax
call   MESSAGEBOX
loc_57:
xor    ax, ax
pop    bp
retn   0Ah
WinMain endp
```

Somewhat extended example from the previous section.

#### 3.30.4 Example #4

```
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
    return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
    return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
    return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    func1 (123, 456, 789);
    func2 (600000, 700000, 800000);
    func3 (600000, 700000, 800000, 123);
    return 0;
};
```

```
func1      proc near
c          = word ptr 4
b          = word ptr 6
a          = word ptr 8

        push    bp
        mov     bp, sp
        mov     ax, [bp+a]
        imul   [bp+b]
        add    ax, [bp+c]
        pop    bp
        retn   6
func1      endp

func2      proc near
arg_0      = word ptr 4
arg_2      = word ptr 6
arg_4      = word ptr 8
arg_6      = word ptr 0Ah
arg_8      = word ptr 0Ch
arg_A      = word ptr 0Eh

        push    bp
        mov     bp, sp
```

### 3.30. WINDOWS 16-BIT

```
        mov    ax, [bp+arg_8]
        mov    dx, [bp+arg_A]
        mov    bx, [bp+arg_4]
        mov    cx, [bp+arg_6]
        call   sub_B2 ; long 32-bit multiplication
        add    ax, [bp+arg_0]
        adc    dx, [bp+arg_2]
        pop    bp
        retn  12
func2
endp

func3      proc near

arg_0       = word ptr 4
arg_2       = word ptr 6
arg_4       = word ptr 8
arg_6       = word ptr 0Ah
arg_8       = word ptr 0Ch
arg_A       = word ptr 0Eh
arg_C       = word ptr 10h

        push   bp
        mov    bp, sp
        mov    ax, [bp+arg_A]
        mov    dx, [bp+arg_C]
        mov    bx, [bp+arg_6]
        mov    cx, [bp+arg_8]
        call   sub_B2 ; long 32-bit multiplication
        mov    cx, [bp+arg_2]
        add    cx, ax
        mov    bx, [bp+arg_4]
        adc    bx, dx      ; BX=high part, CX=low part
        mov    ax, [bp+arg_0]
        cwd
        mov    ax, cx      ; AX=low part d, DX=high part d
        sub    cx, ax
        mov    ax, cx
        sbb    bx, dx
        mov    dx, bx
        pop    bp
        retn  14
func3
endp

WinMain    proc near
        push   bp
        mov    bp, sp
        mov    ax, 123
        push   ax
        mov    ax, 456
        push   ax
        mov    ax, 789
        push   ax
        call   func1
        mov    ax, 9       ; high part of 600000
        push   ax
        mov    ax, 27C0h   ; low part of 600000
        push   ax
        mov    ax, 0Ah     ; high part of 700000
        push   ax
        mov    ax, 0AE60h  ; low part of 700000
        push   ax
        mov    ax, 0Ch     ; high part of 800000
        push   ax
        mov    ax, 3500h   ; low part of 800000
        push   ax
        call   func2
        mov    ax, 9       ; high part of 600000
        push   ax
        mov    ax, 27C0h   ; low part of 600000
        push   ax
        mov    ax, 0Ah     ; high part of 700000
```

### 3.30. WINDOWS 16-BIT

```
push    ax
mov     ax, 0AE60h ; low part of 700000
push    ax
mov     ax, 0Ch      ; high part of 800000
push    ax
mov     ax, 3500h   ; low part of 800000
push    ax
mov     ax, 7Bh      ; 123
push    ax
call    func3
xor    ax, ax      ; return 0
pop    bp
retn    0Ah
WinMain
endp
```

32-bit values (the `long` data type implies 32 bits, while `int` is 16-bit) in 16-bit code (both MS-DOS and Win16) are passed in pairs. It is just like when 64-bit values are used in a 32-bit environment ([1.28 on page 397](#)).

`sub_B2` here is a library function written by the compiler's developers that does "long multiplication", i.e., multiplies two 32-bit values. Other compiler functions that do the same are listed here: [.5 on page 1026](#), [.4 on page 1026](#).

The `ADD / ADC` instruction pair is used for addition of compound values: `ADD` may set/clear the `CF` flag, and `ADC` uses it after.

The `SUB / SBB` instruction pair is used for subtraction: `SUB` may set/clear the `CF` flag, `SBB` uses it after. 32-bit values are returned from functions in the `DX:AX` register pair.

Constants are also passed in pairs in `WinMain()` here.

The `int`-typed 123 constant is first converted according to its sign into a 32-bit value using the `CWD` instruction.

#### 3.30.5 Example #5

```
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};
```

### 3.30. WINDOWS 16-BIT

```
void PASCAL remove_digits (char *s)
{
    while (*s)
    {
        if (*s>='0' && *s<='9')
            *s='-';
        s++;
    };
}

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    string_compare ("asd", "def");
    string_compare_far ("asd", "def");
    remove_digits (str);
    MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
    return 0;
}
```

```
string_compare proc near

arg_0 = word ptr 4
arg_2 = word ptr 6

    push    bp
    mov     bp, sp
    push    si
    mov     si, [bp+arg_0]
    mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
    mov     al, [bx]
    cmp     al, [si]
    jz      short loc_1C
    xor     ax, ax
    jmp     short loc_2B

loc_1C: ; CODE XREF: string_compare+Ej
    test   al, al
    jz      short loc_22
    jnz    short loc_27

loc_22: ; CODE XREF: string_compare+16j
    mov     ax, 1
    jmp     short loc_2B

loc_27: ; CODE XREF: string_compare+18j
    inc     bx
    inc     si
    jmp     short loc_12

loc_2B: ; CODE XREF: string_compare+12j
    ; string_compare+1Dj
    pop     si
    pop     bp
    retn   4
string_compare endp

string_compare_far proc near ; CODE XREF: WinMain+18p

arg_0 = word ptr 4
arg_2 = word ptr 6
```

### 3.30. WINDOWS 16-BIT

```
arg_4 = word ptr 8
arg_6 = word ptr 0Ah

push    bp
mov     bp, sp
push    si
mov     si, [bp+arg_0]
mov     bx, [bp+arg_4]

loc_3A: ; CODE XREF: string_compare_far+35j
    mov     es, [bp+arg_6]
    mov     al, es:[bx]
    mov     es, [bp+arg_2]
    cmp     al, es:[si]
    jz      short loc_4C
    xor     ax, ax
    jmp     short loc_67

loc_4C: ; CODE XREF: string_compare_far+16j
    mov     es, [bp+arg_6]
    cmp     byte ptr es:[bx], 0
    jz      short loc_5E
    mov     es, [bp+arg_2]
    cmp     byte ptr es:[si], 0
    jnz     short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
    mov     ax, 1
    jmp     short loc_67

loc_63: ; CODE XREF: string_compare_far+2Cj
    inc     bx
    inc     si
    jmp     short loc_3A

loc_67: ; CODE XREF: string_compare_far+1Aj
    ; string_compare_far+31j
    pop     si
    pop     bp
    retn    8
string_compare_far endp

remove_digits proc near ; CODE XREF: WinMain+1Fp
arg_0 = word ptr 4

push    bp
mov     bp, sp
mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
    mov     al, [bx]
    test    al, al
    jz      short loc_86
    cmp     al, 30h ; '0'
    jb     short loc_83
    cmp     al, 39h ; '9'
    ja     short loc_83
    mov     byte ptr [bx], 2Dh ; '-'

loc_83: ; CODE XREF: remove_digits+Ej
    ; remove_digits+12j
    inc     bx
    jmp     short loc_72

loc_86: ; CODE XREF: remove_digits+Aj
```

### 3.30. WINDOWS 16-BIT

```
pop      bp
retn    2
remove_digits    endp

WinMain proc near ; CODE XREF: start+EDp
    push    bp
    mov     bp, sp
    mov     ax, offset aAsd ; "asd"
    push    ax
    mov     ax, offset aDef ; "def"
    push    ax
    call   string_compare
    push    ds
    mov     ax, offset aAsd ; "asd"
    push    ax
    push    ds
    mov     ax, offset aDef ; "def"
    push    ax
    call   string_compare_far
    mov     ax, offset aHello1234World ; "hello 1234 world"
    push    ax
    call   remove_digits
    xor    ax, ax
    push    ax
    push    ds
    mov     ax, offset aHello1234World ; "hello 1234 world"
    push    ax
    push    ds
    mov     ax, offset aCaption ; "caption"
    push    ax
    mov     ax, 3 ; MB_YESNOCANCEL
    push    ax
    call   MESSAGEBOX
    xor    ax, ax
    pop    bp
    retn    0Ah
WinMain endp
```

Here we see a difference between the so-called “near” pointers and the “far” pointers: another weird artifact of segmented memory in 16-bit 8086.

You can read more about it here: [10.6 on page 992](#).

“near” pointers are those which point within the current data segment. Hence, the `string_compare()` function takes only two 16-bit pointers, and accesses the data from the segment that `DS` points to (The `mov al, [bx]` instruction actually works like `mov al, ds:[bx]` — `DS` is implicit here).

“far” pointers are those which may point to data in another memory segment.

Hence `string_compare_far()` takes the 16-bit pair as a pointer, loads the high part of it in the `ES` segment register and accesses the data through it (`mov al, es:[bx]`). “far” pointers are also used in my

`MessageBox()` win16 example: [3.30.2 on page 655](#). Indeed, the Windows kernel is not aware which data segment to use when accessing text strings, so it need the complete information.

The reason for this distinction is that a compact program may use just one 64kb data segment, so it doesn’t need to pass the high part of the address, which is always the same. A bigger program may use several 64kb data segments, so it needs to specify the segment of the data each time.

It’s the same story for code segments. A compact program may have all executable code within one 64kb-segment, then all functions in it will be called using the `CALL NEAR` instruction, and the code flow will be returned using `RETN`. But if there are several code segments, then the address of the function is to be specified by a pair, it is to be called using the `CALL FAR` instruction, and the code flow is to be returned using `RETF`.

This is what is set in the compiler by specifying “memory model”.

The compilers targeting MS-DOS and Win16 have specific libraries for each memory model: they differ by pointer types for code and data.

**3.30.6 Example #6**

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    struct tm *t;
    time_t unix_time;

    unix_time=time(NULL);

    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->tm_mday,
             t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};
```

```
WinMain    proc near

var_4      = word ptr -4
var_2      = word ptr -2

    push    bp
    mov     bp, sp
    push    ax
    push    ax
    xor    ax, ax
    call   time_
    mov     [bp+var_4], ax    ; low part of UNIX time
    mov     [bp+var_2], dx    ; high part of UNIX time
    lea     ax, [bp+var_4]    ; take a pointer of high part
    call   localtime_
    mov     bx, ax            ; t
    push    word ptr [bx]     ; second
    push    word ptr [bx+2]    ; minute
    push    word ptr [bx+4]    ; hour
    push    word ptr [bx+6]    ; day
    push    word ptr [bx+8]    ; month
    mov     ax, [bx+0Ah]       ; year
    add    ax, 1900
    push    ax
    mov     ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
    push    ax
    mov     ax, offset strbuf
    push    ax
    call   sprintf_
    add    sp, 10h
    xor    ax, ax            ; NULL
    push    ax
    push    ds
    mov     ax, offset strbuf
    push    ax
    push    ds
    mov     ax, offset aCaption ; "caption"
    push    ax
    xor    ax, ax            ; MB_OK
    push    ax
```

### 3.30. WINDOWS 16-BIT

```
call    MESSAGEBOX
xor    ax, ax
mov    sp, bp
pop    bp
retn   0Ah
WinMain endp
```

UNIX time is a 32-bit value, so it is returned in the `DX:AX` register pair and stored in two local 16-bit variables. Then a pointer to the pair is passed to the `localtime()` function. The `localtime()` function has a `struct tm` allocated somewhere in the guts of the C library, so only a pointer to it is returned.

By the way, this also implies that the function cannot be called again until its results are used.

For the `time()` and `localtime()` functions, a Watcom calling convention is used here: the first four arguments are passed in the `AX`, `DX`, `BX` and `CX`, registers, and the rest arguments are via the stack.

The functions using this convention are also marked by underscore at the end of their name.

`sprintf()` does not use the `PASCAL` calling convention, nor the Watcom one, so the arguments are passed in the normal `cdecl` way ([6.1.1 on page 733](#)).

### Global variables

This is the same example, but now these variables are global:

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    unix_time=time(NULL);

    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->tm_mday,
             t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};
```

```
unix_time_low    dw 0
unix_time_high   dw 0
t                dw 0

WinMain         proc near
    push    bp
    mov     bp, sp
    xor    ax, ax
    call   time_
    mov    unix_time_low, ax
    mov    unix_time_high, dx
    mov    ax, offset unix_time_low
    call   localtime_
    mov    bx, ax
    mov    t, ax           ; will not be used in future...
    push   word ptr [bx]   ; seconds
```

### 3.30. WINDOWS 16-BIT

```
push    word ptr [bx+2]      ; minutes
push    word ptr [bx+4]      ; hour
push    word ptr [bx+6]      ; day
push    word ptr [bx+8]      ; month
mov     ax, [bx+0Ah]        ; year
add     ax, 1900
push    ax
mov     ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
push    ax
mov     ax, offset strbuf
push    ax
call   sprintf_
add    sp, 10h
xor    ax, ax      ; NULL
push    ax
push    ds
mov     ax, offset strbuf
push    ax
push    ds
mov     ax, offset aCaption ; "caption"
push    ax
xor    ax, ax      ; MB_OK
push    ax
call   MESSAGEBOX
xor    ax, ax      ; return 0
pop    bp
retn  0Ah
WinMain endp
```

t is not to be used, but the compiler emitted the code which stores the value.

Because it is not sure, maybe that value will eventually be used in some other module.

# Chapter 4

## Java

### 4.1 Java

#### 4.1.1 Introduction

There are some well-known decompilers for Java (or [JVM](#) bytecode in general) <sup>1</sup>.

The reason is the decompilation of [JVM](#)-bytecode is somewhat easier than for lower level x86 code:

- There is much more information about the data types.
- The [JVM](#) memory model is much more rigorous and outlined.
- The Java compiler don't do any optimizations (the [JVM JIT](#)<sup>2</sup> does them at runtime), so the bytecode in the class files is usually pretty readable.

When can the knowledge of [JVM](#) be useful?

- Quick-and-dirty patching tasks of class files without the need to recompile the decompiler's results.
- Analyzing obfuscated code.
- Building your own obfuscator.
- Building a compiler codegenerator (back-end) targeting [JVM](#) (like Scala, Clojure, etc. <sup>3</sup>).

Let's start with some simple pieces of code. JDK 1.7 is used everywhere, unless mentioned otherwise.

This is the command used to decompile class files everywhere:

```
javap -c -verbose .
```

This is the book I used while preparing all examples: [Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] <sup>4</sup>.

#### 4.1.2 Returning a value

Probably the simplest Java function is the one which returns some value.

Oh, and we must keep in mind that there are no "free" functions in Java in common sense, they are "methods".

Each method is related to some class, so it's not possible to define a method outside of a class.

But we'll call them "functions" anyway, for simplicity.

```
public class ret
{
    public static int main(String[] args)
    {
```

<sup>1</sup>For example, JAD: <http://varaneckas.com/jad/>

<sup>2</sup>Just-in-time compilation

<sup>3</sup>Full list: [http://en.wikipedia.org/wiki/List\\_of\\_JVM\\_languages](http://en.wikipedia.org/wiki/List_of_JVM_languages)

<sup>4</sup>Also available as <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>; <http://docs.oracle.com/javase/specs/jvms/se7/html/>

#### 4.1. JAVA

```
        return 0;
    }
}
```

Let's compile it:

```
javac ret.java
```

...and decompile it using the standard Java utility:

```
javap -c -verbose ret.class
```

And we get:

Listing 4.1: JDK 1.7 (excerpt)

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
      0: iconst_0
      1: ireturn
```

The Java developers decided that 0 is one of the busiest constants in programming, so there is a separate short one-byte `iconst_0` instruction which pushes 0

<sup>5</sup>. There are also `iconst_1` (which pushes 1), `iconst_2`, etc., up to `iconst_5`.

There is also `iconst_m1` which pushes -1.

The stack is used in JVM for passing data to called functions and also for return values. So `iconst_0` pushes 0 into the stack. `ireturn` returns an integer value (*i* in name means *integer*) from the TOS<sup>6</sup>.

Let's rewrite our example slightly, now we return 1234:

```
public class ret
{
    public static int main(String[] args)
    {
        return 1234;
    }
}
```

...we get:

Listing 4.2: JDK 1.7 (excerpt)

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
      0: sipush      1234
      3: ireturn
```

`sipush` (*short integer*) pushes 1234 into the stack. *short* in name implies a 16-bit value is to be pushed. The number 1234 indeed fits well in a 16-bit value.

What about larger values?

```
public class ret
{
    public static int main(String[] args)
    {
        return 12345678;
    }
}
```

<sup>5</sup>Just like in MIPS, where a separate register for zero constant exists: [1.5.5 on page 26](#).

<sup>6</sup>Top Of Stack

## 4.1. JAVA

Listing 4.3: Constant pool

```
...
#2 = Integer          12345678
...
public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: ldc             #2           // int 12345678
2: ireturn
```

It's not possible to encode a 32-bit number in a JVM instruction opcode, the developers didn't leave such possibility.

So the 32-bit number 12345678 is stored in so called "constant pool" which is, let's say, the library of most used constants (including strings, objects, etc.).

This way of passing constants is not unique to JVM.

MIPS, ARM and other RISC CPUs also can't encode a 32-bit number in a 32-bit opcode, so the RISC CPU code (including MIPS and ARM) has to construct the value in several steps, or to keep it in the data segment: [1.32.3 on page 442](#), [1.33.1 on page 445](#).

MIPS code also traditionally has a constant pool, named "literal pool", the segments are called ".lit4" (for 32-bit single precision floating point number constants) and ".lit8" (for 64-bit double precision floating point number constants).

Let's try some other data types!

Boolean:

```
public class ret
{
    public static boolean main(String[] args)
    {
        return true;
    }
}
```

```
public static boolean main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iconst_1
1: ireturn
```

This JVM bytecode is no different from one returning integer 1.

32-bit data slots in the stack are also used here for boolean values, like in C/C++.

But one could not use returned boolean value as integer or vice versa — type information is stored in the class file and checked at runtime.

It's the same story with a 16-bit *short*:

```
public class ret
{
    public static short main(String[] args)
    {
        return 1234;
    }
}
```

```
public static short main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: sipush      1234
3: ireturn
```

#### 4.1. JAVA

...and *char*!

```
public class ret
{
    public static char main(String[] args)
    {
        return 'A';
    }
}
```

```
public static char main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: bipush      65
2: ireturn
```

`bipush` means “push byte”. Needless to say that a *char* in Java is 16-bit UTF-16 character, and it’s equivalent to *short*, but the ASCII code of the “A” character is 65, and it’s possible to use the instruction for pushing a byte in the stack.

Let’s also try a *byte*:

```
public class retc
{
    public static byte main(String[] args)
    {
        return 123;
    }
}
```

```
public static byte main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: bipush      123
2: ireturn
```

One may ask, why bother with a 16-bit *short* data type which internally works as a 32-bit integer?

Why use a *char* data type if it is the same as a *short* data type?

The answer is simple: for data type control and source code readability.

A *char* may essentially be the same as a *short*, but we quickly grasp that it’s a placeholder for an UTF-16 character, and not for some other integer value.

When using *short*, we show everyone that the variable’s range is limited by 16 bits.

It’s a very good idea to use the *boolean* type where needed to, instead of the C-style *int*.

There is also a 64-bit integer data type in Java:

```
public class ret3
{
    public static long main(String[] args)
    {
        return 1234567890123456789L;
    }
}
```

Listing 4.4: Constant pool

```
...
#2 = Long          1234567890123456789L
...
```

```
public static long main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
```

#### 4.1. JAVA

```
0: ldc2_w          #2           // long 1234567890123456789l  
3: lreturn
```

The 64-bit number is also stored in a constant pool, `ldc2_w` loads it and `lreturn` (*long return*) returns it.

The `ldc2_w` instruction is also used to load double precision floating point numbers (which also occupy 64 bits) from a constant pool:

```
public class ret  
{  
    public static double main(String[] args)  
    {  
        return 123.456d;  
    }  
}
```

Listing 4.5: Constant pool

```
...  
#2 = Double      123.456d  
...  
  
public static double main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
stack=2, locals=1, args_size=1  
0: ldc2_w          #2           // double 123.456d  
3: dreturn
```

`dreturn` stands for “return double”.

And finally, a single precision floating point number:

```
public class ret  
{  
    public static float main(String[] args)  
    {  
        return 123.456f;  
    }  
}
```

Listing 4.6: Constant pool

```
...  
#2 = Float      123.456f  
...  
  
public static float main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
stack=1, locals=1, args_size=1  
0: ldc            #2           // float 123.456f  
2: freturn
```

The `ldc` instruction used here is the same one as for loading 32-bit integer numbers from a constant pool.

`freturn` stands for “return float”.

Now what about function that return nothing?

```
public class ret  
{  
    public static void main(String[] args)  
    {  
        return;  
    }  
}
```

## 4.1. JAVA

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=0, locals=1, args_size=1
      0: return
```

This means that the `return` instruction is used to return control without returning an actual value.

Knowing all this, it's very easy to deduce the function's (or method's) returning type from the last instruction.

### 4.1.3 Simple calculating functions

Let's continue with a simple calculating functions.

```
public class calc
{
    public static int half(int a)
    {
        return a/2;
}
```

Here's the output when the `iconst_2` instruction is used:

```
public static int half(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
      0: iload_0
      1: iconst_2
      2: idiv
      3: ireturn
```

`iload_0` takes the zeroth function argument and pushes it to the stack.

`iconst_2` pushes 2 in the stack. After the execution of these two instructions, this is how stack looks like:

```
+---+
TOS ->| 2 |
+---+
| a |
+---+
```

`idiv` just takes the two values at the `TOS`, divides one by the other and leaves the result at `TOS`:

```
+-----+
TOS ->| result |
+-----+
```

`ireturn` takes it and returns.

Let's proceed with double precision floating point numbers:

```
public class calc
{
    public static double half_double(double a)
    {
        return a/2.0;
}
```

#### 4.1. JAVA

Listing 4.7: Constant pool

```
...
#2 = Double          2.0d
...
public static double half_double(double);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=2, args_size=1
0: dload_0
1: ldc2_w           #2                  // double 2.0d
4: ddiv
5: dreturn
```

It's the same, but the `ldc2_w` instruction is used to load the constant 2.0 from the constant pool.

Also, the other three instructions have the *d* prefix, meaning they work with *double* data type values.

Let's now use a function with two arguments:

```
public class calc
{
    public static int sum(int a, int b)
    {
        return a+b;
    }
}
```

```
public static int sum(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
1: iload_1
2: iadd
3: ireturn
```

`iload_0` loads the first function argument (a), `iload_1` —second (b).

Here is the stack after the execution of both instructions:

```
+---+
TOS ->| b |
+---+
| a |
+---+
```

`iadd` adds the two values and leaves the result at **TOS**:

```
+-----+
TOS ->| result |
+-----+
```

Let's extend this example to the *long* data type:

```
public static long lsum(long a, long b)
{
    return a+b;
}
```

...we got:

```
public static long lsum(long, long);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=4, args_size=2
0: lload_0
1: lload_2
2: ladd
3: lreturn
```

#### 4.1. JAVA

The second `lload` instruction takes the second argument from the 2nd slot.

That's because a 64-bit *long* value occupies exactly two 32-bit slots.

Slightly more advanced example:

```
public class calc
{
    public static int mult_add(int a, int b, int c)
    {
        return a*b+c;
}
```

```
public static int mult_add(int, int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=3, args_size=3
0: iload_0
1: iload_1
2: imul
3: iload_2
4: iadd
5: ireturn
```

The first step is multiplication. The product is left at the [TOS](#):

```
+-----+
TOS ->| product |
+-----+
```

`iload_2` loads the third argument (c) in the stack:

```
+-----+
TOS ->|     c   |
+-----+
| product |
+-----+
```

Now the `iadd` instruction can add the two values.

#### 4.1.4 JVM memory model

x86 and other low-level environments use the stack for argument passing and as a local variables storage.

JVM is slightly different.

It has:

- Local variable array ([LVA](#)<sup>7</sup>). Used as storage for incoming function arguments and local variables.

Instructions like `iload_0` load values from it.

`istore` stores values in it. At the beginning the function arguments are stored: starting at 0 or at 1 (if the zeroth argument is occupied by *this* pointer).

Then the local variables are allocated.

Each slot has size of 32-bit.

Hence, values of *long* and *double* data types occupy two slots.

- Operand stack (or just "stack"). It's used for computations and passing arguments while calling other functions.

Unlike low-level environments like x86, it's not possible to access the stack without using instructions which explicitly pushes or pops values to/from it.

- Heap. It is used as storage for objects and arrays.

These 3 areas are isolated from each other.

<sup>7</sup>(Java) Local Variable Array

## 4.1.5 Simple function calling

`Math.random()` returns a pseudorandom number in range of [0.0 ...1.0), but let's say that for some reason we need to devise a function that returns a number in range of [0.0 ...0.5):

```
public class HalfRandom
{
    public static double f()
    {
        return Math.random()/2;
    }
}
```

Listing 4.8: Constant pool

```
...
#2 = Methodref      #18.#19    //  java/lang/Math.random:()D
#3 = Double         2.0d
...
#12 = Utf8          ()D
...
#18 = Class         #22       //  java/lang/Math
#19 = NameAndType   #23:#12   //  random:()D
#22 = Utf8          java/lang/Math
#23 = Utf8          random
```

```
public static double f();
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=0, args_size=0
      0: invokestatic  #2           // Method java/lang/Math.random:()D
      3: ldc2_w        #3           // double 2.0d
      6: ddiv
      7: dreturn
```

`invokestatic` calls the `Math.random()` function and leaves the result at the [TOS](#).

Then the result is divided by 2.0 and returned.

But how is the function name encoded?

It's encoded in the constant pool using a `Methodref` expression.

It defines the class and method names.

The first field of `Methodref` points to a `Class` expression which, in turn, points to the usual text string ("java/lang/Math").

The second `Methodref` expression points to a `NameAndType` expression which also has two links to the strings.

The first string is "random", which is the name of the method.

The second string is "()D", which encodes the function's type. It means that it returns a *double* value (hence the *D* in the string).

This is the way 1) JVM can check data for type correctness; 2) Java decompilers can restore data types from a compiled class file.

Now let's try the "Hello, world!" example:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```

Listing 4.9: Constant pool

```
...
#2 = Fieldref      #16.#17      //  java/lang/System.out:Ljava/io/PrintStream;
#3 = String        #18          //  Hello, World
#4 = Methodref     #19.#20      //  java/io/PrintStream.println:(Ljava/lang/String;)V
...
#16 = Class        #23          //  java/lang/System
#17 = NameAndType  #24:#25      //  out:Ljava/io/PrintStream;
#18 = Utf8          Hello, World
#19 = Class        #26          //  java/io/PrintStream
#20 = NameAndType  #27:#28      //  println:(Ljava/lang/String;)V
...
#23 = Utf8          java/lang/System
#24 = Utf8          out
#25 = Utf8          Ljava/io/PrintStream;
#26 = Utf8          java/io/PrintStream
#27 = Utf8          println
#28 = Utf8          (Ljava/lang/String;)V
...
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
    0: getstatic   #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc         #3          // String Hello, World
    5: invokevirtual #4         // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    8: return
```

`ldc` at offset 3 takes a pointer to the “Hello, World” string in the constant pool and pushes in the stack. It’s called a *reference* in the Java world, but it’s rather a pointer, or an address

<sup>8</sup>

The familiar `invokevirtual` instruction takes the information about the `println` function (or method) from the constant pool and calls it.

As we may know, there are several `println` methods, one for each data type.

Our case is the version of `println` intended for the *String* data type.

But what about the first `getstatic` instruction?

This instruction takes a *reference* (or address of) a field of the object `System.out` and pushes it in the stack.

This value is acts like the *this* pointer for the `println` method.

Thus, internally, the `println` method takes two arguments for input: 1) *this*, i.e., a pointer to an object; 2) the address of the “Hello, World” string.

Indeed, `println()` is called as a method within an initialized `System.out` object.

For convenience, the `javap` utility writes all this information in the comments.

#### 4.1.6 Calling `beep()`

This is a simple calling of two functions without arguments:

```
public static void main(String[] args)
{
    java.awt.Toolkit.getDefaultToolkit().beep();
}
```

<sup>8</sup>About difference in pointers and *reference's* in C++ see: [3.19.3 on page 562](#).

## 4.1. JAVA

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
      0: invokestatic #2          // Method java.awt.Toolkit.getDefaultToolkit:()Ljava.awt.Toolkit;
    ↳ Toolkit;
      3: invokevirtual #3        // Method java.awt.Toolkit.beep:()V
      6: return
```

First `invokestatic` at offset 0 calls

`java.awt.Toolkit.getDefaultToolkit()`, which returns a reference to an object of class `Toolkit`.

The `invokevirtual` instruction at offset 3 calls the `beep()` method of this class.

### 4.1.7 Linear congruent PRNG

Let's try a simple pseudorandom numbers generator, which we already considered once in the book ([1.23 on page 339](#)):

```
public class LCG
{
    public static int rand_state;

    public void my_srand (int init)
    {
        rand_state=init;
    }

    public static int RNG_a=1664525;
    public static int RNG_c=1013904223;

    public int my_rand ()
    {
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
    }
}
```

There are couple of class fields which are initialized at start.

But how? In `javap` output we can find the class constructor:

```
static {};
flags: ACC_STATIC
Code:
  stack=1, locals=0, args_size=0
    0: ldc           #5          // int 1664525
    2: putstatic     #3          // Field RNG_a:I
    5: ldc           #6          // int 1013904223
    7: putstatic     #4          // Field RNG_c:I
   10: return
```

That's the way variables are initialized.

`RNG_a` occupies the 3rd slot in the class and `RNG_c` —4th, and `putstatic` puts the constants there.

The `my_srand()` function just stores the input value in `rand_state`:

```
public void my_srand(int);
flags: ACC_PUBLIC
Code:
  stack=1, locals=2, args_size=2
    0: iload_1
    1: putstatic     #2          // Field rand_state:I
    4: return
```

#### 4.1. JAVA

`iload_1` takes the input value and pushes it into stack. But why not `iload_0`?

It's because this function may use fields of the class, and so *this* is also passed to the function as a zeroth argument.

The field `rand_state` occupies the 2nd slot in the class, so `putstatic` copies the value from the TOS into the 2nd slot.

Now `my_rand()`:

```
public int my_rand();
flags: ACC_PUBLIC
Code:
stack=2, locals=1, args_size=1
 0: getstatic    #2          // Field rand_state:I
 3: getstatic    #3          // Field RNG_a:I
 6: imul
 7: putstatic    #2          // Field rand_state:I
10: getstatic    #2          // Field rand_state:I
13: getstatic    #4          // Field RNG_c:I
16: iadd
17: putstatic    #2          // Field rand_state:I
20: getstatic    #2          // Field rand_state:I
23: sipush      32767
26: iand
27: ireturn
```

It just loads all the values from the object's fields, does the operations and updates `rand_state`'s value using the `putstatic` instruction.

At offset 20, `rand_state` is reloaded again (because it has been dropped from the stack before, by `putstatic`).

This looks like non-efficient code, but be sure, the [JVM](#) is usually good enough to optimize such things really well.

#### 4.1.8 Conditional jumps

Now let's proceed to conditional jumps.

```
public class abs
{
    public static int abs(int a)
    {
        if (a<0)
            return -a;
        return a;
    }
}
```

```
public static int abs(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
 0: iload_0
 1: ifge           7
 4: iload_0
 5: ineg
 6: ireturn
 7: iload_0
 8: ireturn
```

`ifge` jumps to offset 7 if the value at TOS is greater or equal to 0.

Don't forget, any `ifXX` instruction pops the value (to be compared) from the stack.

`ineg` just negates value at TOS.

#### 4.1. JAVA

Another example:

```
public static int min (int a, int b)
{
    if (a>b)
        return b;
    return a;
}
```

We get:

```
public static int min(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
1: iload_1
2: if_icmple    7
5: iload_1
6: ireturn
7: iload_0
8: ireturn
```

`if_icmple` pops two values and compares them. If the second one is lesser than (or equal to) the first, a jump to offset 7 is performed.

When we define `max()` function ...

```
public static int max (int a, int b)
{
    if (a>b)
        return a;
    return b;
}
```

...the resulting code is the same, but the last two `iload` instructions (at offsets 5 and 7) are swapped:

```
public static int max(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
1: iload_1
2: if_icmple    7
5: iload_0
6: ireturn
7: iload_1
8: ireturn
```

A more advanced example:

```
public class cond
{
    public static void f(int i)
    {
        if (i<100)
            System.out.print("<100");
        if (i==100)
            System.out.print("==100");
        if (i>100)
            System.out.print(">100");
        if (i==0)
            System.out.print("==0");
    }
}
```

```
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
```

## 4.1. JAVA

```
stack=2, locals=1, args_size=1
 0: iload_0
 1: bipush      100
 3: if_icmpge   14           // Field java/lang/System.out:Ljava/io/PrintStream;
 6: getstatic    #2           // String <100
 9: ldc          #3           // Method java/io/PrintStream.print:(Ljava/lang/String;)V
11: invokevirtual #4           // Field java/lang/System.out:Ljava/io/PrintStream;
14: iload_0
15: bipush      100
17: if_icmpne   28           // Field java/lang/System.out:Ljava/io/PrintStream;
20: getstatic    #2           // String ==100
23: ldc          #5           // Method java/io/PrintStream.print:(Ljava/lang/String;)V
25: invokevirtual #4           // Field java/lang/System.out:Ljava/io/PrintStream;
28: iload_0
29: bipush      100
31: if_icmple   42           // Field java/lang/System.out:Ljava/io/PrintStream;
34: getstatic    #2           // String >100
37: ldc          #6           // Method java/io/PrintStream.print:(Ljava/lang/String;)V
39: invokevirtual #4           // Field java/lang/System.out:Ljava/io/PrintStream;
42: iload_0
43: ifne        54           // Field java/lang/System.out:Ljava/io/PrintStream;
46: getstatic    #2           // String ==0
49: ldc          #7           // Method java/io/PrintStream.print:(Ljava/lang/String;)V
51: invokevirtual #4           // Field java/lang/System.out:Ljava/io/PrintStream;
54: return
```

`if_icmpge` pops two values and compares them. If the second one is larger than the first, a jump to offset 14 is performed.

`if_icmpne` and `if_icmple` work just the same, but implement different conditions.

There is also a `ifne` instruction at offset 43.

Its name is misnomer, it would've be better to name it `ifnz` (jump if the value at `TOS` is not zero).

And that is what it does: it jumps to offset 54 if the input value is not zero.

If zero, the execution flow proceeds to offset 46, where the “==0” string is printed.

N.B.: [JVM](#) has no unsigned data types, so the comparison instructions operate only on signed integer values.

### 4.1.9 Passing arguments

Let's extend our `min() / max()` example:

```
public class minmax
{
    public static int min (int a, int b)
    {
        if (a>b)
            return b;
        return a;
    }

    public static int max (int a, int b)
    {
        if (a>b)
            return a;
        return b;
    }

    public static void main(String[] args)
    {
        int a=123, b=456;
        int max_value=max(a, b);
        int min_value=min(a, b);
        System.out.println(min_value);
        System.out.println(max_value);
    }
}
```

#### 4.1. JAVA

```
    }  
}
```

Here is `main()` function code:

```
public static void main(java.lang.String[]);  
  flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=2, locals=5, args_size=1  
  0: bipush      123  
  2: istore_1  
  3: sipush      456  
  6: istore_2  
  7: iload_1  
  8: iload_2  
  9: invokestatic #2          // Method max:(II)I  
 12: istore_3  
13: iload_1  
14: iload_2  
15: invokestatic #3          // Method min:(II)I  
18: istore      4  
20: getstatic     #4          // Field java/lang/System.out:Ljava/io/PrintStream;  
23: iload      4  
25: invokevirtual #5          // Method java/io/PrintStream.println:(I)V  
28: getstatic     #4          // Field java/lang/System.out:Ljava/io/PrintStream;  
31: iload_3  
32: invokevirtual #5          // Method java/io/PrintStream.println:(I)V  
35: return
```

Arguments are passed to the other function in the stack, and the return value is left on [TOS](#).

#### 4.1.10 Bitfields

All bit-wise operations work just like in any other [ISA](#):

```
public static int set (int a, int b)  
{  
    return a | 1<<b;  
}  
  
public static int clear (int a, int b)  
{  
    return a & (~(1<<b));  
}
```

```
public static int set(int, int);  
  flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=3, locals=2, args_size=2  
  0: iload_0  
  1: iconst_1  
  2: iload_1  
  3: ishl  
  4: ior  
  5: ireturn  
  
public static int clear(int, int);  
  flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=3, locals=2, args_size=2  
  0: iload_0  
  1: iconst_1  
  2: iload_1  
  3: ishl  
  4: iconst_m1  
  5: ixor  
  6: iand  
  7: ireturn
```

#### 4.1. JAVA

`iconst_m1` loads -1 in the stack, it's the same as the `0xFFFFFFFF` number.

XORing with `0xFFFFFFFF` has the same effect of inverting all bits ([2.5 on page 461](#)).

Let's extend all data types to 64-bit `long`:

```
public static long lset (long a, int b)
{
    return a | 1<<b;
}

public static long lclear (long a, int b)
{
    return a & (~(1<<b));
}
```

```
public static long lset(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=3, args_size=2
0: lload_0
1: iconst_1
2: iload_2
3: ishl
4: i2l
5: lor
6: lreturn

public static long lclear(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=3, args_size=2
0: lload_0
1: iconst_1
2: iload_2
3: ishl
4: iconst_m1
5: ixor
6: i2l
7: land
8: lreturn
```

The code is the same, but instructions with `/` prefix are used, which operate on 64-bit values.

Also, the second argument of the function still is of type `int`, and when the 32-bit value in it needs to be promoted to 64-bit value the `i2l` instruction is used, which essentially extend the value of an `integer` type to a `long` one.

#### 4.1.11 Loops

```
public class Loop
{
    public static void main(String[] args)
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(i);
        }
    }
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=1
0: iconst_1
1: istore_1
2: iload_1
```

#### 4.1. JAVA

```
3: bipush      10
5: if_icmpgt   21
8: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
11: iload_1
12: invokevirtual #3        // Method java/io/PrintStream.println:(I)V
15: iinc         1, 1
18: goto         2
21: return
```

`iconst_1` loads 1 into `TOS`, `istore_1` stores it in the `LVA` at slot 1.

Why not the zeroth slot? Because the `main()` function has one argument (array of `String`) and a pointer to it (or *reference*) is now in the zeroth slot.

So, the *i* local variable will always be in 1st slot.

Instructions at offsets 3 and 5 compare *i* with 10.

If *i* is larger, execution flow passes to offset 21, where the function ends.

If it's not, `println` is called.

*i* is then reloaded at offset 11, for `println`.

By the way, we call the `println` method for an *integer*, and we see this in the comments: "(I)V" (*I* means *integer* and *V* means the return type is *void*).

When `println` finishes, *i* is incremented at offset 15.

The first operand of the instruction is the number of a slot (1), the second is the number (1) to add to the variable.

`goto` is just GOTO, it jumps to the beginning of the loop's body offset 2.

Let's proceed with a more complex example:

```
public class Fibonacci
{
    public static void main(String[] args)
    {
        int limit = 20, f = 0, g = 1;

        for (int i = 1; i <= limit; i++)
        {
            f = f + g;
            g = f - g;
            System.out.println(f);
        }
    }
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=5, args_size=1
0: bipush      20
2: istore_1
3: iconst_0
4: istore_2
5: iconst_1
6: istore_3
7: iconst_1
8: istore      4
10: iload       4
12: iload_1
13: if_icmpgt   37
16: iload_2
17: iload_3
18: iadd
19: istore_2
20: iload_2
```

#### 4.1. JAVA

```
21: iload_3
22: isub
23: istore_3
24: getstatic #2          // Field java/lang/System.out:Ljava/io/PrintStream;
27: iload_2
28: invokevirtual #3      // Method java/io/PrintStream.println:(I)V
31: iinc      4, 1
34: goto      10
37: return
```

Here is a map of the LVA slots:

- 0 — the sole argument of `main()`
- 1 — *limit*, always contains 20
- 2 — *f*
- 3 — *g*
- 4 — *i*

We can see that the Java compiler allocates variables in LVA slots in the same order they were declared in the source code.

There are separate `istore` instructions for accessing slots 0, 1, 2 and 3, but not for 4 and larger, so there is `istore` with an additional operand at offset 8 which takes the slot number as an operand.

It's the same with `iload` at offset 10.

But isn't it dubious to allocate another slot for the *limit* variable, which always contains 20 (so it's a constant in essence), and reload its value so often?

JVM JIT compiler is usually good enough to optimize such things.

Manual intervention in the code is probably not worth it.

#### 4.1.12 switch()

The `switch()` statement is implemented with the `tableswitch` instruction:

```
public static void f(int a)
{
    switch (a)
    {
        case 0: System.out.println("zero"); break;
        case 1: System.out.println("one\n"); break;
        case 2: System.out.println("two\n"); break;
        case 3: System.out.println("three\n"); break;
        case 4: System.out.println("four\n"); break;
        default: System.out.println("something unknown\n"); break;
    }
}
```

As simple, as possible:

```
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: iload_0
1: tableswitch { // 0 to 4
    0: 36
    1: 47
    2: 58
    3: 69
    4: 80
    default: 91
}
36: getstatic #2          // Field java/lang/System.out:Ljava/io/PrintStream;
39: ldc      #3          // String zero
```

## 4.1. JAVA

```
41: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
44: goto    99
47: getstatic #2      // Field java/lang/System.out:Ljava/io/PrintStream;
50: ldc     #5       // String one\n
52: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
55: goto    99
58: getstatic #2      // Field java/lang/System.out:Ljava/io/PrintStream;
61: ldc     #6       // String two\n
63: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
66: goto    99
69: getstatic #2      // Field java/lang/System.out:Ljava/io/PrintStream;
72: ldc     #7       // String three\n
74: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
77: goto    99
80: getstatic #2      // Field java/lang/System.out:Ljava/io/PrintStream;
83: ldc     #8       // String four\n
85: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
88: goto    99
91: getstatic #2      // Field java/lang/System.out:Ljava/io/PrintStream;
94: ldc     #9       // String something unknown\n
96: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
99: return
```

### 4.1.13 Arrays

#### Simple example

Let's first create an array of 10 integers and fill it:

```
public static void main(String[] args)
{
    int a[]=new int[10];
    for (int i=0; i<10; i++)
        a[i]=i;
    dump (a);
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
0: bipush      10
2: newarray     int
4: astore_1
5: iconst_0
6: istore_2
7: iload_2
8: bipush      10
10: if_icmpge   23
13: aload_1
14: iload_2
15: iload_2
16: iastore
17: iinc       2, 1
20: goto       7
23: aload_1
24: invokestatic #4      // Method dump:([I)V
27: return
```

The `newarray` instruction creates an array object of 10 *int* elements.

The array's size is set with `bipush` and left at [TOS](#).

The array's type is set in `newarray` instruction's operand.

After `newarray`'s execution, a *reference* (or pointer) to the newly created array in the heap is left at the [TOS](#).

#### 4.1. JAVA

`astore_1` stores the *reference* to the 1st slot in **LVA**.

The second part of the `main()` function is the loop which stores *i* into the corresponding array element.

`aload_1` gets a *reference* of the array and places it in the stack.

`iastore` then stores the integer value from the stack in the array, *reference* of which is currently in **TOS**.

The third part of the `main()` function calls the `dump()` function.

An argument for it is prepared by `aload_1` (offset 23).

Now let's proceed to the `dump()` function:

```
public static void dump(int a[])
{
    for (int i=0; i<a.length; i++)
        System.out.println(a[i]);
}
```

```
public static void dump(int[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
  0: iconst_0
  1: istore_1
  2: iload_1
  3: aload_0
  4: arraylength
  5: if_icmpge   23
  8: getstatic   #2      // Field java/lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iload_1
13: iaload
14: invokevirtual #3      // Method java/io/PrintStream.println:(I)V
17: iinc          1, 1
20: goto          2
23: return
```

The incoming *reference* to the array is in the zeroth slot.

The `a.length` expression in the source code is converted to an `arraylength` instruction: it takes a *reference* to the array and leaves the array size at **TOS**.

`iaload` at offset 13 is used to load array elements, it requires to array *reference* be present in the stack (prepared by `aload_0` at 11), and also an index (prepared by `iload_1` at offset 12).

Needless to say, instructions prefixed with `a` may be mistakenly comprehended as *array* instructions.

It's not correct. These instructions works with *references* to objects.

And arrays and strings are objects too.

### Summing elements of array

Another example:

```
public class ArraySum
{
    public static int f (int[] a)
    {
        int sum=0;
        for (int i=0; i<a.length; i++)
            sum=sum+a[i];
        return sum;
    }
}
```

## 4.1. JAVA

```
public static int f(int[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
0:  iconst_0
1:  istore_1
2:  iconst_0
3:  istore_2
4:  iload_2
5:  aload_0
6:  arraylength
7:  if_icmpge    22
10: iload_1
11: aload_0
12: iload_2
13: iaload
14: iadd
15: istore_1
16: iinc      2, 1
19: goto      4
22: iload_1
23: ireturn
```

LVA slot 0 contains a *reference* to the input array.

LVA slot 1 contains the local variable *sum*.

### The only argument of the `main()` function is an array too

We'll be using the only argument of the `main()` function, which is an array of strings:

```
public class UseArgument
{
    public static void main(String[] args)
    {
        System.out.print("Hi, ");
        System.out.print(args[1]);
        System.out.println(". How are you?");
    }
}
```

The zeroth argument is the program's name (like in C/C++, etc.), so the 1st argument supplied by the user is 1st.

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=1, args_size=1
0: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc         #3      // String Hi,
5: invokevirtual #4      // Method java/io/PrintStream.print:(Ljava/lang/String;)V
8: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iconst_1
13: aaload
14: invokevirtual #4      // Method java/io/PrintStream.print:(Ljava/lang/String;)V
17: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
20: ldc         #5      // String . How are you?
22: invokevirtual #6      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
25: return
```

`aload_0` at 11 loads a *reference* of the zeroth LVA slot (1st and only `main()` argument).

`iconst_1` and `aaload` at 12 and 13 take a *reference* to the first (counting at 0) element of array.

The *reference* to the string object is at TOS at offset 14, and it is taken from there by `println` method.

#### 4.1. JAVA

##### Pre-initialized array of strings

```
class Month
{
    public static String[] months =
    {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    };

    public String get_month (int i)
    {
        return months[i];
    }
}
```

The `get_month()` function is simple: Функция `get_month()` проста:

```
public java.lang.String get_month(int);
flags: ACC_PUBLIC
Code:
stack=2, locals=2, args_size=2
0: getstatic    #2          // Field months:[Ljava/lang/String;
3: iload_1
4: aaload
5: areturn
```

`aaload` operates on an array of *references*.

Java String are objects, so the a-instructions are used to operate on them.

`areturn` returns a *reference* to a `String` object.

How is the `months[]` array initialized?

```
static {};
flags: ACC_STATIC
Code:
stack=4, locals=0, args_size=0
0: bipush      12
2: anewarray   #3          // class java/lang/String
5: dup
6: iconst_0
7: ldc         #4          // String January
9: aastore
10: dup
11: iconst_1
12: ldc         #5          // String February
14: aastore
15: dup
16: iconst_2
17: ldc         #6          // String March
19: aastore
20: dup
21: iconst_3
22: ldc         #7          // String April
24: aastore
25: dup
26: iconst_4
```

#### 4.1. JAVA

```
27: ldc          #8      // String May
29: aastore
30: dup
31: iconst_5
32: ldc          #9      // String June
34: aastore
35: dup
36: bipush       6
38: ldc          #10     // String July
40: aastore
41: dup
42: bipush       7
44: ldc          #11     // String August
46: aastore
47: dup
48: bipush       8
50: ldc          #12     // String September
52: aastore
53: dup
54: bipush       9
56: ldc          #13     // String October
58: aastore
59: dup
60: bipush       10
62: ldc          #14     // String November
64: aastore
65: dup
66: bipush       11
68: ldc          #15     // String December
70: aastore
71: putstatic    #2      // Field months:[Ljava/lang/String;
74: return
```

`anewarray` creates a new array of *references* (hence a prefix).

The object's type is defined in the `anewarray`'s operand, it is the "java/lang/String" string.

The `bipush 12` before `anewarray` sets the array's size.

We see here a new instruction for us: `dup`.

It's a standard instruction in stack computers (including the Forth programming language) which just duplicates the value at [TOS](#).

By the way, FPU 80x87 is also a stack computer and it has similar instruction - `FDUP`.

It is used here to duplicate a *reference* to an array, because the `aastore` instruction pops the *reference* to array from the stack, but subsequent `aastore` will need it again.

The Java compiler concluded that it's better to generate a `dup` instead of generating a `getstatic` instruction before each array store operation (i.e., 11 times).

`aastore` puts a *reference* (to string) into the array at an index which is taken from [TOS](#).

Finally, `putstatic` puts *reference* to the newly created array into the second field of our object, i.e., *months* field.

### Variadic functions

Variadic functions actually use arrays:

```
public static void f(int... values)
{
    for (int i=0; i<values.length; i++)
        System.out.println(values[i]);
}
```

#### 4.1. JAVA

```
public static void main(String[] args)
{
    f (1,2,3,4,5);
}
```

```
public static void f(int...);
flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
Code:
stack=3, locals=2, args_size=1
 0:  iconst_0
 1:  istore_1
 2:  iload_1
 3:  aload_0
 4:  arraylength
 5:  if_icmpge   23
 8:  getstatic   #2          // Field java/lang/System.out:Ljava/io/PrintStream;
11:  aload_0
12:  iload_1
13:  iaload
14:  invokevirtual #3        // Method java/io/PrintStream.println:(I)V
17:  iinc           1, 1
20:  goto            2
23:  return
```

f() just takes an array of integers using `aload_0` at offset 3.

Then it gets the array's size, etc.

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=1, args_size=1
 0:  iconst_5
 1:  newarray      int
 3:  dup
 4:  iconst_0
 5:  iconst_1
 6:  iastore
 7:  dup
 8:  iconst_1
 9:  iconst_2
10:  iastore
11:  dup
12:  iconst_2
13:  iconst_3
14:  iastore
15:  dup
16:  iconst_3
17:  iconst_4
18:  iastore
19:  dup
20:  iconst_4
21:  iconst_5
22:  iastore
23:  invokestatic #4        // Method f:([I)V
26:  return
```

The array is constructed in `main()` using the `newarray` instruction, then it's filled, and `f()` is called.

Oh, by the way, array object is not destroyed at the end of `main()`.

There are no destructors in Java at all, because the JVM has a garbage collector which does this automatically, when it feels it needs to.

What about the `format()` method?

It takes two arguments at input: a string and an array of objects:

```
public PrintStream format(String format, Object... args)
```

#### 4.1. JAVA

( <http://docs.oracle.com/javase/tutorial/java/data/numberformat.html> )

Let's see:

```
public static void main(String[] args)
{
    int i=123;
    double d=123.456;
    System.out.format("int: %d double: %f.%n", i, d);
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=7, locals=4, args_size=1
  0: bipush      123
  2: istore_1
  3: ldc2_w      #2          // double 123.456
  6: dstore_2
  7: getstatic   #4          // Field java/lang/System.out:Ljava/io/PrintStream;
 10: ldc         #5          // String int: %d double: %f.%n
 12: istrue
 13: anewarray   #6          // class java/lang/Object
 16: dup
 17: istrue
 18: iload_1
 19: invokestatic #7          // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
 22: aastore
 23: dup
 24: istrue
 25: dload_2
 26: invokestatic #8          // Method java/lang/Double.valueOf:(D)Ljava/lang/Double;
 29: aastore
 30: invokevirtual #9          // Method java/io/PrintStream.format:(Ljava/lang/String;[L
  ↴ Ljava/lang/Object;)Ljava/io/PrintStream;
 33: pop
 34: return
```

So values of the *int* and *double* types are first promoted to `Integer` and `Double` objects using the `valueOf` methods.

The `format()` method needs objects of type `Object` at input, and since the `Integer` and `Double` classes are derived from the root `Object` class, they suitable for elements in the input array.

On the other hand, an array is always homogeneous, i.e., it can't hold elements of different types, which makes it impossible to push *int* and *double* values in it.

An array of `Object` objects is created at offset 13, an `Integer` object is added to the array at offset 22, and a `Double` object is added to the array at offset 29.

The penultimate `pop` instruction discards the element at `TOS`, so when `return` is executed, the stack becomes empty (or balanced).

### Two-dimensional arrays

Two-dimensional arrays in Java are just one-dimensional arrays of *references* to another one-dimensional arrays.

Let's create a two-dimensional array:

```
public static void main(String[] args)
{
    int[][] a = new int[5][10];
    a[1][2]=3;
}
```

#### 4.1. JAVA

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
  0:  iconst_5
  1:  bipush      10
  3:  multianewarray #2,  2      // class "[[I"
  7:  astore_1
  8:  aload_1
  9:  iconst_1
 10: aaload
 11: iconst_2
 12: iconst_3
 13: iastore
 14: return
```

It's created using the `multianewarray` instruction: the object's type and dimensionality are passed as operands.

The array's size (10\*5) is left in stack (using the instructions `iconst_5` and `bipush`).

A reference to row #1 is loaded at offset 10 (`iconst_1` and `aaload`).

The column is chosen using `iconst_2` at offset 11.

The value to be written is set at offset 12.

`iastore` at 13 writes the array's element.

How it is an element accessed?

```
public static int get12 (int[][][] in)
{
    return in[1][2];
```

```
public static int get12(int[][][]);
  flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=1, args_size=1
  0:  aload_0
  1:  iconst_1
  2:  aaload
  3:  iconst_2
  4:  iaload
  5:  ireturn
```

A Reference to the array's row is loaded at offset 2, the column is set at offset 3, then `iaload` loads the array's element.

#### Three-dimensional arrays

Three-dimensional arrays are just one-dimensional arrays of references to one-dimensional arrays of references.

```
public static void main(String[] args)
{
    int[][][] a = new int[5][10][15];
    a[1][2][3]=4;
    get_elem(a);
}
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
Code:
```

#### 4.1. JAVA

```
stack=3, locals=2, args_size=1
 0: iconst_5
 1: bipush      10
 3: bipush      15
 5: multianewarray #2,  3      // class "[[[I"
 9: astore_1
10: aload_1
11: iconst_1
12: aaload
13: iconst_2
14: aaload
15: iconst_3
16: iconst_4
17: iastore
18: aload_1
19: invokestatic #3           // Method get_elem:([[I)I
22: pop
23: return
```

Now it takes two `aaload` instructions to find right *reference*:

```
public static int get_elem (int[][][] a)
{
    return a[1][2][3];
}
```

```
public static int get_elem(int[][][]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
 0: aload_0
 1: iconst_1
 2: aaload
 3: iconst_2
 4: aaload
 5: iconst_3
 6: iaload
 7: ireturn
```

#### Summary

Is it possible to do a buffer overflow in Java?

No, because the array's length is always present in an array object, array bounds are controlled, and an exception is to be raised in case of out-of-bounds access.

There are no multi-dimensional arrays in Java in the C/C++ sense, so Java is not very suited for fast scientific computations.

#### 4.1.14 Strings

##### First example

Strings are objects and are constructed in the same way as other objects (and arrays).

```
public static void main(String[] args)
{
    System.out.println("What is your name?");
    String input = System.console().readLine();
    System.out.println("Hello, "+input);
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
```

#### 4.1. JAVA

```
stack=3, locals=2, args_size=1
 0: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
 3: ldc          #3      // String What is your name?
 5: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
 8: invokestatic  #5      // Method java/lang/System.console():Ljava/io/Console;
11: invokevirtual #6      // Method java/io/Console.readLine():Ljava/lang/String;
14: astore_1
15: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
18: new          #7      // class java/lang/StringBuilder
21: dup
22: invokespecial #8      // Method java/lang/StringBuilder."<init>":()V
25: ldc          #9      // String Hello,
27: invokevirtual #10     // Method java/lang/StringBuilder.append:(Ljava/lang/String)Ljava/lang/StringBuilder;
   ;Ljava/lang/StringBuilder;
30: aload_1
31: invokevirtual #10     // Method java/lang/StringBuilder.append:(Ljava/lang/String)Ljava/lang/StringBuilder;
   ;Ljava/lang/StringBuilder;
34: invokevirtual #11     // Method java/lang/StringBuilder.toString():Ljava/lang/String;
   ;String;
37: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
40: return
```

The `readLine()` method is called at offset 11, a reference to string (which is supplied by the user) is then stored at **TOS**.

At offset 14 the *reference to string* is stored in slot 1 of **LVA**.

The string the user entered is reloaded at offset 30 and concatenated with the “Hello,” string using the `StringBuilder` class.

The constructed string is then printed using `println` at offset 37.

#### Second example

Another example:

```
public class strings
{
    public static char test (String a)
    {
        return a.charAt(3);
    };

    public static String concat (String a, String b)
    {
        return a+b;
    }
}
```

```
public static char test(java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
 0: aload_0
 1: iconst_3
 2: invokevirtual #2      // Method java/lang/String.charAt:(I)C
 5: ireturn
```

The string concatenation is performed using `StringBuilder`:

```
public static java.lang.String concat(java.lang.String, java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
 0: new          #3      // class java/lang/StringBuilder
 3: dup
 4: invokespecial #4      // Method java/lang/StringBuilder."<init>":()V
```

#### 4.1. JAVA

```
7: aload_0
8: invokevirtual #5      // Method java/lang/StringBuilder.append:(Ljava/lang/String;)
   ;)Ljava/lang/StringBuilder;
11: aload_1
12: invokevirtual #5      // Method java/lang/StringBuilder.append:(Ljava/lang/String;)
   ;)Ljava/lang/StringBuilder;
15: invokevirtual #6      // Method java/lang/StringBuilder.toString:()Ljava/lang/
   ;String;
18: areturn
```

Another example:

```
public static void main(String[] args)
{
    String s="Hello!";
    int n=123;
    System.out.println("s=" + s + " n=" + n);
}
```

And again, the strings are constructed using the `StringBuilder` class and its `append` method, then the constructed string is passed to `println`:

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
  0: ldc           #2      // String Hello!
  2: astore_1
  3: bipush        123
  5: istore_2
  6: getstatic     #3      // Field java/lang/System.out:Ljava/io/PrintStream;
  9: new           #4      // class java/lang/StringBuilder
 12: dup
 13: invokespecial #5      // Method java/lang/StringBuilder."<init>":()V
 16: ldc           #6      // String s=
 18: invokevirtual #7      // Method java/lang/StringBuilder.append:(Ljava/lang/String;)
   ;)Ljava/lang/StringBuilder;
 21: aload_1
 22: invokevirtual #7      // Method java/lang/StringBuilder.append:(Ljava/lang/String;)
   ;)Ljava/lang/StringBuilder;
 25: ldc           #8      // String n=
 27: invokevirtual #7      // Method java/lang/StringBuilder.append:(Ljava/lang/String;)
   ;)Ljava/lang/StringBuilder;
 30: iload_2
 31: invokevirtual #9      // Method java/lang/StringBuilder.append:(I)Ljava/lang/
   ;StringBuilder;
 34: invokevirtual #10     // Method java/lang/StringBuilder.toString:()Ljava/lang/
   ;String;
 37: invokevirtual #11     // Method java/io/PrintStream.println:(Ljava/lang/String;)V
 40: return
```

#### 4.1.15 Exceptions

Let's rework our `Month` example ([4.1.13 on page 687](#)) a bit:

Listing 4.10: IncorrectMonthException.java

```
public class IncorrectMonthException extends Exception
{
    private int index;

    public IncorrectMonthException(int index)
    {
        this.index = index;
    }
    public int getIndex()
    {
```

#### 4.1. JAVA

```
        return index;  
    }  
}
```

Listing 4.11: Month2.java

```
class Month2  
{  
    public static String[] months =  
    {  
        "January",  
        "February",  
        "March",  
        "April",  
        "May",  
        "June",  
        "July",  
        "August",  
        "September",  
        "October",  
        "November",  
        "December"  
    };  
  
    public static String get_month (int i) throws IncorrectMonthException  
    {  
        if (i<0 || i>11)  
            throw new IncorrectMonthException(i);  
        return months[i];  
    };  
  
    public static void main (String[] args)  
    {  
        try  
        {  
            System.out.println(get_month(100));  
        }  
        catch(IncorrectMonthException e)  
        {  
            System.out.println("incorrect month index: "+ e.getIndex());  
            e.printStackTrace();  
        }  
    };  
}
```

Essentially, `IncorrectMonthException.class` has just an object constructor and one getter method.

The `IncorrectMonthException` class is derived from `Exception`, so the `IncorrectMonthException` constructor first calls the constructor of the `Exception` class, then it puts incoming integer value into the sole `IncorrectMonthException` class field:

```
public IncorrectMonthException(int);  
flags: ACC_PUBLIC  
Code:  
stack=2, locals=2, args_size=2  
0: aload_0  
1: invokespecial #1           // Method java/lang/Exception."<init>":()V  
4: aload_0  
5: iload_1  
6: putfield      #2           // Field index:I  
9: return
```

`getIndex()` is just a getter. A reference to `IncorrectMonthException` is passed in the zeroth [LVA](#) slot (`this`), `aload_0` takes it, `getfield` loads an integer value from the object, `ireturn` returns it.

```
public int getIndex();  
flags: ACC_PUBLIC  
Code:
```

#### 4.1. JAVA

```
stack=1, locals=1, args_size=1
 0: aload_0
 1: getfield      #2      // Field index:I
 4: ireturn
```

Now let's take a look at `get_month()` in `Month2.class`:

Listing 4.12: Month2.class

```
public static java.lang.String get_month(int) throws IncorrectMonthException;
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=1, args_size=1
 0: iload_0
 1: iflt      10
 4: iload_0
 5: bipush    11
 7: if_icmple  19
10: new      #2      // class IncorrectMonthException
13: dup
14: iload_0
15: invokespecial #3      // Method IncorrectMonthException."<init>":(I)V
18: athrow
19: getstatic    #4      // Field months:[Ljava/lang/String;
22: iload_0
23: aaload
24: areturn
```

`iflt` at offset 1 is *if less than*.

In case of invalid index, a new object is created using the `new` instruction at offset 10.

The object's type is passed as an operand to the instruction (which is `IncorrectMonthException`).

Then its constructor is called, and index is passed via `TOS` (offset 15).

When the control flow is offset 18, the object is already constructed, so now the `athrow` instruction takes a reference to the newly constructed object and signals to `JVM` to find the appropriate exception handler.

The `athrow` instruction doesn't return the control flow here, so at offset 19 there is another `basic block`, not related to exceptions business, where we can get from offset 7.

How do handlers work?

`main()` in `Month2.class`:

Listing 4.13: Month2.class

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
 0: getstatic    #5      // Field java/lang/System.out:Ljava/io/PrintStream;
 3: bipush     100
 5: invokestatic #6      // Method get_month:(I)Ljava/lang/String;
 8: invokevirtual #7      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
11: goto      47
14: astore_1
15: getstatic    #5      // Field java/lang/System.out:Ljava/io/PrintStream;
18: new       #8      // class java/lang/StringBuilder
21: dup
22: invokespecial #9      // Method java/lang/StringBuilder."<init>":()V
25: ldc       #10
27: invokevirtual #11      // Method java/lang/StringBuilder.append:(Ljava/lang/String<
↳ ;)Ljava/lang/StringBuilder;
30: aload_1
31: invokevirtual #12      // Method IncorrectMonthException.getIndex:()I
34: invokevirtual #13      // Method java/lang/StringBuilder.append:(I)Ljava/lang/<
↳ StringBuilder;
37: invokevirtual #14      // Method java/lang/StringBuilder.toString:()Ljava/lang/<
↳ String;
40: invokevirtual #7      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

#### 4.1. JAVA

```
43: aload_1
44: invokevirtual #15           // Method IncorrectMonthException.printStackTrace:()V
47: return
Exception table:
  from   to target type
    0     11    14  Class IncorrectMonthException
```

Here is the `Exception table`, which defines that from offsets 0 to 11 (inclusive) an exception `IncorrectMonthException` may happen, and if it does, the control flow is to be passed to offset 14. Indeed, the main program ends at offset 11.

At offset 14 the handler starts. It's not possible to get here, there are no conditional/unconditional jumps to this area.

But [JVM](#) will transfer the execution flow here in case of an exception.

The very first `astore_1` (at 14) takes the incoming *reference* to the exception object and stores it in [LVA](#) slot 1.

Later, the `getIndex()` method (of this exception object) will be called at offset 31.

The *reference* to the current exception object is passed right before that (offset 30).

The rest of the code is does just string manipulation: first the integer value returned by `getIndex()` is converted to string by the `toString()` method, then it's concatenated with the "incorrect month index:" text string (like we saw before), then `println()` and `printStackTrace()` are called.

After `printStackTrace()` finishes, the exception is handled and we can continue with the normal execution.

At offset 47 there is a `return` which finishes the `main()` function, but there could be any other code which would execute as if no exceptions were raised.

Here is an example on how IDA shows exception ranges:

Listing 4.14: from some random .class file found on the author's computer

```
.catch java/io/FileNotFoundException from met001_335 to met001_360\
using met001_360
  .catch java/io/FileNotFoundException from met001_185 to met001_214\
using met001_214
    .catch java/io/FileNotFoundException from met001_181 to met001_192\
using met001_195
      .catch java/io/FileNotFoundException from met001_155 to met001_176\
using met001_176
        .catch java/io/FileNotFoundException from met001_83 to met001_129 using \
met001_129
          .catch java/io/FileNotFoundException from met001_42 to met001_66 using \
met001_69
            .catch java/io/FileNotFoundException from met001_begin to met001_37\
using met001_37
```

#### 4.1.16 Classes

Simple class:

Listing 4.15: test.java

```
public class test
{
    public static int a;
    private static int b;

    public test()
    {
        a=0;
        b=0;
    }
}
```

#### 4.1. JAVA

```
public static void set_a (int input)
{
    a=input;
}
public static int get_a ()
{
    return a;
}
public static void set_b (int input)
{
    b=input;
}
public static int get_b ()
{
    return b;
}
```

The constructor just sets both fields to zero:

```
public test();
flags: ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #1          // Method java/lang/Object."<init>":()V
4: iconst_0
5: putstatic     #2          // Field a:I
8: iconst_0
9: putstatic     #3          // Field b:I
12: return
```

Setter of `a`:

```
public static void set_a(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: putstatic     #2          // Field a:I
4: return
```

Getter of `a`:

```
public static int get_a();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=0, args_size=0
0: getstatic     #2          // Field a:I
3: ireturn
```

Setter of `b`:

```
public static void set_b(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: putstatic     #3          // Field b:I
4: return
```

Getter of `b`:

```
public static int get_b();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=0, args_size=0
0: getstatic     #3          // Field b:I
3: ireturn
```

## 4.1. JAVA

There is no difference in the code which works with public and private fields.

But this type information is present in the `.class` file, and it's not possible to access private fields from everywhere.

Let's create an object and call its method:

Listing 4.16: ex1.java

```
public class ex1
{
    public static void main(String[] args)
    {
        test obj=new test();
        obj.set_a(1234);
        System.out.println(obj.a);
    }
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=1
  0: new           #2      // class test
  3: dup
  4: invokespecial #3      // Method test."<init>":()V
  7: astore_1
  8: aload_1
  9: pop
 10: sipush        1234
 13: invokestatic #4      // Method test.set_a:(I)V
 16: getstatic     #5      // Field java/lang/System.out:Ljava/io/PrintStream;
 19: aload_1
 20: pop
 21: getstatic     #6      // Field test.a:I
 24: invokevirtual #7      // Method java/io/PrintStream.println:(I)V
 27: return
```

The `new` instruction creates an object, but doesn't call the constructor (it is called at offset 4).

The `set_a()` method is called at offset 16.

The `a` field is accessed using the `getstatic` instruction at offset 21.

### 4.1.17 Simple patching

#### First example

Let's proceed with a simple code patching task.

```
public class nag
{
    public static void nag_screen()
    {
        System.out.println("This program is not registered");
    };
    public static void main(String[] args)
    {
        System.out.println("Greetings from the mega-software");
        nag_screen();
    }
}
```

How would we remove the printing of "This program is not registered" string?

Let's load the `.class` file into IDA:

#### 4.1. JAVA

```
; Segment type: Pure code
.method public static nag_screen()V
.limit stack 2
.line 4
• 178 000 002 | ldc "This program is not registered"
• 018 003      ldc "Greetings from the mega-software"
• 182 000 004  invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
               .line 5
• 177          return
• ??? ??? ???+ .end method
• ??? ??? ???+
???
```

```
; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.line 8
• 178 000 002 | ldc "This program is not registered"
• 018 005      ldc "Greetings from the mega-software"
• 182 000 004  invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
               .line 9
• 184 000 006  invokestatic nag.nag_screen()V
               .line 10
• 177          return
???
```

Figure 4.1: IDA

Let's patch the first byte of the function to 177 (which is the `return` instruction's opcode):

```
; Segment type: Pure code
.method public static nag_screen()V
.limit stack 2
.line 4
nag_screen:
    return
• 177
• 000
• 002
• 018 003  ldc "This program is not registered"
• 182 000 004  invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
               .line 5
• 177          return
• ??? ??? ???+ .end method
• ??? ??? ???+
???
```

Figure 4.2: IDA

But that doesn't work (JRE 1.7):

```
Exception in thread "main" java.lang.VerifyError: Expecting a stack map frame
Exception Details:
  Location:
    nag.nag_screen()V @1: nop
  Reason:
    Error exists in the bytecode
Bytecode:
  0000000: b100 0212 03b6 0004 b1
              at java.lang.Class.getDeclaredMethods0(Native Method)
              at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
              at java.lang.Class.getMethod0(Class.java:2856)
              at java.lang.Class.getMethod(Class.java:1668)
```

## 4.1. JAVA

```
at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:494)
at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:486)
```

Perhaps JVM has some other checks related to the stack maps.

OK, let's patch it differently by removing the call to `nag()`:

```
; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.line 8
• 178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
• 018 005        ldc "Greetings from the mega-software"
• 182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9
• 000            nop
• 000            nop
• 000            nop
.line 10
• 177            return
=====
#
```

Figure 4.3: IDA

0 is the opcode for NOP.

Now that works!

## Second example

## Another simple crackme example:

```
public class password
{
    public static void main(String[] args)
    {
        System.out.println("Please enter the password");
        String input = System.console().readLine();
        if (input.equals("secret"))
            System.out.println("password is correct");
        else
            System.out.println("password is not correct");
    }
}
```

Let's load it in IDA:

#### 4.1. JAVA

```
; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
• 178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
• 018 003        ldc "Please enter the password"
• 182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
• 184 000 005    invokestatic java/lang/System.console()Ljava/io/Console;
• 182 000 006    invokevirtual java/io/Console.readLine()Ljava/lang/String;
• 076            astore_1 ; met002_slot001
.line 5
• 043            aload_1 ; met002_slot001
• 018 007        ldc "secret"
• 182 000 008    invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
• 153 000 014    ifeq met002_35
.line 6
• 178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
• 018 009        ldc "password is correct"
• 182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
• 167 000 011    goto met002_43
.line 8

met002_35:                                ; CODE XREF: main+21↑j
178 000 002    .stack use locals
              locals Object java/lang/String
              .end stack
              getstatic java/lang/System.out Ljava/io/PrintStream;
              ldc "password is not correct"
              invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9
```

Figure 4.4: IDA

We see here the `ifeq` instruction which does the job.

Its name stands for *if equal*, and this is misnomer, a better name would be `ifz` (*if zero*), i.e, if value at `TOS` is zero, then do the jump.

In our example, it jumps if the password is not correct (the `equals` method returns `False`, which is 0).

The very first idea is to patch this instruction.

There are two bytes in `ifeq` opcode, which encode the jump offset.

To make this instruction a NOP, we must set the 3rd byte to the value of 3 (because by adding 3 to the current address we will always jump to the next instruction, since the `ifeq` instruction's length is 3 bytes):

#### 4.1. JAVA

```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
• 178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
• 018 003        ldc "Please enter the password"
• 182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
• 184 000 005    invokestatic java/lang/System.console()Ljava/io/Console;
• 182 000 006    invokevirtual java/io/Console.readLine()Ljava/lang/String;
• 076            astore_1 ; met002_slot001
.line 5
• 043            aload_1 ; met002_slot001
• 018 007        ldc "secret"
• 182 000 008    invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
• 153 000 003    ifeq met002_24
.line 6

met002_24:                                ; CODE XREF: main+21↑j
• 178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
• 018 009        ldc "password is correct"
• 182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
• 167 000 011    goto met002_43
.line 8
178 000 002    .stack use locals
                locals Object java/lang/String
                .end stack
                getstatic java/lang/System.out Ljava/io/PrintStream;
                ldc "password is not correct"
                invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9

```

Figure 4.5: IDA

That doesn't work (JRE 1.7):

```

Exception in thread "main" java.lang.VerifyError: Expecting a stackmap frame at branch target ↴ 24
Exception Details:
  Location:
    password.main([Ljava/lang/String;)V @21: ifeq
  Reason:
    Expected stackmap frame at this location.
Bytecode:
  0000000: b200 0212 03b6 0004 b800 05b6 0006 4c2b
  0000010: 1207 b600 0899 0003 b200 0212 09b6 0004
  0000020: a700 0bb2 0002 120a b600 04b1
Stackmap Table:
  append_frame(@35, Object[#20])
  same_frame(@43)

  at java.lang.Class.getDeclaredMethods0(Native Method)
  at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
  at java.lang.Class.getMethod0(Class.java:2856)
  at java.lang.Class.getMethod(Class.java:1668)
  at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:494)
  at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:486)

```

But it must be mentioned that it worked in JRE 1.6.

We can also try to replace all 3 `ifeq` opcode bytes with zero bytes (`NOP`), and it still won't work.

Seems like there are more stack map checks in JRE 1.7.

OK, we'll replace the whole call to the `equals` method with the `iconst_1` instruction plus a pack of `NOPs`:

#### 4.1. JAVA

```
; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
    getstatic java/lang/System.out Ljava/io/PrintStream;
    ldc "Please enter the password"
    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
    invokestatic java/lang/System.console()Ljava/io/Console;
    invokevirtual java/io/Console.readLine()Ljava/lang/String;
    astore_1 ; met002_slot001
.line 5
    iconst_1
    nop
    nop
    nop
    nop
    nop
    ifeq met002_35
.line 6
    getstatic java/lang/System.out Ljava/io/PrintStream;
    ldc "password is correct"
    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
    goto met002_43
.line 8

met002_35:                                ; CODE XREF: main+21↑j
178 000 002      .stack use locals
                           locals Object java/lang/String
                           .end stack
```

Figure 4.6: IDA

1 needs always to be in the TOS when the `ifeq` instruction is executed, so `ifeq` would never jump. This works.

#### 4.1.18 Summary

What is missing in Java in comparison to C/C++?

- Structures: use classes.
- Unions: use class hierarchies.
- Unsigned data types. By the way, this makes cryptographic algorithms somewhat harder to implement in Java.
- Function pointers.

# Chapter 5

## Finding important/interesting stuff in the code

Minimalism it is not a prominent feature of modern software.

But not because the programmers are writing a lot, but because a lot of libraries are commonly linked statically to executable files. If all external libraries were shifted into an external DLL files, the world would be different. (Another reason for C++ are the [STL](#) and other template libraries.)

Thus, it is very important to determine the origin of a function, if it is from standard library or well-known library (like Boost<sup>1</sup>, libpng<sup>2</sup>), or if it is related to what we are trying to find in the code.

It is just absurd to rewrite all code in C/C++ to find what we're looking for.

One of the primary tasks of a reverse engineer is to find quickly the code he/she needs.

The [IDA](#) disassembler allow us to search among text strings, byte sequences and constants. It is even possible to export the code to .lst or .asm text files and then use `grep`, `awk`, etc.

When you try to understand what some code is doing, this easily could be some open-source library like libpng. So when you see some constants or text strings which look familiar, it is always worth to *google* them. And if you find the opensource project where they are used, then it's enough just to compare the functions. It may solve some part of the problem.

For example, if a program uses XML files, the first step may be determining which XML library is used for processing, since the standard (or well-known) libraries are usually used instead of self-made one.

For example, the author of these lines once tried to understand how the compression/decompression of network packets works in SAP 6.0. It is a huge software, but a detailed [PDB](#) with debugging information is present, and that is convenient. He finally came to the idea that one of the functions, that was called *CsDecomprLZC*, was doing the decompression of network packets. Immediately he tried to *google* its name and he quickly found the function was used in MaxDB (it is an open-source SAP project)<sup>3</sup>.

<http://www.google.com/search?q=CsDecomprLZC>

Astoundingly, MaxDB and SAP 6.0 software shared likewise code for the compression/decompression of network packets.

### 5.1 Identification of executable files

#### 5.1.1 Microsoft Visual C++

MSVC versions and DLLs that can be imported:

---

<sup>1</sup><http://go.yurichev.com/17036>

<sup>2</sup><http://go.yurichev.com/17037>

<sup>3</sup>More about it in relevant section ([8.10.1 on page 884](#))

## 5.1. IDENTIFICATION OF EXECUTABLE FILES

Marketing ver.	Internal ver.	CL.EXE ver.	DLLs imported	Release date
6	6.0	12.00	msvcrt.dll msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll msvcp120.dll	October 17, 2013

msvcp\*.dll has C++-related functions, so if it is imported, this is probably a C++ program.

### Name mangling

The names usually start with the `?` symbol.

You can read more about MSVC's [name mangling](#) here: [3.19.1 on page 546](#).

### 5.1.2 GCC

Aside from \*NIX targets, GCC is also present in the win32 environment, in the form of Cygwin and MinGW.

### Name mangling

Names usually start with the `_Z` symbols.

You can read more about GCC's [name mangling](#) here: [3.19.1 on page 546](#).

### Cygwin

cygwin1.dll is often imported.

### MinGW

msvcrt.dll may be imported.

### 5.1.3 Intel Fortran

libifcoremd.dll, libifportmd.dll and libiomp5md.dll (OpenMP support) may be imported.

libifcoremd.dll has a lot of functions prefixed with `for_`, which means *Fortran*.

### 5.1.4 Watcom, OpenWatcom

#### Name mangling

Names usually start with the `W` symbol.

For example, that is how the method named "method" of the class "class" that does not have any arguments and returns *void* is encoded:

```
W?method$_class$n__v
```

## 5.1.5 Borland

Here is an example of Borland Delphi's and C++Builder's [name mangling](#):

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviisiitliiiii
@TrueColorTo16BitN$qpviisiitliiiii
@DIB24BitTo8BitBitmap$qpviisiitliiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

The names always start with the `@` symbol, then we have the class name came, method name, and encoded the types of the arguments of the method.

These names can be in the .exe imports, .dll exports, debug data, etc.

Borland Visual Component Libraries (VCL) are stored in .bpl files instead of .dll ones, for example, vcl50.bpl, rtl60.bpl.

Another DLL that might be imported: BORLNDMM.DLL.

## Delphi

Almost all Delphi executables has the “Boolean” text string at the beginning of the code segment, along with other type names.

This is a very typical beginning of the `CODE` segment of a Delphi program, this block came right after the win32 PE file header:

00000400	04 10 40 00 03 07 42 6f	6f 6c 65 61 6e 01 00 00	...@...Boolean...
00000410	00 00 01 00 00 00 00 10	40 00 05 46 61 6c 73 65	.....@..False
00000420	04 54 72 75 65 8d 40 00	2c 10 40 00 09 08 57 69	.True@...@...Wi
00000430	64 65 43 68 61 72 03 00	00 00 00 ff ff 00 00 90	deChar.....
00000440	44 10 40 00 02 04 43 68	61 72 01 00 00 00 00 ff	D@...Char.....
00000450	00 00 00 90 58 10 40 00	01 08 53 6d 61 6c 6c 69	....X@...Smalli
00000460	6e 74 02 00 80 ff ff ff	7f 00 00 90 70 10 40 00	nt.....p@.
00000470	01 07 49 6e 74 65 67 65	72 04 00 00 00 80 ff ff	..Integer.....
00000480	ff 7f 8b c0 88 10 40 00	01 04 42 79 74 65 01 00	.....@...Byte..
00000490	00 00 00 ff 00 00 00 90	9c 10 40 00 01 04 57 6f	.....@...Wo
000004a0	72 64 03 00 00 00 00 ff	ff 00 00 90 b0 10 40 00	rd.....@.
000004b0	01 08 43 61 72 64 69 6e	61 6c 05 00 00 00 00 ff	..Cardinal.....
000004c0	ff ff ff 90 c8 10 40 00	10 05 49 6e 74 36 34 00	.....@...Int64.
000004d0	00 00 00 00 00 00 80 ff	ff ff ff ff ff 7f 90	.....
000004e0	e4 10 40 00 04 08 45 78	74 65 6e 64 65 64 02 90	...@...Extended..
000004f0	f4 10 40 00 04 06 44 6f	75 62 6c 65 01 8d 40 00	...@...Double..@.
00000500	04 11 40 00 04 08 43 75	72 72 65 6e 63 79 04 90	...@...Currency..
00000510	14 11 40 00 0a 06 73 74	72 69 6e 67 20 11 40 00	...@...string .@.
00000520	0b 0a 57 69 64 65 53 74	72 69 6e 67 30 11 40 00	..WideString0.@.
00000530	0c 07 56 61 72 69 61 6e	74 8d 40 00 40 11 40 00	.Variant.@.@@.
00000540	0c 0a 4f 6c 65 56 61 72	69 61 6e 74 98 11 40 00	..OleVariant..@.
00000550	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00000560	00 00 00 00 00 00 00 00	00 00 00 00 98 11 40 00	.....@.
00000570	04 00 00 00 00 00 00 00	18 4d 40 00 24 4d 40 00	.....M@.\$M@.
00000580	28 4d 40 00 2c 4d 40 00	20 4d 40 00 68 4a 40 00	(M@.,M@. M@.hJ@.
00000590	84 4a 40 00 c0 4a 40 00	07 54 4f 62 6a 65 63 74	.J@..J@..TObject
000005a0	a4 11 40 00 07 07 54 4f	62 6a 65 63 74 98 11 40	...@...TObject..@
000005b0	00 00 00 00 00 00 00 06	53 79 73 74 65 6d 00 00	.....System..
000005c0	c4 11 40 00 0f 0a 49 49	6e 74 65 72 66 61 63 65	...@...IInterface
000005d0	00 00 00 00 01 00 00 00	00 00 00 00 00 00 c0 00	.....

## 5.2. COMMUNICATION WITH OUTER WORLD (FUNCTION LEVEL)

000005e0	00 00 00 00 46 06 53 79	73 74 65 6d 03 00 ff ff	.....F.System....
000005f0	f4 11 40 00 0f 09 49 44	69 73 70 61 74 63 68 c0	...@...IDispatch.
00000600	11 40 00 01 00 04 02 00	00 00 00 00 c0 00 00 00	...@.....
00000610	00 00 00 46 06 53 79 73	74 65 6d 04 00 ff ff 90	...F.System....
00000620	cc 83 44 24 04 f8 e9 51	6c 00 00 83 44 24 04 f8	..D\$...Ql...D\$..
00000630	e9 6f 6c 00 00 83 44 24	04 f8 e9 79 6c 00 00 cc	.ol...D\$...yl...
00000640	cc 21 12 40 00 2b 12 40	00 35 12 40 00 01 00 00	!..@.+.@.5.@....
00000650	00 00 00 00 00 00 00 00	00 c0 00 00 00 00 00 00	.....
00000660	46 41 12 40 00 08 00 00	00 00 00 00 00 8d 40 00	FA.@.....@.
00000670	bc 12 40 00 4d 12 40 00	00 00 00 00 00 00 00 00	...@.M.@.....
00000680	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00000690	bc 12 40 00 0c 00 00 00	4c 11 40 00 18 4d 40 00	...@.....L.@..M@.
000006a0	50 7e 40 00 5c 7e 40 00	2c 4d 40 00 20 4d 40 00	P~@.\~@.,M@. M@.
000006b0	6c 7e 40 00 84 4a 40 00	c0 4a 40 00 11 54 49 6e	l~@..J@..J@..TIn
000006c0	74 65 72 66 61 63 65 64	4f 62 6a 65 63 74 8b c0	terfacedObject..
000006d0	d4 12 40 00 07 11 54 49	6e 74 65 72 66 61 63 65	...@...TInterface
000006e0	64 4f 62 6a 65 63 74 bc	12 40 00 a0 11 40 00 00	dObject..@...@..
000006f0	00 06 53 79 73 74 65 6d	00 00 8b c0 00 13 40 00	..System.....@.
00000700	11 0b 54 42 6f 75 6e 64	41 72 72 61 79 04 00 00	..TBoundArray...
00000710	00 00 00 00 03 00 00 00	00 6c 10 40 00 06 53 79	.....l.@..Sy
00000720	73 74 65 6d 28 13 40 00	04 09 54 44 61 74 65 54	stem(.@...TDateT
00000730	69 6d 65 01 ff 25 48 e0	c4 00 8b c0 ff 25 44 e0	ime..%H.....%D.

The first 4 bytes of the data segment (`DATA`) can be `00 00 00 00`, `32 13 8B C0` or `FF FF FF FF`.

This information can be useful when dealing with packed/encrypted Delphi executables.

### 5.1.6 Other known DLLs

- `vcomp*.dll`—Microsoft's implementation of OpenMP.

## 5.2 Communication with outer world (function level)

It's often advisable to track function arguments and return values in debugger or [DBI](#). For example, the author once tried to understand meaning of some obscure function, which happens to be incorrectly implemented bubble sort. (It worked correctly, but slower.) Meanwhile, watching inputs and outputs of this function helps instantly to understand what it does.

## 5.3 Communication with the outer world (win32)

Sometimes it's enough to observe some function's inputs and outputs in order to understand what it does. That way you can save time.

Files and registry access: for the very basic analysis, Process Monitor<sup>4</sup> utility from SysInternals can help.

For the basic analysis of network accesses, Wireshark<sup>5</sup> can be useful.

But then you will have to look inside anyway.

The first thing to look for is which functions from the OS's APIs and standard libraries are used.

If the program is divided into a main executable file and a group of DLL files, sometimes the names of the functions in these DLLs can help.

If we are interested in exactly what can lead to a call to `MessageBox()` with specific text, we can try to find this text in the data segment, find the references to it and find the points from which the control may be passed to the `MessageBox()` call we're interested in.

If we are talking about a video game and we're interested in which events are more or less random in it, we may try to find the `rand()` function or its replacements (like the Mersenne twister algorithm) and find

<sup>4</sup><http://go.yurichev.com/17301>

<sup>5</sup><http://go.yurichev.com/17303>

### 5.3. COMMUNICATION WITH THE OUTER WORLD (WIN32)

the places from which those functions are called, and more importantly, how are the results used. One example: [8.2](#).

But if it is not a game, and `rand()` is still used, it is also interesting to know why. There are cases of unexpected `rand()` usage in data compression algorithms (for encryption imitation): [blog.yurichev.com](http://blog.yurichev.com).

#### **5.3.1 Often used functions in the Windows API**

These functions may be among the imported. It is worth to note that not every function might be used in the code that was written by the programmer. A lot of functions might be called from library functions and [CRT](#) code.

Some functions may have the `-A` suffix for the ASCII version and `-W` for the Unicode version.

- Registry access (`advapi32.dll`): `RegEnumKeyEx`, `RegEnumValue`, `RegGetValue`, `RegOpenKeyEx`, `RegQueryValueEx`.
- Access to text .ini-files (`kernel32.dll`): `GetPrivateProfileString`.
- Dialog boxes (`user32.dll`): `MessageBox`, `MessageBoxEx`, `CreateDialog`, `SetDlgItemText`, `GetDlgItemText`.
- Resources access ([6.5.2 on page 762](#)): (`user32.dll`): `LoadMenu`.
- TCP/IP networking (`ws2_32.dll`): `WSARecv`, `WSASend`.
- File access (`kernel32.dll`): `CreateFile`, `ReadFile`, `ReadFileEx`, `WriteFile`, `WriteFileEx`.
- High-level access to the Internet (`wininet.dll`): `WinHttpOpen`.
- Checking the digital signature of an executable file (`wintrust.dll`): `WinVerifyTrust`.
- The standard MSVC library (if it's linked dynamically) (`msvcr*.dll`): `assert`, `itoa`, `Itoa`, `open`, `printf`, `read`, `strcmp`, `atol`, `atoi`, `fopen`, `fread`, `fwrite`, `memcmp`, `rand`, `strlen`, `strrchr`.

#### **5.3.2 Extending trial period**

Registry access functions are frequent targets for those who try to crack trial period of some software, which may save installation date/time into registry.

Another popular target are `GetLocalTime()` and `GetSystemTime()` functions: a trial software, at each startup, must check current date/time somehow anyway.

#### **5.3.3 Removing nag dialog box**

A popular way to find out what causing popping nag dialog box is intercepting `MessageBox()`, `CreateDialog()` and `CreateWindow()` functions.

#### **5.3.4 tracer: Intercepting all functions in specific module**

There are INT3 breakpoints in the [tracer](#), that are triggered only once, however, they can be set for all functions in a specific DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Or, let's set INT3 breakpoints on all functions with the `xml` prefix in their name:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

On the other side of the coin, such breakpoints are triggered only once. Tracer will show the call of a function, if it happens, but only once. Another drawback—it is impossible to see the function's arguments.

Nevertheless, this feature is very useful when you know that the program uses a DLL, but you do not know which functions are actually used. And there are a lot of functions.

For example, let's see, what does the `uptime` utility from cygwin use:

## 5.4. STRINGS

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Thus we may see all that cygwin1.dll library functions that were called at least once, and where from:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!0EP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!0EP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!0EP+0xbba (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!0EP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!0EP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!0EP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!0EP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!0EP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!0EP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!0EP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!0EP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!0EP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!0EP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!0EP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!0EP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!0EP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!0EP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!0EP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!0EP+0x4c3 (0x4014c3))
```

## 5.4 Strings

### 5.4.1 Text strings

#### C/C++

The normal C strings are zero-terminated (ASCIIZ-strings).

The reason why the C string format is as it is (zero-terminated) is apparently historical. In [Dennis M. Ritchie, *The Evolution of the Unix Time-sharing System*, (1979)] we read:

A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

In Hiew or FAR Manager these strings looks like this:

```
int main()
{
    printf ("Hello, world!\n");
}
```

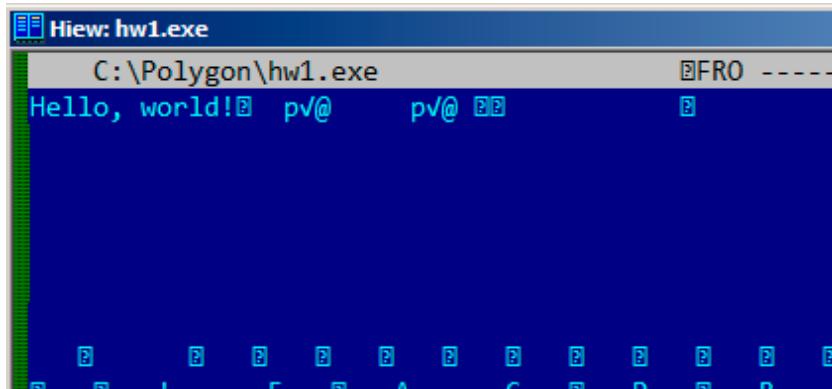


Figure 5.1: Hiew

## 5.4. STRINGS

### Borland Delphi

The string in Pascal and Borland Delphi is preceded by an 8-bit or 32-bit string length.

For example:

Listing 5.1: Delphi

```
CODE:00518AC8          dd 19h
CODE:00518ACC aLoading__Plea db 'Loading... , please wait.',0
...
CODE:00518AFC          dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

### Unicode

Often, what is called Unicode is a methods for encoding strings where each character occupies 2 bytes or 16 bits. This is a common terminological mistake. Unicode is a standard for assigning a number to each character in the many writing systems of the world, but does not describe the encoding method.

The most popular encoding methods are: UTF-8 (is widespread in Internet and \*NIX systems) and UTF-16LE (is used in Windows).

### UTF-8

UTF-8 is one of the most successful methods for encoding characters. All Latin symbols are encoded just like in ASCII, and the symbols beyond the ASCII table are encoded using several bytes. 0 is encoded as before, so all standard C string functions work with UTF-8 strings just like any other string.

Let's see how the symbols in various languages are encoded in UTF-8 and how it looks like in FAR, using the 437 codepage<sup>6</sup>:

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic) أَنَا قَادِرٌ عَلَى أَكْل الزُّجاج وَ هَذَا لَا يَؤْلِمُنِي .
(Hebrew) אָנוּ יִכּוֹל אַזְגָּה דְּקָרְבָּה וְזָהָה אָ.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

<sup>6</sup>The example and translations was taken from here: <http://go.yurichev.com/17304>

## 5.4. STRINGS

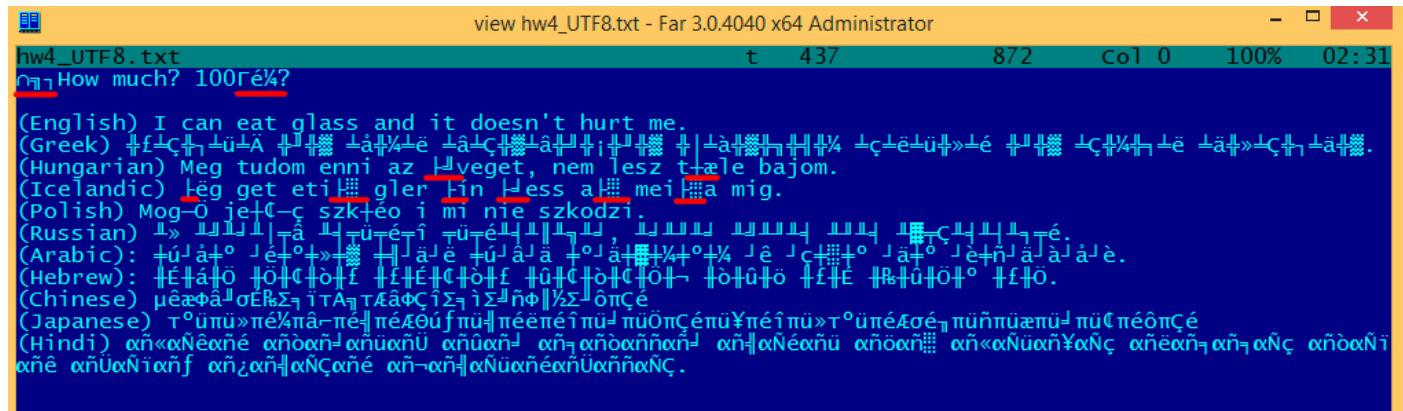


Figure 5.2: FAR: UTF-8

As you can see, the English language string looks the same as it is in ASCII.

The Hungarian language uses some Latin symbols plus symbols with diacritic marks.

These symbols are encoded using several bytes, these are underscored with red. It's the same story with the Icelandic and Polish languages.

There is also the “Euro” currency symbol at the start, which is encoded with 3 bytes.

The rest of the writing systems here have no connection with Latin.

At least in Russian, Arabic, Hebrew and Hindi we can see some recurring bytes, and that is not surprise: all symbols from a writing system are usually located in the same Unicode table, so their code begins with the same numbers.

At the beginning, before the “How much?” string we see 3 bytes, which are in fact the [BOM<sup>7</sup>](#). The [BOM](#) defines the encoding system to be used.

## UTF-16LE

Many win32 functions in Windows have the suffixes `-A` and `-W`. The first type of functions works with normal strings, the other with UTF-16LE strings (*wide*).

In the second case, each symbol is usually stored in a 16-bit value of type *short*.

The Latin symbols in UTF-16 strings look in Hiew or FAR like they are interleaved with zero byte:

```
int wmain()
{
    wprintf (L"Hello, world!\n");
}
```

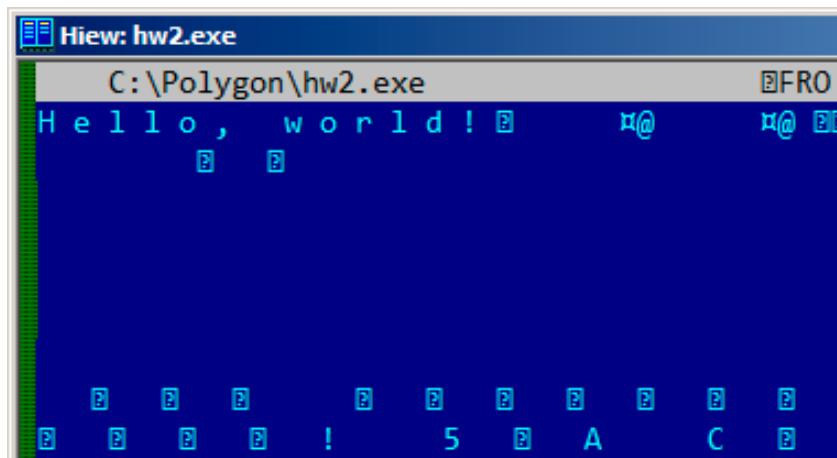


Figure 5.3: Hiew

<sup>7</sup>Byte order mark

## 5.4. *STRINGS*

We can see this often in Windows NT system files:

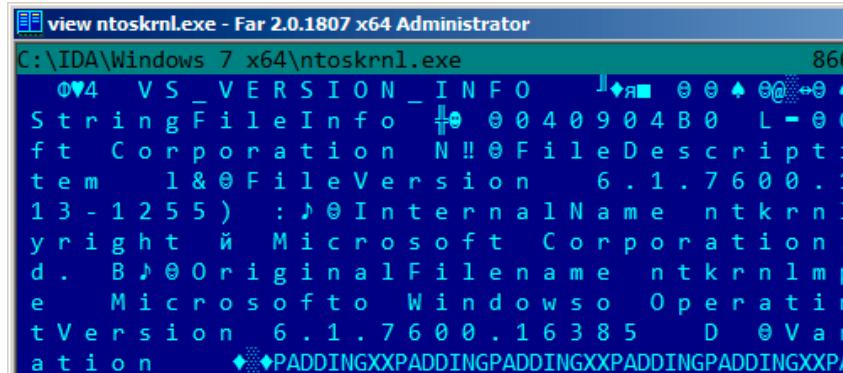


Figure 5.4: Hiew

Strings with characters that occupy exactly 2 bytes are called “Unicode” in IDA:

```
.data:0040E000 aHelloWorld:  
.data:0040E000                 unicode 0, <Hello, world!>  
.data:0040E000                 dw 0Ah, 0
```

Here is how the Russian language string is encoded in UTF-16LE:

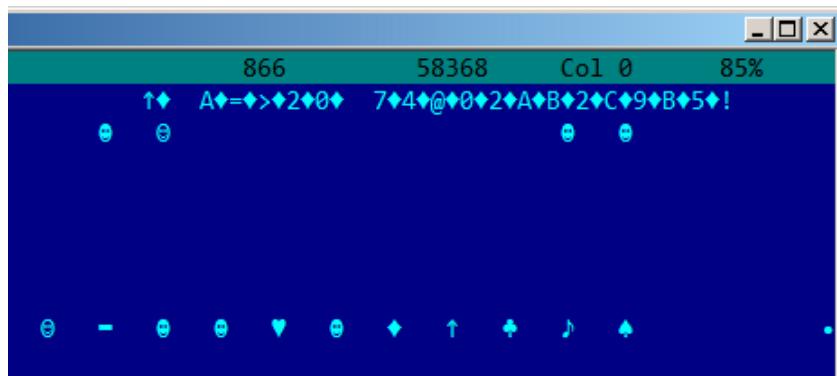


Figure 5.5: Hiew: UTF-16LE

What we can easily spot is that the symbols are interleaved by the diamond character (which has the ASCII code of 4). Indeed, the Cyrillic symbols are located in the fourth Unicode plane<sup>8</sup>. Hence, all Cyrillic symbols in UTF-16LE are located in the `0x400-0x4FF` range.

Let's go back to the example with the string written in multiple languages. Here is how it looks like in UTF-16LE.

<sup>8</sup>wikipedia

## 5.4. *STRINGS*

Figure 5.6: FAR: UTF-16LE

Here we can also see the **BOM** at the beginning. All Latin characters are interleaved with a zero byte. Some characters with diacritic marks (Hungarian and Icelandic languages) are also underscored in red.

## Base64

The base64 encoding is highly popular for the cases when you have to transfer binary data as a text string. In essence, this algorithm encodes 3 binary bytes into 4 printable characters: all 26 Latin letters (both lower and upper case), digits, plus sign ("+") and slash sign ("/"), 64 characters in total.

One distinctive feature of base64 strings is that they often (but not always) end with 1 or 2 padding equality symbol(s) ("="), for example:

AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuWs=

wVjbbVSfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lA0uQ==

The equality sign ("=") is never encountered in the middle of base64-encoded strings.

Now example of manual encoding. Let's encode 0x00, 0x11, 0x22, 0x33 hexadecimal bytes into base64 string:

```
$ echo -n "\x00\x11\x22\x33" | base64  
ABEiMw==
```

Let's put all 4 bytes in binary form, then regroup them into 6-bit groups:

	00	11	22	33				
	0000000000000100010010001000110011?????????????????????							
	A	B	E	i	M	w	=	=

Three first bytes (0x00, 0x11, 0x22) can be encoded into 4 base64 characters ("ABEi"), but the last one (0x33) — cannot be, so it's encoded using two characters ("Mw") and **padding** symbol ("=") is added twice to pad the last group to 4 characters. Hence, length of all correct base64 strings are always divisible by 4.

## 5.4. STRINGS

Base64 is often used when binary data needs to be stored in XML. "Armored" (i.e., in text form) PGP keys and signatures are encoded using base64.

Some people tries to use base64 to obfuscate strings: <http://blog.sec-consult.com/2016/01/deliberately.html><sup>9</sup>.

There are utilities for scanning an arbitrary binary files for base64 strings. One such utility is base64scanner<sup>10</sup>.

Another encoding system which was much more popular in UseNet and FidoNet is Uuencoding. Binary files are still encoded in Uuencode format in Phrack magazine. It offers mostly the same features, but is different from base64 in the sense that file name is also stored in header.

By the way: there is also close sibling to base64: base32, alphabet of which has 10 digits and 26 Latin characters. One well-known usage of it is onion addresses<sup>11</sup>, like: <http://3g2upl4pq6kufc4m.onion/>. URL can't have mixed-case Latin characters, so apparently, this is why Tor developers used base32.

### 5.4.2 Finding strings in binary

Actually, the best form of Unix documentation is frequently running the **strings** command over a program's object code. Using **strings**, you can get a complete list of the program's hard-coded file name, environment variables, undocumented options, obscure error messages, and so forth.

The Unix-Haters Handbook

The standard UNIX *strings* utility is quick-n-dirty way to see strings in file. For example, these are some strings from OpenSSH 7.2 sshd executable file:

```
...
0123
0123456789
0123456789abcdefABCDEF.:/
%02x

...
%.100s, line %lu: Bad permitopen specification <%.100s>
%.100s, line %lu: invalid criteria
%.100s, line %lu: invalid tun device
...
%.200s/.ssh/environment
...
2886173b9c9b6fdbdeda7a247cd636db38deaa.debug
$2a$06$r3.juUhZDlIbQa02dS9FuYxL1W9M81R1Tc92PoSNmzvpEqLkLGrK
...
3des-cbc
...
Bind to port %s on %s.
Bind to port %s on %s failed: %.200s.
/bin/login
/bin/sh
/bin/sh /etc/ssh/sshrc
...
D$4PQWR1
D$4PUj
D$4PV
D$4PVj
D$4PW
D$4PWj
D$4X
D$4XZj
D$4Y
...
diffie-hellman-group-exchange-sha1
```

<sup>9</sup> <http://archive.is/nDCas>

<sup>10</sup> <https://github.com/dennis714/base64scanner>

<sup>11</sup> <https://trac.torproject.org/projects/tor/wiki/doc/HiddenServiceNames>

## 5.4. STRINGS

```
diffie-hellman-group-exchange-sha256
digests
D$IPV
direct-streamlocal
direct-streamlocal@openssh.com
...
FFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A6...
...
```

There are options, error messages, file paths, imported dynamic modules and functions, some other strange strings (keys?) There is also unreadable noise—x86 code sometimes has chunks consisting of printable ASCII characters, up to 8 characters.

Of course, OpenSSH is open-source program. But looking at readable strings inside of some unknown binary is often a first step of analysis.

*grep* can be applied as well.

Hiew has the same capability (Alt-F6), as well as Sysinternals ProcessMonitor.

### 5.4.3 Error/debug messages

Debugging messages are very helpful if present. In some sense, the debugging messages are reporting what's going on in the program right now. Often these are `printf()`-like functions, which write to log-files, or sometimes do not writing anything but the calls are still present since the build is not a debug one but *release* one.

If local or global variables are dumped in debug messages, it might be helpful as well since it is possible to get at least the variable names. For example, one of such function in Oracle RDBMS is `ksdwrt()`.

Meaningful text strings are often helpful. The [IDA](#) disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen<sup>12</sup>.

The error messages may help us as well. In Oracle RDBMS, errors are reported using a group of functions. You can read more about them here: [blog.yurichev.com](http://blog.yurichev.com).

It is possible to find quickly which functions report errors and in which conditions.

By the way, this is often the reason for copy-protection systems to inarticulate cryptic error messages or just error numbers. No one is happy when the software cracker quickly understand why the copy-protection is triggered just by the error message.

One example of encrypted error messages is here: [8.5.2 on page 821](#).

### 5.4.4 Suspicious magic strings

Some magic strings which are usually used in backdoors looks pretty suspicious.

For example, there was a backdoor in the TP-Link WR740 home router<sup>13</sup>. The backdoor can activated using the following URL:

[http://192.168.0.1/userRpmNatDebugRpm26525557/start\\_art.html](http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html).

Indeed, the “userRpmNatDebugRpm26525557” string is present in the firmware.

This string was not googleable until the wide disclosure of information about the backdoor.

You would not find this in any [RFC](#)<sup>14</sup>.

You would not find any computer science algorithm which uses such strange byte sequences.

And it doesn't look like an error or debugging message.

So it's a good idea to inspect the usage of such weird strings.

<sup>12</sup>[blog.yurichev.com](http://blog.yurichev.com)

<sup>13</sup><http://sekurak.pl/tp-link-httptftp-backdoor/>

<sup>14</sup>Request for Comments

## 5.5. CALLS TO ASSERT()

Sometimes, such strings are encoded using base64.

So it's a good idea to decode them all and to scan them visually, even a glance should be enough.

More precise, this method of hiding backdoors is called “security through obscurity”.

## 5.5 Calls to assert()

Sometimes the presence of the `assert()` macro is useful too: commonly this macro leaves source file name, line number and condition in the code.

The most useful information is contained in the assert’s condition, we can deduce variable names or structure field names from it. Another useful piece of information are the file names—we can try to deduce what type of code is there. Also it is possible to recognize well-known open-source libraries by the file names.

Listing 5.2: Example of informative assert() calls

```
.text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D cmp  edx, 1
.text:107D4B30 jz   short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...
.text:107D52CA mov  edx, [ebp-4]
.text:107D52CD and  edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz   short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30      ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...
.text:107D6759 mov  cx, [eax+6]
.text:107D675D cmp  ecx, 0Ch
.text:107D6760 jle  short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c  ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

It is advisable to “google” both the conditions and file names, which can lead us to an open-source library. For example, if we “google” “`sp->lzw_nbits <= BITS_MAX`”, this predictably gives us some open-source code that’s related to the LZW compression.

## 5.6 Constants

Humans, including programmers, often use round numbers like 10, 100, 1000, in real life as well as in the code.

The practicing reverse engineer usually know them well in hexadecimal representation: `0b10=0xA`, `0b100=0x64`, `0b1000=0x3E8`, `0b10000=0x2710`.

The constants `0xFFFFFFFF` (`0b101`) and `0xFFFFFFFF` (`0b01`) are also popular—those are composed of alternating bits.

## 5.6. CONSTANTS

That may help to distinguish some signal from a signal where all bits are turned on (0b1111 ...) or off (0b0000 ...). For example, the `0x55AA` constant is used at least in the boot sector, [MBR<sup>15</sup>](#), and in the [ROM](#) of IBM-compatible extension cards.

Some algorithms, especially cryptographical ones use distinct constants, which are easy to find in code using [IDA](#).

For example, the MD5<sup>16</sup> algorithm initializes its own internal variables like this:

```
var int h0 := 0x67452301  
var int h1 := 0xEFCDAB89  
var int h2 := 0x98BADCFC  
var int h3 := 0x10325476
```

If you find these four constants used in the code in a row, it is highly probable that this function is related to MD5.

Another example are the CRC16/CRC32 algorithms, whose calculation algorithms often use precomputed tables like this one:

Listing 5.3: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */  
u16 const crc16_table[256] = {  
    0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,  
    0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,  
    0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0xE40,  
    ...
```

See also the precomputed table for CRC32: [3.5 on page 481](#).

In tableless CRC algorithms well-known polynomials are used, for example, 0xEDB88320 for CRC32.

### 5.6.1 Magic numbers

A lot of file formats define a standard file header where a *magic number(s)*<sup>17</sup> is used, single one or even several.

For example, all Win32 and MS-DOS executables start with the two characters “MZ”<sup>18</sup>.

At the beginning of a MIDI file the “MThd” signature must be present. If we have a program which uses MIDI files for something, it’s very likely that it must check the file for validity by checking at least the first 4 bytes.

This could be done like this: (*buf* points to the beginning of the loaded file in memory)

```
cmp [buf], 0x6468544D ; "MThd"  
jnz _error_not_a_MIDI_file
```

...or by calling a function for comparing memory blocks like `memcmp()` or any other equivalent code up to a `CMPSB` ([1.6 on page 1015](#)) instruction.

When you find such point you already can say where the loading of the MIDI file starts, also, we could see the location of the buffer with the contents of the MIDI file, what is used from the buffer, and how.

### Dates

Often, one may encounter number like `0x19870116`, which is clearly looks like a date (year 1987, 1th month (January), 16th day). This may be someone’s birthday (a programmer, his/her relative, child), or some other important date. The date may also be written in a reverse order, like `0x16011987`. American-style dates are also popular, like `0x01161987`.

Well-known example is `0x19540119` (magic number used in UFS2 superblock structure), which is a birth-day of Marshall Kirk McKusick, prominent FreeBSD contributor.

<sup>15</sup>Master Boot Record

<sup>16</sup>[wikipedia](#)

<sup>17</sup>[wikipedia](#)

<sup>18</sup>[wikipedia](#)

## 5.6. CONSTANTS

Stuxnet uses the number “19790509” (not as 32-bit number, but as string, though), and this led to speculation that the malware is connected to Israel <sup>19</sup>

Also, numbers like those are very popular in amateur-grade cryptography, for example, excerpt from the *secret function* internals from HASP3 dongle <sup>20</sup>:

```
void xor_pwd(void)
{
    int i;

    pwd^=0x09071966;
    for(i=0;i<8;i++)
    {
        al_buf[i]= pwd & 7; pwd = pwd >> 3;
    }
};

void emulate_func2(unsigned short seed)
{
    int i, j;
    for(i=0;i<8;i++)
    {
        ch[i] = 0;

        for(j=0;j<8;j++)
        {
            seed *= 0x1989;
            seed += 5;
            ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);
        }
    }
}
```

## DHCP

This applies to network protocols as well. For example, the DHCP protocol’s network packets contains the so-called *magic cookie*: `0x63538263`. Any code that generates DHCP packets somewhere must embed this constant into the packet. If we find it in the code we may find where this happens and, not only that. Any program which can receive DHCP packet must verify the *magic cookie*, comparing it with the constant.

For example, let’s take the `dhcpcore.dll` file from Windows 7 x64 and search for the constant. And we can find it, twice: it seems that the constant is used in two functions with descriptive names

`DhcpExtractOptionsForValidation()` and `DhcpExtractFullOptions()`:

Listing 5.4: `dhcpcore.dll` (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h ; DATA XREF: ↴
    ↴ DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h ; DATA XREF: ↴
    ↴ DhcpExtractFullOptions+97
```

And here are the places where these constants are accessed:

Listing 5.5: `dhcpcore.dll` (Windows 7 x64)

```
.text:000007FF6480875F mov     eax, [rsi]
.text:000007FF64808761 cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767 jnz     loc_7FF64817179
```

And:

Listing 5.6: `dhcpcore.dll` (Windows 7 x64)

```
.text:000007FF648082C7 mov     eax, [r12]
.text:000007FF648082CB cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1 jnz     loc_7FF648173AF
```

<sup>19</sup>This is a date of execution of Habib Elghanian, persian jew.

<sup>20</sup><https://web.archive.org/web/20160311231616/http://www.woodmann.com/fravia/bayu3.htm>

## 5.6.2 Specific constants

Sometimes, there is a specific constant for some type of code. For example, the author once dug into a code, where number 12 was encountered suspiciously often. Size of many arrays is 12, or multiple of 12 (24, etc). As it turned out, that code takes 12-channel audio file at input and process it.

And vice versa: for example, if a program works with text field which has length of 120 bytes, there has to be a constant 120 or 119 somewhere in the code. If UTF-16 is used, then  $2 \cdot 120$ . If a code works with network packets of fixed size, it's good idea to search for this constant in the code as well.

This is also true for amateur cryptography (license keys, etc). If encrypted block has size of  $n$  bytes, you may want to try to find occurrences of this number throughout the code. Also, if you see a piece of code which is been repeated  $n$  times in loop during execution, this may be encryption/decryption routine.

## 5.6.3 Searching for constants

It is easy in [IDA](#): Alt-B or Alt-I. And for searching for a constant in a big pile of files, or for searching in non-executable files, there is a small utility called *binary grep*<sup>21</sup>.

## 5.7 Finding the right instructions

If the program is utilizing FPU instructions and there are very few of them in the code, one can try to check each one manually with a debugger.

For example, we may be interested how Microsoft Excel calculates the formulae entered by user. For example, the division operation.

If we load excel.exe (from Office 2010) version 14.0.4756.1000 into [IDA](#), make a full listing and to find every `FDIV` instruction (except the ones which use constants as a second operand—obviously, they do not suit us):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we see that there are 144 of them.

We can enter a string like `= (1/3)` in Excel and check each instruction.

By checking each instruction in a debugger or [tracer](#) (one may check 4 instruction at a time), we get lucky and the sought-for instruction is just the 14th:

```
.text:3011E919 DC 33          fdiv    qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```

`ST(0)` holds the first argument (1) and second one is in `[EBX]`.

The instruction after `FDIV` (`FSTP`) writes the result in memory:

```
.text:3011E91B DD 1E          fstp    qword ptr [esi]
```

If we set a breakpoint on it, we can see the result:

<sup>21</sup>[GitHub](#)

## 5.8. SUSPICIOUS CODE PATTERNS

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

Also as a practical joke, we can modify it on the fly:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B, set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel shows 666 in the cell, finally convincing us that we have found the right point.

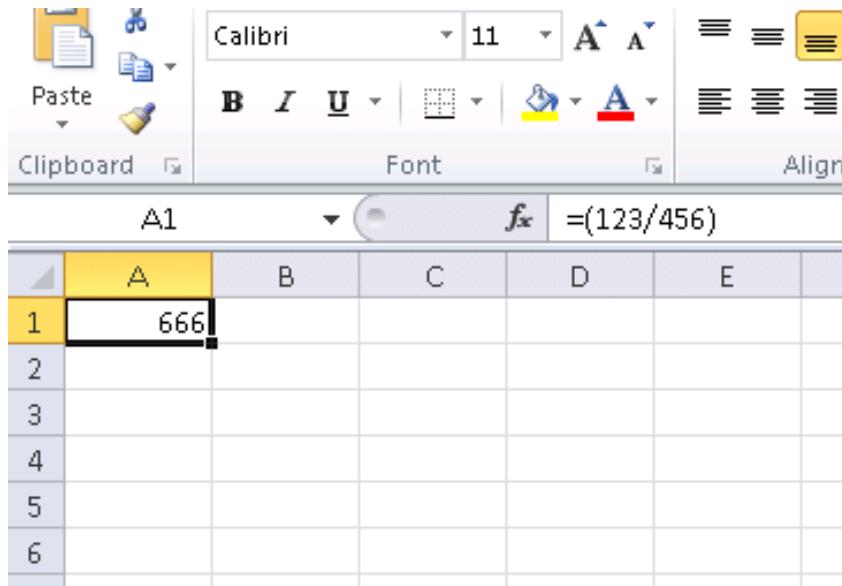


Figure 5.7: The practical joke worked

If we try the same Excel version, but in x64, we will find only 12 FDIV instructions there, and the one we looking for is the third one.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC, set(st0,666)
```

It seems that a lot of division operations of *float* and *double* types, were replaced by the compiler with SSE instructions like DIVSD (DIVSD is present 268 times in total).

## 5.8 Suspicious code patterns

### 5.8.1 XOR instructions

Instructions like X0R op, op (for example, X0R EAX, EAX) are usually used for setting the register value to zero, but if the operands are different, the “exclusive or” operation is executed.

## 5.8. SUSPICIOUS CODE PATTERNS

This operation is rare in common programming, but widespread in cryptography, including amateur one. It's especially suspicious if the second operand is a big number.

This may point to encrypting/decrypting, checksum computing, etc.

One exception to this observation worth noting is the “canary” ([1.20.3 on page 283](#)). Its generation and checking are often done using the `XOR` instruction.

This AWK script can be used for processing `IDA` listing (.lst) files:

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="esp") if ($4!="ebp") { print $1, $2, tmp, ", ", $4 } }' filename.lst
```

It is also worth noting that this kind of script can also match incorrectly disassembled code ([3.17 on page 537](#)).

### 5.8.2 Hand-written assembly code

Modern compilers do not emit the `LOOP` and `RCL` instructions. On the other hand, these instructions are well-known to coders who like to code directly in assembly language. If you spot these, it can be said that there is a high probability that this fragment of code was hand-written. Such instructions are marked as (M) in the instructions list in this appendix: [1.6 on page 1009](#).

Also the function prologue/epilogue are not commonly present in hand-written assembly.

Commonly there is no fixed system for passing arguments to functions in the hand-written code.

Example from the Windows 2003 kernel (ntoskrnl.exe file):

```
MultiplyTest proc near ; CODE XREF: Get386Stepping
    xor    cx, cx
loc_620555:           ; CODE XREF: MultiplyTest+E
    push   cx
    call   Multiply
    pop    cx
    jb    short locret_620563
    loop  loc_620555
    clc
locret_620563:        ; CODE XREF: MultiplyTest+C
    retn
MultiplyTest endp

Multiply    proc near ; CODE XREF: MultiplyTest+5
    mov    ecx, 81h
    mov    eax, 417A000h
    mul    ecx
    cmp    edx, 2
    stc
    jnz    short locret_62057F
    cmp    eax, 0FE7A000h
    stc
    jnz    short locret_62057F
    clc
locret_62057F:         ; CODE XREF: Multiply+10
                     ; Multiply+18
    retn
Multiply    endp
```

Indeed, if we look in the `WRK22` v1.2 source code, this code can be found easily in file `WRK-v1.2\base\ntos\ke\i386\cpu.asm`.

<sup>22</sup>Windows Research Kernel

## 5.9 Using magic numbers while tracing

Often, our main goal is to understand how the program uses a value that has been either read from file or received via network. The manual tracing of a value is often a very labor-intensive task. One of the simplest techniques for this (although not 100% reliable) is to use your own *magic number*.

This resembles X-ray computed tomography in some sense: a radiocontrast agent is injected into the patient's blood, which is then used to improve the visibility of the patient's internal structure in to the X-rays. It is well known how the blood of healthy humans percolates in the kidneys and if the agent is in the blood, it can be easily seen on tomography, how blood is percolating, and are there any stones or tumors.

We can take a 32-bit number like `0x0badf00d`, or someone's birth date like `0x11101979` and write this 4-byte number to some point in a file used by the program we investigate.

Then, while tracing this program with [tracer](#) in code coverage mode, with the help of [grep](#) or just by searching in the text file (of tracing results), we can easily see where the value has been used and how.

Example of *grepable* [tracer](#) results in cc mode:

```
0x150bf66 (_kziaia+0x14), e=      1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=      1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=      1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=      1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=      1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

This can be used for network packets as well. It is important for the *magic number* to be unique and not to be present in the program's code.

Aside of the [tracer](#), DosBox (MS-DOS emulator) in heavydebug mode is able to write information about all registers' states for each executed instruction of the program to a plain text file<sup>23</sup>, so this technique may be useful for DOS programs as well.

## 5.10 Loops

Whenever your program works with some kind of file, or buffer of some size, it has to be some kind of decrypting/processing loop inside of the code.

This is a real example of [tracer](#) tool output. There was a code which loads some kind of encrypted file of 258 bytes. I run it with the intention to get each instruction counts (a [DBI](#) tool will serve much better these days). And I quickly found a piece of code, which executed 259/258 times:

```
...
0x45a6b5 e= 1 [FS: MOV [0], EAX] EAX=0x218fb08
0x45a6bb e= 1 [MOV [EBP-254h], ECX] ECX=0x218fbdb8
0x45a6c1 e= 1 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbdb8
0x45a6c7 e= 1 [CMP [EAX+14h], 0] [EAX+14h]=0x102
0x45a6cb e= 1 [JZ 45A9F2h] ZF=false
0x45a6d1 e= 1 [MOV [EBP-0Dh], 1]
0x45a6d5 e= 1 [XOR ECX, ECX] ECX=0x218fbdb8
0x45a6d7 e= 1 [MOV [EBP-14h], CX] CX=0
0x45a6db e= 1 [MOV [EBP-18h], 0]
0x45a6e2 e= 1 [JMP 45A6EDh]
0x45a6e4 e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a6e7 e= 258 [ADD EDX, 1] EDX=0..5 (248 items skipped) 0xfd..0x101
0x45a6ea e= 258 [MOV [EBP-18h], EDX] EDX=1..6 (248 items skipped) 0xfe..0x102
0x45a6ed e= 259 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbdb8
0x45a6f3 e= 259 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (249 items skipped) 0xfe..0x102
0x45a6f6 e= 259 [CMP ECX, [EAX+14h]] ECX=0..5 (249 items skipped) 0xfe..0x102 [EAX+14h]=0x102
0x45a6f9 e= 259 [JNB 45A727h] CF=false,true
0x45a6fb e= 258 [MOV EDX, [EBP-254h]] [EBP-254h]=0x218fbdb8
0x45a701 e= 258 [MOV EAX, [EDX+10h]] [EDX+10h]=0x21ee4c8
0x45a704 e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a707 e= 258 [ADD ECX, 1] ECX=0..5 (248 items skipped) 0xfd..0x101
0x45a70a e= 258 [IMUL ECX, ECX, 1Fh] ECX=1..6 (248 items skipped) 0xfe..0x102
```

<sup>23</sup>See also my blog post about this DosBox feature: [blog.yurichev.com](http://blog.yurichev.com)

## 5.11. OTHER THINGS

```
0x45a70d e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a710 e= 258 [MOVZX EAX, [EAX+EDX]] [EAX+EDX]=1..6 (156 items skipped) 0xf3, 0xf8, 0xf9, 0..
    ↴ xfc, 0xfd
0x45a714 e= 258 [XOR EAX, ECX] EAX=1..6 (156 items skipped) 0xf3, 0xf8, 0xf9, 0xfc, 0xfd ECX=0..
    ↴ x1f, 0x3e, 0x5d, 0x7c, 0x9b (248 items skipped) 0x1ec2, 0x1ee1, 0x1f00, 0x1f1f, 0x1f3e
0x45a716 e= 258 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbdb8
0x45a71c e= 258 [MOV EDX, [ECX+10h]] [ECX+10h]=0x21ee4c8
0x45a71f e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a722 e= 258 [MOV [EDX+ECX], AL] AL=0..5 (77 items skipped) 0xe2, 0xee, 0xef, 0xf7, 0xfc
0x45a725 e= 258 [JMP 45A6E4h]
0x45a727 e= 1 [PUSH 5]
0x45a729 e= 1 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbdb8
0x45a72f e= 1 [CALL 45B500h]
0x45a734 e= 1 [MOV ECX, EAX] EAX=0x218fbdb8
0x45a736 e= 1 [CALL 45B710h]
0x45a73b e= 1 [CMP EAX, 5] EAX=5
...
...
```

As it turns out, this is the decrypting loop.

## 5.11 Other things

### 5.11.1 General idea

A reverse engineer should try to be in programmer's shoes as often as possible. To take his/her viewpoint and ask himself, how would one solve some task the specific case.

### 5.11.2 Order of functions in binary code

All functions located in a single .c or .cpp-file are compiled into corresponding object (.o) file. Later, linker puts all object files it needs together, not changing order or functions in them. As a consequence, if you see two or more consecutive functions, it means, that they were placed together in a single source code file (unless you're on border of two object files, of course.) This means these functions have something in common, that they are from the same API level, from same library, etc.

### 5.11.3 Tiny functions

Tiny functions like empty functions ([1.3 on page 5](#)) or function which returns just "true" (1) or "false" (0) ([1.4 on page 7](#)) are very common, and almost all decent compiler tends put only one such function into resulting executable code even if there was several similar functions in source code. So, whenever you see a tiny function consisting just of `mov eax, 1 / ret` which is referenced (and can be called) from many places, which are seems unconnected to each other, this may be a result of such optimization.

### 5.11.4 C++

[RTTI](#) ([3.19.1 on page 561](#))-data may be also useful for C++ class identification.

### 5.11.5 Some binary file patterns

All examples here were prepared on the Windows with active code page 437 <sup>24</sup> in console. Binary files internally may look visually different if another code page is set.

<sup>24</sup>[https://en.wikipedia.org/wiki/Code\\_page\\_437](https://en.wikipedia.org/wiki/Code_page_437)

**Arrays**

Sometimes, we can clearly spot an array of 16/32/64-bit values visually, in hex editor.

Here is an example of array of 16-bit values. We see that the first byte in pair is 7 or 8, and the second looks random:

E:\...\3affacde09fe21c28f1543db51145b.dat	h	1252	2175000	Col 0	23%	21:25		
000007CA70:	EF	07	C6	07	D6	07	26	08
000007CA80:	CC	07	AA	07	A2	07	AC	07
000007CA90:	09	08	CA	07	31	07	5E	07
000007CAA0:	E6	07	BD	07	D8	07	2F	08
000007CAB0:	B3	07	91	07	8B	07	97	07
000007CAC0:	03	08	CB	07	4C	07	61	07
000007CAD0:	E0	07	BB	07	DC	07	33	08
000007CAE0:	A4	07	84	07	81	07	90	07
000007CAF0:	FF	07	CD	07	65	07	69	07
000007CB00:	DE	07	BC	07	DF	07	33	08
000007CB10:	9F	07	82	07	81	07	93	07
000007CB20:	FE	07	CE	07	7E	07	78	07
000007CB30:	DE	07	BD	07	DF	07	32	08
000007CB40:	A1	07	87	07	88	07	9B	07
000007CB50:	02	08	CF	07	93	07	89	07
000007CB60:	E4	07	C0	07	DD	07	2D	08
000007CB70:	A9	07	90	07	91	07	A3	07
000007CB80:	04	08	D0	07	A7	07	9C	07
000007CB90:	E8	07	C7	07	DF	07	29	08
000007CBA0:	B4	07	9B	07	9B	07	AB	07
000007CBB0:	03	08	D5	07	BB	07	B3	07
000007CBC0:	EA	07	CD	07	E3	07	25	08
000007CBD0:	C1	07	A6	07	A5	07	B3	07
000007CBE0:	01	08	DC	07	CE	07	C8	07

1Help 2Wrap 3Quit 4Text 5 6Edit 7Search 80EM 9 10Quit

Figure 5.8: FAR: array of 16-bit values

I used a file containing 12-channel signal digitized using 16-bit ADC<sup>25</sup>.

<sup>25</sup>Analog-to-digital converter

## 5.11. OTHER THINGS

And here is an example of very typical MIPS code.

As we may recall, every MIPS (and also ARM in ARM mode or ARM64) instruction has size of 32 bits (or 4 bytes), so such code is array of 32-bit values.

By looking at this screenshot, we may see some kind of pattern.

Vertical red lines are added for clarity:

```

Hiew: FW96650A.bin
FW96650A.bin          FRO ----- 00005000
00005000: A0 B0 02 3C-04 00 BE AF-40 00 43 8C-21 F0 A0 03 aвв<п@ CM!Ёав
00005010: FF 1F 02 3C-21 E8 C0 03-FF FF 42 34-24 10 62 00 вв<!шв B4$вв
00005020: 00 A0 03 3C-25 10 43 00-04 00 BE 8F-08 00 E0 03 ав<%вв вв<Пв вв
00005030: 08 00 BD 27-F8 FF BD 27-A0 B0 02 3C-04 00 BE AF ввЛво Лв·ав<пвЛв
00005040: 48 00 43 8C-21 F0 A0 03-FF 1F 02 3C-21 E8 C0 03 Н CM!Ёав вв<!шв
00005050: FF FF 42 34-24 10 62 00-00 A0 03 3C-25 10 43 00 B4$вв ав<%вв
00005060: 04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27 вв<Пв ввЛво Лв
00005070: 21 10 00 00-04 00 BE AF-08 00 80 14-21 F0 A0 03 !вв вв<пв Ав!Ёав
00005080: A0 B0 03 3C-21 E8 C0 03-44 29 02 7C-3C 00 62 AC ав<!швD|< вв
00005090: 04 00 BE 8F-08 00 E0 03-08 00 BD 27-01 00 03 24 вв<Пв ввЛво вв$
000050A0: 44 29 62 7C-A0 B0 03 3C-21 E8 C0 03-3C 00 62 AC D|ав<!шв вв вв
000050B0: 04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27 вв<Пв ввЛво Лв
000050C0: A0 B0 02 3C-04 00 BE AF-84 00 43 8C-21 F0 A0 03 ав<пв СМ!Ёав
000050D0: 21 E8 C0 03-C4 FF 03 7C-84 00 43 AC-04 00 BE 8F !шв- вв|Д СМв вв
000050E0: 08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C вв ввЛво Лв·ав<
000050F0: 04 00 BE AF-20 00 43 8C-21 F0 A0 03-01 00 04 24 вв<пв СМ!Ёав вв$
00005100: 21 E8 C0 03-44 08 83 7C-20 00 43 AC-04 00 BE 8F !швДвГ| СМв вв
00005110: 08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C вв ввЛво Лв·ав<
00005120: 04 00 BE AF-20 00 43 8C-21 F0 A0 03-21 E8 C0 03 вв<пв СМ!Ёав!шв
00005130: 44 08 03 7C-20 00 43 AC-04 00 BE 8F-08 00 E0 03 D| СМв вв вв
00005140: 08 00 BD 27-F8 FF BD 27-A0 B0 03 3C-04 00 BE AF вв<Пв ввЛво Лв
00005150: 10 00 62 8C-01 00 08 24-04 A5 02 7D-08 00 09 24 вв вв{ } вв вв
00005160: 10 00 62 AC-04 7B 22 7D-04 48 02 7C-04 84 02 7D вв вв вв вв
00005170: 10 00 62 AC-21 F0 A0 03-21 18 00 00-A0 B0 0B 3C вв вв!Ёав! вв
00005180: 51 00 0A 24-02 00 88 94-00 00 89 94-00 44 08 00 Q вв ИФ ЙФ Д|
00005190: 25 40 09 01-01 00 63 24-14 00 68 AD-F9 FF 6A 14 %@ вв вв вв
000051A0: 04 00 84 24-21 18 00 00-A0 B0 0A 3C-07 00 09 24 вв вв вв вв
000051B0: 02 00 A4 94-00 00 A8 94-00 24 04 00-25 20 88 00 вв вв вв вв

```

1Global 2FilBlk 3CryBlk 4ReLoad 5 6String 7Direct 8Table 9 10Leave 11

Figure 5.9: Hiew: very typical MIPS code

Another example of such pattern here is book: [9.5 on page 969](#).

**Sparse files**

This is sparse file with data scattered amidst almost empty file. Each space character here is in fact zero byte (which is looks like space). This is a file to program FPGA (Altera Stratix GX device). Of course, files like these can be compressed easily, but formats like this one are very popular in scientific and engineering software where efficient access is important while compactness is not.

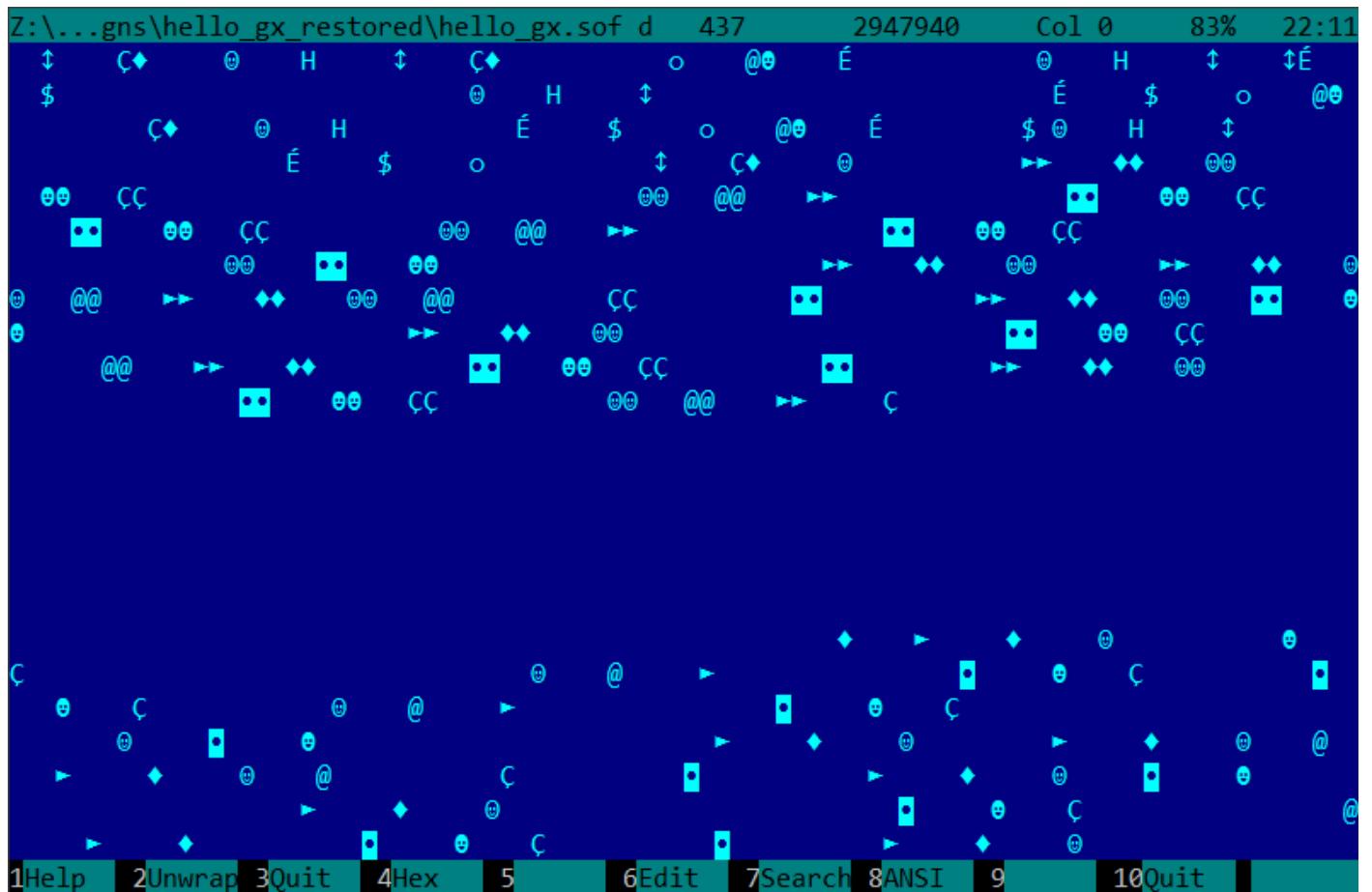


Figure 5.10: FAR: Sparse file

**Compressed file**

This file is just some compressed archive. It has relatively high entropy and visually looks just chaotic. This is how compressed and/or encrypted files looks like.

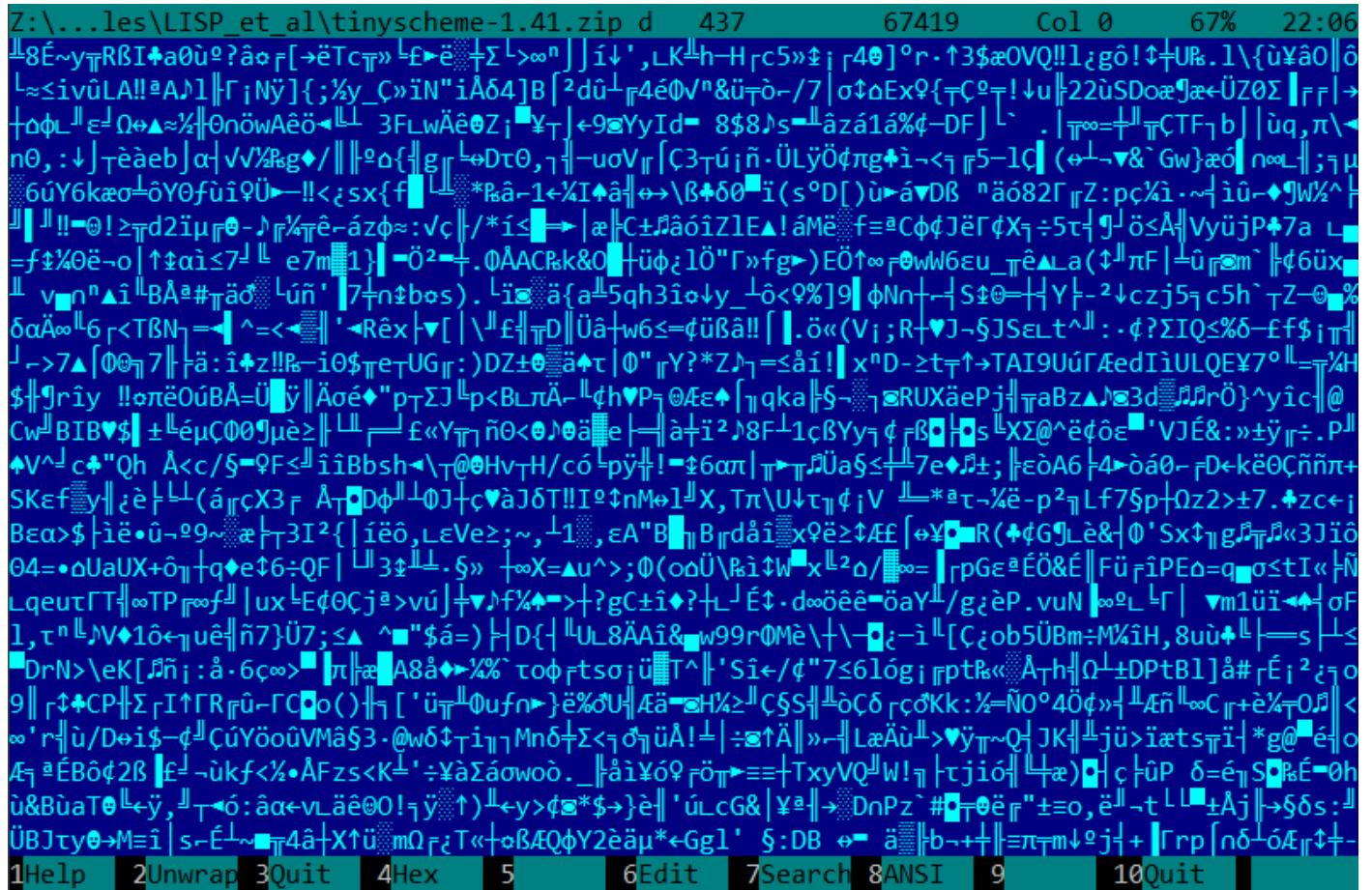


Figure 5.11: FAR: Compressed file

## 5.11. OTHER THINGS

### CDFS<sup>26</sup>

OS installations are usually distributed as ISO files which are copies of CD/DVD discs. Filesystem used is named CDFS, here is you see file names mixed with some additional data. This can be file sizes, pointers to another directories, file attributes, etc. This is how typical filesystems may look internally.

```
Z:\...untu\ubuntu-15.10-desktop-i386.iso d 437 1226964992 Col 0 0% 21:59

ä # # • • s¤§»‡; @ @ 00 SP•@Jn
PX$@mA Am@ @ TF→@Jss¤§»‡, s¤§»‡, s¤§»‡, CE@$ $ φ
φ ` # # • • s¤§»‡; @ @ 000PX$@mA Am@ @ TF→@Jss¤§»‡, s¤§»‡, n % % • • s¤§»‡; @ @ 0+DISKPX$@mA Am@ @ TF
s¤§»‡, s¤§»‡, n % % • • s¤§»‡; @ @ 0+DISKPX$@mA Am@ @ TF→@Jss¤§»‡, s¤§»‡, NM@0 .diskn & & • • s¤§»‡; @ @ 0+BOOT PX$@mA Am@ @
TF→@Jss¤§»‡% s¤§»‡, s¤§»‡% NM@0 boot r ( ( ( • • s¤§»‡; @ @ 0
TF→@Jss¤§»‡( s¤§»‡, s¤§»‡( NM@0 boot r ( ( ( • • s¤§»‡; @ @ 0
↑CASPER PX$@mA Am@ @ TF→@Jss¤§»‡( s¤§»‡, s¤§»‡( NM@0 casper n )
) • • s¤§»‡; @ @ 0+DISTSPX$@mA Am@ @ TF→@Jss¤§»‡→ s¤§»‡ s¤§»‡→ s¤§»‡→ NM@0 distsr 1 1 • • s¤§»‡; @ @ 0+INSTALLPX$@mA Am@ @
TF→@Jss¤§»‡( s¤§»‡, s¤§»‡( NM@0 installv 2 2 H H s¤§»‡; @ @ 0+ISOLINUX PX$@mA Am@ @
TF→@Jss¤§»‡( s¤§»‡( NM@0 isolinux z p@ @p`J
s¤§»‡; @ @ 0+MD5SUM.TXT PX$@$ü ü$@ @ TF→@Jss¤§»‡: s¤§»‡; s¤§»‡:
: NM@0 md5sum.txt n ; ; • • s¤§»‡; @ @ 0+PICS PX$@mA Am@ @
TF→@Jss¤§»‡→ s¤§»‡→ s¤§»‡→ NM@0 pics n < < • • s¤§»‡; @ @ 0+POOL PX$@mA
Am@ @ TF→@Jss¤§»‡→ s¤§»‡→ s¤§»‡→ NM@0 pool r W W • • s¤§»‡; @ @
@ 0+PRESEEDPX$@mA Am@ @ TF→@Jss¤§»‡→ s¤§»‡→ s¤§»‡→ NM@0 preseed
è S@ @S@ as¤§»‡; @ @ 0+README.DISKDEFINES PX$@$ü ü$@ @
TF→@Jss¤§»‡→ s¤§»‡: s¤§»‡→ NM@0 README.diskdefines x ≤ ≤ s¤§»‡; @ @ 0+UBUNT
U.PX$@mi im@ @ TF→@Jss¤§»‡→ s¤§»‡→ s¤§»‡→ NM@0 ubuntuSL@ @ @
```

1Help 2Unwrap 3Quit 4Hex 5 6Edit 7Search 8ANSI 9 10Quit

Figure 5.12: FAR: ISO file: Ubuntu 15 installation [CD<sup>27</sup>](#)

<sup>26</sup>Compact Disc File System

**32-bit x86 executable code**

This is how 32-bit x86 executable code looks like. It has not very high entropy, because some bytes occurred more often than others.

The screenshot shows a window titled "universal-USB-Installer-1.9.6.0.exe" with the file path "Z:\...". The window displays assembly code in a monospaced font. The code is highly compressed and contains many repeated bytes. At the bottom of the window, there is a menu bar with the following items:

- 1Help
- 2Unwrap
- 3Quit
- 4Hex
- 5
- 6Edit
- 7Search
- 8ANSI
- 9
- 10Quit

Figure 5.13: FAR: Executable 32-bit x86 code

## 5.11. OTHER THINGS

### BMP graphics files

BMP files are not compressed, so each byte (or group of bytes) describes each pixel. I've found this picture somewhere inside my installed Windows 8.1:



Figure 5.14: Example picture

You see that this picture has some pixels which unlikely can be compressed very good (around center), but there are long one-color lines at top and bottom. Indeed, lines like these also looks as lines during viewing the file:

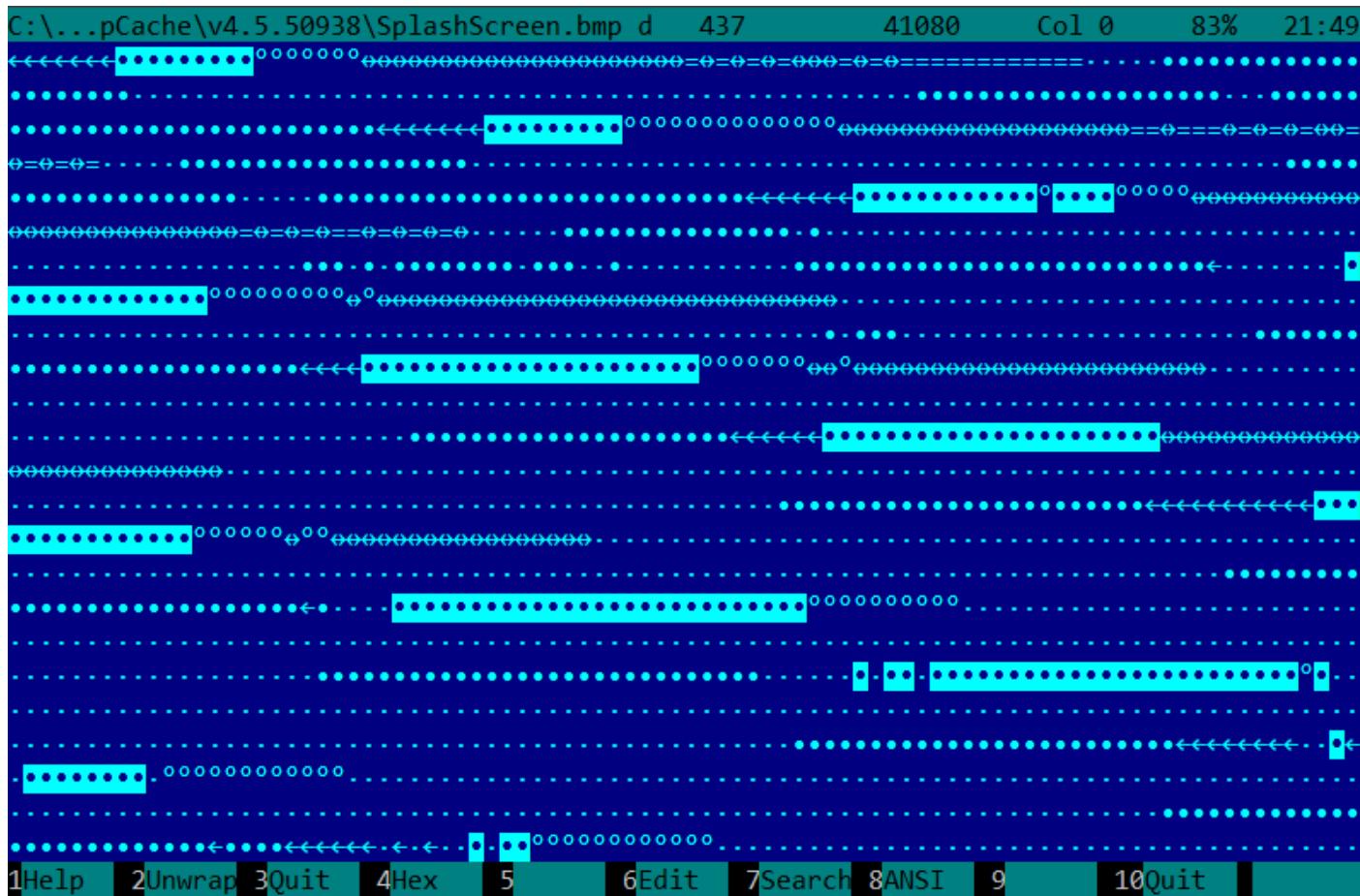


Figure 5.15: BMP file fragment

#### 5.11.6 Memory “snapshots” comparing

The technique of the straightforward comparison of two memory snapshots in order to see changes was often used to hack 8-bit computer games and for hacking “high score” files.

## 5.11. OTHER THINGS

For example, if you had a loaded game on an 8-bit computer (there isn't much memory on these, but the game usually consumes even less memory) and you know that you have now, let's say, 100 bullets, you can do a "snapshot" of all memory and back it up to some place. Then shoot once, the bullet count goes to 99, do a second "snapshot" and then compare both: it must be a byte somewhere which has been 100 at the beginning, and now it is 99.

Considering the fact that these 8-bit games were often written in assembly language and such variables were global, it can be said for sure which address in memory has holding the bullet count. If you searched for all references to the address in the disassembled game code, it was not very hard to find a piece of code **decrementing** the bullet count, then to write a **NOP** instruction there, or a couple of **NOP**-s, and then have a game with 100 bullets forever. Games on these 8-bit computers were commonly loaded at the constant address, also, there were not much different versions of each game (commonly just one version was popular for a long span of time), so enthusiastic gamers knew which bytes must be overwritten (using the BASIC's instruction **POKE**) at which address in order to hack it. This led to "cheat" lists that contained **POKE** instructions, published in magazines related to 8-bit games. See also: [wikipedia](#).

Likewise, it is easy to modify "high score" files, this does not work with just 8-bit games. Notice your score count and back up the file somewhere. When the "high score" count gets different, just compare the two files, it can even be done with the DOS utility FC<sup>28</sup> ("high score" files are often in binary form).

There will be a point where a couple of bytes are different and it is easy to see which ones are holding the score number. However, game developers are fully aware of such tricks and may defend the program against it.

Somewhat similar example in this book is: [9.3 on page 957](#).

## Windows registry

It is also possible to compare the Windows registry before and after a program installation.

It is a very popular method of finding which registry elements are used by the program. Perhaps, this is the reason why the "windows registry cleaner" shareware is so popular.

## Blink-comparator

Comparison of files or memory snapshots remind us blink-comparator <sup>29</sup>: a device used by astronomers in past, intended to find moving celestial objects.

Blink-comparator allows to switch quickly between two photographies shot in different time, so astronomer would spot the difference visually.

By the way, Pluto was discovered by blink-comparator in 1930.

---

<sup>28</sup>MS-DOS utility for comparing binary files

<sup>29</sup><http://go.yurichev.com/17348>

# Chapter 6

## OS-specific

### 6.1 Arguments passing methods (calling conventions)

#### 6.1.1 cdecl

This is the most popular method for passing arguments to functions in the C/C++ languages.

The caller also must return the value of the **stack pointer** (`ESP`) to its initial state after the **callee** function exits.

Listing 6.1: cdecl

```
push arg3  
push arg2  
push arg1  
call function  
add esp, 12 ; returns ESP
```

#### 6.1.2 stdcall

It's almost the same as *cdecl*, with the exception that the **callee** must set `ESP` to the initial state by executing the `RET x` instruction instead of `RET`, where `x = arguments number * sizeof(int)`<sup>1</sup>. The **caller** is not adjusting the **stack pointer**, there are no `add esp, x` instruction.

Listing 6.2: stdcall

```
push arg3  
push arg2  
push arg1  
call function  
  
function:  
... do something ...  
ret 12
```

The method is ubiquitous in win32 standard libraries, but not in win64 (see below about win64).

For example, we can take the function from [1.85 on page 98](#) and change it slightly by adding the `__stdcall` modifier:

```
int __stdcall f2 (int a, int b, int c)  
{  
    return a*b+c;  
};
```

It is to be compiled in almost the same way as [1.86 on page 98](#), but you will see `RET 12` instead of `RET`. `SP` is not updated in the **caller**.

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

As a consequence, the number of function arguments can be easily deduced from the `RETN n` instruction: just divide  $n$  by 4.

Listing 6.3: MSVC 2010

```
_a$ = 8          ; size = 4
_b$ = 12         ; size = 4
_c$ = 16         ; size = 4
_f2@12 PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    imul   eax, DWORD PTR _b$[ebp]
    add    eax, DWORD PTR _c$[ebp]
    pop    ebp
    ret    12
_f2@12 ENDP

; ...
    push    3
    push    2
    push    1
    call    _f2@12
    push    eax
    push    OFFSET $SG81369
    call    _printf
    add    esp, 8
```

### Functions with variable number of arguments

`printf()`-like functions are, probably, the only case of functions with a variable number of arguments in C/C++, but it is easy to illustrate an important difference between *cdecl* and *stdcall* with their help. Let's start with the idea that the compiler knows the argument count of each `printf()` function call.

However, the called `printf()`, which is already compiled and located in MSVCRT.DLL (if we talk about Windows), does not have any information about how much arguments were passed, however it can determine it from the format string.

Thus, if `printf()` would be a *stdcall* function and restored *stack pointer* to its initial state by counting the number of arguments in the format string, this could be a dangerous situation, when one programmer's typo can provoke a sudden program crash. Thus it is not suitable for such functions to use *stdcall*, *cdecl* is better.

### 6.1.3 fastcall

That's the general naming for the method of passing some arguments via registers and the rest via the stack. It worked faster than *cdecl/stdcall* on older CPUs (because of smaller stack pressure). It may not help to gain any significant performance on latest (much more complex) CPUs, however.

It is not standardized, so the various compilers can do it differently. It's a well known caveat: if you have two DLLs and the one uses another one, and they are built by different compilers with different *fastcall* calling conventions, you can expect problems.

Both MSVC and GCC pass the first and second arguments via `ECX` and `EDX` and the rest of the arguments via the stack.

The *stack pointer* must be restored to its initial state by the *callee* (like in *stdcall*).

Listing 6.4: fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
```

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

```
.. do something ..  
ret 4
```

For example, we may take the function from [1.85 on page 98](#) and change it slightly by adding a `__fastcall` modifier:

```
int __fastcall f3 (int a, int b, int c)  
{  
    return a*b+c;  
};
```

Here is how it is to be compiled:

Listing 6.5: Optimizing MSVC 2010 /Ob0

```
_c$ = 8      ; size = 4  
@f3@12 PROC  
; _a$ = ecx  
; _b$ = edx  
    mov    eax, ecx  
    imul   eax, edx  
    add    eax, DWORD PTR _c$[esp-4]  
    ret    4  
@f3@12 ENDP  
  
; ...  
  
    mov    edx, 2  
    push   3  
    lea    ecx, DWORD PTR [edx-1]  
    call   @f3@12  
    push   eax  
    push   OFFSET $SG81390  
    call   _printf  
    add    esp, 8
```

We see that the `callee` returns `SP` by using the `RETN` instruction with an operand.

Which implies that the number of arguments can be deduced easily here as well.

### GCC `regparm`

It is the evolution of `fastcall`<sup>2</sup> in some sense. With the `-mregparm` option it is possible to set how many arguments are to be passed via registers (3 is the maximum). Thus, the `EAX`, `EDX` and `ECX` registers are to be used.

Of course, if the number the of arguments is less than 3, not all 3 registers are to be used.

The `caller` restores the `stack pointer` to its initial state.

For example, see ([1.22.1 on page 306](#)).

### Watcom/OpenWatcom

Here it is called “register calling convention”. The first 4 arguments are passed via the `EAX`, `EDX`, `EBX` and `ECX` registers. All the rest—via the stack.

These functions has an underscore appended to the function name in order to distinguish them from those having a different calling convention.

<sup>2</sup><http://go.yurichev.com/17040>

### 6.1.4 **thiscall**

This is passing the object's *this* pointer to the function-method, in C++.

In MSVC, *this* is usually passed in the ECX register.

In GCC, the *this* pointer is passed as the first function-method argument. Thus it will be very visible that internally: all function-methods have an extra argument.

For an example, see ([3.19.1 on page 546](#)).

### 6.1.5 **x86-64**

#### Windows x64

The method of for passing arguments in Win64 somewhat resembles `fastcall`. The first 4 arguments are passed via RCX, RDX, R8 and R9, the rest—via the stack. The *caller* also must prepare space for 32 bytes or 4 64-bit values, so then the *callee* can save there the first 4 arguments. Short functions may use the arguments' values just from the registers, but larger ones may save their values for further use.

The *caller* also must return the *stack pointer* into its initial state.

This calling convention is also used in Windows x86-64 system DLLs (instead of `stdcall` in win32).

Example:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
    printf ("%d %d %d %d %d %d\n", a, b, c, d, e, f, g);
}

int main()
{
    f1(1,2,3,4,5,6,7);
}
```

Listing 6.6: MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d', 0aH, 00H

main PROC
    sub    rsp, 72

    mov    DWORD PTR [rsp+48], 7
    mov    DWORD PTR [rsp+40], 6
    mov    DWORD PTR [rsp+32], 5
    mov    r9d, 4
    mov    r8d, 3
    mov    edx, 2
    mov    ecx, 1
    call   f1

    xor    eax, eax
    add    rsp, 72
    ret    0
main ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1    PROC
$LN3:   mov    DWORD PTR [rsp+32], r9d
```

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

```

    mov    DWORD PTR [rsp+24], r8d
    mov    DWORD PTR [rsp+16], edx
    mov    DWORD PTR [rsp+8], ecx
    sub    rsp, 72

    mov    eax, DWORD PTR g$[rsp]
    mov    DWORD PTR [rsp+56], eax
    mov    eax, DWORD PTR f$[rsp]
    mov    DWORD PTR [rsp+48], eax
    mov    eax, DWORD PTR e$[rsp]
    mov    DWORD PTR [rsp+40], eax
    mov    eax, DWORD PTR d$[rsp]
    mov    DWORD PTR [rsp+32], eax
    mov    r9d, DWORD PTR c$[rsp]
    mov    r8d, DWORD PTR b$[rsp]
    mov    edx, DWORD PTR a$[rsp]
    lea    rcx, OFFSET FLAT:$SG2937
    call   printf

    add    rsp, 72
    ret    0
f1    ENDP

```

Here we clearly see how 7 arguments are passed: 4 via registers and the remaining 3 via the stack.

The code of the f1() function's prologue saves the arguments in the “scratch space”—a space in the stack intended exactly for this purpose.

This is arranged so because the compiler cannot be sure that there will be enough registers to use without these 4, which will otherwise be occupied by the arguments until the function's execution end.

The “scratch space” allocation in the stack is the caller's duty.

Listing 6.7: Optimizing MSVC 2012 /O**b**

```

$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1    PROC
$LN3:
    sub    rsp, 72

    mov    eax, DWORD PTR g$[rsp]
    mov    DWORD PTR [rsp+56], eax
    mov    eax, DWORD PTR f$[rsp]
    mov    DWORD PTR [rsp+48], eax
    mov    eax, DWORD PTR e$[rsp]
    mov    DWORD PTR [rsp+40], eax
    mov    DWORD PTR [rsp+32], r9d
    mov    r9d, r8d
    mov    r8d, edx
    mov    edx, ecx
    lea    rcx, OFFSET FLAT:$SG2777
    call   printf

    add    rsp, 72
    ret    0
f1    ENDP

main  PROC
sub    rsp, 72

    mov    edx, 2
    mov    DWORD PTR [rsp+48], 7
    mov    DWORD PTR [rsp+40], 6

```

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

```
lea    r9d, QWORD PTR [rdx+2]
lea    r8d, QWORD PTR [rdx+1]
lea    ecx, QWORD PTR [rdx-1]
mov    DWORD PTR [rsp+32], 5
call   f1

xor    eax, eax
add    rsp, 72
ret    0
main  ENDP
```

If we compile the example with optimizations, it is to be almost the same, but the “scratch space” will not be used, because it won’t be needed.

Also take a look on how MSVC 2012 optimizes the loading of primitive values into registers by using LEA ([.1.6 on page 1011](#)). MOV would be 1 byte longer here (5 instead of 4).

Another example of such thing is: [8.1.1 on page 796](#).

### Windows x64: Passing *this* (C/C++)

The *this* pointer is passed in RCX, the first argument of the method is in RDX, etc. For an example see: [3.19.1 on page 548](#).

### Linux x64

The way arguments are passed in Linux for x86-64 is almost the same as in Windows, but 6 registers are used instead of 4 (RDI, RSI, RDX, RCX, R8, R9) and there is no “scratch space”, although the callee may save the register values in the stack, if it needs/wants to.

Listing 6.8: Optimizing GCC 4.7.3

```
.LC0:
    .string "%d %d %d %d %d %d\n"
f1:
    sub    rsp, 40
    mov    eax, DWORD PTR [rsp+48]
    mov    DWORD PTR [rsp+8], r9d
    mov    r9d, ecx
    mov    DWORD PTR [rsp], r8d
    mov    ecx, esi
    mov    r8d, edx
    mov    esi, OFFSET FLAT:.LC0
    mov    edx, edi
    mov    edi, 1
    mov    DWORD PTR [rsp+16], eax
    xor    eax, eax
    call   __printf_chk
    add    rsp, 40
    ret
main:
    sub    rsp, 24
    mov    r9d, 6
    mov    r8d, 5
    mov    DWORD PTR [rsp], 7
    mov    ecx, 4
    mov    edx, 3
    mov    esi, 2
    mov    edi, 1
    call   f1
    add    rsp, 24
    ret
```

N.B.: here the values are written into the 32-bit parts of the registers (e.g., EAX) but not in the whole 64-bit register (RAX). This is because each write to the low 32-bit part of a register automatically clears the high 32 bits. Supposedly, it was decided in AMD to do so to simplify porting code to x86-64.

**6.1.6 Return values of *float* and *double* type**

In all conventions except in Win64, the values of type *float* or *double* are returned via the FPU register ST(0).

In Win64, the values of *float* and *double* types are returned in the low 32 or 64 bits of the XMM0 register.

**6.1.7 Modifying arguments**

Sometimes, C/C++ programmers (not limited to these PLs, though), may ask, what can happen if they modify the arguments?

The answer is simple: the arguments are stored in the stack, that is where the modification takes place.

The calling functions is not using them after the callee's exit (the author of these lines has never seen any such case in his practice).

```
#include <stdio.h>

void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
}
```

Listing 6.9: MSVC 2012

```
_a$ = 8                                ; size = 4
_b$ = 12                               ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _a$[ebp], eax
    mov     ecx, DWORD PTR _a$[ebp]
    push    ecx
    push    OFFSET $SG2938 ; '%d', 0aH
    call    _printf
    add     esp, 8
    pop     ebp
    ret     0
_f ENDP
```

So yes, one can modify the arguments easily. Of course, if it is not references in C++ ([3.19.3 on page 562](#)), and if you don't modify data to which a pointer points to, then the effect will not propagate outside the current function.

Theoretically, after the callee's return, the caller could get the modified argument and use it somehow. Maybe if it is written directly in assembly language.

For example, code like this will be generated by usual C/C++ compiler:

```
push    456      ; will be b
push    123      ; will be a
call    f         ; f() modifies its first argument
add    esp, 2*4
```

We can rewrite this code like:

```
push    456      ; will be b
push    123      ; will be a
call    f         ; f() modifies its first argument
pop     eax
add    esp, 4
; EAX=1st argument of f() modified in f()
```

Hard to imagine, why anyone would need this, but this is possible in practice. Nevertheless, the C/C++ languages standards don't offer any way to do so.

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

### 6.1.8 Taking a pointer to function argument

...even more than that, it's possible to take a pointer to the function's argument and pass it to another function:

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
}
```

It's hard to understand how it works until we can see the code:

Listing 6.10: Optimizing MSVC 2010

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f      PROC
    lea    eax, DWORD PTR _a$[esp-4] ; just get the address of value in local stack
    push   eax                      ; and pass it to modify_a()
    call   _modify_a
    mov    ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
    push   ecx                      ; and pass it to printf()
    push   OFFSET $SG2796 ; '%d'
    call   _printf
    add    esp, 12
    ret    0
_f      ENDP
```

The address of the place in the stack where *a* has been passed is just passed to another function. It modifies the value addressed by the pointer and then `printf()` prints the modified value.

The observant reader might ask, what about calling conventions where the function's arguments are passed in registers?

That's a situation where the *Shadow Space* is used.

The input value is copied from the register to the *Shadow Space* in the local stack, and then this address is passed to the other function:

Listing 6.11: Optimizing MSVC 2012 x64

```
$SG2994 DB      '%d', 0aH, 00H

a$ = 48
_f      PROC
    mov    DWORD PTR [rsp+8], ecx ; save input value in Shadow Space
    sub    rsp, 40
    lea    rcx, QWORD PTR a$[rsp] ; get address of value and pass it to modify_a()
    call   modify_a
    mov    edx, DWORD PTR a$[rsp] ; reload value from Shadow Space and pass it to printf
    ↴ ()
    lea    rcx, OFFSET FLAT:$SG2994 ; '%d'
    call   printf
    add    rsp, 40
    ret    0
_f      ENDP
```

GCC also stores the input value in the local stack:

Listing 6.12: Optimizing GCC 4.9.1 x64

```
.LC0:
    .string "%d\n"
f:
```

## 6.2. THREAD LOCAL STORAGE

```
sub    rsp, 24
mov    DWORD PTR [rsp+12], edi ; store input value to the local stack
lea    rdi, [rsp+12]           ; take an address of the value and pass it to modify_a()
        (
call   modify_a
mov    edx, DWORD PTR [rsp+12] ; reload value from the local stack and pass it to printf()
        )
printf()
    mov    esi, OFFSET FLAT:.LC0    ; '%d'
    mov    edi, 1
    xor    eax, eax
    call   __printf_chk
    add    rsp, 24
    ret
```

GCC for ARM64 does the same, but this space is called *Register Save Area* here:

Listing 6.13: Optimizing GCC 4.9.1 ARM64

```
f:
stp    x29, x30, [sp, -32]!
add    x29, sp, 0          ; setup FP
add    x1, x29, 32         ; calculate address of variable in Register Save Area
str    w0, [x1,-4]!        ; store input value there
mov    x0, x1              ; pass address of variable to the modify_a()
bl    modify_a
ldr    w1, [x29,28]        ; load value from the variable and pass it to printf()
adrp  x0, .LC0 ; '%d'
add    x0, x0, :lo12:.LC0
bl    printf              ; call printf()
ldp    x29, x30, [sp], 32
ret
.LC0:
.string "%d\n"
```

By the way, a similar usage of the *Shadow Space* is also considered here: [3.14.1 on page 520](#).

## 6.2 Thread Local Storage

TLS is a data area, specific to each thread. Every thread can store what it needs there. One well-known example is the C standard global variable *errno*.

Multiple threads may simultaneously call functions which return an error code in *errno*, so a global variable will not work correctly here for multi-threaded programs, so *errno* must be stored in the [TLS](#).

In the C++11 standard, a new *thread\_local* modifier was added, showing that each thread has its own version of the variable, it can be initialized, and it is located in the [TLS](#)<sup>3</sup>:

Listing 6.14: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
    std::cout << tmp << std::endl;
}
```

Compiled in MinGW GCC 4.8.1, but not in MSVC 2012.

If we talk about PE files, in the resulting executable file, the *tmp* variable is to be allocated in the section devoted to the [TLS](#).

<sup>3</sup> C11 also has thread support, optional though

## 6.2.1 Linear congruential generator revisited

The pseudorandom number generator we considered earlier [1.23 on page 339](#) has a flaw: it's not thread-safe, because it has an internal state variable which can be read and/or modified in different threads simultaneously.

### Win32

#### Uninitialized TLS data

One solution is to add `__declspec( thread )` modifier to the global variable, then it will be allocated in the [TLS](#) (line 9):

```

1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8
9 __declspec( thread ) uint32_t rand_state;
10
11 void my_srand (uint32_t init)
12 {
13     rand_state=init;
14 }
15
16 int my_rand ()
17 {
18     rand_state=rand_state*RNG_a;
19     rand_state=rand_state+RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     my_srand(0x12345678);
26     printf ("%d\n", my_rand());
27 }
```

Hiew shows us that there is a new PE section in the executable file: `.tls`.

Listing 6.15: Optimizing MSVC 2013 x86

```

_TLS    SEGMENT
_rand_state DD 01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG84851 DB      '%d', 0Ah, 00H
_DATA   ENDS
_TEXT   SEGMENT

_init$ = 8      ; size = 4
_my_srand PROC
; FS:0=address of TIB
    mov     eax, DWORD PTR fs:_tls_array  ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
    mov     ecx, DWORD PTR _tls_index
    mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    mov     eax, DWORD PTR _init$[esp-4]
    mov     DWORD PTR _rand_state[ecx], eax
    ret     0
_my_srand ENDP
```

## 6.2. THREAD LOCAL STORAGE

```
_my_rand PROC  
; FS:0=address of TIB  
    mov     eax, DWORD PTR fs:_tls_array ; displayed in IDA as FS:2Ch  
; EAX=address of TLS of process  
    mov     ecx, DWORD PTR __tls_index  
    mov     ecx, DWORD PTR [eax+ecx*4]  
; ECX=current TLS segment  
    imul    eax, DWORD PTR _rand_state[ecx], 1664525  
    add    eax, 1013904223           ; 3c6ef35fH  
    mov    DWORD PTR _rand_state[ecx], eax  
    and    eax, 32767            ; 00007ffffH  
    ret    0  
_my_rand ENDP  
  
_TEXT    ENDS
```

`rand_state` is now in the [TLS](#) segment, and each thread has its own version of this variable.

Here is how it's accessed: load the address of the [TIB](#) from FS:2Ch, then add an additional index (if needed), then calculate the address of the [TLS](#) segment.

Then it's possible to access the `rand_state` variable through the ECX register, which points to an unique area in each thread.

The `FS:` selector is familiar to every reverse engineer, it is specially used to always point to [TIB](#), so it would be fast to load the thread-specific data.

The `GS:` selector is used in Win64 and the address of the [TLS](#) is 0x58:

Listing 6.16: Optimizing MSVC 2013 x64

```
_TLS    SEGMENT  
rand_state DD  01H DUP (?)  
_TLS    ENDS  
  
_DATA   SEGMENT  
$SG85451 DB  '%d', 0ah, 00h  
_DATA   ENDS  
  
_TEXT   SEGMENT  
  
init$ = 8  
my_srand PROC  
    mov     edx, DWORD PTR __tls_index  
    mov     rax, QWORD PTR gs:88 ; 58h  
    mov     r8d, OFFSET FLAT:rand_state  
    mov     rax, QWORD PTR [rax+r8d*8]  
    mov     DWORD PTR [r8+rax], ecx  
    ret    0  
my_srand ENDP  
  
my_rand PROC  
    mov     rax, QWORD PTR gs:88 ; 58h  
    mov     ecx, DWORD PTR __tls_index  
    mov     edx, OFFSET FLAT:rand_state  
    mov     rcx, QWORD PTR [rax+rcx*8]  
    imul    eax, DWORD PTR [rcx+rdx], 1664525 ; 0019660dH  
    add    eax, 1013904223           ; 3c6ef35fH  
    mov    DWORD PTR [rcx+rdx], eax  
    and    eax, 32767            ; 00007ffffH  
    ret    0  
my_rand ENDP  
  
_TEXT   ENDS
```

### Initialized [TLS](#) data

## 6.2. THREAD LOCAL STORAGE

Let's say, we want to set some fixed value to `rand_state`, so in case the programmer forgets to, the `rand_state` variable would be initialized to some constant anyway (line 9):

```
1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8
9 __declspec( thread ) uint32_t rand_state=1234;
10
11 void my_srand (uint32_t init)
12 {
13     rand_state = init;
14 }
15
16 int my_rand ()
17 {
18     rand_state = rand_state * RNG_a;
19     rand_state = rand_state + RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     printf ("%d\n", my_rand());
26 }
```

The code is not different from what we already saw, but in IDA we see:

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls          segment para public 'DATA' use32
.tls:00404000             assume cs:_tls
.tls:00404000             ;org 404000h
.tls:00404000 TlsStart      db    0          ; DATA XREF: .rdata:TlsDirectory
.tls:00404001             db    0
.tls:00404002             db    0
.tls:00404003             db    0
.tls:00404004             dd   1234
.tls:00404008 TlsEnd        db    0          ; DATA XREF: .rdata:TlsEnd_ptr
...
...
```

1234 is there and every time a new thread starts, a new **TLS** is allocated for it, and all this data, including 1234, will be copied there.

This is a typical scenario:

- Thread A is started. A **TLS** is created for it, 1234 is copied to `rand_state`.
- The `my_rand()` function is called several times in thread A.  
`rand_state` is different from 1234.
- Thread B is started. A **TLS** is created for it, 1234 is copied to `rand_state`, while thread A has a different value in the same variable.

### TLS callbacks

But what if the variables in the **TLS** have to be filled with some data that must be prepared in some unusual way?

Let's say, we've got the following task: the programmer can forget to call the `my_srand()` function to initialize the **PRNG**, but the generator has to be initialized at start with something truly random, instead of 1234. This is a case in which **TLS** callbacks can be used.

## 6.2. THREAD LOCAL STORAGE

The following code is not very portable due to the hack, but nevertheless, you get the idea.

What we do here is define a function (`tls_callback()`) which is to be called *before* the process and/or thread start.

The function initializes the PRNG with the value returned by `GetTickCount()` function.

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    // rand_state is already initialized at the moment (using GetTickCount())
    printf ("%d\n", my_rand());
};
```

Let's see it in IDA:

Listing 6.17: Optimizing MSVC 2013

```
.text:00401020 TlsCallback_0    proc near          ; DATA XREF: .rdata:TlsCallbacks
.text:00401020                 call    ds:GetTickCount
.text:00401026                 push    eax
.text:00401027                 call    my_srand
.text:0040102C                 pop    ecx
.text:0040102D                 retn    0Ch
.text:0040102D TlsCallback_0    endp

...
.rdata:004020C0 TlsCallbacks    dd offset TlsCallback_0 ; DATA XREF: .rdata:TlsCallbacks_ptr
...
.rdata:00402118 TlsDirectory   dd offset TlsStart
.rdata:0040211C TlsEnd_ptr     dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr   dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

TLS callback functions are sometimes used in unpacking routines to obscure their processing.

### 6.3. SYSTEM CALLS (SYSCALL-S)

Some people may be confused and be in the dark that some code executed right before the [OEP](#)<sup>4</sup>.

#### Linux

Here is how a thread-local global variable is declared in GCC:

```
_thread uint32_t rand_state=1234;
```

This is not the standard C/C++ modifier, but a rather GCC-specific one <sup>5</sup>.

The `GS:` selector is also used to access the [TLS](#), but in a somewhat different way:

Listing 6.18: Optimizing GCC 4.8.1 x86

```
.text:08048460 my_srand    proc near
.text:08048460
.text:08048460 arg_0        = dword ptr  4
.text:08048460
.text:08048460             mov     eax, [esp+arg_0]
.text:08048464             mov     gs:0FFFFFFFCh, eax
.text:0804846A             retn
.text:0804846A my_srand    endp

.text:08048470 my_rand     proc near
.text:08048470             imul   eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B             add    eax, 3C6EF35Fh
.text:08048480             mov    gs:0FFFFFFFCh, eax
.text:08048486             and    eax, 7FFFh
.text:0804848B             retn
.text:0804848B my_rand    endp
```

More about it: [Ulrich Drepper, *ELF Handling For Thread-Local Storage*, (2013)]<sup>6</sup>.

## 6.3 System calls (syscall-s)

As we know, all running processes inside an [OS](#) are divided into two categories: those having full access to the hardware (“kernel space”) and those that do not (“user space”).

The [OS](#) kernel and usually the drivers are in the first category.

All applications are usually in the second category.

For example, Linux kernel is in *kernel space*, but Glibc in *user space*.

This separation is crucial for the safety of the [OS](#): it is very important not to give to any process the possibility to screw up something in other processes or even in the [OS](#) kernel. On the other hand, a failing driver or error inside the [OS](#)’s kernel usually leads to a kernel panic or [BSOD](#)<sup>7</sup>.

The protection in the x86 processors allows to separate everything into 4 levels of protection (rings), but both in Linux and in Windows only two are used: ring0 (“kernel space”) and ring3 (“user space”).

System calls (syscall-s) are a point where these two areas are connected.

It can be said that this is the main [API](#) provided to applications.

As in [Windows NT](#), the syscalls table resides in the [SSDT](#)<sup>8</sup>.

The usage of syscalls is very popular among shellcode and computer viruses authors, because it is hard to determine the addresses of needed functions in the system libraries, but it is easier to use syscalls. However, much more code has to be written due to the lower level of abstraction of the [API](#).

It is also worth noting that the syscall numbers may be different in various OS versions.

<sup>4</sup>Original Entry Point

<sup>5</sup><http://go.yurichev.com/17062>

<sup>6</sup>Also available as <http://go.yurichev.com/17272>

<sup>7</sup>Blue Screen of Death

<sup>8</sup>System Service Dispatch Table

## 6.3.1 Linux

In Linux, a syscall is usually called via `int 0x80`. The call's number is passed in the `EAX` register, and any other parameters —in the other registers.

Listing 6.19: A simple example of the usage of two syscalls

```
section .text
global _start

_start:
    mov     edx,len ; buffer len
    mov     ecx,msg ; buffer
    mov     ebx,1    ; file descriptor. 1 is for stdout
    mov     eax,4    ; syscall number. 4 is for sys_write
    int     0x80

    mov     eax,1    ; syscall number. 1 is for sys_exit
    int     0x80

section .data

msg    db  'Hello, world!',0xa
len    equ $ - msg
```

Compilation:

```
nasm -f elf32 1.s
ld 1.o
```

The full list of syscalls in Linux: <http://go.yurichev.com/17319>.

For system calls interception and tracing in Linux, strace( [7.2.3 on page 790](#)) can be used.

## 6.3.2 Windows

Here they are called via `int 0x2e` or using the special x86 instruction `SYSENTER`.

The full list of syscalls in Windows: <http://go.yurichev.com/17320>.

Further reading:

“Windows Syscall Shellcode” by Piotr Bania: <http://go.yurichev.com/17321>.

## 6.4 Linux

### 6.4.1 Position-independent code

While analyzing Linux shared (.so) libraries, one may frequently spot this code pattern:

Listing 6.20: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near          ; CODE XREF: sub_17350+3
.text:0012D5E3                           ; sub_173CC+4 ...
.text:0012D5E3             mov     ebx, [esp+0]
.text:0012D5E6             retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...
.text:000576C0 sub_576C0      proc near          ; CODE XREF: tmpfile+73
...
.text:000576C0             push    ebp
.text:000576C1             mov     ecx, large gs:0
.text:000576C8             push    edi
```

## 6.4. LINUX

```
.text:000576C9          push    esi
.text:000576CA          push    ebx
.text:000576CB          call    __x86_get_pc_thunk_bx
.text:000576D0          add     ebx, 157930h
.text:000576D6          sub     esp, 9Ch

...
.text:000579F0          lea     eax, (a_gen_tempname - 1AF000h)[ebx] ; "__gen_tempname"
.text:000579F6          mov     [esp+0ACh+var_A0], eax
.text:000579FA          lea     eax, (a_SysdepsPosix - 1AF000h)[ebx] ; "../sysdeps/
    ↳ posix/tempname.c"
.text:00057A00          mov     [esp+0ACh+var_A8], eax
.text:00057A04          lea     eax, (aInvalidKindIn_ - 1AF000h)[ebx] ; "! \"invalid ↳
    ↳ KIND in __gen_tempname\""
.text:00057A0A          mov     [esp+0ACh+var_A4], 14Ah
.text:00057A12          mov     [esp+0ACh+var_AC], eax
.text:00057A15          call   __assert_fail
```

All pointers to strings are corrected by some constants and the value in `EBX`, which is calculated at the beginning of each function.

This is the so-called **PIC**, it is intended to be executable if placed at any random point of memory, that is why it cannot contain any absolute memory addresses.

**PIC** was crucial in early computer systems and is still crucial today in embedded systems without virtual memory support (where all processes are placed in a single continuous memory block).

It is also still used in \*NIX systems for shared libraries, since they are shared across many processes while loaded in memory only once. But all these processes can map the same shared library at different addresses, so that is why a shared library has to work correctly without using any absolute addresses.

Let's do a simple experiment:

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
}
```

Let's compile it in GCC 4.7.3 and see the resulting .so file in [IDA](#):

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 6.21: GCC 4.7.3

```
.text:00000440          public  __x86_get_pc_thunk_bx
.text:00000440  __x86_get_pc_thunk_bx proc near           ; CODE XREF: _init_proc+4
.text:00000440                                         ; deregister_tm_clones+4 ...
.text:00000440          mov     ebx, [esp+0]
.text:00000443          retn
.text:00000443  __x86_get_pc_thunk_bx endp

.text:00000570          public  f1
.text:00000570  f1      proc near
.text:00000570
.text:00000570  var_1C      = dword ptr -1Ch
.text:00000570  var_18      = dword ptr -18h
.text:00000570  var_14      = dword ptr -14h
.text:00000570  var_8       = dword ptr -8
.text:00000570  var_4       = dword ptr -4
.text:00000570  arg_0       = dword ptr 4
.text:00000570
.text:00000570          sub     esp, 1Ch
.text:00000573          mov     [esp+1Ch+var_8], ebx
.text:00000573          call   __x86_get_pc_thunk_bx
```

## 6.4. LINUX

```
.text:0000057C          add    ebx, 1A84h
.text:00000582          mov    [esp+1Ch+var_4], esi
.text:00000586          mov    eax, ds:(global_variable_ptr - 2000h)[ebx]
.text:0000058C          mov    esi, [eax]
.text:0000058E          lea    eax, (aReturningD - 2000h)[ebx] ; "returning %d\n"
.text:00000594          add    esi, [esp+1Ch+arg_0]
.text:00000598          mov    [esp+1Ch+var_18], eax
.text:0000059C          mov    [esp+1Ch+var_1C], 1
.text:000005A3          mov    [esp+1Ch+var_14], esi
.text:000005A7          call   __printf_chk
.text:000005AC          mov    eax, esi
.text:000005AE          mov    ebx, [esp+1Ch+var_8]
.text:000005B2          mov    esi, [esp+1Ch+var_4]
.text:000005B6          add    esp, 1Ch
.text:000005B9          retn
.text:000005B9 f1       endp
```

That's it: the pointers to «returning %d\n» and *global\_variable* are to be corrected at each function execution.

The `__x86_get_pc_thunk_bx()` function returns in `EBX` the address of the point after a call to itself (`0x57C` here).

That's a simple way to get the value of the program counter (`EIP`) at some point. The `0x1A84` constant is related to the difference between this function's start and the so-called *Global Offset Table Procedure Linkage Table* (GOT PLT), the section right after the *Global Offset Table* (GOT), where the pointer to *global\_variable* is. [IDA](#) shows these offsets in their processed form to make them easier to understand, but in fact the code is:

```
.text:00000577          call   __x86_get_pc_thunk_bx
.text:0000057C          add    ebx, 1A84h
.text:00000582          mov    [esp+1Ch+var_4], esi
.text:00000586          mov    eax, [ebx-0Ch]
.text:0000058C          mov    esi, [eax]
.text:0000058E          lea    eax, [ebx-1A30h]
```

Here `EBX` points to the `GOT PLT` section and to calculate a pointer to *global\_variable* (which is stored in the `GOT`), `0xC` must be subtracted.

To calculate pointer to the «returning %d\n» string, `0x1A30` must be subtracted.

By the way, that is the reason why the AMD64 instruction set supports RIP<sup>9</sup>-relative addressing — to simplify PIC-code.

Let's compile the same C code using the same GCC version, but for x64.

[IDA](#) would simplify the resulting code but would suppress the RIP-relative addressing details, so we are going to use `objdump` instead of IDA to see everything:

```
00000000000000720 <f1>:
720: 48 8b 05 b9 08 20 00  mov    rax,QWORD PTR [rip+0x2008b9]      # 200fe0 <_DYNAMIC+0>
     ↓ x1d0>
727: 53                      push   rbx
728: 89 fb                  mov    ebx,edi
72a: 48 8d 35 20 00 00 00  lea    rsi,[rip+0x20]      # 751 <_fini+0x9>
731: bf 01 00 00 00          mov    edi,0x1
736: 03 18                  add    ebx,DWORD PTR [rax]
738: 31 c0                  xor    eax,eax
73a: 89 da                  mov    edx,ebx
73c: e8 df fe ff ff          call   620 <__printf_chk@plt>
741: 89 d8                  mov    eax,ebx
743: 5b                      pop    rbx
744: c3                      ret
```

`0x2008b9` is the difference between the address of the instruction at `0x720` and *global\_variable*, and `0x20` is the difference between the address of the instruction at `0x72A` and the «returning %d\n» string.

<sup>9</sup>program counter in AMD64

## 6.4. LINUX

As you might see, the need to recalculate addresses frequently makes execution slower (it is better in x64, though).

So it is probably better to link statically if you care about performance [see: Agner Fog, *Optimizing software in C++* (2015)].

## Windows

The PIC mechanism is not used in Windows DLLs. If the Windows loader needs to load DLL on another base address, it “patches” the DLL in memory (at the *FIXUP* places) in order to correct all addresses.

This implies that several Windows processes cannot share an once loaded DLL at different addresses in different process’ memory blocks — since each instance that’s loaded in memory is *fixed* to work only at these addresses..

### 6.4.2 *LD\_PRELOAD* hack in Linux

This allows us to load our own dynamic libraries before others, even before system ones, like `libc.so.6`.

This, in turn, allows us to “substitute” our written functions before the original ones in the system libraries. For example, it is easy to intercept all calls to `time()`, `read()`, `write()`, etc.

Let’s see if we can fool the *uptime* utility. As we know, it tells how long the computer has been working. With the help of `strace` ([7.2.3 on page 790](#)), it is possible to see that the utility takes this information the `/proc/uptime` file:

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)      = 3
lseek(3, 0, SEEK_SET)              = 0
read(3, "416166.86 414629.38\n", 2047) = 20
...
```

It is not a real file on disk, it is a virtual one and its contents are generated on fly in the Linux kernel. There are just two numbers:

```
$ cat /proc/uptime
416690.91 415152.03
```

What we can learn from Wikipedia <sup>[10](#)</sup>:

The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.

Let’s try to write our own dynamic library with the `open()`, `read()`, `close()` functions working as we need.

At first, our `open()` will compare the name of the file to be opened with what we need and if it is so, it will write down the descriptor of the file opened.

Second, `read()`, if called for this file descriptor, will substitute the output, and in the rest of the cases will call the original `read()` from `libc.so.6`. And also `close()`, will note if the file we are currently following is to be closed.

We are going to use the `dlopen()` and `dlsym()` functions to determine the original function addresses in `libc.so.6`.

We need them because we must pass control to the “real” functions.

On the other hand, if we intercepted `strcmp()` and monitored each string comparisons in the program, then we would have to implement our version of `strcmp()`, and not use the original function <sup>[11](#)</sup>, that would be easier.

<sup>10</sup>[wikipedia](#)

<sup>11</sup>For example, here is how simple `strcmp()` interception works in this article <sup>[12](#)</sup> written by Yong Huang

## 6.4. LINUX

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool initied = false;

_Noreturn void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);

    vprintf (fmt, va);
    exit(0);
};

static void find_original_functions ()
{
    if (initied)
        return;

    libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
    if (libc_handle==NULL)
        die ("can't open libc.so.6\n");

    open_ptr = dlsym (libc_handle, "open");
    if (open_ptr==NULL)
        die ("can't find open()\n");

    close_ptr = dlsym (libc_handle, "close");
    if (close_ptr==NULL)
        die ("can't find close()\n");

    read_ptr = dlsym (libc_handle, "read");
    if (read_ptr==NULL)
        die ("can't find read()\n");

    initied = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
    find_original_functions();

    int fd=(*open_ptr)(pathname, flags);
    if (strcmp(pathname, "/proc/uptime")==0)
        opened_fd=fd; // that's our file! record its file descriptor
    else
        opened_fd=0;
    return fd;
};

int close(int fd)
{
    find_original_functions();

    if (fd==opened_fd)
        opened_fd=0; // the file is not opened anymore
    return (*close_ptr)(fd);
```

## 6.5. WINDOWS NT

```
};

ssize_t read(int fd, void *buf, size_t count)
{
    find_original_functions();

    if (opened_fd!=0 && fd==opened_fd)
    {
        // that's our file!
        return sprintf (buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
    };
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};
```

( [Source code at GitHub](#) )

Let's compile it as common dynamic library:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Let's run *uptime* while loading our library before the others:

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

And we see:

```
01:23:02 up 24855 days, 3:14, 3 users, load average: 0.00, 0.01, 0.05
```

If the *LD\_PRELOAD*

environment variable always points to the filename and path of our library, it is to be loaded for all starting programs.

More examples:

- Very simple interception of the strcmp() (Yong Huang) <http://go.yurichev.com/17143>
- Kevin Pulo—Fun with LD\_PRELOAD. A lot of examples and ideas. [yurichev.com](http://yurichev.com)
- File functions interception for compression/decompression files on fly (zlibc). <http://go.yurichev.com/17146>

## 6.5 Windows NT

### 6.5.1 CRT (win32)

Does the program execution start right at the `main()` function? No, it does not.

If we would open any executable file in IDA or HIEW, we can see OEP pointing to some another code block.

This code is doing some maintenance and preparations before passing control flow to our code. It is called startup-code or CRT code (C RunTime).

The `main()` function takes an array of the arguments passed on the command line, and also one with environment variables. But in fact a generic string is passed to the program, the CRT code finds the spaces in it and cuts it in parts. The CRT code also prepares the environment variables array `envp`.

As for GUI<sup>13</sup> win32 applications, `WinMain` is used instead of `main()`, having its own arguments:

```
int CALLBACK WinMain(
    _In_ HINSTANCE hInstance,
    _In_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nCmdShow
);
```

<sup>13</sup>Graphical user interface

## 6.5. WINDOWS NT

The CRT code prepares them as well.

Also, the number returned by the `main()` function is the exit code.

It may be passed in CRT to the `ExitProcess()` function, which takes the exit code as an argument.

Usually, each compiler has its own CRT code.

Here is a typical CRT code for MSVC 2008.

```
1  __tmainCRTStartup proc near
2
3  var_24 = dword ptr -24h
4  var_20 = dword ptr -20h
5  var_1C = dword ptr -1Ch
6  ms_exc = CPPEH_RECORD ptr -18h
7
8      push    14h
9      push    offset stru_4092D0
10     call    __SEH_prolog4
11     mov     eax, 5A4Dh
12     cmp     ds:400000h, ax
13     jnz    short loc_401096
14     mov     eax, ds:40003Ch
15     cmp     dword ptr [eax+400000h], 4550h
16     jnz    short loc_401096
17     mov     ecx, 10Bh
18     cmp     [eax+400018h], cx
19     jnz    short loc_401096
20     cmp     dword ptr [eax+400074h], 0Eh
21     jbe    short loc_401096
22     xor     ecx, ecx
23     cmp     [eax+4000E8h], ecx
24     setnz  cl
25     mov     [ebp+var_1C], ecx
26     jmp    short loc_40109A
27
28
29 loc_401096: ; CODE XREF: __tmainCRTStartup+18
30             ; __tmainCRTStartup+29 ...
31     and    [ebp+var_1C], 0
32
33 loc_40109A: ; CODE XREF: __tmainCRTStartup+50
34     push   1
35     call    _heap_init
36     pop    ecx
37     test   eax, eax
38     jnz    short loc_4010AE
39     push   1Ch
40     call    _fast_error_exit
41     pop    ecx
42
43 loc_4010AE: ; CODE XREF: __tmainCRTStartup+60
44     call    __mtinit
45     test   eax, eax
46     jnz    short loc_4010BF
47     push   10h
48     call    _fast_error_exit
49     pop    ecx
50
51 loc_4010BF: ; CODE XREF: __tmainCRTStartup+71
52     call    sub_401F2B
53     and    [ebp+ms_exc.disabled], 0
54     call    __ioinit
55     test   eax, eax
56     jge    short loc_4010D9
57     push   1Bh
58     call    __amsg_exit
59     pop    ecx
60
```

## 6.5. WINDOWS NT

```
61 loc_4010D9: ; CODE XREF: __tmainCRTStartup+8B
62     call    ds:GetCommandLineA
63     mov     dword_40B7F8, eax
64     call    __crtGetEnvironmentStringsA
65     mov     dword_40AC60, eax
66     call    __setargv
67     test   eax, eax
68     jge    short loc_4010FF
69     push   8
70     call    __amsg_exit
71     pop    ecx
72
73 loc_4010FF: ; CODE XREF: __tmainCRTStartup+B1
74     call    __setenvp
75     test   eax, eax
76     jge    short loc_401110
77     push   9
78     call    __amsg_exit
79     pop    ecx
80
81 loc_401110: ; CODE XREF: __tmainCRTStartup+C2
82     push   1
83     call    __cinit
84     pop    ecx
85     test   eax, eax
86     jz    short loc_401123
87     push   eax
88     call    __amsg_exit
89     pop    ecx
90
91 loc_401123: ; CODE XREF: __tmainCRTStartup+D6
92     mov    eax, envp
93     mov    dword_40AC80, eax
94     push  eax          ; envp
95     push  argv          ; argv
96     push  argc          ; argc
97     call  _main
98     add   esp, 0Ch
99     mov    [ebp+var_20], eax
100    cmp   [ebp+var_1C], 0
101    jnz  short $LN28
102    push  eax          ; uExitCode
103    call  $LN32
104
105 $LN28:    ; CODE XREF: __tmainCRTStartup+105
106    call  __cexit
107    jmp   short loc_401186
108
109
110 $LN27:    ; DATA XREF: .rdata:stru_4092D0
111    mov    eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
112    mov    ecx, [eax]
113    mov    ecx, [ecx]
114    mov    [ebp+var_24], ecx
115    push  eax
116    push  ecx
117    call  __XcptFilter
118    pop   ecx
119    pop   ecx
120
121 $LN24:
122     retn
123
124
125 $LN14:    ; DATA XREF: .rdata:stru_4092D0
126    mov    esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function 401044
127    mov    eax, [ebp+var_24]
128    mov    [ebp+var_20], eax
129    cmp   [ebp+var_1C], 0
130    jnz  short $LN29
```

## 6.5. WINDOWS NT

```
131     push    eax          ; int
132     call    __exit
133
134
135 $LN29:   ; CODE XREF: __tmainCRTStartup+135
136     call    __c_exit
137
138 loc_401186: ; CODE XREF: __tmainCRTStartup+112
139     mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
140     mov     eax, [ebp+var_20]
141     call    __SEH_epilog4
142     retn
```

Here we can see calls to `GetCommandLineA()` (line 62), then to `setargv()` (line 66) and `setenvp()` (line 74), which apparently fill the global variables `argc`, `argv`, `envp`.

Finally, `main()` is called with these arguments (line 97).

There are also calls to functions with self-describing names like `heap_init()` (line 35), `ioinit()` (line 54).

The `heap` is indeed initialized in the [CRT](#). If you try to use `malloc()` in a program without CRT, it will exit abnormally with the following error:

```
runtime error R6030
- CRT not initialized
```

Global object initializations in C++ is also occur in the [CRT](#) before the execution of `main()`: [3.19.4 on page 568](#).

The value that `main()` returns is passed to `cexit()`, or in `$LN32`, which in turn calls `doexit()`.

Is it possible to get rid of the [CRT](#)? Yes, if you know what you are doing.

The [MSVC](#)'s linker has the `/ENTRY` option for setting an entry point.

```
#include <windows.h>

int main()
{
    MessageBox (NULL, "hello, world", "caption", MB_OK);
}
```

Let's compile it in MSVC 2008.

```
cl no_crt.c user32.lib /link /entry:main
```

We are getting a runnable .exe with size 2560 bytes, that has a PE header in it, instructions calling `MessageBox`, two strings in the data segment, the `MessageBox` function imported from `user32.dll` and nothing else.

This works, but you cannot write `WinMain` with its 4 arguments instead of `main()`.

To be precise, you can, but the arguments are not prepared at the moment of execution.

By the way, it is possible to make the .exe even shorter by aligning the PE sections at less than the default 4096 bytes.

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Linker says:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

We get an .exe that's 720 bytes. It can be executed in Windows 7 x86, but not in x64 (an error message will be shown when you try to execute it).

With even more efforts, it is possible to make the executable even shorter, but as you can see, compatibility problems arise quickly.

## 6.5.2 Win32 PE

PE is an executable file format used in Windows. The difference between .exe, .dll and .sys is that .exe and .sys usually do not have exports, only imports.

A [DLL<sup>14</sup>](#), just like any other PE-file, has an entry point ([OEP](#)) (the function `DllMain()` is located there) but this function usually does nothing. .sys is usually a device driver. As of drivers, Windows requires the checksum to be present in the PE file and for it to be correct [<sup>15</sup>](#).

Starting at Windows Vista, a driver's files must also be signed with a digital signature. It will fail to load otherwise.

Every PE file begins with tiny DOS program that prints a message like "This program cannot be run in DOS mode."—if you run this program in DOS or Windows 3.1 ([OS](#)-es which are not aware of the PE format), this message will be printed.

### Terminology

- Module—a separate file, .exe or .dll.
- Process—a program loaded into memory and currently running. Commonly consists of one .exe file and bunch of .dll files.
- Process memory—the memory a process works with. Each process has its own. There usually are loaded modules, memory of the stack, [heap\(s\)](#), etc.
- [VA<sup>16</sup>](#)—an address which is to be used in program while runtime.
- Base address (of module)—the address within the process memory at which the module is to be loaded. [OS](#) loader may change it, if the base address is already occupied by another module just loaded before.
- [RVA<sup>17</sup>](#)—the [VA](#)-address minus the base address.  
Many addresses in PE-file tables use [RVA](#)-addresses.
- [IAT<sup>18</sup>](#)—an array of addresses of imported symbols [<sup>19</sup>](#). Sometimes, the `IMAGE_DIRECTORY_ENTRY_IAT` data directory points at the [IAT](#). It is worth noting that [IDA](#) (as of 6.1) may allocate a pseudo-section named `.idata` for [IAT](#), even if the [IAT](#) is a part of another section!
- [INT<sup>20</sup>](#)—an array of names of symbols to be imported<sup>21</sup>.

### Base address

The problem is that several module authors can prepare DLL files for others to use and it is not possible to reach an agreement which addresses is to be assigned to whose modules.

So that is why if two necessary DLLs for a process have the same base address, one of them will be loaded at this base address, and the other—at some other free space in process memory, and each virtual addresses in the second DLL will be corrected.

With [MSVC](#) the linker often generates the .exe files with a base address of `0x400000` [<sup>22</sup>](#), and with the code section starting at `0x401000`. This means that the [RVA](#) of the start of the code section is `0x1000`.

DLLs are often generated by MSVC's linker with a base address of `0x10000000` [<sup>23</sup>](#).

There is also another reason to load modules at various base addresses, in this case random ones. It is [ASLR<sup>24</sup>](#).

<sup>14</sup>Dynamic-link library

<sup>15</sup>For example, Hiew ([7.1 on page 788](#)) can calculate it

<sup>16</sup>Virtual Address

<sup>17</sup>Relative Virtual Address

<sup>18</sup>Import Address Table

<sup>19</sup>Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

<sup>20</sup>Import Name Table

<sup>21</sup>Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

<sup>22</sup>The origin of this address choice is described here: [MSDN](#)

<sup>23</sup>This can be changed by the /BASE linker option

<sup>24</sup>wikipedia

## 6.5. WINDOWS NT

A shellcode trying to get executed on a compromised system must call system functions, hence, know their addresses.

In older OS (in Windows NT line: before Windows Vista), system DLL (like kernel32.dll, user32.dll) were always loaded at known addresses, and if we also recall that their versions rarely changed, the addresses of functions were fixed and shellcode could call them directly.

In order to avoid this, the ASLR method loads your program and all modules it needs at random base addresses, different every time.

ASLR support is denoted in a PE file by setting the flag

`IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` [see Mark Russinovich, *Microsoft Windows Internals*].

## Subsystem

There is also a *Subsystem* field, usually it is:

- native<sup>25</sup> (.sys-driver),
- console (console application) or
- GUI (non-console).

## OS version

A PE file also specifies the minimal Windows version it needs in order to be loadable.

The table of version numbers stored in the PE file and corresponding Windows codenames is here<sup>26</sup>.

For example, MSVC 2005 compiles .exe files for running on Windows NT4 (version 4.00), but MSVC 2008 does not (the generated files have a version of 5.00, at least Windows 2000 is needed to run them).

MSVC 2012 generates .exe files of version 6.00 by default, targeting at least Windows Vista. However, by changing the compiler's options<sup>27</sup>, it is possible to force it to compile for Windows XP.

## Sections

Division in sections, as it seems, is present in all executable file formats.

It is devised in order to separate code from data, and data—from constant data.

- Either the `IMAGE_SCN_CNT_CODE` or `IMAGE_SCN_MEM_EXECUTE` flags will be set on the code section—this is executable code.
- On data section—`IMAGE_SCN_CNT_INITIALIZED_DATA`, `IMAGE_SCN_MEM_READ` and `IMAGE_SCN_MEM_WRITE` flags.
- On an empty section with uninitialized data—`IMAGE_SCN_CNT_UNINITIALIZED_DATA`, `IMAGE_SCN_MEM_READ` and `IMAGE_SCN_MEM_WRITE`.
- On a constant data section (one that's protected from writing), the flags `IMAGE_SCN_CNT_INITIALIZED_DATA` and `IMAGE_SCN_MEM_READ` can be set, but not `IMAGE_SCN_MEM_WRITE`. A process going to crash if it tries to write to this section.

Each section in PE-file may have a name, however, it is not very important. Often (but not always) the code section is named `.text`, the data section—`.data`, the constant data section — `.rdata` (readable data). Other popular section names are:

- `.idata`—imports section. IDA may create a pseudo-section named like this: [6.5.2 on the previous page](#).
- `.edata`—exports section (rare)

<sup>25</sup>Meaning, the module use Native API instead of Win32

<sup>26</sup>[wikipedia](#)

<sup>27</sup>[MSDN](#)

## 6.5. WINDOWS NT

- `.pdata` — section holding all information about exceptions in Windows NT for MIPS, IA64 and x64: [6.5.3 on page 782](#)
- `.reloc` — relocs section
- `.bss` — uninitialized data ([BSS](#))
- `.tls` — thread local storage ([TLS](#))
- `.rsrc` — resources
- `.CRT` — may present in binary files compiled by ancient MSVC versions

PE file packers/encryptors often garble section names or replace the names with their own.

MSVC allows you to declare data in arbitrarily named section <sup>28</sup>.

Some compilers and linkers can add a section with debugging symbols and other debugging information (MinGW for instance). However it is not so in latest versions of MSVC (separate PDB files are used there for this purpose).

That is how a PE section is described in the file:

```
typedef struct _IMAGE_SECTION_HEADER {  
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];  
    union {  
        DWORD PhysicalAddress;  
        DWORD VirtualSize;  
    } Misc;  
    DWORD VirtualAddress;  
    DWORD SizeOfRawData;  
    DWORD PointerToRawData;  
    DWORD PointerToRelocations;  
    DWORD PointerToLinenumbers;  
    WORD NumberOfRelocations;  
    WORD NumberOfLinenumbers;  
    DWORD Characteristics;  
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

<sup>29</sup>

A word about terminology: *PointerToRawData* is called “Offset” in Hiew and *VirtualAddress* is called “RVA” there.

## Data section

Data section in file can be smaller than in memory. For example, some variables can be initialized, some are not. Compiler and linker will collect them all into one section, but the first part of it is initialized and allocated in file, while another is absent in file (of course, to make it smaller). *VirtualSize* will be equal to the size of section in memory, and *SizeOfRawData* — to size of section in file.

IDA can show the border between initialized and not initialized parts like that:

```
...  
.data:10017FFA db 0  
.data:10017FFB db 0  
.data:10017FFC db 0  
.data:10017FFD db 0  
.data:10017FFE db 0  
.data:10017FFF db 0  
.data:10018000 db ? ;  
.data:10018001 db ? ;  
.data:10018002 db ? ;  
.data:10018003 db ? ;  
.data:10018004 db ? ;  
.data:10018005 db ? ;
```

<sup>28</sup>[MSDN](#)

<sup>29</sup>[MSDN](#)

## Relocations (relocs)

AKA FIXUP-s (at least in Hiew).

They are also present in almost all executable file formats <sup>30</sup>. Exceptions are shared dynamic libraries compiled with PIC, or any other PIC-code.

What are they for?

Obviously, modules can be loaded on various base addresses, but how to deal with global variables, for example? They must be accessed by address. One solution is position-independent code ([6.4.1 on page 747](#)). But it is not always convenient.

That is why a relocations table is present. There the addresses of points that must be corrected are enumerated, in case of loading at a different base address.

For example, there is a global variable at address `0x410000` and this is how it is accessed:

<code>A1 00 00 41 00</code>	<code>mov</code>	<code>eax, [00041000]</code>
-----------------------------	------------------	------------------------------

The base address of the module is `0x400000`, the RVA of the global variable is `0x10000`.

If the module is loaded at base address `0x500000`, the real address of the global variable must be `0x510000`.

As we can see, the address of variable is encoded in the instruction `MOV`, after the byte `0xA1`.

That is why the address of the 4 bytes after `0xA1`, is written in the relocs table.

If the module is loaded at a different base address, the OS loader enumerates all addresses in the table, finds each 32-bit word the address points to, subtracts the original base address from it (we get the RVA here), and adds the new base address to it.

If a module is loaded at its original base address, nothing happens.

All global variables can be treated like that.

Relocs may have various types, however, in Windows for x86 processors, the type is usually `IMAGE_REL_BASED_HIGHLOW`.

By the way, relocs are darkened in Hiew, for example: [fig.1.21](#).

OllyDbg underlines the places in memory to which relocs are to be applied, for example: [fig.1.52](#).

## Exports and imports

As we all know, any executable program must use the OS's services and other DLL-libraries somehow.

It can be said that functions from one module (usually DLL) must be connected somehow to the points of their calls in other modules (.exe-file or another DLL).

For this, each DLL has an "exports" table, which consists of functions plus their addresses in a module.

And every .exe file or DLL has "imports", a table of functions it needs for execution including list of DLL filenames.

After loading the main .exe-file, the OS loader processes imports table: it loads the additional DLL-files, finds function names among the DLL exports and writes their addresses down in the IAT of the main .exe-module.

As we can see, during loading the loader must compare a lot of function names, but string comparison is not a very fast procedure, so there is a support for "ordinals" or "hints", which are function numbers stored in the table, instead of their names.

That is how they can be located faster when loading a DLL. Ordinals are always present in the "export" table.

---

<sup>30</sup>Even in .exe files for MS-DOS

## 6.5. WINDOWS NT

For example, a program using the [MFC<sup>31</sup>](#) library usually loads mfc\*.dll by ordinals, and in such programs there are no MFC function names in INT.

When loading such programs in [IDA](#), it will ask for a path to the mfc\*.dll files in order to determine the function names.

If you don't tell [IDA](#) the path to these DLLs, there will be *mfc80\_123* instead of function names.

### Imports section

Often a separate section is allocated for the imports table and everything related to it (with name like `.idata`), however, this is not a strict rule.

Imports are also a confusing subject because of the terminological mess. Let's try to collect all information in one place.

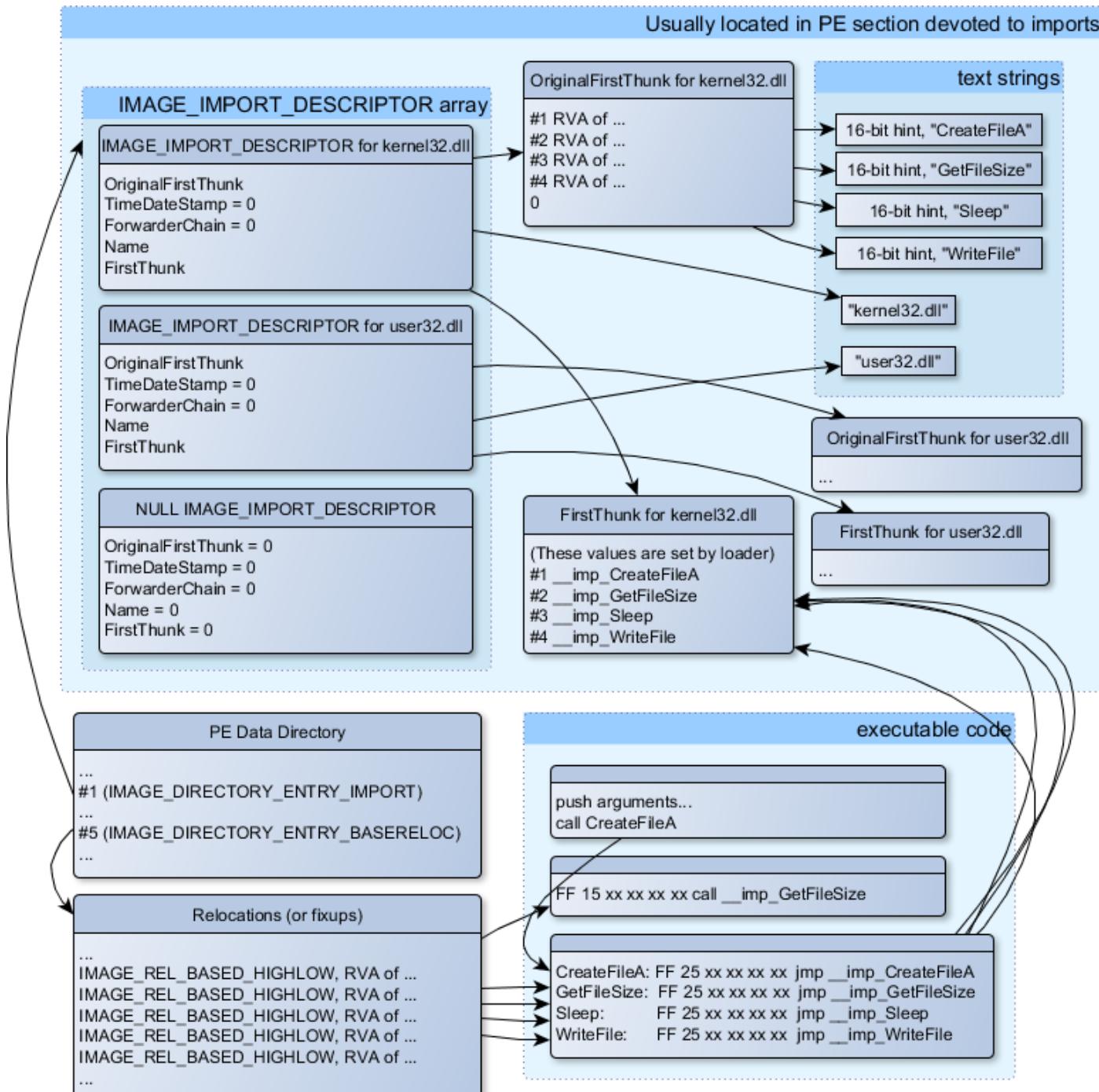


Figure 6.1: A scheme that unites all PE-file structures related to imports

<sup>31</sup>Microsoft Foundation Classes

## 6.5. WINDOWS NT

The main structure is the array *IMAGE\_IMPORT\_DESCRIPTOR*. Each element for each DLL being imported.

Each element holds the [RVA](#) address of the text string (DLL name) (*Name*).

*OriginalFirstThunk* is the [RVA](#) address of the *INT* table. This is an array of [RVA](#) addresses, each of which points to a text string with a function name. Each string is prefixed by a 16-bit integer (“hint”)—“ordinal” of function.

While loading, if it is possible to find a function by ordinal, then the strings comparison will not occur. The array is terminated by zero.

There is also a pointer to the *IAT* table named *FirstThunk*, it is just the [RVA](#) address of the place where the loader writes the addresses of the resolved functions.

The points where the loader writes addresses are marked by [IDA](#) like this: *\_imp\_CreateFileA*, etc.

There are at least two ways to use the addresses written by the loader.

- The code will have instructions like *call \_imp\_CreateFileA*, and since the field with the address of the imported function is a global variable in some sense, the address of the *call* instruction (plus 1 or 2) is to be added to the relocs table, for the case when the module is loaded at a different base address.

But, obviously, this may enlarge relocs table significantly.

Because there are might be a lot of calls to imported functions in the module.

Furthermore, large relocs table slows down the process of loading modules.

- For each imported function, there is only one jump allocated, using the [JMP](#) instruction plus a reloc to it. Such points are also called “thunks”.

All calls to the imported functions are just [CALL](#) instructions to the corresponding “thunk”. In this case, additional relocs are not necessary because these [CALL](#)-s have relative addresses and do not need to be corrected.

These two methods can be combined.

Possible, the linker creates individual “thunk”s if there are too many calls to the function, but not done by default.

By the way, the array of function addresses to which *FirstThunk* is pointing is not necessary to be located in the *IAT* section. For example, the author of these lines once wrote the *PE\_add\_import*<sup>32</sup> utility for adding imports to an existing .exe-file.

Some time earlier, in the previous versions of the utility, at the place of the function you want to substitute with a call to another DLL, my utility wrote the following code:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

*FirstThunk* points to the first instruction. In other words, when loading *yourdll.dll*, the loader writes the address of the *function* function right in the code.

It also worth noting that a code section is usually write-protected, so my utility adds the *IMAGE\_SCN\_MEM\_WRITE* flag for code section. Otherwise, the program to crash while loading with error code 5 (access denied).

One might ask: what if I supply a program with a set of DLL files which is not supposed to change (including addresses of all DLL functions), is it possible to speed up the loading process?

Yes, it is possible to write the addresses of the functions to be imported into the *FirstThunk* arrays in advance. The *Timestamp* field is present in the *IMAGE\_IMPORT\_DESCRIPTOR* structure.

If a value is present there, then the loader compares this value with the date-time of the DLL file.

If the values are equal, then the loader does not do anything, and the loading of the process can be faster. This is called “old-style binding”<sup>33</sup>.

<sup>32</sup>[yurichev.com](http://yurichev.com)

<sup>33</sup>[MSDN](#). There is also the “new-style binding”.

## 6.5. WINDOWS NT

The BIND.EXE utility in Windows SDK is for for this. For speeding up the loading of your program, Matt Pietrek in Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]<sup>34</sup>, suggests to do the binding shortly after your program installation on the computer of the end user.

PE-files packers/encryptors may also compress/encrypt imports table.

In this case, the Windows loader, of course, will not load all necessary DLLs.

Therefore, the packer/encryptor does this on its own, with the help of *LoadLibrary()* and the *GetProcAddress()* functions.

That is why these two functions are often present in **IAT** in packed files.

In the standard DLLs from the Windows installation, **IAT** often is located right at the beginning of the PE file. Supposedly, it is made so for optimization.

While loading, the .exe file is not loaded into memory as a whole (recall huge install programs which are started suspiciously fast), it is “mapped”, and loaded into memory in parts as they are accessed.

Probably, Microsoft developers decided it will be faster.

## Resources

Resources in a PE file are just a set of icons, pictures, text strings, dialog descriptions.

Perhaps they were separated from the main code, so all these things could be multilingual, and it would be simpler to pick text or picture for the language that is currently set in the **OS**.

As a side effect, they can be edited easily and saved back to the executable file, even if one does not have special knowledge, by using the ResHack editor, for example ([6.5.2](#)).

## .NET

.NET programs are not compiled into machine code but into a special bytecode. Strictly speaking, there is bytecode instead of the usual x86 code in the .exe file, however, the entry point (**OEP**) points to this tiny fragment of x86 code:

```
jmp mscoree.dll!_CorExeMain
```

The .NET loader is located in mscoree.dll, which processes the PE file.

It was so in all pre-Windows XP **OSes**. Starting from XP, the **OS** loader is able to detect the .NET file and run it without executing that **JMP** instruction <sup>35</sup>.

## TLS

This section holds initialized data for the **TLS** ([6.2 on page 741](#)) (if needed). When a new thread start, its **TLS** data is initialized using the data from this section.

Aside from that, the PE file specification also provides initialization of the **TLS** section, the so-called TLS callbacks.

If they are present, they are to be called before the control is passed to the main entry point (**OEP**).

This is used widely in the PE file packers/encryptors.

<sup>34</sup>Also available as <http://go.yurichev.com/17318>

<sup>35</sup>MSDN

---

**Tools**

- objdump (present in cygwin) for dumping all PE-file structures.
- Hiew ([7.1 on page 788](#)) as editor.
- pefile—Python-library for PE-file processing <sup>[36](#)</sup>.
- ResHack AKA Resource Hacker—resources editor<sup>[37](#)</sup>.
- PE\_add\_import<sup>[38](#)</sup>— simple tool for adding symbol(s) to PE executable import table.
- PE\_patcher<sup>[39](#)</sup>—simple tool for patching PE executables.
- PE\_search\_str\_refs<sup>[40](#)</sup>—simple tool for searching for a function in PE executables which use some text string.

**Further reading**

- Daniel Pistelli—The .NET File Format <sup>[41](#)</sup>

**6.5.3 Windows SEH****Let's forget about MSVC**

In Windows, the [SEH](#) is intended for exceptions handling, nevertheless, it is language-agnostic, not related to C++ or [OOP](#) in any way.

Here we are going to take a look at [SEH](#) in its isolated (from C++ and MSVC extensions) form.

Each running process has a chain of [SEH](#) handlers, [TIB](#) has the address of the last handler.

When an exception occurs (division by zero, incorrect address access, user exception triggered by calling the `RaiseException()` function), the [OS](#) finds the last handler in the [TIB](#) and calls it, passing all information about the [CPU](#) state (register values, etc.) at the moment of the exception.

The exception handler considering the exception, does it see something familiar? If so, it handles the exception.

If not, it signals to the [OS](#) that it cannot handle it and the [OS](#) calls the next handler in the chain, until a handler which is able to handle the exception is found.

At the very end of the chain there a standard handler that shows the well-known dialog box, informing the user about a process crash, some technical information about the [CPU](#) state at the time of the crash, and offering to collect all information and send it to developers in Microsoft.

---

<sup>36</sup><http://go.yurichev.com/17052>

<sup>37</sup><http://go.yurichev.com/17052>

<sup>38</sup><http://go.yurichev.com/17049>

<sup>39</sup>[yurichev.com](http://yurichev.com)

<sup>40</sup>[yurichev.com](http://yurichev.com)

<sup>41</sup><http://go.yurichev.com/17056>

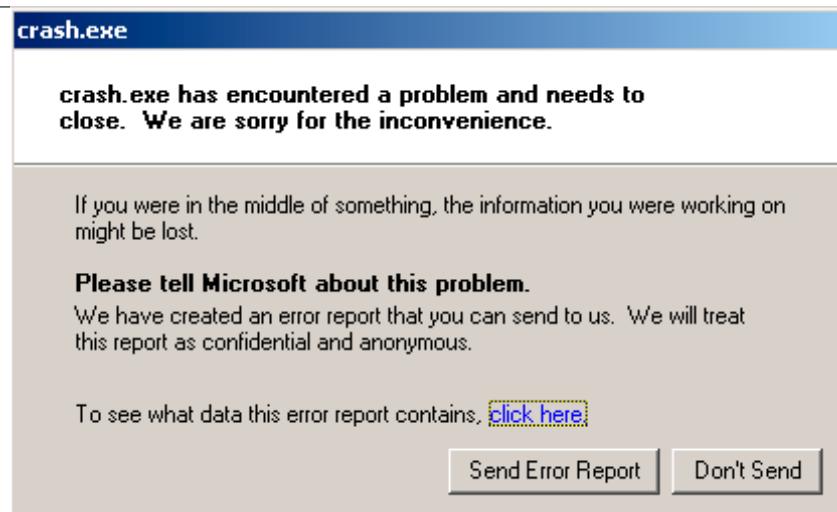


Figure 6.2: Windows XP

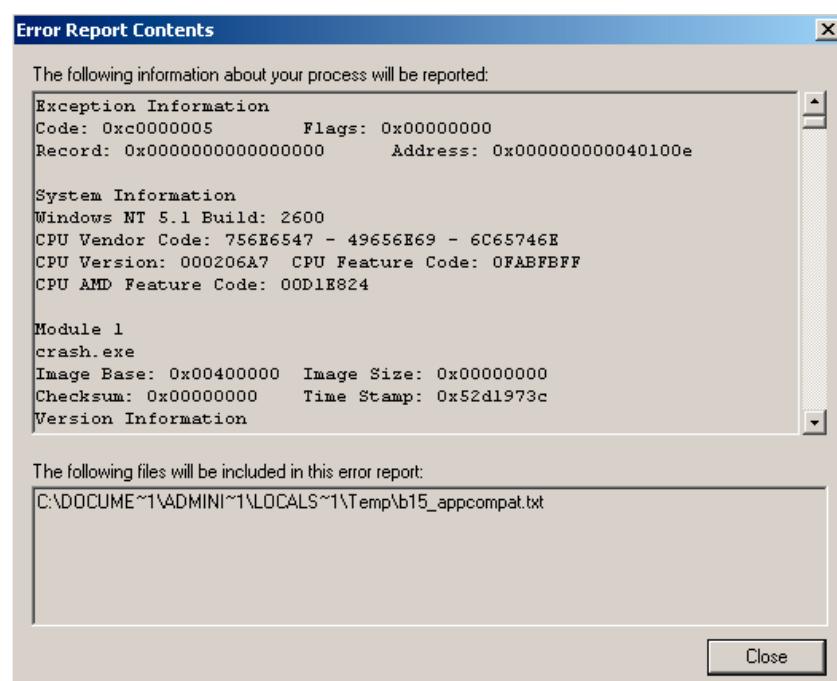


Figure 6.3: Windows XP

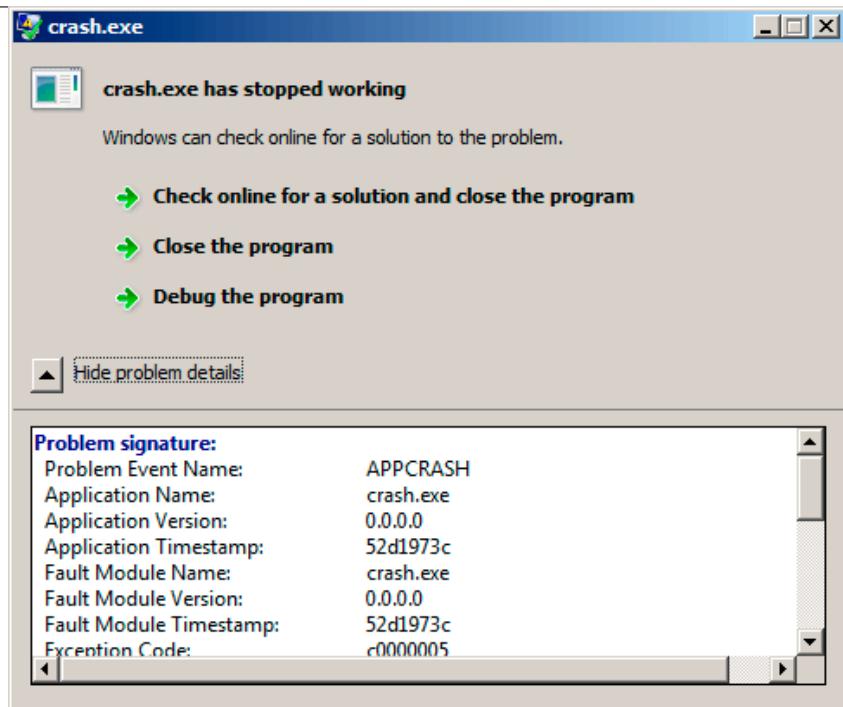


Figure 6.4: Windows 7

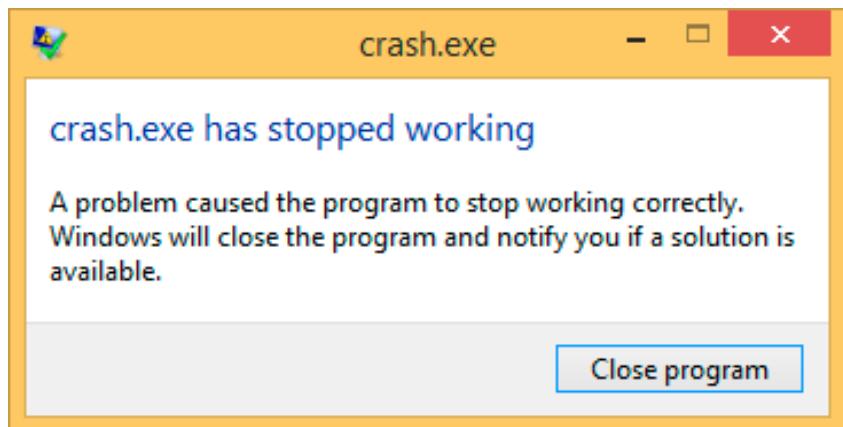


Figure 6.5: Windows 8.1

Earlier, this handler was called Dr. Watson <sup>42</sup>.

By the way, some developers make their own handler that sends information about the program crash to themselves. It is registered with the help of `SetUnhandledExceptionFilter()` and to be called if the OS does not have any other way to handle the exception. An example is Oracle RDBMS—it saves huge dumps reporting all possible information about the CPU and memory state.

Let's write our own primitive exception handler. This example is based on the example from [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]<sup>43</sup>. It must be compiled with the SAFESEH option: `cl seh1.cpp /link /safe seh: no`. More about SAFESEH here: [MSDN](#).

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
```

<sup>42</sup>[wikipedia](#)

<sup>43</sup>Also available as <http://go.yurichev.com/17293>

## 6.5. WINDOWS NT

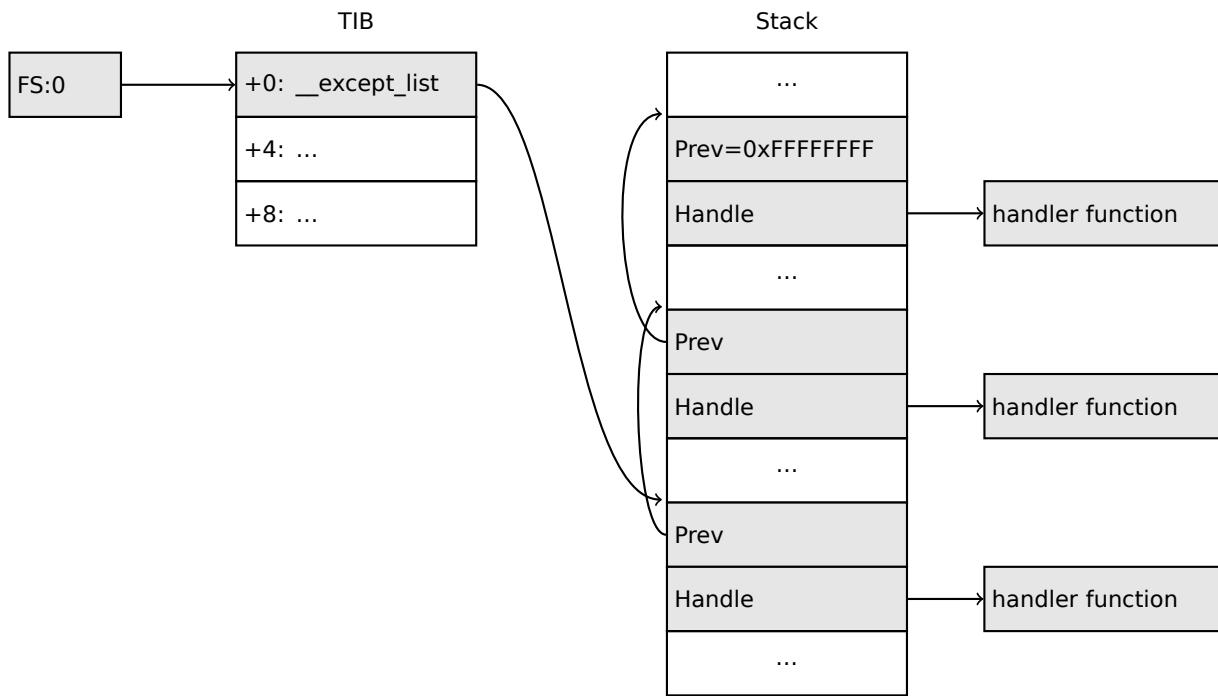
```
        void * DispatcherContext )  
{  
    unsigned i;  
  
    printf ("%s\n", __FUNCTION__);  
    printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);  
    printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);  
    printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->ExceptionAddress);  
  
    if (ExceptionRecord->ExceptionCode==0xE1223344)  
    {  
        printf ("That's for us\n");  
        // yes, we "handled" the exception  
        return ExceptionContinueExecution;  
    }  
    else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION)  
    {  
        printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);  
        // will it be possible to 'fix' it?  
        printf ("Trying to fix wrong pointer address\n");  
        ContextRecord->Eax=(DWORD)&new_value;  
        // yes, we "handled" the exception  
        return ExceptionContinueExecution;  
    }  
    else  
    {  
        printf ("We do not handle this\n");  
        // someone else's problem  
        return ExceptionContinueSearch;  
    };  
}  
  
int main()  
{  
    DWORD handler = (DWORD)except_handler; // take a pointer to our handler  
  
    // install exception handler  
    __asm  
    {  
        push    handler           // make EXCEPTION_REGISTRATION record:  
        push    FS:[0]            // address of handler function  
        mov     FS:[0],ESP         // address of previous handler  
    }  
  
    RaiseException (0xE1223344, 0, 0, NULL);  
  
    // now do something very bad  
    int* ptr=NULL;  
    int val=0;  
    val=*ptr;  
    printf ("val=%d\n", val);  
  
    // deinstall exception handler  
    __asm  
    {  
        mov    eax,[ESP]          // remove our EXCEPTION_REGISTRATION record  
        mov    FS:[0], EAX          // get pointer to previous record  
        add    esp, 8              // install previous record  
        add    esp, 8              // clean our EXCEPTION_REGISTRATION off stack  
    }  
  
    return 0;  
}
```

The FS: segment register is pointing to the [TIB](#) in win32.

The very first element in the [TIB](#) is a pointer to the last handler in the chain. We save it in the stack and store the address of our handler there. The structure is named `_EXCEPTION_REGISTRATION`, it is a simple singly-linked list and its elements are stored right in the stack.

```
\_EXCEPTION\REGISTRATION struc
    prev    dd      ?
    handler dd      ?
\_EXCEPTION\REGISTRATION ends
```

So each “handler” field points to a handler and an each “prev” field points to the previous record in the stack. The last record has `0xFFFFFFFF` (-1) in the “prev” field.



After our handler is installed, we call `RaiseException()`<sup>44</sup>. This is an user exception. The handler checks the code. If the code is `0xE1223344`, it returning `ExceptionContinueExecution`, which means that handler corrected the CPU state (it is usually a correction of the EIP/ESP registers) and the OS can resume the execution of the. If you alter slightly the code so the handler returns `ExceptionContinueSearch`,

then the OS will call the other handlers, and it's unlikely that one who can handle it will be found, since no one will have any information about it (rather about its code). You will see the standard Windows dialog about a process crash.

What is the difference between a system exceptions and a user one? Here are the system ones:

<sup>44</sup>MSDN

## 6.5. WINDOWS NT

as defined in WinBase.h	as defined in ntstatus.h	value
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

That is how the code is defined:

31	29	28	27	16	15	0
S	U	0	Facility code		Error code	

S is a basic status code: 11—error; 10—warning; 01—informational; 00—success. U—whether the code is user code.

That is why we chose 0xE1223344—E<sub>16</sub> (1110<sub>2</sub>) 0xE (1110b) means that it is 1) user exception; 2) error. But to be honest, this example works fine without these high bits.

Then we try to read a value from memory at address 0.

Of course, there is nothing at this address in win32, so an exception is raised.

The very first handler is to be called—yours, and it will know about it first, by checking the code if it's equal to the EXCEPTION\_ACCESS\_VIOLATION constant.

The code that's reading from memory at address 0 looks like this:

Listing 6.23: MSVC 2010

```

...
xor    eax, eax
mov    eax, DWORD PTR [eax] ; exception will occur here
push   eax
push   OFFSET msg
call   _printf
add    esp, 8
...

```

Will it be possible to fix this error “on the fly” and to continue with program execution?

Yes, our exception handler can fix the EAX value and let the OS execute this instruction once again. So that is what we do. printf() prints 1234, because after the execution of our handler EAX is not 0, but contains the address of the global variable new\_value. The execution will resume.

That is what is going on: the memory manager in the CPU signals about an error, the CPU suspends the thread, finds the exception handler in the Windows kernel, which, in turn, starts to call all handlers in the SEH chain, one by one.

We use MSVC 2010 here, but of course, there is no any guarantee that EAX will be used for this pointer.

This address replacement trick is showy, and we considering it here as an illustration of SEH's internals. Nevertheless, it's hard to recall any case where it is used for “on-the-fly” error fixing.

## 6.5. WINDOWS NT

Why SEH-related records are stored right in the stack instead of some other place?

Supposedly because the OS is not needing to care about freeing this information, these records are simply disposed when the function finishes its execution. This is somewhat like alloca(): ([1.7.2 on page 35](#)).

### Now let's get back to MSVC

Supposedly, Microsoft programmers needed exceptions in C, but not in C++, so they added a non-standard C extension to MSVC<sup>45</sup>. It is not related to C++ PL exceptions.

```
_try
{
    ...
}
_except(filter code)
{
    handler code
}
```

"Finally" block may be instead of handler code:

```
_try
{
    ...
}
_finally
{
    ...
}
```

The filter code is an expression, telling whether this handler code corresponds to the exception raised.

If your code is too big and cannot fit into one expression, a separate filter function can be defined.

There are a lot of such constructs in the Windows kernel. Here are a couple of examples from there ([WRK](#)):

Listing 6.24: WRK-v1.2/base/ntos/ob/obwait.c

```
try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
         EXCEPTION_EXECUTE_HANDLER :
         EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}
```

Listing 6.25: WRK-v1.2/base/ntos/cache/cachesub.c

```
try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                 UserBuffer,
                 MorePages ?
                     (PAGE_SIZE - PageOffset) :
                     (ReceivedLength - PageOffset) );

} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                    &Status ) ) {
```

<sup>45</sup>[MSDN](#)

## 6.5. WINDOWS NT

Here is also a filter code example:

Listing 6.26: WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ErrorCode
)

/*++

Routine Description:

    This routine serves as an exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains
                       the real Io Status.

    ErrorCode - A pointer to an NTSTATUS that is to receive the real
                status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/



{
    *ErrorCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
        (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}
```

Internally, SEH is an extension of the OS-supported exceptions. But the handler function is `_except_handler3` (for SEH3) or `_except_handler4` (for SEH4).

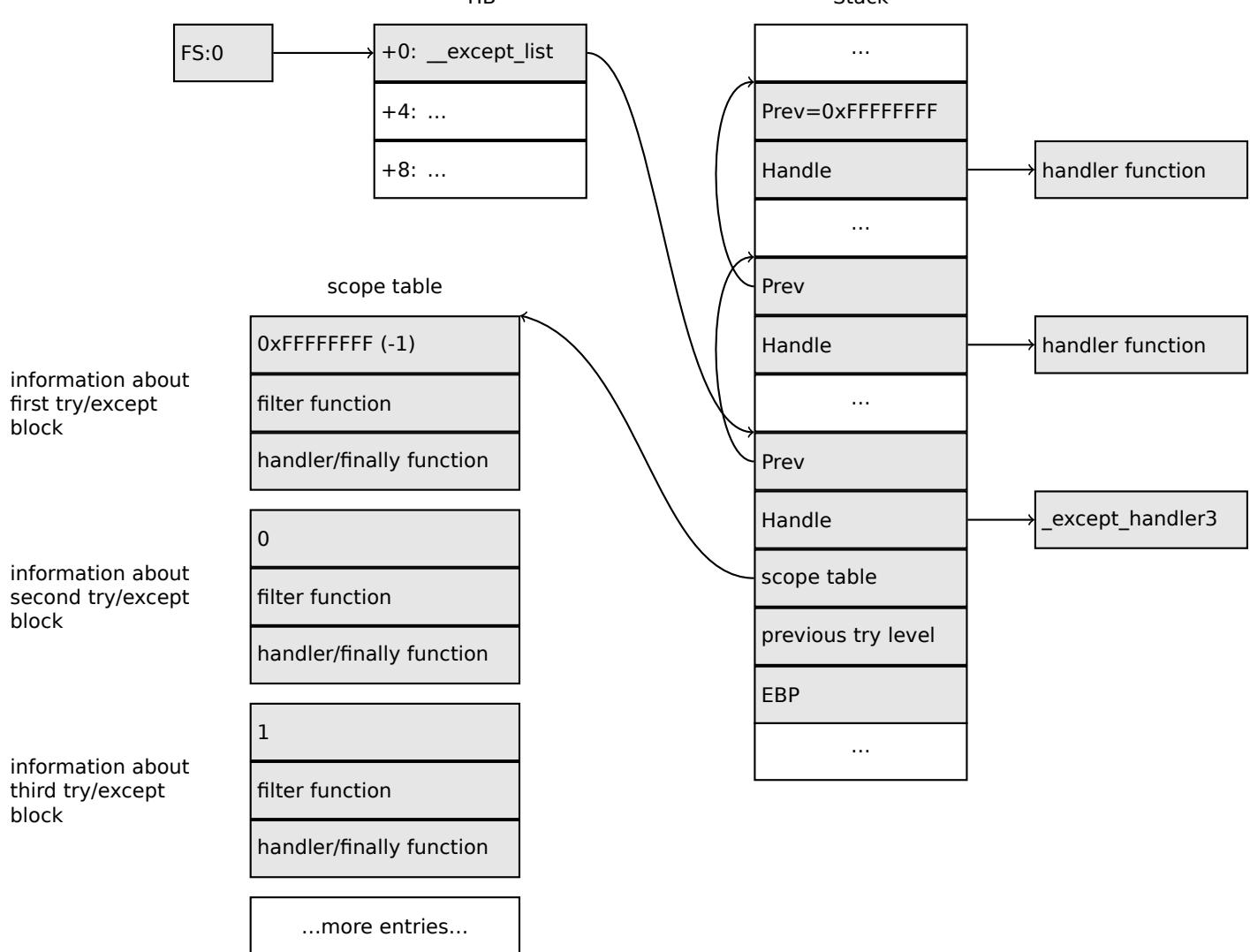
The code of this handler is MSVC-related, it is located in its libraries, or in `msvcr*.dll`. It is very important to know that SEH is a MSVC thing.

Other win32-compilers may offer something completely different.

### SEH3

SEH3 has `_except_handler3` as a handler function, and extends the `_EXCEPTION_REGISTRATION` table, adding a pointer to the *scope table* and *previous try level* variable. SEH4 extends the *scope table* by 4 values for buffer overflow protection.

The *scope table* is a table that consists of pointers to the filter and handler code blocks, for each nested level of *try/except*.



Again, it is very important to understand that the OS takes care only of the `prev/handle` fields, and nothing more.

It is the job of the `_except_handler3` function to read the other fields and `scope table`, and decide which handler to execute and when.

The source code of the `_except_handler3` function is closed.

However, Sanos OS, which has a win32 compatibility layer, has the same functions reimplemented, which are somewhat equivalent to those in Windows<sup>46</sup>. Another reimplementation is present in Wine<sup>47</sup> and ReactOS<sup>48</sup>.

If the `filter` pointer is `NULL`, the `handler` pointer is the pointer to the `finally` code block.

During execution, the `previous try level` value in the stack changes, so `_except_handler3` can get information about the current level of nestedness, in order to know which `scope table` entry to use.

### SEH3: one try/except block example

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
```

<sup>46</sup><http://go.yurichev.com/17058>

<sup>47</sup>[GitHub](#)

<sup>48</sup><http://go.yurichev.com/17060>

## 6.5. WINDOWS NT

```

try
{
    printf("hello #1!\n");
    *p = 13;      // causes an access violation exception;
    printf("hello #2!\n");
}
__except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
          EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
{
    printf("access violation, can't recover\n");
}
}

```

Listing 6.27: MSVC 2003

```

$SG74605 DB      'hello #1!', 0aH, 00H
$SG74606 DB      'hello #2!', 0aH, 00H
$SG74608 DB      'access violation, can''t recover', 0aH, 00H
_DATA    ENDS

; scope table:
CONST     SEGMENT
$T74622   DD      0xffffffffH      ; previous try level
            DD      FLAT:$L74617    ; filter
            DD      FLAT:$L74618    ; handler

CONST     ENDS
_TEXT     SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28    ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
    push    ebp
    mov     ebp, esp
    push    -1                      ; previous try level
    push    OFFSET FLAT:$T74622        ; scope table
    push    OFFSET FLAT:_except_handler3 ; handler
    mov     eax, DWORD PTR fs:_except_list
    push    eax                      ; prev
    mov     DWORD PTR fs:_except_list, esp
    add    esp, -16
; 3 registers to be saved:
    push    ebx
    push    esi
    push    edi
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push    OFFSET FLAT:$SG74605 ; 'hello #1'
    call    _printf
    add    esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    push    OFFSET FLAT:$SG74606 ; 'hello #2'
    call    _printf
    add    esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; previous try level
    jmp    SHORT $L74616

; filter code:
$L74617:
$L74627:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74621[ebp], eax
    mov    eax, DWORD PTR $T74621[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax

```

## 6.5. WINDOWS NT

```
inc    eax
$L74619:
$L74626:
ret    0

; handler code:
$L74618:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74608 ; 'access violation, can't recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous try level back to -1
$L74616:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:_except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
main    ENDP
_TEXT   ENDS
END
```

Here we see how the SEH frame is constructed in the stack. The *scope table* is located in the CONST segment—indeed, these fields are not to be changed. An interesting thing is how the *previous try level* variable has changed. The initial value is 0xFFFFFFFF (-1). The moment when the body of the try statement is opened is marked with an instruction that writes 0 to the variable. The moment when the body of the try statement is closed, -1 is written back to it. We also see the addresses of filter and handler code.

Thus we can easily see the structure of the try/except constructs in the function.

Since the SEH setup code in the function prologue may be shared between many functions, sometimes the compiler inserts a call to the `SEH_prolog()` function in the prologue, which does just that.

The SEH cleanup code is in the `SEH_epilog()` function.

Let's try to run this example in `tracer`:

```
tracer.exe -l:2.exe --dump-seh
```

Listing 6.28: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0x60) handler=0x401088 ↴
    ↴ (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!mainCRTStartup+0x18d) ↴
    ↴ handler=0x401545 (2.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xffffffff GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffff EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll! ↴
    ↴ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16 ↴
    ↴ )
```

## 6.5. WINDOWS NT

We see that the SEH chain consists of 4 handlers.

The first two are located in our example. Two? But we made only one? Yes, another one has been set up in the CRT function `_mainCRTStartup()`, and as it seems that it handles at least FPU exceptions. Its source code can be found in the MSVC installation: `crt/src/winxfltr.c`.

The third is the SEH4 one in ntdll.dll, and the fourth handler is not MSVC-related and is located in ntdll.dll, and has a self-describing function name.

As you can see, there are 3 types of handlers in one chain:

one is not related to MSVC at all (the last one) and two MSVC-related: SEH3 and SEH4.

### SEH3: two try/except blocks example

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}

int main()
{
    int* p = NULL;
    try
    {
        try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13;      // causes an access violation exception;
        }
        __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
                  EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to the question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

Now there are two `try` blocks. So the *scope table* now has two entries, one for each block. *Previous try level* changes as execution flow enters or exits the `try` block.

Listing 6.29: MSVC 2003

```
$SG74606 DB      'in filter. code=0x%08X', 0Ah, 00H
$SG74608 DB      'yes, that is our exception', 0Ah, 00H
```

## 6.5. WINDOWS NT

```
$SG74610 DB      'not our exception', 0aH, 00H
$SG74617 DB      'hello!', 0aH, 00H
$SG74619 DB      '0x112233 raised. now let's crash', 0aH, 00H
$SG74621 DB      'access violation, can't recover', 0aH, 00H
$SG74623 DB      'user exception caught', 0aH, 00H

_code$ = 8      ; size = 4
_ep$ = 12      ; size = 4
_filter_user_exceptions PROC NEAR
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _code$[ebp]
    push    eax
    push    OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call    _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push    OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call    _printf
    add    esp, 4
    mov     eax, 1
    jmp    SHORT $L74605
$L74607:
    push    OFFSET FLAT:$SG74610 ; 'not our exception'
    call    _printf
    add    esp, 4
    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table:
CONST   SEGMENT
$T74644  DD      0xffffffffH ; previous try level for outer block
            DD      FLAT:$L74634 ; outer block filter
            DD      FLAT:$L74635 ; outer block handler
            DD      00H          ; previous try level for inner block
            DD      FLAT:$L74638 ; inner block filter
            DD      FLAT:$L74639 ; inner block handler
CONST   ENDS

$T74643 = -36      ; size = 4
$T74642 = -32      ; size = 4
_p$ = -28      ; size = 4
__$SEHRec$ = -24    ; size = 24
_main    PROC NEAR
    push    ebp
    mov     ebp, esp
    push    -1 ; previous try level
    push    OFFSET FLAT:$T74644
    push    OFFSET FLAT:_except_handler3
    mov     eax, DWORD PTR fs:_except_list
    push    eax
    mov     DWORD PTR fs:_except_list, esp
    add    esp, -20
    push    ebx
    push    esi
    push    edi
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entered. set previous try level to 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entered. set previous try level to 1
    push    OFFSET FLAT:$SG74617 ; 'hello!'
    call    _printf
    add    esp, 4
    push    0
_main    ENDP
```

## 6.5. WINDOWS NT

```
push    0
push    0
push    1122867      ; 00112233H
call    DWORD PTR __imp__RaiseException@16
push    OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s crash'
call    _printf
add    esp, 4
mov    eax, DWORD PTR _p$[ebp]
mov    DWORD PTR [eax], 13
mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level ↵
↳ back to 0
jmp    SHORT $L74615

; inner block filter:
$L74638:
$L74650:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74643[ebp], eax
    mov    eax, DWORD PTR $T74643[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74640:
$L74648:
    ret    0

; inner block handler:
$L74639:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74621 ; 'access violation, can''t recover'
    call    _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level ↵
    ↳ back to 0

$L74615:
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set previous try level ↵
    ↳ back to -1
    jmp    SHORT $L74633

; outer block filter:
$L74634:
$L74651:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74642[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T74642[ebp]
    push   edx
    call    _filter_user_exceptions
    add    esp, 8
$L74636:
$L74649:
    ret    0

; outer block handler:
$L74635:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74623 ; 'user exception caught'
    call    _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set previous try level ↵
    ↳ back to -1
$L74633:
    xor    eax, eax
```

## 6.5. WINDOWS NT

```
mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
mov    DWORD PTR fs:_except_list, ecx
pop    edi
pop    esi
pop    ebx
mov    esp, ebp
pop    ebp
ret    0
_main  ENDP
```

If we set a breakpoint on the `printf()` function, which is called from the handler, we can also see how yet another SEH handler is added.

Perhaps it's another machinery inside the SEH handling process. Here we also see our *scope table* consisting of 2 entries.

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

Listing 6.30: tracer.exe output

```
(0) 3.exe!printf
EAX=0x00000001b EBX=0x000000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x000000000 EDI=0x000000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0xb0) handler=0x40113b ↴
  ↴ (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78) handler=0x401100 ↴
  ↴ (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!mainCRTStartup+0x18d) ↴
  ↴ handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:   GSCookie0ffset=0xffffffff GSCookieXOROffset=0x0
               EHCookie0ffset=0xffffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll! ↴
  ↴ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16 ↴
  ↴ )
```

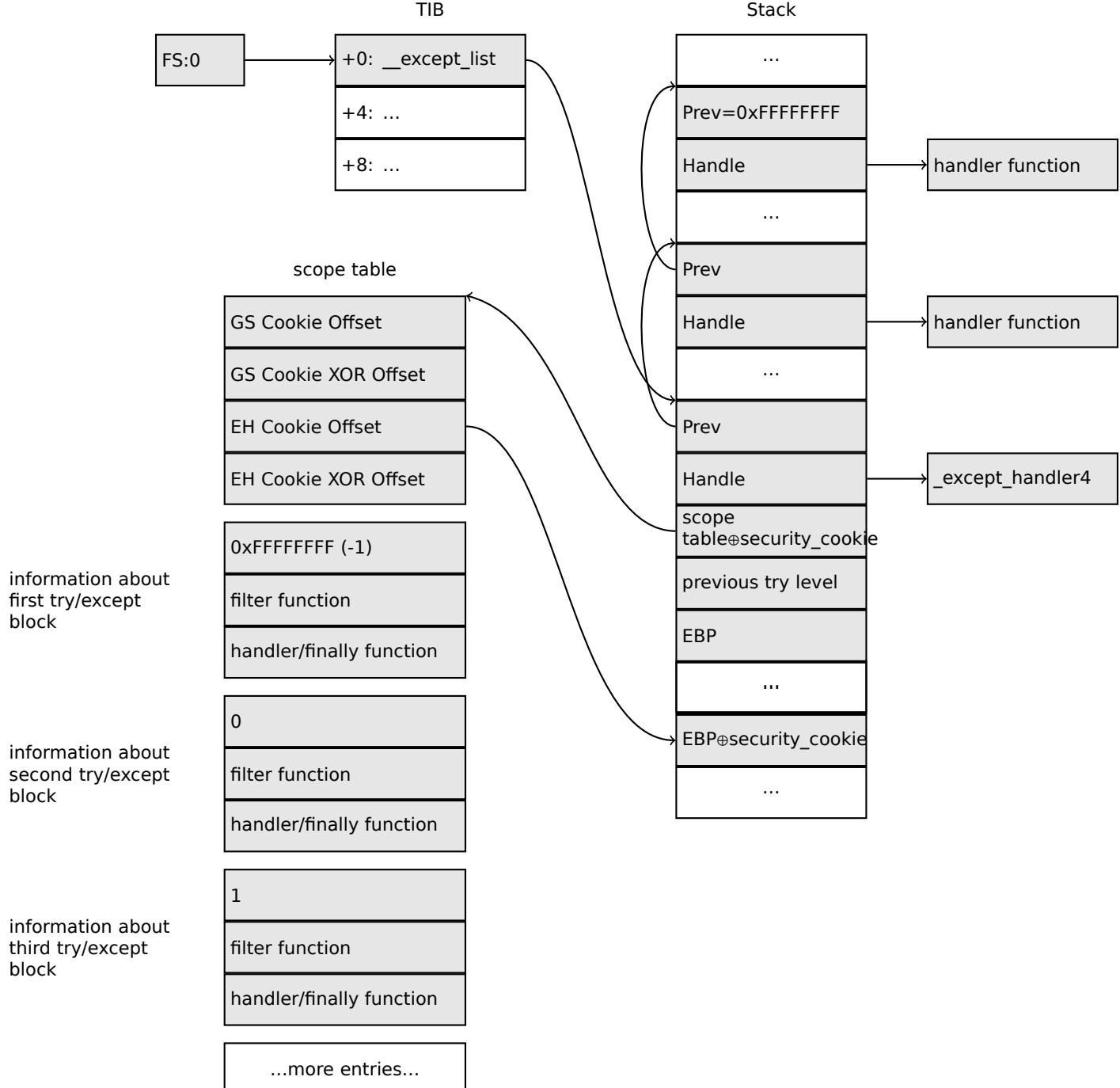
## SEH4

During a buffer overflow ([1.20.2 on page 275](#)) attack, the address of the *scope table* can be rewritten, so starting from MSVC 2005, SEH3 was upgraded to SEH4 in order to have buffer overflow protection. The pointer to the *scope table* is now *xored* with a *security cookie*. The *scope table* was extended to have a header consisting of two pointers to *security cookies*.

Each element has an offset inside the stack of another value: the address of the *stack frame* (EBP) *xored* with the *security\_cookie*, placed in the stack.

This value will be read during exception handling and checked for correctness. The *security cookie* in the stack is random each time, so hopefully a remote attacker can't predict it.

The initial *previous try level* is -2 in SEH4 instead of -1.



Here are both examples compiled in MSVC 2012 with SEH4:

Listing 6.31: MSVC 2012: one try block example

```
$SG85485 DB      'hello #1!', 0aH, 00H
$SG85486 DB      'hello #2!', 0aH, 00H
$SG85488 DB      'access violation, can''t recover', 0aH, 00H

; scope table:
xdata$x          SEGMENT
__sehtable$_main DD 0xfffffffffeH ; GS Cookie Offset
                   00H           ; GS Cookie XOR Offset
                   0xffffffffccH ; EH Cookie Offset
                   00H           ; EH Cookie XOR Offset
                   0xfffffffffeH ; previous try level
                   FLAT:$LN12@main ; filter
                   FLAT:$LN8@main  ; handler
xdata$x          ENDS

$T2 = -36        ; size = 4
_p$ = -32        ; size = 4
tv68 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
```

## 6.5. WINDOWS NT

```
_main PROC
    push    ebp
    mov     ebp, esp
    push    -2
    push    OFFSET __sehtable$_main
    push    OFFSET __except_handler4
    mov     eax, DWORD PTR fs:0
    push    eax
    add     esp, -20
    push    ebx
    push    esi
    push    edi
    mov     eax, DWORD PTR __security_cookie
    xor     DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor     eax, ebp
    push    eax          ; ebp ^ security_cookie
    lea     eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov     DWORD PTR fs:0, eax
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push    OFFSET $SG85485 ; 'hello #1!'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    push    OFFSET $SG85486 ; 'hello #2!'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T2[ebp], eax
    cmp     DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov     DWORD PTR tv68[ebp], 1
    jmp    SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR tv68[ebp], 0
$LN5@main:
    mov     eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret    0

; handler:
$LN8@main:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET $SG85488 ; 'access violation, can''t recover'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov     esp, ebp
    pop    ebp
    ret    0
_main  ENDP
```

## 6.5. WINDOWS NT

Listing 6.32: MSVC 2012: two try blocks example

```
$SG85486 DB      'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB      'yes, that is our exception', 0aH, 00H
$SG85490 DB      'not our exception', 0aH, 00H
$SG85497 DB      'hello!', 0aH, 00H
$SG85499 DB      '0x112233 raised. now let's crash', 0aH, 00H
$SG85501 DB      'access violation, can't recover', 0aH, 00H
$SG85503 DB      'user exception caught', 0aH, 00H

xdata$x      SEGMENT
__sehtable$_main DD 0xfffffffffeH          ; GS Cookie Offset
                  DD 00H                 ; GS Cookie XOR Offset
                  DD 0xffffffffc8H        ; EH Cookie Offset
                  DD 00H                 ; EH Cookie Offset
                  DD 0xfffffffffeH        ; previous try level for outer block
                  DD FLAT:$LN19@main    ; outer block filter
                  DD FLAT:$LN9@main     ; outer block handler
                  DD 00H                 ; previous try level for inner block
                  DD FLAT:$LN18@main    ; inner block filter
                  DD FLAT:$LN13@main    ; inner block handler
xdata$x      ENDS

$T2 = -40        ; size = 4
$T3 = -36        ; size = 4
_p$ = -32        ; size = 4
tv72 = -28        ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push    ebp
    mov     ebp, esp
    push    -2    ; initial previous try level
    push    OFFSET __sehtable$_main
    push    OFFSET __except_handler4
    mov     eax, DWORD PTR fs:0
    push    eax ; prev
    add    esp, -24
    push    ebx
    push    esi
    push    edi
    mov     eax, DWORD PTR __security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax      ; xored pointer to scope table
    xor    eax, ebp                         ; ebp ^ security_cookie
    push    eax
    lea    eax, DWORD PTR __$SEHRec$[ebp+8]       ; pointer to ↴
    ↓ VC_EXCEPTION_REGISTRATION_RECORD
    mov     DWORD PTR fs:0, eax
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try block, setting previous try ↴
    ↓ level=0
    mov     DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try block, setting previous try ↴
    ↓ level=1
    push    OFFSET $SG85497 ; 'hello!'
    call    _printf
    add    esp, 4
    push    0
    push    0
    push    0
    push    1122867 ; 00112233H
    call    DWORD PTR __imp_RaiseException@16
    push    OFFSET $SG85499 ; '0x112233 raised. now let's crash'
    call    _printf
    add    esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set previous try level ↴
    ↓ back to 0
    jmp    SHORT $LN2@main

; inner block filter:
```

## 6.5. WINDOWS NT

```
$LN12@main:
$LN18@main:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T3[ebp], eax
    cmp     DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne     SHORT $LN5@main
    mov     DWORD PTR tv72[ebp], 1
    jmp     SHORT $LN6@main
$LN5@main:
    mov     DWORD PTR tv72[ebp], 0
$LN6@main:
    mov     eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
    ret     0

; inner block handler:
$LN13@main:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET $SG85501 ; 'access violation, can''t recover'
    call    _printf
    add    esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting previous try ↴
    ↴ level back to 0
$LN2@main:
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level ↴
    ↴ back to -2
    jmp     SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN19@main:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T2[ebp], eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    push    ecx
    mov     edx, DWORD PTR $T2[ebp]
    push    edx
    call    _filter_user_exceptions
    add    esp, 8
$LN10@main:
$LN17@main:
    ret     0

; outer block handler:
$LN9@main:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET $SG85503 ; 'user exception caught'
    call    _printf
    add    esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level ↴
    ↴ back to -2
$LN7@main:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov     esp, ebp
    pop    ebp
    ret     0
_main    ENDP

_code$ = 8 ; size = 4
```

## 6.5. WINDOWS NT

```
_ep$ = 12      ; size = 4
_filter_user_exceptions PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _code$[ebp]
    push    eax
    push    OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call    _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne    SHORT $LN2@filter_use
    push    OFFSET $SG85488 ; 'yes, that is our exception'
    call    _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $LN3@filter_use
    jmp    SHORT $LN3@filter_use
$LN2@filter_use:
    push    OFFSET $SG85490 ; 'not our exception'
    call    _printf
    add    esp, 4
    xor    eax, eax
$LN3@filter_use:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP
```

Here is the meaning of the *cookies*: `Cookie Offset` is the difference between the address of the saved EBP value in the stack and the `EBP + security_cookie` value in the stack. `Cookie XOR Offset` is an additional difference between the `EBP + security_cookie` value and what is stored in the stack.

If this equation is not true, the process is to halt due to stack corruption:

`security_cookie ⊕ (CookieXOROffset + address_of_saved_EBP) == stack[address_of_saved_EBP + CookieOffset]`

If `Cookie Offset` is -2, this implies that it is not present.

Cookies checking is also implemented in my [tracer](#), see [GitHub](#) for details.

It is still possible to fall back to SEH3 in the compilers after (and including) MSVC 2005 by setting the `/GS-` option, however, the [CRT](#) code use SEH4 anyway.

## Windows x64

As you might think, it is not very fast to set up the SEH frame at each function prologue. Another performance problem is changing the *previous try level* value many times during the function's execution.

So things are changed completely in x64: now all pointers to `try` blocks, filter and handler functions are stored in another PE segment `.pdata`, and from there the OS's exception handler takes all the information.

Here are the two examples from the previous section compiled for x64:

Listing 6.33: MSVC 2012

```
$SG86276 DB      'hello #1!', 0aH, 00H
$SG86277 DB      'hello #2!', 0aH, 00H
$SG86279 DB      'access violation, can''t recover', 0aH, 00H

pdata   SEGMENT
$pdata$main DD  imagerel $LN9
            DD  imagerel $LN9+61
            DD  imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filter$0 DD imagerel main$filter$0
            DD  imagerel main$filter$0+32
            DD  imagerel $unwind$main$filter$0
pdata   ENDS
```

## 6.5. WINDOWS NT

```

xdata SEGMENT
$unwind$main DD 020609H
    DD 030023206H
    DD imagerel __C_specific_handler
    DD 01H
    DD imagerel $LN9+8
    DD imagerel $LN9+40
    DD imagerel main$filter$0
    DD imagerel $LN9+40
$unwind$main$filter$0 DD 020601H
    DD 050023206H
xdata ENDS

_TEXT SEGMENT
main PROC
$LN9:
    push rbx
    sub  rsp, 32
    xor  ebx, ebx
    lea   rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
    call  printf
    mov   DWORD PTR [rbx], 13
    lea   rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
    call  printf
    jmp   SHORT $LN8@main
$LN6@main:
    lea   rcx, OFFSET FLAT:$SG86279 ; 'access violation, can't recover'
    call  printf
    npad 1 ; align next label
$LN8@main:
    xor  eax, eax
    add  rsp, 32
    pop  rbx
    ret  0
main ENDP
_TEXT ENDS

text$x SEGMENT
main$filter$0 PROC
    push rbp
    sub  rsp, 32
    mov  rbp, rdx
$LN5@main$filter$0:
    mov  rax, QWORD PTR [rcx]
    xor  ecx, ecx
    cmp  DWORD PTR [rax], -1073741819; c0000005H
    sete cl
    mov  eax, ecx
$LN7@main$filter$0:
    add  rsp, 32
    pop  rbp
    ret  0
    int  3
main$filter$0 ENDP
text$x ENDS

```

Listing 6.34: MSVC 2012

\$SG86277 DB	'in filter. code=0x%08X', 0aH, 00H
\$SG86279 DB	'yes, that is our exception', 0aH, 00H
\$SG86281 DB	'not our exception', 0aH, 00H
\$SG86288 DB	'hello!', 0aH, 00H
\$SG86290 DB	'0x112233 raised. now let's crash', 0aH, 00H
\$SG86292 DB	'access violation, can't recover', 0aH, 00H
\$SG86294 DB	'user exception caught', 0aH, 00H

```

pdata SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
    DD imagerel $LN6+73
    DD imagerel $unwind$filter_user_exceptions

```

## 6.5. WINDOWS NT

```
$pdata$main DD imagerel $LN14
    DD imagerel $LN14+95
    DD imagerel $unwind$main
pdata ENDS
pdata SEGMENT
$pdata$main$filter$0 DD imagerel main$filter$0
    DD imagerel main$filter$0+32
    DD imagerel $unwind$main$filter$0
$pdata$main$filter$1 DD imagerel main$filter$1
    DD imagerel main$filter$1+30
    DD imagerel $unwind$main$filter$1
pdata ENDS

xdata SEGMENT
$unwind$filter_user_exceptions DD 020601H
    DD 030023206H
$unwind$main DD 020609H
    DD 030023206H
    DD imagerel __C_specific_handler
    DD 02H
    DD imagerel $LN14+8
    DD imagerel $LN14+59
    DD imagerel main$filter$0
    DD imagerel $LN14+59
    DD imagerel $LN14+8
    DD imagerel $LN14+74
    DD imagerel main$filter$1
    DD imagerel $LN14+74
$unwind$main$filter$0 DD 020601H
    DD 050023206H
$unwind$main$filter$1 DD 020601H
    DD 050023206H
xdata ENDS

_TEXT SEGMENT
main PROC
$LN14:
    push rbx
    sub  rsp, 32
    xor  ebx, ebx
    lea   rcx, OFFSET FLAT:$SG86288 ; 'hello!'
    call printf
    xor  r9d, r9d
    xor  r8d, r8d
    xor  edx, edx
    mov  ecx, 1122867 ; 00112233H
    call QWORD PTR __imp_RaiseException
    lea   rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s crash'
    call printf
    mov  DWORD PTR [rbx], 13
    jmp  SHORT $LN13@main
$LN11@main:
    lea   rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t recover'
    call printf
    npad 1 ; align next label
$LN13@main:
    jmp  SHORT $LN9@main
$LN7@main:
    lea   rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
    call printf
    npad 1 ; align next label
$LN9@main:
    xor  eax, eax
    add  rsp, 32
    pop  rbx
    ret  0
main ENDP

text$x SEGMENT
main$filter$0 PROC
```

## 6.5. WINDOWS NT

```
push    rbp
sub    rsp, 32
mov    rbp, rdx
$LN10@main$filter$:
    mov    rax, QWORD PTR [rcx]
    xor    ecx, ecx
    cmp    DWORD PTR [rax], -1073741819; c0000005H
    sete   cl
    mov    eax, ecx
$LN12@main$filter$:
    add    rsp, 32
    pop    rbp
    ret    0
    int    3
main$filter$0 ENDP

main$filter$1 PROC
    push   rbp
    sub    rsp, 32
    mov    rbp, rdx
$LN6@main$filter$:
    mov    rax, QWORD PTR [rcx]
    mov    rdx, rcx
    mov    ecx, DWORD PTR [rax]
    call   filter_user_exceptions
    npad   1 ; align next label
$LN8@main$filter$:
    add    rsp, 32
    pop    rbp
    ret    0
    int    3
main$filter$1 ENDP
text$x ENDS

_TEXT  SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
    push   rbx
    sub    rsp, 32
    mov    ebx, ecx
    mov    edx, ecx
    lea    rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
    call   printf
    cmp    ebx, 1122867; 00112233H
    jne    SHORT $LN2@filter_use
    lea    rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
    call   printf
    mov    eax, 1
    add    rsp, 32
    pop    rbx
    ret    0
$LN2@filter_use:
    lea    rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
    call   printf
    xor    eax, eax
    add    rsp, 32
    pop    rbx
    ret    0
filter_user_exceptions ENDP
_TEXT  ENDS
```

Read [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]<sup>49</sup> for more detailed information about this.

Aside from exception information, `.pdata` is a section that contains the addresses of almost all function starts and ends, hence it may be useful for a tools targeted at automated analysis.

<sup>49</sup>Also available as <http://go.yurichev.com/17294>

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]<sup>50</sup>, [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]<sup>51</sup>.

### 6.5.4 Windows NT: Critical section

Critical sections in any OS are very important in multithreaded environment, mostly for giving a guarantee that only one thread can access some data in a single moment of time, while blocking other threads and interrupts.

That is how a `CRITICAL_SECTION` structure is declared in [Windows NT](#) line OS:

Listing 6.35: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    // The following three fields control entering and exiting the critical
    // section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;           // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;          // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

That's how `EnterCriticalSection()` function works:

Listing 6.36: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4

var_C      = dword ptr -0Ch
var_8      = dword ptr -8
var_4      = dword ptr -4
arg_0      = dword ptr 8

        mov     edi, edi
        push    ebp
        mov     ebp, esp
        sub     esp, 0Ch
        push    esi
        push    edi
        mov     edi, [ebp+arg_0]
        lea     esi, [edi+4] ; LockCount
        mov     eax, esi
        lock btr dword ptr [eax], 0
        jnb     wait ; jump if CF=0

loc_7DE922DD:
        mov     eax, large fs:18h
        mov     ecx, [eax+24h]
        mov     [edi+0Ch], ecx
        mov     dword ptr [edi+8], 1
        pop     edi
        xor     eax, eax
        pop     esi
        mov     esp, ebp
        pop     ebp
        retn   4

... skipped
```

<sup>50</sup>Also available as <http://go.yurichev.com/17293>

<sup>51</sup>Also available as <http://go.yurichev.com/17294>

## 6.5. WINDOWS NT

The most important instruction in this code fragment is `BTR` (prefixed with `LOCK`):  
the zeroth bit is stored in the `CF` flag and cleared in memory. This is an [atomic operation](#),  
blocking all other CPUs' access to this piece of memory (see the `LOCK` prefix before the `BTR` instruction).  
If the bit at `LockCount` is 1,  
fine, reset it and return from the function: we are in a critical section.  
If not—the critical section is already occupied by other thread, so wait.  
The wait is performed there using `WaitForSingleObject()`.

And here is how the `LeaveCriticalSection()` function works:

Listing 6.37: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near

arg_0      = dword ptr  8

        mov     edi, edi
        push    ebp
        mov     ebp, esp
        push    esi
        mov     esi, [ebp+arg_0]
        add    dword ptr [esi+8], 0FFFFFFFh ; RecursionCount
        jnz    short loc_7DE922B2
        push    ebx
        push    edi
        lea     edi, [esi+4]    ; LockCount
        mov     dword ptr [esi+0Ch], 0
        mov     ebx, 1
        mov     eax, edi
        lock xadd [eax], ebx
        inc     ebx
        cmp     ebx, 0FFFFFFFh
        jnz    loc_7DEA8EB7

loc_7DE922B0:
        pop     edi
        pop     ebx

loc_7DE922B2:
        xor     eax, eax
        pop     esi
        pop     ebp
        retn   4

... skipped
```

`XADD` is “exchange and add”.

In this case, it adds 1 to `LockCount`, meanwhile saves initial value of `LockCount` in the `EBX` register.  
However, value in `EBX` is to be incremented with a help of subsequent `INC EBX`, and it also will be equal to  
the updated value of `LockCount`.

This operation is atomic since it is prefixed by `LOCK` as well, meaning that all other CPUs or CPU cores in  
system are blocked from accessing this point in memory.

The `LOCK` prefix is very important:

without it two threads, each of which works on separate CPU or CPU core can try to enter a critical section  
and to modify the value in memory, which will result in non-deterministic behavior.

# Chapter 7

## Tools

Now that Dennis Yurichev has made this book free (*libre*), it is a contribution to the world of free knowledge and free education. However, for our freedom's sake, we need free (*libre*) reverse engineering tools to replace the proprietary tools described in this book.

---

Richard M. Stallman

### 7.1 Binary analysis

Tools you use when you don't run any process.

- (Free, open-source) *ent*<sup>1</sup>: entropy analyzing tool. Read more about entropy: [9.2 on page 947](#).
- *Hiew*<sup>2</sup>: for small modifications of code in binary files. Has dssembler/issasember.
- (Free, open-source) *GHex*<sup>3</sup>: simple hexadecimal editor for Linux.
- (Free, open-source) *xxd* and *od*: standard UNIX utilities for dumping.
- (Free, open-source) *strings*: \*NIX tool for searching for ASCII strings in binary files, including executable ones. Sysinternals has alternative<sup>4</sup> supporting wide char strings (UTF-16, widely used in Windows).
- (Free, open-source) *Binwalk*<sup>5</sup>: analyzing firmware images.
- (Free, open-source) *binary grep*: a small utility for searching any byte sequence in a big pile of files, including non-executable ones: [GitHub](#). There is also rafind2 in rada.re for the same purpose.

#### 7.1.1 Disassemblers

- *IDA*. An older freeware version is available for download<sup>6</sup>. Hot-keys cheatsheet: [.6.1 on page 1027](#)
- *Binary Ninja*<sup>7</sup>
- (Free, open-source) *zynamics BinNavi*<sup>8</sup>
- (Free, open-source) *objdump*: simple command-line utility for dumping and disassembling.
- (Free, open-source) *readelf*<sup>9</sup>: dump information about ELF file.

---

<sup>1</sup><http://www.fourmilab.ch/random/>

<sup>2</sup>[hiew.ru](http://hiew.ru)

<sup>3</sup><https://wiki.gnome.org/Apps/Ghex>

<sup>4</sup><https://technet.microsoft.com/en-us/sysinternals/strings>

<sup>5</sup><http://binwalk.org/>

<sup>6</sup>[hex-rays.com/products/ida/support/download\\_freeware.shtml](http://hex-rays.com/products/ida/support/download_freeware.shtml)

<sup>7</sup><http://binary.ninja/>

<sup>8</sup><https://www.zynamics.com/binnavi.html>

<sup>9</sup><https://sourceware.org/binutils/docs/binutils/readelf.html>

## 7.1.2 Decompilers

There is only one known, publicly available, high-quality decompiler to C code: *Hex-Rays*: [hex-rays.com/products/decompiler/](http://hex-rays.com/products/decompiler/)

## 7.1.3 Patch comparison/diffing

You may want to use it when you compare original version of some executable and patched one, in order to find what has been patched and why.

- (Free) *zynamics BinDiff*<sup>10</sup>
- (Free, open-source) *Diaphora*<sup>11</sup>

## 7.2 Live analysis

Tools you use on a live system or during running of a process.

### 7.2.1 Debuggers

- (Free) *OllyDbg*. Very popular user-mode win32 debugger<sup>12</sup>. Hot-keys cheatsheet: [.6.2 on page 1027](#)
- (Free, open-source) *GDB*. Not quite popular debugger among reverse engineers, because it's intended mostly for programmers. Some commands: [.6.5 on page 1028](#). There is a visual interface for GDB, "GDB dashboard"<sup>13</sup>.
- (Free, open-source) *LLDB*<sup>14</sup>.
- *WinDbg*<sup>15</sup>: kernel debugger for Windows.
- *IDA* has internal debugger.
- (Free, open-source) *Radare AKA rada.re AKA r2*<sup>16</sup>. A GUI also exists: *ragui*<sup>17</sup>.
- (Free, open-source) *tracer*. The author often uses *tracer*<sup>18</sup> instead of a debugger.

The author of these lines stopped using a debugger eventually, since all he needs from it is to spot function arguments while executing, or registers state at some point. Loading a debugger each time is too much, so a small utility called *tracer* was born. It works from command line, allows intercepting function execution, setting breakpoints at arbitrary places, reading and changing registers state, etc.

N.B.: the *tracer* isn't evolving, because it was developed as a demonstration tool for this book, not as everyday tool.

### 7.2.2 Library calls tracing

*ltrace*<sup>19</sup>.

---

<sup>10</sup><https://www.zynamics.com/software.html>

<sup>11</sup><https://github.com/joxeankoret/diaphora>

<sup>12</sup>[ollydbg.de](http://ollydbg.de)

<sup>13</sup><https://github.com/cyrus-and/gdb-dashboard>

<sup>14</sup><http://lldb.llvm.org/>

<sup>15</sup><https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit>

<sup>16</sup><http://rada.re/r/>

<sup>17</sup><http://radare.org/ragui/>

<sup>18</sup>[yurichev.com](http://yurichev.com)

<sup>19</sup><http://www.ltrace.org/>

## 7.2. LIVE ANALYSIS

### **7.2.3 System calls tracing**

## **strace / dtruss**

It shows which system calls (syscalls( 6.3 on page 746)) are called by a process right now.

For example:

Mac OS X has dtruss for doing the same.

Cygwin also has strace, but as far as it's known, it works only for .exe-files compiled for the cygwin environment itself.

## 7.2.4 Network sniffing

*Sniffing* is intercepting some information you may be interested in.

(Free, open-source) *Wireshark*<sup>20</sup> for network sniffing. It has also capability for USB sniffing<sup>21</sup>.

Wireshark has a younger (or older) brother `tcpdump`<sup>22</sup>, simpler command-line tool.

## 7.2.5 Sysinternals

(Free) Sysinternals (developed by Mark Russinovich) <sup>23</sup>. At least these tools are important and worth studying: Process Explorer, Handle, VMMap, TCPView, Process Monitor.

## 7.2.6 Valgrind

(Free, open-source) a powerful tool for detecting memory leaks: <http://valgrind.org/>. Due to its powerful JIT mechanism, Valgrind is used as a framework for other tools.

## 7.2.7 Emulators

- (Free, open-source) *QEMU*<sup>24</sup>: emulator for various CPUs and architectures.
  - (Free, open-source) *DosBox*<sup>25</sup>: MS-DOS emulator, mostly used for retrogaming.
  - (Free, open-source) *SimH*<sup>26</sup>: emulator of ancient computers, mainframes, etc.

<sup>20</sup> <https://www.wireshark.org/>

<sup>21</sup> <https://wiki.wireshark.org/CaptureSetup/USB>

<sup>22</sup><http://www.tcpdump.org/>

<sup>23</sup><https://technet.microsoft.com/en-us/sysinternals/bb842062>

24 <http://gemu.org>

25 <https://www.dosbox.com/>

<sup>26</sup> <http://simh.trailing-edge.com/>

## 7.3 Other tools

*Microsoft Visual Studio Express* <sup>27</sup>: Stripped-down free version of Visual Studio, convenient for simple experiments.

Some useful options: [.6.3 on page 1027](#).

There is a website named “Compiler Explorer”, allowing to compile small code snippets and see output in various GCC versions and architectures (at least x86, ARM, MIPS): <http://gcc.betab.godbolt.org/>—I would use it myself for the book if I would know about it!

## 7.4 Something missing here?

If you know a great tool not listed here, please drop me a note:

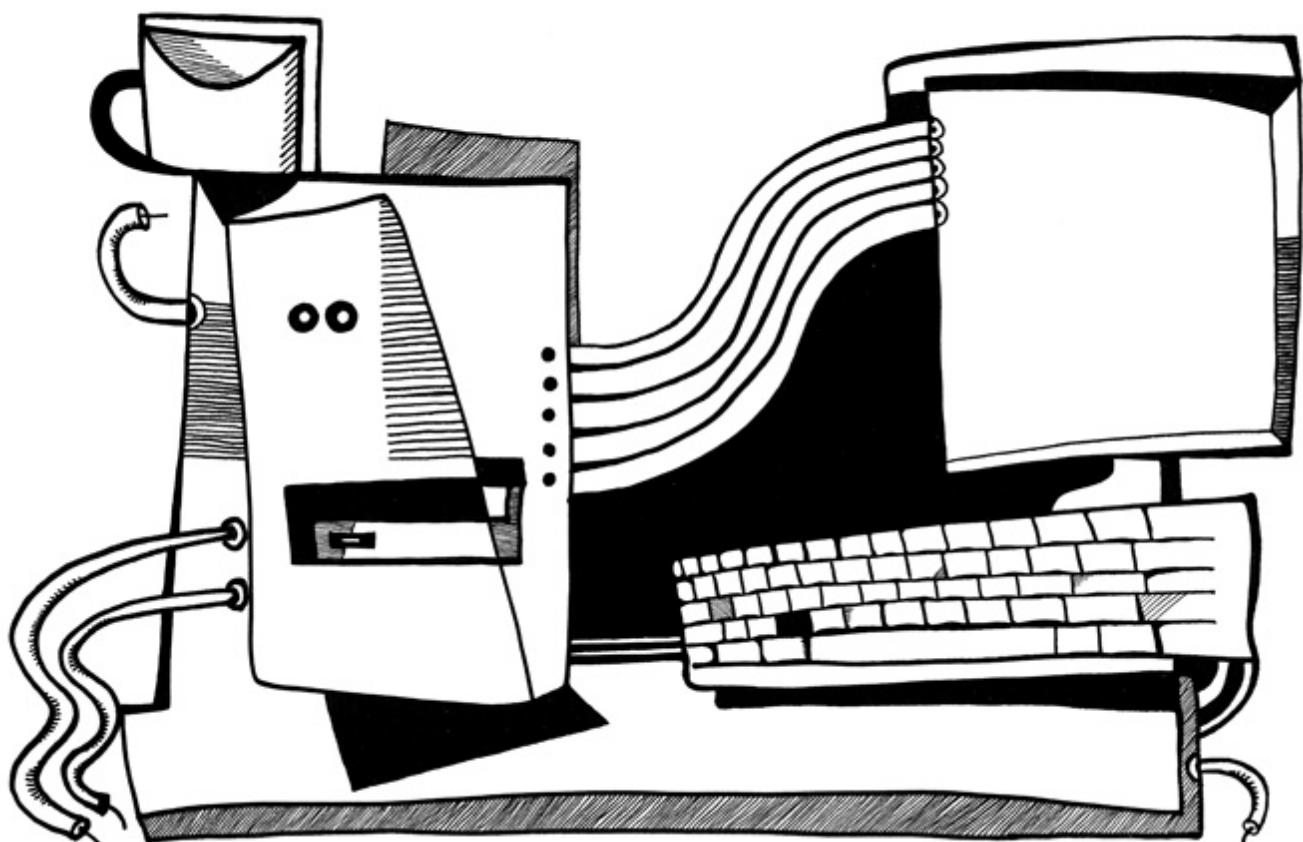
dennis(a)yurichev.com

---

<sup>27</sup>[visualstudio.com/en-US/products/visual-studio-express-vs](http://visualstudio.com/en-US/products/visual-studio-express-vs)

# **Chapter 8**

## **Case studies**



## 8.1 Task manager practical joke (Windows Vista)

Let's see if it's possible to hack Task Manager slightly so it would detect more CPU cores.

Let us first think, how does the Task Manager know the number of cores?

There is the `GetSystemInfo()` win32 function present in win32 userspace which can tell us this. But it's not imported in `taskmgr.exe`.

There is, however, another one in `NTAPI`, `NtQuerySystemInformation()`, which is used in `taskmgr.exe` in several places.

To get the number of cores, one has to call this function with the `SystemBasicInformation` constant as a first argument (which is zero<sup>1</sup>).

The second argument has to point to the buffer which is getting all the information.

So we have to find all calls to the

`NtQuerySystemInformation(0, ?, ?, ?)` function. Let's open `taskmgr.exe` in IDA.

What is always good about Microsoft executables is that IDA can download the corresponding `PDB` file for this executable and show all function names.

It is visible that Task Manager is written in C++ and some of the function names and classes are really speaking for themselves. There are classes `CAdapter`, `CNetPage`, `CPerfPage`, `CProcInfo`, `CProcPage`, `CSvcPage`, `CTaskPage`, `CUserPage`.

Apparently, each class corresponds to each tab in Task Manager.

Let's visit each call and add comment with the value which is passed as the first function argument. We will write "not zero" at some places, because the value there was clearly not zero, but something really different (more about this in the second part of this chapter).

And we are looking for zero passed as argument, after all.

xrefs to __imp_NtQuerySystemInformation			
Dire...	T.	Address	Text
↳ Up	p	wWinMain+50E	call cs:__imp_NtQuerySystemInformation; 0
↳ Up	p	wWinMain+542	call cs:__imp_NtQuerySystemInformation; 2
↳ Up	p	CPerfPage::TimerEvent(void)+200	call cs:__imp_NtQuerySystemInformation; not zero
↳	p	InitPerfInfo(void)+2C	call cs:__imp_NtQuerySystemInformation; 0
↳ D...	p	InitPerfInfo(void)+F0	call cs:__imp_NtQuerySystemInformation; 8
↳ D...	p	CalcCpuTime(int)+5F	call cs:__imp_NtQuerySystemInformation; 8
↳ D...	p	CalcCpuTime(int)+248	call cs:__imp_NtQuerySystemInformation; 2
↳ D...	p	CPerfPage::CalcPhysicalMem(unsigned ...)	call cs:__imp_NtQuerySystemInformation; not zero
↳ D...	p	CPerfPage::CalcPhysicalMem(unsigned ...)	call cs:__imp_NtQuerySystemInformation; not zero
↳ D...	p	CProcPage::GetProcessInfo(void)+2B	call cs:__imp_NtQuerySystemInformation; 5
↳ D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 0
↳ D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 2
↳ D...	p	CProcPage::Initialize(HWND__ *)+201	call cs:__imp_NtQuerySystemInformation; 0
↳ D...	p	CProcPage::GetTaskListEx(void)+3C	call cs:__imp_NtQuerySystemInformation; 5

Figure 8.1: IDA: cross references to `NtQuerySystemInformation()`

Yes, the names are really speaking for themselves.

When we closely investigate each place where

`NtQuerySystemInformation(0, ?, ?, ?)` is called, we quickly find what we need in the `InitPerfInfo()` function:

<sup>1</sup>MSDN

## 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

Listing 8.1: taskmgr.exe (Windows Vista)

```
.text:10000B4B3 xor    r9d, r9d
.text:10000B4B6 lea    rdx, [rsp+0C78h+var_C58] ; buffer
.text:10000B4BB xor    ecx, ecx
.text:10000B4BD lea    ebp, [r9+40h]
.text:10000B4C1 mov    r8d, ebp
.text:10000B4C4 call   cs:_imp_NtQuerySystemInformation ; 0
.text:10000B4CA xor    ebx, ebx
.text:10000B4CC cmp    eax, ebx
.text:10000B4CE jge    short loc_10000B4D7
.text:10000B4D0
.text:10000B4D0 loc_10000B4D0:           ; CODE XREF: InitPerfInfo(void)+97
.text:10000B4D0                 ; InitPerfInfo(void)+AF
xor    al, al
jmp   loc_10000B5EA
.text:10000B4D7 ; -----
.text:10000B4D7 loc_10000B4D7:           ; CODE XREF: InitPerfInfo(void)+36
mov    eax, [rsp+0C78h+var_C50]
.text:10000B4DB mov    esi, ebx
.text:10000B4DD mov    r12d, 3E80h
.text:10000B4E3 mov    cs:?g_PageSize@@3KA, eax ; ulong g_PageSize
.text:10000B4E9 shr    eax, 0Ah
.text:10000B4EC lea    r13, __ImageBase
.text:10000B4F3 imul   eax, [rsp+0C78h+var_C4C]
.text:10000B4F8 cmp    [rsp+0C78h+var_C20], bpl
.text:10000B4FD mov    cs:?g_MEMMax@@3_JA, rax ; __int64 g_MEMMax
.text:10000B504 movzx  eax, [rsp+0C78h+var_C20] ; number of CPUs
.text:10000B509 cmova eax, ebp
.text:10000B50C cmp    al, bl
.text:10000B50E mov    cs:?g_cProcessors@@3EA, al ; uchar g_cProcessors
```

`g_cProcessors` is a global variable, and this name has been assigned by IDA according to the [PDB](#) loaded from Microsoft's symbol server.

The byte is taken from `var_C20`. And `var_C58` is passed to

`NtQuerySystemInformation()` as a pointer to the receiving buffer. The difference between `0xC20` and `0xC58` is `0x38` (56).

Let's take a look at format of the return structure, which we can find in MSDN:

```
typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;
```

This is a x64 system, so each `PVOID` takes 8 byte.

All `reserved` fields in the structure take  $24 + 4 * 8 = 56$  bytes.

Oh yes, this implies that `var_C20` is the local stack is exactly the `NumberOfProcessors` field of the `SYSTEM_BASIC_INFORMATION` structure.

Let's check our guess. Copy `taskmgr.exe` from `C:\Windows\System32` to some other folder (so the `Windows Resource Protection` will not try to restore the patched `taskmgr.exe`).

Let's open it in Hiew and find the place:

### 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

```

01`0000B4F8: 40386C2458      cmp    [rsp][058], bp
01`0000B4FD: 48890544A00100  mov    [00000001, 00025548], rax
01`0000B504: 0FB6442458     movzx  eax, b, [rsp][058]
01`0000B509: 0F47C5          cmova eax, ebp
01`0000B50C: 3AC3            cmp    al, bl
01`0000B50E: 880574950100   mov    [00000001, 00024A88], al
01`0000B514: 7645            jbe   .00000001, 0000B55B --B
01`0000B516: 488BFB          mov    rdi, rbx
01`0000B519: 498BD4          mov    rdx, r12
01`0000B51C: 8BCD            mov    ecx, ebp

```

Figure 8.2: Hiew: find the place to be patched

Let's replace the **M0VZX** instruction with ours. Let's pretend we've got 64 CPU cores.

Add one additional **NOP** (because our instruction is shorter than the original one):

```

00`0000A8F8: 40386C2458      cmp    [rsp][058], bp
00`0000A8FD: 48890544A00100  mov    [000024948], rax
00`0000A904: 66B84000         mov    ax, 00040 ; @'
00`0000A908: 90              nop
00`0000A909: 0F47C5          cmova eax, ebp
00`0000A90C: 3AC3            cmp    al, bl
00`0000A90E: 880574950100   mov    [000023E88], al
00`0000A914: 7645            jbe   00000A95B
00`0000A916: 488BFB          mov    rdi, rbx
00`0000A919: 498BD4          mov    rdx, r12
00`0000A91C: 8BCD            mov    ecx, ebp

```

Figure 8.3: Hiew: patch it

And it works! Of course, the data in the graphs is not correct.

At times, Task Manager even shows an overall CPU load of more than 100%.

## 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

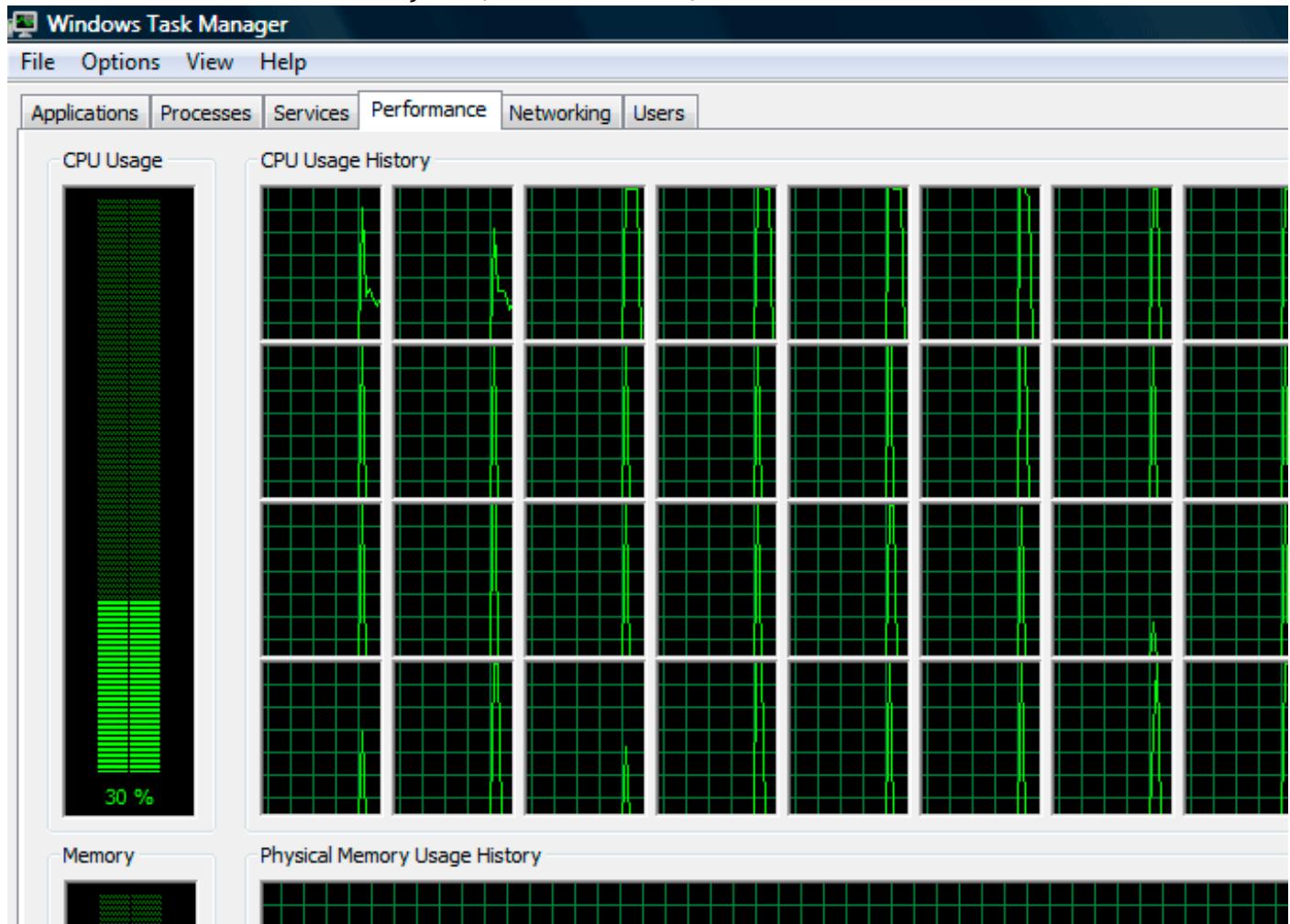


Figure 8.4: Fooled Windows Task Manager

The biggest number Task Manager does not crash with is 64.

Apparently, Task Manager in Windows Vista was not tested on computers with a large number of cores. So there are probably some static data structure(s) inside it limited to 64 cores.

### 8.1.1 Using LEA to load values

Sometimes, `LEA` is used in `taskmgr.exe` instead of `MOV` to set the first argument of `NtQuerySystemInformation()`:

Listing 8.2: taskmgr.exe (Windows Vista)

```
xor    r9d, r9d
div    dword ptr [rsp+4C8h+WndClass.lpfnWndProc]
lea    rdx, [rsp+4C8h+VersionInformation]
lea    ecx, [r9+2]      ; put 2 to ECX
mov    r8d, 138h
mov    ebx, eax
; ECX=SystemPerformanceInformation
call   cs:_imp_NtQuerySystemInformation ; 2
...
mov    r8d, 30h
lea    r9, [rsp+298h+var_268]
lea    rdx, [rsp+298h+var_258]
lea    ecx, [r8-2Dh]    ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
```

### 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

```
call    cs:_imp_NtQuerySystemInformation ; not zero
...
mov    rbp, [rsi+8]
mov    r8d, 20h
lea    r9, [rsp+98h+arg_0]
lea    rdx, [rsp+98h+var_78]
lea    ecx, [r8+2Fh] ; put 0x4F to ECX
mov    [rsp+98h+var_60], ebx
mov    [rsp+98h+var_68], rbp
; ECX=SystemSuperfetchInformation
call    cs:_imp_NtQuerySystemInformation ; not zero
```

Perhaps MSVC did so because machine code of LEA is shorter than MOV REG, 5 (would be 5 instead of 4).

LEA with offset in -128..127 range (offset will occupy 1 byte in opcode) with 32-bit registers is even shorter (for lack of REX prefix)—3 bytes.

Another example of such thing is: [6.1.5 on page 738](#).

## 8.2 Color Lines game practical joke

This is a very popular game with several implementations in existence. We can take one of them, called BallTriX, from 1997, available freely at <http://go.yurichev.com/17311><sup>2</sup>. Here is how it looks:

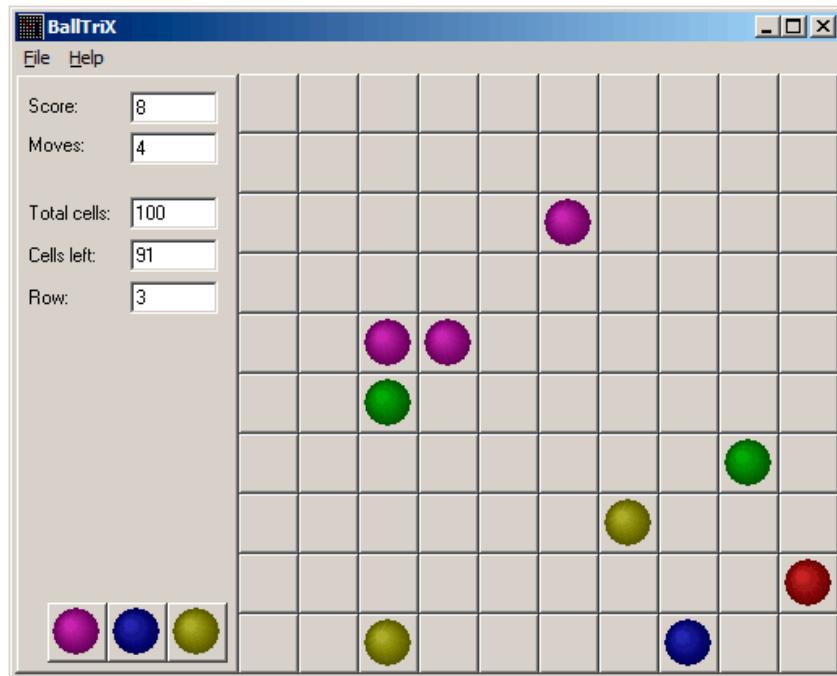


Figure 8.5: This is how the game is usually looks like

<sup>2</sup>Or at <http://go.yurichev.com/17365> or <http://go.yurichev.com/17366>.

## 8.2. COLOR LINES GAME PRACTICAL JOKE

So let's see, is it be possible to find the random generator and do some trick with it. [IDA](#) quickly recognize the standard `_rand` function in `balltrix.exe` at `0x00403DA0`. [IDA](#) also shows that it is called only from one place:

```
.text:00402C9C sub_402C9C    proc near                ; CODE XREF: sub_402ACA+52
.text:00402C9C
.text:00402C9C
.text:00402C9C arg_0        = dword ptr  8
.text:00402C9C
.text:00402C9C             push    ebp
.text:00402C9D             mov     ebp, esp
.text:00402C9F             push    ebx
.text:00402CA0             push    esi
.text:00402CA1             push    edi
.text:00402CA2             mov     eax, dword_40D430
.text:00402CA7             imul   eax, dword_40D440
.text:00402CAE             add    eax, dword_40D5C8
.text:00402CB4             mov     ecx, 32000
.text:00402CB9             cdq
.text:00402CBA             idiv   ecx
.text:00402CBC             mov     dword_40D440, edx
.text:00402CC2             call   _rand
.text:00402CC7             cdq
.text:00402CC8             idiv   [ebp+arg_0]
.text:00402CCB             mov     dword_40D430, edx
.text:00402CD1             mov     eax, dword_40D430
.text:00402CD6             jmp    $+5
.text:00402CDB             pop    edi
.text:00402CDC             pop    esi
.text:00402CDD             pop    ebx
.text:00402CDE             leave
.text:00402CDF             retn
.text:00402CDF sub_402C9C    endp
```

We'll call it "random". Let's not to dive into this function's code yet.

This function is referred from 3 places.

Here are the first two:

```
.text:00402B16             mov     eax, dword_40C03C ; 10 here
.text:00402B1B             push   eax
.text:00402B1C             call   random
.text:00402B21             add    esp, 4
.text:00402B24             inc    eax
.text:00402B25             mov    [ebp+var_C], eax
.text:00402B28             mov    eax, dword_40C040 ; 10 here
.text:00402B2D             push   eax
.text:00402B2E             call   random
.text:00402B33             add    esp, 4
```

Here is the third one:

```
.text:00402BBB             mov     eax, dword_40C058 ; 5 here
.text:00402BC0             push   eax
.text:00402BC1             call   random
.text:00402BC6             add    esp, 4
.text:00402BC9             inc    eax
```

So the function has only one argument.

10 is passed in first two cases and 5 in third. We can also notice that the board has a size of  $10 \times 10$  and there are 5 possible colors. This is it! The standard `rand()` function returns a number in the `0..0x7FFF` range and this is often inconvenient, so many programmers implement their own random functions which returns a random number in a specified range. In our case, the range is  $0..n - 1$  and  $n$  is passed as the sole argument of the function. We can quickly check this in any debugger.

So let's fix the third function call to always return zero. First, we will replace three instructions (`PUSH/CALL/ADD`) by `NOPs`. Then we'll add `XOR EAX, EAX` instruction, to clear the `EAX` register.

## 8.2. COLOR LINES GAME PRACTICAL JOKE

```
.00402BB8: 83C410      add    esp,010
.00402BBB: A158C04000  mov    eax,[00040C058]
.00402BC0: 31C0        xor    eax,eax
.00402BC2: 90          nop
.00402BC3: 90          nop
.00402BC4: 90          nop
.00402BC5: 90          nop
.00402BC6: 90          nop
.00402BC7: 90          nop
.00402BC8: 90          nop
.00402BC9: 40          inc    eax
.00402BCA: 8B4DF8      mov    ecx,[ebp][-8]
.00402BCD: 8D0C49      lea    ecx,[ecx][ecx]*2
.00402BD0: 8B15F4D54000 mov    edx,[00040D5F4]
```

So what we did is we replaced a call to the `random()` function by a code which always returns zero.

### 8.3. MINESWEEPER (WINDOWS XP)

Let's run it now:

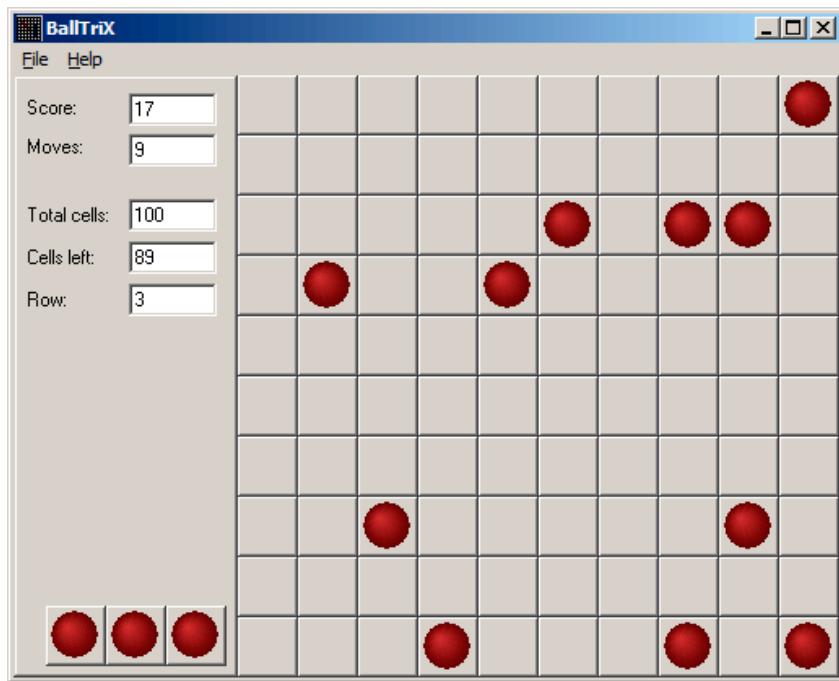


Figure 8.6: Practical joke works

Oh yes, it works<sup>3</sup>.

But why are the arguments to the `random()` functions global variables? That's just because it's possible to change the board size in the game's settings, so these values are not hardcoded. The 10 and 5 values are just defaults.

## 8.3 Minesweeper (Windows XP)

For those who are not very good at playing Minesweeper, we could try to reveal the hidden mines in the debugger.

As we know, Minesweeper places mines randomly, so there has to be some kind of random number generator or a call to the standard `rand()` C-function.

What is really cool about reversing Microsoft products is that there are [PDB](#) file with symbols (function names, etc). When we load `winmine.exe` into [IDA](#), it downloads the [PDB](#) file exactly for this executable and shows all names.

So here it is, the only call to `rand()` is this function:

```
.text:01003940 ; __stdcall Rnd(x)
.text:01003940 _Rnd@4          proc near             ; CODE XREF: StartGame()+53
.text:01003940                                         ; StartGame()+61
.text:01003940
.text:01003940 arg_0           = dword ptr  4
.text:01003940
.text:01003940                                         call    ds:_imp__rand
.text:01003946                                         cdq
.text:01003947                                         idiv   [esp+arg_0]
.text:0100394B                                         mov    eax, edx
.text:0100394D                                         retn   4
.text:0100394D _Rnd@4          endp
```

[IDA](#) named it so, and it was the name given to it by Minesweeper's developers.

The function is very simple:

<sup>3</sup>Author of this book once did this as a joke for his coworkers with the hope that they would stop playing. They didn't.

### 8.3. MINESWEEPER (WINDOWS XP)

```
int Rnd(int limit)
{
    return rand() % limit;
};
```

(There is no “limit” name in the PDB file; we manually named this argument like this.)

So it returns a random value from 0 to a specified limit.

Rnd() is called only from one place, a function called StartGame(), and as it seems, this is exactly the code which place the mines:

```
.text:010036C7          push    _xBoxMac
.text:010036CD          call    _Rnd@4           ; Rnd(x)
.text:010036D2          push    _yBoxMac
.text:010036D8          mov     esi, eax
.text:010036DA          inc    esi
.text:010036DB          call    _Rnd@4           ; Rnd(x)
.text:010036E0          inc    eax
.text:010036E1          mov     ecx, eax
.text:010036E3          shl    ecx, 5            ; ECX=ECX*32
.text:010036E6          test   _rgBlk[ecx+esi], 80h
.text:010036EE          jnz    short loc_10036C7
.text:010036F0          shl    eax, 5            ; EAX=EAX*32
.text:010036F3          lea    eax, _rgBlk[eax+esi]
.text:010036FA          or     byte ptr [eax], 80h
.text:010036FD          dec    _cBombStart
.text:01003703          jnz    short loc_10036C7
```

Minesweeper allows you to set the board size, so the X (xBoxMac) and Y (yBoxMac) of the board are global variables. They are passed to Rnd() and random coordinates are generated. A mine is placed by the OR instruction at 0x010036FA. And if it has been placed before (it's possible if the pair of Rnd() generates a coordinates pair which has been already generated), then TEST and JNZ at 0x010036E6 jumps to the generation routine again.

cBombStart is the global variable containing total number of mines. So this is loop.

The width of the array is 32 (we can conclude this by looking at the SHL instruction, which multiplies one of the coordinates by 32).

The size of the rgBlk global array can be easily determined by the difference between the rgBlk label in the data segment and the next known one. It is 0x360 (864):

```
.data:01005340 _rgBlk          db 360h dup(?)           ; DATA XREF: MainWndProc(x,x,x,x)+574
.data:01005340                 ; DisplayBlk(x,x)+23
.data:010056A0 _Preferences     dd ?                   ; DATA XREF: FixMenus()+
...
```

864/32 = 27.

So the array size is  $27 * 32$ ? It is close to what we know: when we try to set board size to  $100 * 100$  in Minesweeper settings, it fallbacks to a board of size  $24 * 30$ . So this is the maximal board size here. And the array has a fixed size for any board size.

So let's see all this in OllyDbg. We will ran Minesweeper, attaching OllyDbg to it and now we can see the memory dump at the address of the rgBlk array (0x01005340)<sup>4</sup>.

So we got this memory dump of the array:

Address	Hex dump
01005340	10 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350	0F
01005360	10 0F 10 0F 0F 0F 0F 0F 0F 0F
01005370	0F
01005380	10 0F 10 0F 0F 0F 0F 0F 0F 0F
01005390	0F
010053A0	10 0F 0F 0F 0F 0F 0F 0F 8F 0F 10 0F 0F 0F 0F 0F 0F 0F

<sup>4</sup>All addresses here are for Minesweeper for Windows XP SP3 English. They may differ for other service packs.

### 8.3. MINESWEEPER (WINDOWS XP)

```

010053B0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053C0 10 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|10 0F 0F 0F 0F 0F 0F 0F|
010053D0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010053E0 10 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|10 0F 0F 0F 0F 0F 0F 0F|
010053F0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005400 10 0F 0F 8F|0F 0F 8F 0F|0F 0F 0F 10 0F 0F 0F 0F 0F 0F 0F 0F|
01005410 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005420 10 8F 0F 0F|8F 0F 0F 0F|0F 0F 0F 10 0F 0F 0F 0F 0F 0F 0F 0F|
01005430 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005440 10 8F 0F 0F|0F 0F 0F 0F|8F 0F 0F 8F 10 0F 0F 0F 0F 0F 0F 0F|
01005450 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005460 10 0F 0F 0F|0F 0F 0F 0F|8F 0F 0F 8F 10 0F 0F 0F 0F 0F 0F 0F|
01005470 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005480 10 10 10 10|10 10 10 10|10 10 10 10 10 0F 0F 0F 0F 0F 0F 0F|
01005490 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010054A0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010054B0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010054C0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|

```

OllyDbg, like any other hexadecimal editor, shows 16 bytes per line. So each 32-byte array row occupies exactly 2 lines here.

This is beginner level (9\*9 board).

There is some square structure can be seen visually (0x10 bytes).

We will click “Run” in OllyDbg to unfreeze the Minesweeper process, then we’ll click randomly at the Minesweeper window and trapped into mine, but now all mines are visible:

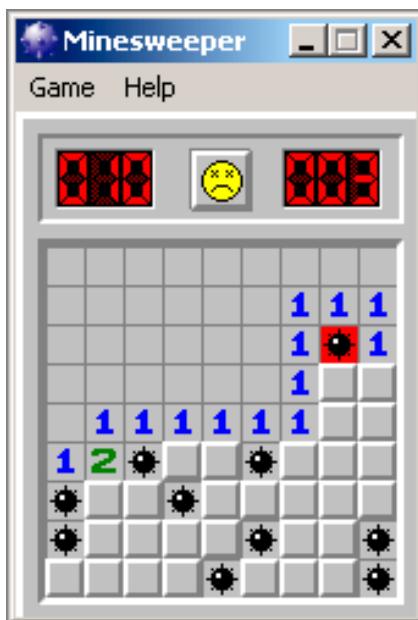


Figure 8.7: Mines

By comparing the mine places and the dump, we can conclude that 0x10 stands for border, 0x0F—empty block, 0x8F—mine.

Now we’ll add comments and also enclose all 0x8F bytes into square brackets:

```

border:
01005340 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350 0F 0F
line #1:
01005360 10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F 0F
01005370 0F 0F
line #2:
01005380 10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F 0F
01005390 0F 0F
line #3:
010053A0 10 0F 0F 0F 0F 0F 0F [8F]0F 10 0F 0F 0F 0F 0F 0F
010053B0 0F 0F

```

### 8.3. MINESWEEPER (WINDOWS XP)

```
line #4:  
010053C0  10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F  
010053D0  0F  
line #5:  
010053E0  10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F  
010053F0  0F  
line #6:  
01005400  10 0F 0F[8F]0F 0F[8F]0F 0F 0F 10 0F 0F 0F 0F 0F  
01005410  0F  
line #7:  
01005420  10[8F]0F 0F[8F]0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F  
01005430  0F  
line #8:  
01005440  10[8F]0F 0F 0F 0F[8F]0F 0F[8F]10 0F 0F 0F 0F 0F  
01005450  0F  
line #9:  
01005460  10 0F 0F 0F 0F[8F]0F 0F 0F[8F]10 0F 0F 0F 0F 0F  
01005470  0F  
border:  
01005480  10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F  
01005490  0F 0F
```

Now we'll remove all *border bytes* (0x10) and what's beyond those:

```
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F 0F 0F 0F 0F[8F]0F  
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F[8F]0F 0F[8F]0F 0F 0F  
[8F]0F 0F[8F]0F 0F 0F 0F 0F  
[8F]0F 0F 0F 0F[8F]0F 0F[8F]  
0F 0F 0F 0F[8F]0F 0F 0F[8F]
```

Yes, these are mines, now it can be clearly seen and compared with the screenshot.

### 8.3. MINESWEEPER (WINDOWS XP)

What is interesting is that we can modify the array right in OllyDbg. We can remove all mines by changing all 0x8F bytes by 0x0F, and here is what we'll get in Minesweeper:

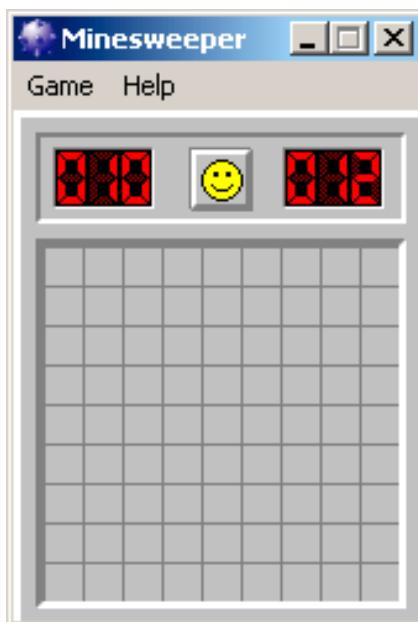


Figure 8.8: All mines are removed in debugger

We can also move all of them to the first line:

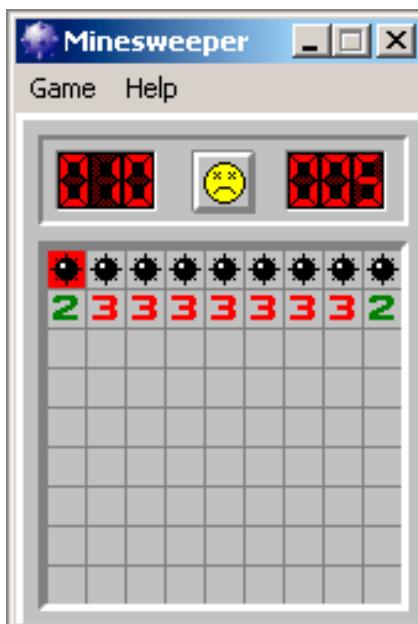


Figure 8.9: Mines set in debugger

Well, the debugger is not very convenient for eavesdropping (which is our goal anyway), so we'll write a small utility to dump the contents of the board:

```
// Windows XP MineSweeper cheater
// written by dennis(a)yurichev.com for http://beginners.re/ book
#include <windows.h>
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
    int i, j;
    HANDLE h;
    DWORD PID, address, rd;
    BYTE board[27][32];
```

### 8.3. MINESWEEPER (WINDOWS XP)

```
if (argc!=3)
{
    printf ("Usage: %s <PID> <address>\n", argv[0]);
    return 0;
};

assert (argv[1]!=NULL);
assert (argv[2]!=NULL);

assert (sscanf (argv[1], "%d", &PID)==1);
assert (sscanf (argv[2], "%x", &address)==1);

h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, FALSE, PID);

if (h==NULL)
{
    DWORD e=GetLastError();
    printf ("OpenProcess error: %08X\n", e);
    return 0;
};

if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &rd )!=TRUE)
{
    printf ("ReadProcessMemory() failed\n");
    return 0;
};

for (i=1; i<26; i++)
{
    if (board[i][0]==0x10 && board[i][1]==0x10)
        break; // end of board
    for (j=1; j<31; j++)
    {
        if (board[i][j]==0x10)
            break; // board border
        if (board[i][j]==0x8F)
            printf ("*");
        else
            printf (" ");
    };
    printf ("\n");
};

CloseHandle (h);
};
```

Just set the [PID<sup>5</sup>](#) <sup>6</sup> and the address of the array (`0x01005340` for Windows XP SP3 English) and it will dump it <sup>7</sup>.

It attaches itself to a win32 process by [PID](#) and just reads process memory at the address.

#### 8.3.1 Finding grid automatically

This is kind of nuisance to set address each time when we run our utility. Also, various Minesweeper versions may have the array on different address. Knowing the fact that there is always a border (0x10 bytes), we can just find it in memory:

```
// find frame to determine the address
process_mem=(BYTE*)malloc(process_mem_size);
assert (process_mem!=NULL);

if (ReadProcessMemory (h, (LPVOID)start_addr, process_mem, process_mem_size, &rd )!=TRUE)
    ↴
```

<sup>5</sup>Program/process ID

<sup>6</sup>PID it can be seen in Task Manager (enable it in "View → Select Columns")

<sup>7</sup>The compiled executable is here: [beginners.re](#)

## 8.4. HACKING WINDOWS CLOCK

```
{  
    printf ("ReadProcessMemory() failed\n");  
    return 0;  
};  
  
// for 9*9 grid.  
// FIXME: slow!  
for (i=0; i<process_mem_size; i++)  
{  
    if (memcmp(process_mem+i, "\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x10", 32) == 0)  
    {  
        // found  
        address=start_addr+i;  
        break;  
    };  
};  
if (address==0)  
{  
    printf ("Can't determine address of frame (and grid)\n");  
    return 0;  
}  
else  
{  
    printf ("Found frame and grid at 0x%x\n", address);  
};
```

Full source code: [https://github.com/dennis714/RE-for-beginners/blob/master/examples/minesweeper/minesweeper\\_cheater2.c](https://github.com/dennis714/RE-for-beginners/blob/master/examples/minesweeper/minesweeper_cheater2.c).

### 8.3.2 Exercises

- Why do the *border bytes* (0x10) exist in the array?  
What they are for if they are not visible in Minesweeper's interface? How could it work without them?
  - As it turns out, there are more values possible (for open blocks, for flagged by user, etc). Try to find the meaning of each one.
  - Modify my utility so it can remove all mines or set them in a fixed pattern that you want in the Minesweeper process currently running.

## 8.4 Hacking Windows clock

Sometimes I did some kind of first April prank for my coworkers.

Let's find, if we could do something with Windows clock? Can we force to go clock hands backwards?

First of all, when you click on date/time in status bar, a C:\WINDOWS\SYSTEM32\TIME DATE.CPL module gets executed, which is usual executable PE file.

Let's see, how it draw hands? When I open the file (from Windows 7) in Resource Hacker, there are clock faces, but with no hands:

## 8.4. HACKING WINDOWS CLOCK

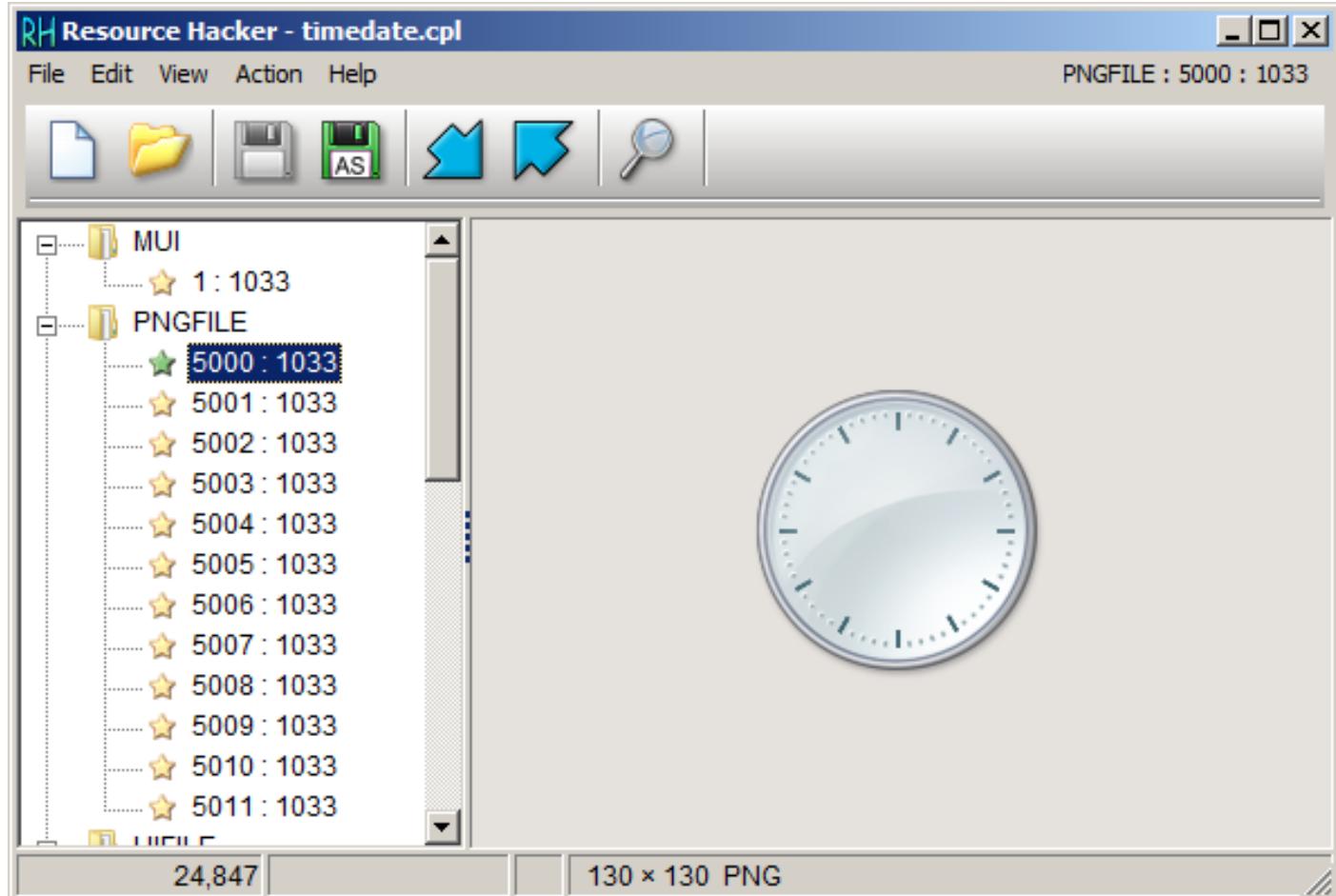


Figure 8.10: Resource Hacker

OK, what we know? How to draw a clock hand? All they are started at the middle of circle, ending with its border. Hence, we must calculate coordinates of a point on circle's border. From school-level mathematics we may recall that we have to use sine/cosine functions to draw circle, or at least square root. There are no such things in *TIMEDATE.CPL*, at least at first glance. But, thanks to Microsoft debugging PDB files, I can find a function named *CAnalogClock::DrawHand()*, which calls *Gdiplus::Graphics::DrawLine()* at least twice.

Here is its code:

```
.text:6EB9DBC7 ; private: enum Gdiplus::Status __thiscall CAnalogClock::_DrawHand(class ↴
    ↴ Gdiplus::Graphics *, int, struct ClockHand const &, class Gdiplus::Pen *)
.text:6EB9DBC7 ?_DrawHand@CAnalogClock@@QAE?↗
    ↴ AW4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@@Z proc near
.text:6EB9DBC7                                     ; CODE XREF: CAnalogClock::_ClockPaint(↗
    ↴ HDC__ *)+163                                ; CAnalogClock::_ClockPaint(HDC__ *)+18B
.text:6EB9DBC7
.text:6EB9DBC7
.text:6EB9DBC7 var_10      = dword ptr -10h
.text:6EB9DBC7 var_C       = dword ptr -0Ch
.text:6EB9DBC7 var_8       = dword ptr -8
.text:6EB9DBC7 var_4       = dword ptr -4
.text:6EB9DBC7 arg_0       = dword ptr 8
.text:6EB9DBC7 arg_4       = dword ptr 0Ch
.text:6EB9DBC7 arg_8       = dword ptr 10h
.text:6EB9DBC7 arg_C       = dword ptr 14h
.text:6EB9DBC7
.text:6EB9DBC7             mov     edi, edi
.text:6EB9DBC9             push    ebp
.text:6EB9DBCA             mov     ebp, esp
.text:6EB9DBC9             sub     esp, 10h
.text:6EB9DBC9             mov     eax, [ebp+arg_4]
.text:6EB9DBD2             push    ebx
.text:6EB9DBD3             push    esi
```

#### 8.4. HACKING WINDOWS CLOCK

```

.text:6EB9DBD4          push    edi
.text:6EB9DBD5          cdq
.text:6EB9DBD6          push    3Ch
.text:6EB9DBD8          mov     esi, ecx
.text:6EB9DBDA          pop    ecx
.text:6EB9DBDB          idiv   ecx
.text:6EB9DBDD          push    2
.text:6EB9DBDF          lea    ebx, table[edx*8]
.text:6EB9DBE6          lea    eax, [edx+1Eh]
.text:6EB9DBE9          cdq
.text:6EB9DBEA          idiv   ecx
.text:6EB9DBEC          mov     ecx, [ebp+arg_0]
.text:6EB9DBEF          mov     [ebp+var_4], ebx
.text:6EB9DBF2          lea    eax, table[edx*8]
.text:6EB9DBF9          mov     [ebp+arg_4], eax
.text:6EB9DBFC          call    ?SetInterpolationMode@Graphics@Gdiplus@@QAE?`  

    ↳ AW4Status@2@W4InterpolationMode@2@Z ; Gdiplus::Graphics::SetInterpolationMode(Gdiplus::`  

    ↳ InterpolationMode)
.text:6EB9DC01          mov     eax, [esi+70h]
.text:6EB9DC04          mov     edi, [ebp+arg_8]
.text:6EB9DC07          mov     [ebp+var_10], eax
.text:6EB9DC0A          mov     eax, [esi+74h]
.text:6EB9DC0D          mov     [ebp+var_C], eax
.text:6EB9DC10          mov     eax, [edi]
.text:6EB9DC12          sub    eax, [edi+8]
.text:6EB9DC15          push   8000      ; nDenominator
.text:6EB9DC1A          push   eax        ; nNumerator
.text:6EB9DC1B          push   dword ptr [ebx+4] ; nNumber
.text:6EB9DC1E          mov    ebx, ds:_imp_MulDiv@12 ; MulDiv(x,x,x)
.text:6EB9DC24          call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC26          add    eax, [esi+74h]
.text:6EB9DC29          push   8000      ; nDenominator
.text:6EB9DC2E          mov    [ebp+arg_8], eax
.text:6EB9DC31          mov    eax, [edi]
.text:6EB9DC33          sub    eax, [edi+8]
.text:6EB9DC36          push   eax        ; nNumerator
.text:6EB9DC37          mov    eax, [ebp+var_4]
.text:6EB9DC3A          push   dword ptr [eax] ; nNumber
.text:6EB9DC3C          call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC3E          add    eax, [esi+70h]
.text:6EB9DC41          mov    ecx, [ebp+arg_0]
.text:6EB9DC44          mov    [ebp+var_8], eax
.text:6EB9DC47          mov    eax, [ebp+arg_8]
.text:6EB9DC4A          mov    [ebp+var_4], eax
.text:6EB9DC4D          lea    eax, [ebp+var_8]
.text:6EB9DC50          push   eax
.text:6EB9DC51          lea    eax, [ebp+var_10]
.text:6EB9DC54          push   eax
.text:6EB9DC55          push   [ebp+arg_C]
.text:6EB9DC58          call    ?DrawLine@Graphics@Gdiplus@@QAE?`  

    ↳ AW4Status@2@PBVPen@2@ABVPoint@2@1@Z ; Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *,`  

    ↳ Gdiplus::Point const &,Gdiplus::Point const &)
.text:6EB9DC5D          mov    ecx, [edi+8]
.text:6EB9DC60          test   ecx, ecx
.text:6EB9DC62          jbe    short loc_6EB9DCAA
.text:6EB9DC64          test   eax, eax
.text:6EB9DC66          jnz    short loc_6EB9DCAA
.text:6EB9DC68          mov    eax, [ebp+arg_4]
.text:6EB9DC6B          push   8000      ; nDenominator
.text:6EB9DC70          push   ecx        ; nNumerator
.text:6EB9DC71          push   dword ptr [eax+4] ; nNumber
.text:6EB9DC74          call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC76          add    eax, [esi+74h]
.text:6EB9DC79          push   8000      ; nDenominator
.text:6EB9DC7E          push   dword ptr [edi+8] ; nNumerator
.text:6EB9DC81          mov    [ebp+arg_8], eax
.text:6EB9DC84          mov    eax, [ebp+arg_4]
.text:6EB9DC87          push   dword ptr [eax] ; nNumber
.text:6EB9DC89          call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC8B          add    eax, [esi+70h]

```

#### 8.4. HACKING WINDOWS CLOCK

```
.text:6EB9DC8E          mov     ecx, [ebp+arg_0]
.text:6EB9DC91          mov     [ebp+var_8], eax
.text:6EB9DC94          mov     eax, [ebp+arg_8]
.text:6EB9DC97          mov     [ebp+var_4], eax
.text:6EB9DC9A          lea     eax, [ebp+var_8]
.text:6EB9DC9D          push    eax
.text:6EB9DC9E          lea     eax, [ebp+var_10]
.text:6EB9DCA1          push    eax
.text:6EB9DCA2          push    [ebp+arg_C]
.text:6EB9DCA5          call    ?DrawLine@Graphics@Gdiplus@@QAE?_
                           ↳ AW4Status@2@PBVPen@2@ABVPoint@2@1@Z ; Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *, 
                           ↳ Gdiplus::Point const &,Gdiplus::Point const &)
.text:6EB9DCAA
.text:6EB9DCAA loc_6EB9DCAA:           ; CODE XREF: CAnalogClock::_DrawHand( 
                           ↳ Gdiplus::Graphics *,int,ClockHand const &,Gdiplus::Pen *)+9B
.text:6EB9DCAA          ; CAnalogClock::_DrawHand(Gdiplus:: 
                           ↳ Graphics *,int,ClockHand const &,Gdiplus::Pen *)+9F
.text:6EB9DCAA          pop     edi
.text:6EB9DCAB          pop     esi
.text:6EB9DCAC          pop     ebx
.text:6EB9DCAD          leave
.text:6EB9DCAE          retn    10h
.text:6EB9DCAE ?_DrawHand@CAnalogClock@@AAE?_
                           ↳ AW4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@@Z endp
.text:6EB9DCAE
```

We can see that *DrawLine()* arguments are dependent on result of *MulDiv()* function and a *table[]* table (name is mine), which has 8-byte elements (look at *LEA*'s second operand).

What is inside of *table[]*?

```
.text:6EB87890 ; int table[]
.text:6EB87890 table      dd 0
.text:6EB87894      dd 0FFFFE0C1h
.text:6EB87898      dd 344h
.text:6EB8789C      dd 0FFFFE0ECh
.text:6EB878A0      dd 67Fh
.text:6EB878A4      dd 0xFFFFE16Fh
.text:6EB878A8      dd 9A8h
.text:6EB878AC      dd 0xFFFFE248h
.text:6EB878B0      dd 0CB5h
.text:6EB878B4      dd 0xFFFFE374h
.text:6EB878B8      dd 0F9Fh
.text:6EB878BC      dd 0xFFFFE4F0h
.text:6EB878C0      dd 125Eh
.text:6EB878C4      dd 0xFFFFE6B8h
.text:6EB878C8      dd 14E9h
```

It's referenced only from *DrawHand()* function at has 120 32-bit words or 60 32-bit pairs... wait, 60? Let's take a closer look at these values. First of all, I'll zap 6 pairs or 12 32-bit words with zeros, and then I'll put patched *TIMEDATE.CPL* into *C:\WINDOWS\SYSTEM32*. (You may need to set owner of the *\*TIMEDATE.CPL\** file to your primary user account (instead of *TrustedInstaller*), and also, boot in safe mode with command prompt so you can copy the file, which is usually locked.)

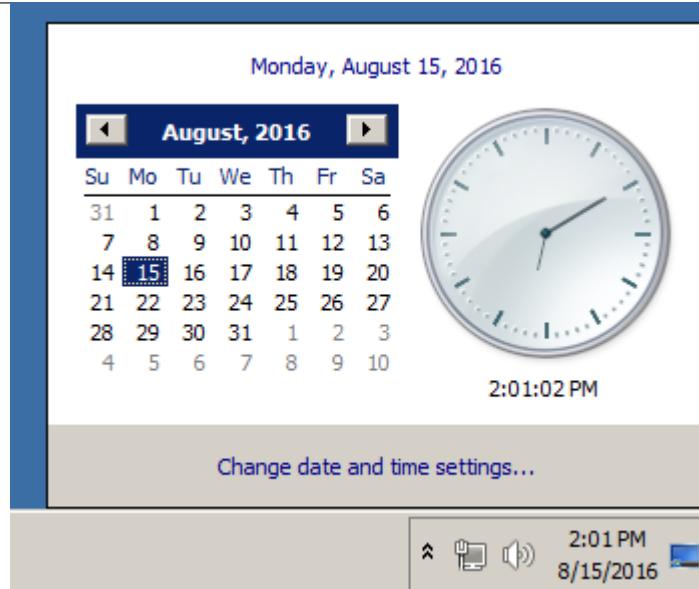


Figure 8.11: Attempt to run

Now when any hand is located at 0-5 seconds/minutes, it's invisible! However, opposite (shorter) part of second hand is visible and moving. When any hand is outside of this area, hand is visible as usual.

Let's take even closer look at the table in Mathematica. I have copy-pasted table from the *TIMEDATE.CPL* to a *tbl* file (480 bytes). We will take for granted the fact that these are signed values, because half of elements are below zero (0xFFFFE0C1h, etc.). If these values would be unsigned, they would be suspiciously huge.

```
In[]:= tbl = BinaryReadList["~/.../tbl", "Integer32"]

Out[]={0, -7999, 836, -7956, 1663, -7825, 2472, -7608, 3253, -7308, 3999, \
-6928, 4702, -6472, 5353, -5945, 5945, -5353, 6472, -4702, 6928, \
-4000, 7308, -3253, 7608, -2472, 7825, -1663, 7956, -836, 8000, 0, \
7956, 836, 7825, 1663, 7608, 2472, 7308, 3253, 6928, 4000, 6472, \
4702, 5945, 5353, 5353, 5945, 4702, 6472, 3999, 6928, 3253, 7308, \
2472, 7608, 1663, 7825, 836, 7956, 0, 7999, -836, 7956, -1663, 7825, \
-2472, 7608, -3253, 7308, -4000, 6928, -4702, 6472, -5353, 5945, \
-5945, 5353, -6472, 4702, -6928, 3999, -7308, 3253, -7608, 2472, \
-7825, 1663, -7956, 836, -7999, 0, -7956, -836, -7825, -1663, -7608, \
-2472, -7308, -3253, -6928, -4000, -6472, -4702, -5945, -5353, -5353, \
-5945, -4702, -6472, -3999, -6928, -3253, -7308, -2472, -7608, -1663, \
-7825, -836, -7956}

In[]:= Length(tbl]
Out[]= 120
```

Let's treat two consecutive 32-bit values as pair:

```
In[]:= pairs = Partition[tbl, 2]
Out[]={{0, -7999}, {836, -7956}, {1663, -7825}, {2472, -7608}, \
{3253, -7308}, {3999, -6928}, {4702, -6472}, {5353, -5945}, {5945, \
-5353}, {6472, -4702}, {6928, -4000}, {7308, -3253}, {7608, -2472}, \
{7825, -1663}, {7956, -836}, {8000, 0}, {7956, 836}, {7825, \
1663}, {7608, 2472}, {7308, 3253}, {6928, 4000}, {6472, \
4702}, {5945, 5353}, {5353, 5945}, {4702, 6472}, {3999, \
6928}, {3253, 7308}, {2472, 7608}, {1663, 7825}, {836, 7956}, {0, \
7999}, {-836, 7956}, {-1663, 7825}, {-2472, 7608}, {-3253, \
7308}, {-4000, 6928}, {-4702, 6472}, {-5353, 5945}, {-5945, \
5353}, {-6472, 4702}, {-6928, 3999}, {-7308, 3253}, {-7608, \
2472}, {-7825, 1663}, {-7956, 836}, {-7999, \
0}, {-7956, -836}, {-7825, -1663}, {-7608, -2472}, {-7308, -3253}, \
{-6928, -4000}, {-6472, -4702}, {-5945, -5353}, {-5353, -5945}, \
{-4702, -6472}, {-3999, -6928}, {-3253, -7308}, {-2472, -7608}, \
{-1663, -7825}, {-836, -7956}}
```

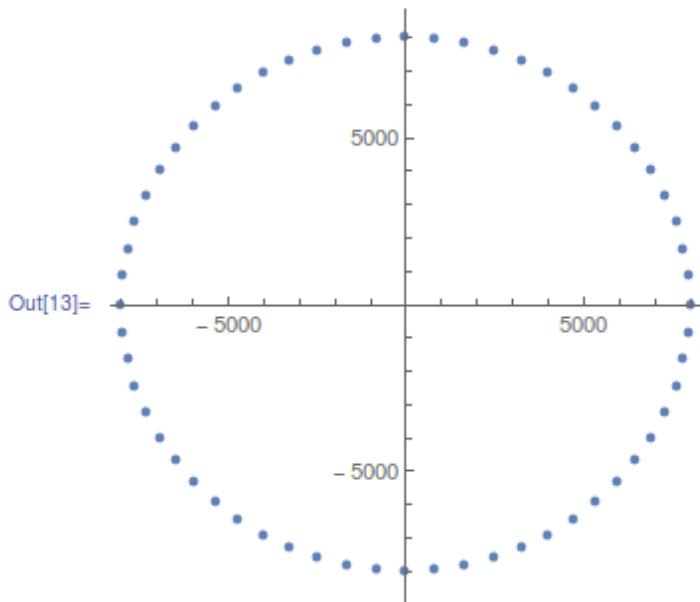
```
In[]:= Length[pairs]
```

#### 8.4. HACKING WINDOWS CLOCK

```
Out[] = 60
```

Let's try to treat each pair as X/Y coordinate and draw all 60 pairs, and also first 15 pairs:

```
In[13]:= ListPlot[pairs, AspectRatio -> Full, ImageSize -> {300, 300}]
```



```
In[27]:= ListPlot[pairs[[1 ;; 15]], AspectRatio -> Full, ImageSize -> {300, 300}]
```

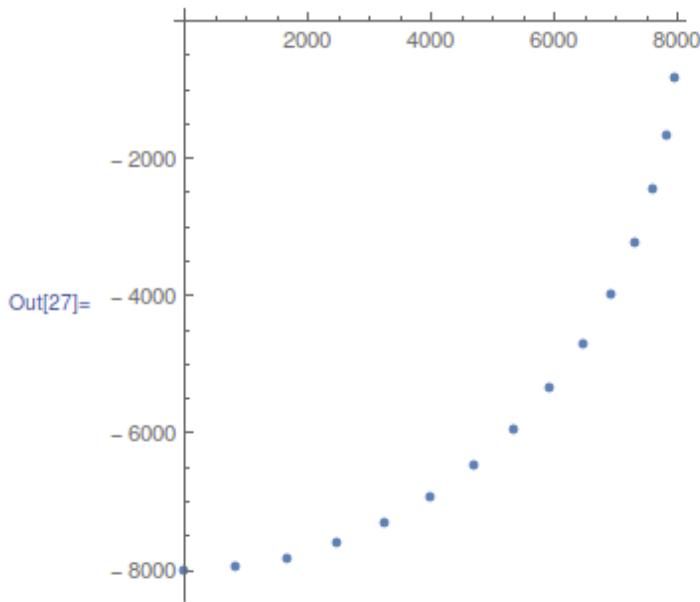


Figure 8.12: Mathematica

Now this is something! Each pair is just coordinate. First 15 pairs are coordinates for  $\frac{1}{4}$  of circle. Perhaps, Microsoft developers precalculated all coordinates and put them into table.

Now I can understand why when I zapped 6 pairs, hands were invisible at that area: in fact, hands were drawn, they just had zero length, because hand started at 0:0 coordinate and ended there.

#### The prank (practical joke)

Given all that, how would we force hands to go counterclockwise? In fact, this is simple, we need just to rotate the table, so each hand, instead of drawing at place of first second, would be drawing at place of 59th second.

#### 8.4. HACKING WINDOWS CLOCK

I made the patcher a long time ago, at the very beginning of 2000s, for Windows 2000. Hard to believe, it still works for Windows 7, perhaps, the table hasn't been changed since then!

Patcher source code: [https://github.com/dennis714/random\\_notes/blob/master/timedate/time\\_pt.c](https://github.com/dennis714/random_notes/blob/master/timedate/time_pt.c).

Now I can see all hands goes backwards:

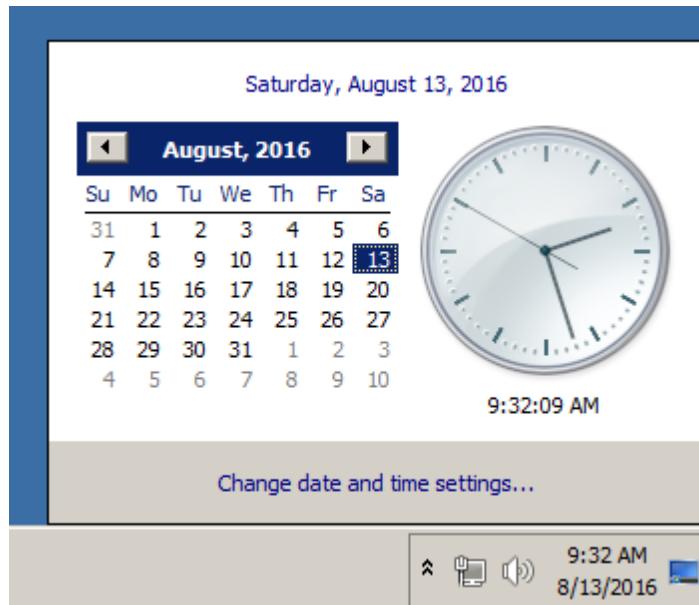


Figure 8.13: Now it works

Well, there is no animation in this article, but if you look closer, you can see, that hands are in fact shows correct time, but the whole clock face is rotated vertically, like we see it from the inside of clock.

#### Windows 2000 leaked source code

So I did the patcher and then Windows 2000 source code has been leaked (I can't force you to trust me, though). Let's take a look on source code if that function and table.

The file is *win2k/private/shell/cpls/utc/clock.c*:

```
//
// Array containing the sine and cosine values for hand positions.
//
POINT rCircleTable[] =
{
    { 0,      -7999},
    { 836,    -7956},
    { 1663,   -7825},
    { 2472,   -7608},
    { 3253,   -7308},
...
    { -4702,  -6472},
    { -3999,  -6928},
    { -3253,  -7308},
    { -2472,  -7608},
    { -1663,  -7825},
    { -836 ,  -7956},
};

///////////////////////////////
// DrawHand
// Draws the hands of the clock.
/////////////////////////////
```

## 8.5. DONGLES

```
void DrawHand(
    HDC hDC,
    int pos,
    HPEN hPen,
    int scale,
    int patMode,
    PCLKSTR np)
{
    LPOINT lppt;
    int radius;

    MoveTo(hDC, np->clockCenter.x, np->clockCenter.y);
    radius = MulDiv(np->clockRadius, scale, 100);
    lppt = rCircleTable + pos;
    SetROP2(hDC, patMode);
    SelectObject(hDC, hPen);

    LineTo( hDC,
            np->clockCenter.x + MulDiv(lppt->x, radius, 8000),
            np->clockCenter.y + MulDiv(lppt->y, radius, 8000) );
}
```

Now it's clear: coordinates has been precalculated as if clock face has height and width of  $2 \cdot 8000$ , and then it's rescaled to current clock face radius using *MulDiv()* function.

POINT structure<sup>8</sup> is a structure of two 32-bit values, first is *x*, second is *y*.

## 8.5 Dongles

The author of these lines, occasionally did software copy-protection **dongle** replacements, or “dongle emulators” and here are couple examples of how it’s happening.

About one of the cases about Rocket and Z3 that is not present here, you can read here: [http://yurichev.com/tmp/SAT\\_SMT\\_DRAFT.pdf](http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf).

### 8.5.1 Example #1: MacOS Classic and PowerPC

Here is an example of a program for MacOS Classic <sup>9</sup>, for PowerPC. The company who developed the software product has disappeared a long time ago, so the (legal) customer was afraid of physical dongle damage.

While running without a dongle connected, a message box with the text “Invalid Security Device” appeared.

Luckily, this text string could easily be found in the executable binary file.

Let’s pretend we are not very familiar both with Mac OS Classic and PowerPC, but will try anyway.

**IDA** opened the executable file smoothly, reported its type as “PEF (Mac OS or Be OS executable)” (indeed, it is a standard Mac OS Classic file format).

By searching for the text string with the error message, we’ve got into this code fragment:

```
...
seg000:000C87FC 38 60 00 01    li      %r3, 1
seg000:000C8800 48 03 93 41    bl      check1
seg000:000C8804 60 00 00 00    nop
seg000:000C8808 54 60 06 3F    clrlwi. %r0, %r3, 24
seg000:000C880C 40 82 00 40    bne    OK
seg000:000C8810 80 62 9F D8    lwz    %r3, TC_aInvalidSecurityDevice
...
```

<sup>8</sup>[https://msdn.microsoft.com/en-us/library/windows/desktop/dd162805\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd162805(v=vs.85).aspx)

<sup>9</sup>pre-UNIX MacOS

## 8.5. DONGLES

Yes, this is PowerPC code.

The CPU is a very typical 32-bit [RISC](#) of 1990s era.

Each instruction occupies 4 bytes (just as in MIPS and ARM) and the names somewhat resemble MIPS instruction names.

`check1()` is a function name we'll give to it later. `BL` is *Branch Link* instruction, e.g., intended for calling subroutines.

The crucial point is the `BNE` instruction which jumps if the dongle protection check passes or not if an error occurs: then the address of the text string gets loaded into the `r3` register for the subsequent passing into a message box routine.

From the [Steve Zucker, SunSoft and Kari Karhi, IBM, *SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement*, (1995)]<sup>10</sup> we will find out that the `r3` register is used for return values (and `r4`, in case of 64-bit values).

Another yet unknown instruction is `CLRLWI`. From [*PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, (2000)]<sup>11</sup> we'll learn that this instruction does both clearing and loading. In our case, it clears the 24 high bits from the value in `r3` and puts them in `r0`, so it is analogical to `MOVZX` in x86 ([1.17.1 on page 202](#)), but it also sets the flags, so `BNE` can check them afterwards.

Let's take a look into the `check1()` function:

```
seg000:00101B40          check1: # CODE XREF: seg000:00063E7Cp
seg000:00101B40          # sub_64070+160p ...
seg000:00101B40
seg000:00101B40          .set arg_8, 8
seg000:00101B40
seg000:00101B40 7C 08 02 A6      mflr    %r0
seg000:00101B44 90 01 00 08      stw     %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0      stwu   %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39      bl      check2
seg000:00101B50 60 00 00 00      nop
seg000:00101B54 80 01 00 48      lwz     %r0, 0x40+arg_8(%sp)
seg000:00101B58 38 21 00 40      addi   %sp, %sp, 0x40
seg000:00101B5C 7C 08 03 A6      mtlr   %r0
seg000:00101B60 4E 80 00 20      blr
seg000:00101B60          # End of function check1
```

As you can see in [IDA](#), that function is called from many places in the program, but only the `r3` register's value is checked after each call.

All this function does is to call the other function, so it is a [thunk function](#): there are function prologue and epilogue, but the `r3` register is not touched, so `check1()` returns what `check2()` returns.

`BLR`<sup>12</sup> looks like the return from the function, but since [IDA](#) does the function layout, we probably do not need to care about this.

Since it is a typical [RISC](#), it seems that subroutines are called using a [link register](#), just like in ARM.

The `check2()` function is more complex:

```
seg000:00118684          check2: # CODE XREF: check1+Cp
seg000:00118684
seg000:00118684          .set var_18, -0x18
seg000:00118684          .set var_C, -0xC
seg000:00118684          .set var_8, -8
seg000:00118684          .set var_4, -4
seg000:00118684          .set arg_8, 8
seg000:00118684
seg000:00118684 93 E1 FF FC  stw     %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6  mflr   %r0
seg000:0011868C 83 E2 95 A8  lwz     %r31, off_1485E8 # dword_24B704
seg000:00118690          .using dword_24B704, %r31
seg000:00118690 93 C1 FF F8  stw     %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4  stw     %r29, var_C(%sp)
```

<sup>10</sup>Also available as [http://yurichev.com/mirrors/PowerPC/elfspec\\_ppc.pdf](http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf)

<sup>11</sup>Also available as [http://yurichev.com/mirrors/PowerPC/6xx\\_pem.pdf](http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf)

<sup>12</sup>(PowerPC) Branch to Link Register

## 8.5. DONGLES

```

seg000:00118698 7C 7D 1B 78    mr      %r29, %r3
seg000:0011869C 90 01 00 08    stw     %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E    clrlwi  %r0, %r3, 24
seg000:001186A4 28 00 00 01    cmplwi  %r0, 1
seg000:001186A8 94 21 FF B0    stwu   %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C    bne    loc_1186B8
seg000:001186B0 38 60 00 01    li     %r3, 1
seg000:001186B4 48 00 00 6C    b      exit
seg000:001186B8
seg000:001186B8      loc_1186B8: # CODE XREF: check2+28j
seg000:001186B8 48 00 03 D5    bl     sub_118A8C
seg000:001186BC 60 00 00 00    nop
seg000:001186C0 3B C0 00 00    li     %r30, 0
seg000:001186C4
seg000:001186C4      skip:    # CODE XREF: check2+94j
seg000:001186C4 57 C0 06 3F    clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18    beq    loc_1186E0
seg000:001186CC 38 61 00 38    addi   %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00    lwz    %r4, dword_24B704
seg000:001186D4 48 00 C0 55    bl     .RBEFINDNEXT
seg000:001186D8 60 00 00 00    nop
seg000:001186DC 48 00 00 1C    b      loc_1186F8
seg000:001186E0
seg000:001186E0      loc_1186E0: # CODE XREF: check2+44j
seg000:001186E0 80 BF 00 00    lwz    %r5, dword_24B704
seg000:001186E4 38 81 00 38    addi   %r4, %sp, 0x50+var_18
seg000:001186E8 38 60 08 C2    li     %r3, 0x1234
seg000:001186EC 48 00 BF 99    bl     .RBEFINDFIRST
seg000:001186F0 60 00 00 00    nop
seg000:001186F4 3B C0 00 01    li     %r30, 1
seg000:001186F8
seg000:001186F8      loc_1186F8: # CODE XREF: check2+58j
seg000:001186F8 54 60 04 3F    clrlwi. %r0, %r3, 16
seg000:001186FC 41 82 00 0C    beq    must_jump
seg000:00118700 38 60 00 00    li     %r3, 0          # error
seg000:00118704 48 00 00 1C    b      exit
seg000:00118708
seg000:00118708      must_jump: # CODE XREF: check2+78j
seg000:00118708 7F A3 EB 78    mr      %r3, %r29
seg000:0011870C 48 00 00 31    bl     check3
seg000:00118710 60 00 00 00    nop
seg000:00118714 54 60 06 3F    clrlwi. %r0, %r3, 24
seg000:00118718 41 82 FF AC    beq    skip
seg000:0011871C 38 60 00 01    li     %r3, 1
seg000:00118720
seg000:00118720      exit:   # CODE XREF: check2+30j
seg000:00118720          # check2+80j
seg000:00118720 80 01 00 58    lwz    %r0, 0x50+arg_8(%sp)
seg000:00118724 38 21 00 50    addi   %sp, %sp, 0x50
seg000:00118728 83 E1 FF FC    lwz    %r31, var_4(%sp)
seg000:0011872C 7C 08 03 A6    mtlr   %r0
seg000:00118730 83 C1 FF F8    lwz    %r30, var_8(%sp)
seg000:00118734 83 A1 FF F4    lwz    %r29, var_C(%sp)
seg000:00118738 4E 80 00 20    blr
seg000:00118738          # End of function check2

```

We are lucky again: some function names are left in the executable (debug symbols section)? Hard to say while we are not very familiar with the file format, maybe it is some kind of PE exports? ([6.5.2 on page 759](#)),

like `.RBEFINDNEXT()` and `.RBEFINDFIRST()`.

Eventually these functions call other functions with names like `.GetNextDeviceViaUSB()`, `.USBSendPKT()`, so these are clearly dealing with an USB device.

There is even a function named `.GetNextEve3Device()`—sounds familiar, there was a Sentinel Eve3 dongle for ADB port (present on Macs) in 1990s.

Let's first take a look on how the r3 register is set before return, while ignoring everything else.

We know that a “good” r3 value has to be non-zero, zero r3 leads the execution flow to the message box

## 8.5. DONGLES

with an error message.

There are two `li %r3, 1` instructions present in the function and one `li %r3, 0` (*Load Immediate*, i.e., loading a value into a register). The first instruction is at `0x001186B0` —and frankly speaking, it's hard to say what it means.

What we see next is, however, easier to understand: `.RBEFINDFIRST()` is called: if it fails, 0 is written into `r3` and we jump to `exit`, otherwise another function is called (`check3()`)—if it fails too, `.RBEFINDNEXT()` is called, probably in order to look for another USB device.

N.B.: `clrlwi. %r0, %r3, 16` it is analogical to what we already saw, but it clears 16 bits, i.e., `.RBEFINDFIRST()` probably returns a 16-bit value.

`B` (stands for *branch*) unconditional jump.

`BEQ` is the inverse instruction of `BNE`.

Let's see `check3()`:

```
seg000:0011873C          check3: # CODE XREF: check2+88p
seg000:0011873C
seg000:0011873C          .set var_18, -0x18
seg000:0011873C          .set var_C, -0xC
seg000:0011873C          .set var_8, -8
seg000:0011873C          .set var_4, -4
seg000:0011873C          .set arg_8, 8
seg000:0011873C
seg000:0011873C 93 E1 FF FC  stw    %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6  mflr   %r0
seg000:00118744 38 A0 00 00  li     %r5, 0
seg000:00118748 93 C1 FF F8  stw    %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8  lwz    %r30, off_1485E8 # dword_24B704
seg000:00118750
seg000:00118750 93 A1 FF F4  stw    %r29, var_C(%sp)
seg000:00118754 3B A3 00 00  addi   %r29, %r3, 0
seg000:00118758 38 60 00 00  li     %r3, 0
seg000:0011875C 90 01 00 08  stw    %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0  stwu   %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00  lwz    %r6, dword_24B704
seg000:00118768 38 81 00 38  addi   %r4, %sp, 0x50+var_18
seg000:0011876C 48 00 C0 5D  bl     .RBEREAD
seg000:00118770 60 00 00 00  nop
seg000:00118774 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C  beq    loc_118784
seg000:0011877C 38 60 00 00  li     %r3, 0
seg000:00118780 48 00 02 F0  b      exit
seg000:00118784
seg000:00118784          loc_118784: # CODE XREF: check3+3Cj
seg000:00118784 A0 01 00 38  lhz    %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2  cmplwi %r0, 0x1100
seg000:0011878C 41 82 00 0C  beq    loc_118798
seg000:00118790 38 60 00 00  li     %r3, 0
seg000:00118794 48 00 02 DC  b      exit
seg000:00118798
seg000:00118798          loc_118798: # CODE XREF: check3+50j
seg000:00118798 80 DE 00 00  lwz    %r6, dword_24B704
seg000:0011879C 38 81 00 38  addi   %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01  li     %r3, 1
seg000:001187A4 38 A0 00 00  li     %r5, 0
seg000:001187A8 48 00 C0 21  bl     .RBEREAD
seg000:001187AC 60 00 00 00  nop
seg000:001187B0 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C  beq    loc_1187C0
seg000:001187B8 38 60 00 00  li     %r3, 0
seg000:001187BC 48 00 02 B4  b      exit
seg000:001187C0
seg000:001187C0          loc_1187C0: # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38  lhz    %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B  cmplwi %r0, 0x09AB
seg000:001187C8 41 82 00 0C  beq    loc_1187D4
```

## 8.5. DONGLES

```

seg000:001187CC 38 60 00 00 li      %r3, 0
seg000:001187D0 48 00 02 A0 b       exit
seg000:001187D4 loc_1187D4: # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9 bl      sub_B7BAC
seg000:001187D8 60 00 00 00 nop
seg000:001187DC 54 60 06 3E clrlwi %r0, %r3, 24
seg000:001187E0 2C 00 00 05 cmpwi   %r0, 5
seg000:001187E4 41 82 01 00 beq     loc_1188E4
seg000:001187E8 40 80 00 10 bge     loc_1187F8
seg000:001187EC 2C 00 00 04 cmpwi   %r0, 4
seg000:001187F0 40 80 00 58 bge     loc_118848
seg000:001187F4 48 00 01 8C b       loc_118980
seg000:001187F8 loc_1187F8: # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 0B cmpwi   %r0, 0xB
seg000:001187FC 41 82 00 08 beq     loc_118804
seg000:00118800 48 00 01 80 b       loc_118980
seg000:00118804
seg000:00118804 loc_118804: # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00 lwz     %r6, dword_24B704
seg000:00118808 38 81 00 38 addi    %r4, %sp, 0x50+var_18
seg000:0011880C 38 60 00 08 li      %r3, 8
seg000:00118810 38 A0 00 00 li      %r5, 0
seg000:00118814 48 00 BF B5 bl     .RBEREAD
seg000:00118818 60 00 00 00 nop
seg000:0011881C 54 60 04 3F clrlwi %r0, %r3, 16
seg000:00118820 41 82 00 0C beq     loc_11882C
seg000:00118824 38 60 00 00 li      %r3, 0
seg000:00118828 48 00 02 48 b       exit
seg000:0011882C loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38 lhz     %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30 cmplwi %r0, 0xFE0
seg000:00118834 41 82 00 0C beq     loc_118840
seg000:00118838 38 60 00 00 li      %r3, 0
seg000:0011883C 48 00 02 34 b       exit
seg000:00118840
seg000:00118840 loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01 li      %r3, 1
seg000:00118844 48 00 02 2C b       exit
seg000:00118848
seg000:00118848 loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00 lwz     %r6, dword_24B704
seg000:0011884C 38 81 00 38 addi    %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A li      %r3, 0xA
seg000:00118854 38 A0 00 00 li      %r5, 0
seg000:00118858 48 00 BF 71 bl     .RBEREAD
seg000:0011885C 60 00 00 00 nop
seg000:00118860 54 60 04 3F clrlwi %r0, %r3, 16
seg000:00118864 41 82 00 0C beq     loc_118870
seg000:00118868 38 60 00 00 li      %r3, 0
seg000:0011886C 48 00 02 04 b       exit
seg000:00118870 loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38 lhz     %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3 cmplwi %r0, 0xA6E1
seg000:00118878 41 82 00 0C beq     loc_118884
seg000:0011887C 38 60 00 00 li      %r3, 0
seg000:00118880 48 00 01 F0 b       exit
seg000:00118884
seg000:00118884 loc_118884: # CODE XREF: check3+13Cj
seg000:00118884 57 BF 06 3E clrlwi %r31, %r29, 24
seg000:00118888 28 1F 00 02 cmplwi %r31, 2
seg000:0011888C 40 82 00 0C bne    loc_118898
seg000:00118890 38 60 00 01 li      %r3, 1
seg000:00118894 48 00 01 DC b       exit
seg000:00118898
seg000:00118898 loc_118898: # CODE XREF: check3+150j
seg000:00118898 80 DE 00 00 lwz     %r6, dword_24B704

```

## 8.5. DONGLES

```

seg000:0011889C 38 81 00 38 addi    %r4, %sp, 0x50+var_18
seg000:001188A0 38 60 00 0B li      %r3, 0xB
seg000:001188A4 38 A0 00 00 li      %r5, 0
seg000:001188A8 48 00 BF 21 bl      .RBREAD
seg000:001188AC 60 00 00 00 nop
seg000:001188B0 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:001188B4 41 82 00 0C beq    loc_1188C0
seg000:001188B8 38 60 00 00 li      %r3, 0
seg000:001188BC 48 00 01 B4 b       exit
seg000:001188C0
seg000:001188C0          loc_1188C0: # CODE XREF: check3+178j
seg000:001188C0 A0 01 00 38 lhz    %r0, 0x50+var_18(%sp)
seg000:001188C4 28 00 23 1C cmplwi %r0, 0x1C20
seg000:001188C8 41 82 00 0C beq    loc_1188D4
seg000:001188CC 38 60 00 00 li      %r3, 0
seg000:001188D0 48 00 01 A0 b       exit
seg000:001188D4
seg000:001188D4          loc_1188D4: # CODE XREF: check3+18Cj
seg000:001188D4 28 1F 00 03 cmplwi %r31, 3
seg000:001188D8 40 82 01 94 bne    error
seg000:001188DC 38 60 00 01 li      %r3, 1
seg000:001188E0 48 00 01 90 b       exit
seg000:001188E4
seg000:001188E4          loc_1188E4: # CODE XREF: check3+A8j
seg000:001188E4 80 DE 00 00 lwz    %r6, dword_24B704
seg000:001188E8 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:001188EC 38 60 00 0C li      %r3, 0xC
seg000:001188F0 38 A0 00 00 li      %r5, 0
seg000:001188F4 48 00 BE D5 bl      .RBREAD
seg000:001188F8 60 00 00 00 nop
seg000:001188FC 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118900 41 82 00 0C beq    loc_11890C
seg000:00118904 38 60 00 00 li      %r3, 0
seg000:00118908 48 00 01 68 b       exit
seg000:0011890C
seg000:0011890C          loc_11890C: # CODE XREF: check3+1C4j
seg000:0011890C A0 01 00 38 lhz    %r0, 0x50+var_18(%sp)
seg000:00118910 28 00 1F 40 cmplwi %r0, 0x40FF
seg000:00118914 41 82 00 0C beq    loc_118920
seg000:00118918 38 60 00 00 li      %r3, 0
seg000:0011891C 48 00 01 54 b       exit
seg000:00118920
seg000:00118920          loc_118920: # CODE XREF: check3+1D8j
seg000:00118920 57 BF 06 3E clrlwi %r31, %r29, 24
seg000:00118924 28 1F 00 02 cmplwi %r31, 2
seg000:00118928 40 82 00 0C bne    loc_118934
seg000:0011892C 38 60 00 01 li      %r3, 1
seg000:00118930 48 00 01 40 b       exit
seg000:00118934
seg000:00118934          loc_118934: # CODE XREF: check3+1ECj
seg000:00118934 80 DE 00 00 lwz    %r6, dword_24B704
seg000:00118938 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:0011893C 38 60 00 0D li      %r3, 0xD
seg000:00118940 38 A0 00 00 li      %r5, 0
seg000:00118944 48 00 BE 85 bl      .RBREAD
seg000:00118948 60 00 00 00 nop
seg000:0011894C 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118950 41 82 00 0C beq    loc_11895C
seg000:00118954 38 60 00 00 li      %r3, 0
seg000:00118958 48 00 01 18 b       exit
seg000:0011895C
seg000:0011895C          loc_11895C: # CODE XREF: check3+214j
seg000:0011895C A0 01 00 38 lhz    %r0, 0x50+var_18(%sp)
seg000:00118960 28 00 07 CF cmplwi %r0, 0xFC7
seg000:00118964 41 82 00 0C beq    loc_118970
seg000:00118968 38 60 00 00 li      %r3, 0
seg000:0011896C 48 00 01 04 b       exit
seg000:00118970
seg000:00118970          loc_118970: # CODE XREF: check3+228j
seg000:00118970 28 1F 00 03 cmplwi %r31, 3

```

## 8.5. DONGLES

```

seg000:00118974 40 82 00 F8 bne    error
seg000:00118978 38 60 00 01 li      %r3, 1
seg000:0011897C 48 00 00 F4 b      exit
seg000:00118980
seg000:00118980          loc_118980: # CODE XREF: check3+B8j
seg000:00118980          # check3+C4j
seg000:00118980 80 DE 00 00 lwz    %r6, dword_24B704
seg000:00118984 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:00118988 3B E0 00 00 li     %r31, 0
seg000:0011898C 38 60 00 04 li     %r3, 4
seg000:00118990 38 A0 00 00 li     %r5, 0
seg000:00118994 48 00 BE 35 bl    .RBEREAD
seg000:00118998 60 00 00 00 nop
seg000:0011899C 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:001189A0 41 82 00 0C beq    loc_1189AC
seg000:001189A4 38 60 00 00 li     %r3, 0
seg000:001189A8 48 00 00 C8 b     exit
seg000:001189AC
seg000:001189AC          loc_1189AC: # CODE XREF: check3+264j
seg000:001189AC A0 01 00 38 lhz    %r0, 0x50+var_18(%sp)
seg000:001189B0 28 00 1D 6A cmplwi %r0, 0xAED0
seg000:001189B4 40 82 00 0C bne    loc_1189C0
seg000:001189B8 3B E0 00 01 li     %r31, 1
seg000:001189BC 48 00 00 14 b     loc_1189D0
seg000:001189C0
seg000:001189C0          loc_1189C0: # CODE XREF: check3+278j
seg000:001189C0 28 00 18 28 cmplwi %r0, 0x2818
seg000:001189C4 41 82 00 0C beq    loc_1189D0
seg000:001189C8 38 60 00 00 li     %r3, 0
seg000:001189CC 48 00 00 A4 b     exit
seg000:001189D0
seg000:001189D0          loc_1189D0: # CODE XREF: check3+280j
seg000:001189D0          # check3+288j
seg000:001189D0 57 A0 06 3E clrlwi %r0, %r29, 24
seg000:001189D4 28 00 00 02 cmplwi %r0, 2
seg000:001189D8 40 82 00 20 bne    loc_1189F8
seg000:001189DC 57 E0 06 3F clrlwi. %r0, %r31, 24
seg000:001189E0 41 82 00 10 beq    good2
seg000:001189E4 48 00 4C 69 bl    sub_11D64C
seg000:001189E8 60 00 00 00 nop
seg000:001189EC 48 00 00 84 b     exit
seg000:001189F0
seg000:001189F0          good2:   # CODE XREF: check3+2A4j
seg000:001189F0 38 60 00 01 li     %r3, 1
seg000:001189F4 48 00 00 7C b     exit
seg000:001189F8
seg000:001189F8          loc_1189F8: # CODE XREF: check3+29Cj
seg000:001189F8 80 DE 00 00 lwz    %r6, dword_24B704
seg000:001189FC 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:00118A00 38 60 00 05 li     %r3, 5
seg000:00118A04 38 A0 00 00 li     %r5, 0
seg000:00118A08 48 00 BD C1 bl    .RBEREAD
seg000:00118A0C 60 00 00 00 nop
seg000:00118A10 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118A14 41 82 00 0C beq    loc_118A20
seg000:00118A18 38 60 00 00 li     %r3, 0
seg000:00118A1C 48 00 00 54 b     exit
seg000:00118A20
seg000:00118A20          loc_118A20: # CODE XREF: check3+2D8j
seg000:00118A20 A0 01 00 38 lhz    %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3 cmplwi %r0, 0xD300
seg000:00118A28 40 82 00 0C bne    loc_118A34
seg000:00118A2C 3B E0 00 01 li     %r31, 1
seg000:00118A30 48 00 00 14 b     good1
seg000:00118A34
seg000:00118A34          loc_118A34: # CODE XREF: check3+2ECj
seg000:00118A34 28 00 1A EB cmplwi %r0, 0xEBA1
seg000:00118A38 41 82 00 0C beq    good1
seg000:00118A3C 38 60 00 00 li     %r3, 0
seg000:00118A40 48 00 00 30 b     exit

```

## 8.5. DONGLES

```
seg000:00118A44          good1:      # CODE XREF: check3+2F4j
seg000:00118A44          # check3+2FCj
seg000:00118A44
seg000:00118A44 57 A0 06 3E  clrlwi %r0, %r29, 24
seg000:00118A48 28 00 00 03  cmplwi %r0, 3
seg000:00118A4C 40 82 00 20  bne   error
seg000:00118A50 57 E0 06 3F  clrlwi %r0, %r31, 24
seg000:00118A54 41 82 00 10  beq   good
seg000:00118A58 48 00 4B F5  bl    sub_11D64C
seg000:00118A5C 60 00 00 00  nop
seg000:00118A60 48 00 00 10  b     exit
seg000:00118A64
seg000:00118A64          good:      # CODE XREF: check3+318j
seg000:00118A64 38 60 00 01  li   %r3, 1
seg000:00118A68 48 00 00 08  b    exit
seg000:00118A6C
seg000:00118A6C          error:     # CODE XREF: check3+19Cj
seg000:00118A6C          # check3+238j ...
seg000:00118A6C 38 60 00 00  li   %r3, 0
seg000:00118A70
seg000:00118A70          exit:      # CODE XREF: check3+44j
seg000:00118A70          # check3+58j ...
seg000:00118A70 80 01 00 58  lwz   %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50  addi  %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC  lwz   %r31, var_4(%sp)
seg000:00118A7C 7C 08 03 A6  mtlr  %r0
seg000:00118A80 83 C1 FF F8  lwz   %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4  lwz   %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20  blr
seg000:00118A88          # End of function check3
```

There are a lot of calls to `.RBEREAD()`.

Perhaps, the function returns some values from the dongle, so they are compared here with some hard-coded variables using `CMPLWI`.

We also see that the r3 register is also filled before each call to `.RBEREAD()` with one of these values: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Probably a memory address or something like that?

Yes, indeed, by googling these function names it is easy to find the Sentinel Eve3 dongle manual!

Perhaps we don't even have to learn any other PowerPC instructions: all this function does is just call `.RBEREAD()`, compare its results with the constants and returns 1 if the comparisons are fine or 0 otherwise.

OK, all we've got is that `check1()` has always to return 1 or any other non-zero value.

But since we are not very confident in our knowledge of PowerPC instructions, we are going to be careful: we will patch the jumps in `check2()` at `0x001186FC` and `0x00118718`.

At `0x001186FC` we'll write bytes 0x48 and 0 thus converting the `BEQ` instruction in an `B` (unconditional jump): we can spot its opcode in the code without even referring to *[PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, (2000)]*<sup>13</sup>.

At `0x00118718` we'll write 0x60 and 3 zero bytes, thus converting it to a `NOP` instruction: Its opcode we could spot in the code too.

And now it all works without a dongle connected.

In summary, such small modifications can be done with `IDA` and minimal assembly language knowledge.

### 8.5.2 Example #2: SCO OpenServer

An ancient software for SCO OpenServer from 1997 developed by a company that disappeared a long time ago.

There is a special dongle driver to be installed in the system, that contains the following text strings: "Copyright 1989, Rainbow Technologies, Inc., Irvine, CA" and "Sentinel Integrated Driver Ver. 3.0".

<sup>13</sup>Also available as [http://yurichev.com/mirrors/PowerPC/6xx\\_pem.pdf](http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf)

## 8.5. DONGLES

After the installation of the driver in SCO OpenServer, these device files appear in the /dev filesystem:

```
/dev/rbsl8  
/dev/rbsl9  
/dev/rbsl10
```

The program reports an error without dongle connected, but the error string cannot be found in the executables.

Thanks to [IDA](#), it is easy to load the COFF executable used in SCO OpenServer.

Let's also try to find "rbsl" string and indeed, found it in this code fragment:

```
.text:00022AB8    public SSQC  
.text:00022AB8 SSQC    proc near ; CODE XREF: SSQ+7p  
.text:00022AB8  
.text:00022AB8 var_44 = byte ptr -44h  
.text:00022AB8 var_29 = byte ptr -29h  
.text:00022AB8 arg_0  = dword ptr  8  
.text:00022AB8  
.text:00022AB8     push    ebp  
.text:00022AB9     mov     ebp, esp  
.text:00022ABB     sub     esp, 44h  
.text:00022ABE     push    edi  
.text:00022ABF     mov     edi, offset unk_4035D0  
.text:00022AC4     push    esi  
.text:00022AC5     mov     esi, [ebp+arg_0]  
.text:00022AC8     push    ebx  
.text:00022AC9     push    esi  
.text:00022ACA     call    strlen  
.text:00022ACF     add    esp, 4  
.text:00022AD2     cmp    eax, 2  
.text:00022AD7     jnz    loc_22BA4  
.text:00022ADD     inc    esi  
.text:00022ADE     mov    al, [esi-1]  
.text:00022AE1     movsx  eax, al  
.text:00022AE4     cmp    eax, '3'  
.text:00022AE9     jz     loc_22B84  
.text:00022AEF     cmp    eax, '4'  
.text:00022AF4     jz     loc_22B94  
.text:00022AFA     cmp    eax, '5'  
.text:00022AFF     jnz    short loc_22B6B  
.text:00022B01     movsx  ebx, byte ptr [esi]  
.text:00022B04     sub    ebx, '0'  
.text:00022B07     mov    eax, 7  
.text:00022B0C     add    eax, ebx  
.text:00022B0E     push   eax  
.text:00022B0F     lea    eax, [ebp+var_44]  
.text:00022B12     push   offset aDevSlD  ; "/dev/sl%d"  
.text:00022B17     push   eax  
.text:00022B18     call   nl_sprintf  
.text:00022B1D     push   0          ; int  
.text:00022B1F     push   offset aDevRbsl8 ; char *  
.text:00022B24     call   _access  
.text:00022B29     add    esp, 14h  
.text:00022B2C     cmp    eax, 0xFFFFFFFFh  
.text:00022B31     jz     short loc_22B48  
.text:00022B33     lea    eax, [ebx+7]  
.text:00022B36     push   eax  
.text:00022B37     lea    eax, [ebp+var_44]  
.text:00022B3A     push   offset aDevRbslD ; "/dev/rbsl%d"  
.text:00022B3F     push   eax  
.text:00022B40     call   nl_sprintf  
.text:00022B45     add    esp, 0Ch  
.text:00022B48 loc_22B48: ; CODE XREF: SSQC+79j  
.text:00022B48     mov    edx, [edi]  
.text:00022B4A     test  edx, edx  
.text:00022B4C     jle   short loc_22B57  
.text:00022B4E     push   edx          ; int  
.text:00022B4F     call   _close
```

## 8.5. DONGLES

```
.text:00022B54      add    esp, 4
.text:00022B57 loc_22B57: ; CODE XREF: SSQC+94j
.text:00022B57      push   2          ; int
.text:00022B59      lea    eax, [ebp+var_44]
.text:00022B5C      push   eax         ; char *
.text:00022B5D      call   _open
.text:00022B62      add    esp, 8
.text:00022B65      test  eax, eax
.text:00022B67      mov    [edi], eax
.text:00022B69      jge   short loc_22B78
.text:00022B6B loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B      mov    eax, 0FFFFFFFh
.text:00022B70      pop    ebx
.text:00022B71      pop    esi
.text:00022B72      pop    edi
.text:00022B73      mov    esp, ebp
.text:00022B75      pop    ebp
.text:00022B76      retn
.text:00022B78
.text:00022B78 loc_22B78: ; CODE XREF: SSQC+B1j
.text:00022B78      pop    ebx
.text:00022B79      pop    esi
.text:00022B7A      pop    edi
.text:00022B7B      xor   eax, eax
.text:00022B7D      mov    esp, ebp
.text:00022B7F      pop    ebp
.text:00022B80      retn
.text:00022B84
.text:00022B84 loc_22B84: ; CODE XREF: SSQC+31j
.text:00022B84      mov    al, [esi]
.text:00022B86      pop    ebx
.text:00022B87      pop    esi
.text:00022B88      pop    edi
.text:00022B89      mov    ds:byte_407224, al
.text:00022B8E      mov    esp, ebp
.text:00022B90      xor   eax, eax
.text:00022B92      pop    ebp
.text:00022B93      retn
.text:00022B94
.text:00022B94 loc_22B94: ; CODE XREF: SSQC+3Cj
.text:00022B94      mov    al, [esi]
.text:00022B96      pop    ebx
.text:00022B97      pop    esi
.text:00022B98      pop    edi
.text:00022B99      mov    ds:byte_407225, al
.text:00022B9E      mov    esp, ebp
.text:00022BA0      xor   eax, eax
.text:00022BA2      pop    ebp
.text:00022BA3      retn
.text:00022BA4
.text:00022BA4 loc_22BA4: ; CODE XREF: SSQC+1Fj
.text:00022BA4      movsx  eax, ds:byte_407225
.text:00022BAB      push   esi
.text:00022BAC      push   eax
.text:00022BAD      movsx  eax, ds:byte_407224
.text:00022BB4      push   eax
.text:00022BB5      lea    eax, [ebp+var_44]
.text:00022BB8      push   offset a46CCS    ; "46%C%C%$"
.text:00022BBD      push   eax
.text:00022BBE      call   nl_sprintf
.text:00022BC3      lea    eax, [ebp+var_44]
.text:00022BC6      push   eax
.text:00022BC7      call   strlen
.text:00022BCC      add    esp, 18h
.text:00022BCF      cmp   eax, 1Bh
.text:00022BD4      jle   short loc_22BDA
.text:00022BD6      mov    [ebp+var_29], 0
.text:00022BDA
```

## 8.5. DONGLES

```
.text:00022BDA loc_22BDA: ; CODE XREF: SSQC+11Cj
.text:00022BDA     lea    eax, [ebp+var_44]
.text:00022BDD     push   eax
.text:00022BDE     call   strlen
.text:00022BE3     push   eax          ; unsigned int
.text:00022BE4     lea    eax, [ebp+var_44]
.text:00022BE7     push   eax          ; void *
.text:00022BE8     mov    eax, [edi]
.text:00022BEA     push   eax          ; int
.text:00022BEB     call   _write
.text:00022BF0     add    esp, 10h
.text:00022BF3     pop    ebx
.text:00022BF4     pop    esi
.text:00022BF5     pop    edi
.text:00022BF6     mov    esp, ebp
.text:00022BF8     pop    ebp
.text:00022BF9     retn
.text:00022BFA     db    0Eh dup(90h)
.text:00022BFA SSQC    endp
```

Yes, indeed, the program needs to communicate with the driver somehow.

The only place where the `SSQC()` function is called is the [thunk function](#):

```
.text:0000DBE8     public SSQ
.text:0000DBE8 SSQ    proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8          ; sys_info+CBp ...
.text:0000DBE8
.text:0000DBE8 arg_0  = dword ptr 8
.text:0000DBE8
.text:0000DBE8     push   ebp
.text:0000DBE9     mov    ebp, esp
.text:0000DBEB     mov    edx, [ebp+arg_0]
.text:0000DBEE     push   edx
.text:0000DBEF     call   SSQC
.text:0000DBF4     add    esp, 4
.text:0000DBF7     mov    esp, ebp
.text:0000DBF9     pop    ebp
.text:0000DBFA     retn
.text:0000DBFB SSQ    endp
```

`SSQ()` can be called from at least 2 functions.

One of these is:

```
.data:0040169C _51_52_53      dd offset aPressAnyKeyT_0 ; DATA XREF: init_sys+392r
.data:0040169C                  ; sys_info+A1r
.data:0040169C                  ; "PRESS ANY KEY TO CONTINUE: "
.data:004016A0      dd offset a51       ; "51"
.data:004016A4      dd offset a52       ; "52"
.data:004016A8      dd offset a53       ; "53"

...
.data:004016B8 _3C_or_3E      dd offset a3c       ; DATA XREF: sys_info:loc_D67Br
.data:004016B8                  ; "3C"
.data:004016BC      dd offset a3e       ; "3E"

; these names we gave to the labels:
.data:004016C0 answers1        dd 6B05h      ; DATA XREF: sys_info+E7r
.data:004016C4          dd 3D87h
.data:004016C8 answers2        dd 3Ch       ; DATA XREF: sys_info+F2r
.data:004016CC          dd 832h
.data:004016D0 _C_and_B       db 0Ch       ; DATA XREF: sys_info+BAr
.data:004016D0          ; sys_info:OKr
.data:004016D1 byte_4016D1    db 0Bh       ; DATA XREF: sys_info+FD
.data:004016D2          db 0

...
```

## 8.5. DONGLES

```

.text:0000D652          xor    eax, eax
.text:0000D654          mov    al, ds:ctl_port
.text:0000D659          mov    ecx, _51_52_53[eax*4]
.text:0000D660          push   ecx
.text:0000D661          call   SSQ
.text:0000D666          add    esp, 4
.text:0000D669          cmp    eax, 0FFFFFFFh
.text:0000D66E          jz    short loc_D6D1
.text:0000D670          xor    ebx, ebx
.text:0000D672          mov    al, _C_and_B
.text:0000D677          test   al, al
.text:0000D679          jz    short loc_D6C0
.text:0000D67B
.loc_D67B: ; CODE XREF: sys_info+106j
.text:0000D67B          mov    eax, _3C_or_3E[ebx*4]
.text:0000D682          push   eax
.text:0000D683          call   SSQ
.text:0000D688          push   offset a4g      ; "4G"
.text:0000D68D          call   SSQ
.text:0000D692          push   offset a0123456789 ; "0123456789"
.text:0000D697          call   SSQ
.text:0000D69C          add    esp, 0Ch
.text:0000D69F          mov    edx, answers1[ebx*4]
.text:0000D6A6          cmp    eax, edx
.text:0000D6A8          jz    short OK
.text:0000D6AA          mov    ecx, answers2[ebx*4]
.text:0000D6B1          cmp    eax, ecx
.text:0000D6B3          jz    short OK
.text:0000D6B5          mov    al, byte_4016D1[ebx]
.text:0000D6BB          inc    ebx
.text:0000D6BC          test   al, al
.text:0000D6BE          jnz   short loc_D67B
.text:0000D6C0
.loc_D6C0: ; CODE XREF: sys_info+C1j
.text:0000D6C0          inc    ds:ctl_port
.text:0000D6C6          xor    eax, eax
.text:0000D6C8          mov    al, ds:ctl_port
.text:0000D6CD          cmp    eax, edi
.text:0000D6CF          jle   short loc_D652
.text:0000D6D1
.loc_D6D1: ; CODE XREF: sys_info+98j
.text:0000D6D1          ; sys_info+B6j
.text:0000D6D1          mov    edx, [ebp+var_8]
.text:0000D6D4          inc    edx
.text:0000D6D5          mov    [ebp+var_8], edx
.text:0000D6D8          cmp    edx, 3
.text:0000D6DB          jle   loc_D641
.text:0000D6E1
.loc_D6E1: ; CODE XREF: sys_info+16j
.text:0000D6E1          ; sys_info+51j ...
.text:0000D6E1          pop    ebx
.text:0000D6E2          pop    edi
.text:0000D6E3          mov    esp, ebp
.text:0000D6E5          pop    ebp
.text:0000D6E6          retn
.text:0000D6E8  OK:    ; CODE XREF: sys_info+F0j
.text:0000D6E8          ; sys_info+FBj
.text:0000D6E8          mov    al, _C_and_B[ebx]
.text:0000D6EE          pop    ebx
.text:0000D6EF          pop    edi
.text:0000D6F0          mov    ds:ctl_model, al
.text:0000D6F5          mov    esp, ebp
.text:0000D6F7          pop    ebp
.text:0000D6F8          retn
.text:0000D6F8  sys_info        endp

```

“3C” and “3E” sound familiar: there was a Sentinel Pro dongle by Rainbow with no memory, providing only one crypto-hashing secret function.

You can read a short description of what hash function is here: [2.10 on page 465](#).

## 8.5. DONGLES

But let's get back to the program.

So the program can only check the presence or absence of a connected dongle.

No other information can be written to such dongle, as it has no memory. The two-character codes are commands (we can see how the commands are handled in the `SSQC()` function) and all other strings are hashed inside the dongle, being transformed into a 16-bit number. The algorithm was secret, so it was not possible to write a driver replacement or to remake the dongle hardware that would emulate it perfectly.

However, it is always possible to intercept all accesses to it and to find what constants the hash function results are compared to.

But we need to say that it is possible to build a robust software copy protection scheme based on secret cryptographic hash-function: let it encrypt/decrypt the data files your software uses.

But let's get back to the code.

Codes 51/52/53 are used for LPT printer port selection. 3x/4x are used for "family" selection (that's how Sentinel Pro dongles are differentiated from each other: more than one dongle can be connected to a LPT port).

The only non-2-character string passed to the hashing function is "0123456789".

Then, the result is compared against the set of valid results.

If it is correct, 0xC or 0xB is to be written into the global variable `ctl_model`.

Another text string that gets passed is "PRESS ANY KEY TO CONTINUE: ", but the result is not checked. Hard to say why, probably by mistake <sup>14</sup>.

Let's see where the value from the global variable `ctl_mode` is used.

One such place is:

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14    = dword ptr -14h
.text:0000D708 var_10    = byte ptr -10h
.text:0000D708 var_8     = dword ptr -8
.text:0000D708 var_2     = word ptr -2
.text:0000D708
.text:0000D708         push    ebp
.text:0000D709         mov     eax, ds:net_env
.text:0000D70E         mov     ebp, esp
.text:0000D710         sub     esp, 1Ch
.text:0000D713         test    eax, eax
.text:0000D715         jnz    short loc_D734
.text:0000D717         mov     al, ds:ctl_model
.text:0000D71C         test    al, al
.text:0000D71E         jnz    short loc_D77E
.text:0000D720         mov     [ebp+var_8], offset aIeCvulnvv0kgT_ ; "Ie-cvulnvV\\b0KG]T_"
.text:0000D727         mov     edx, 7
.text:0000D72C         jmp    loc_D7E7

...
.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7         ; prep_sys+33j
.text:0000D7E7         push    edx
.text:0000D7E8         mov     edx, [ebp+var_8]
.text:0000D7EB         push    20h
.text:0000D7ED         push    edx
.text:0000D7EE         push    16h
.text:0000D7F0         call    err_warn
.text:0000D7F5         push    offset station_sem
.text:0000D7FA         call    ClosSem
.text:0000D7FF         call    startup_err
```

If it is 0, an encrypted error message is passed to a decryption routine and printed.

The error string decryption routine seems a simple [xor-ing](#):

<sup>14</sup>What a strange feeling: to find bugs in such ancient software.

## 8.5. DONGLES

```

.text:0000A43C err_warn    proc near                ; CODE XREF: prep_sys+E8p
.text:0000A43C                                         ; prep_sys2+2Fp ...
.text:0000A43C
.text:0000A43C var_55      = byte ptr -55h
.text:0000A43C var_54      = byte ptr -54h
.text:0000A43C arg_0       = dword ptr  8
.text:0000A43C arg_4       = dword ptr  0Ch
.text:0000A43C arg_8       = dword ptr  10h
.text:0000A43C arg_C       = dword ptr  14h
.text:0000A43C
.text:0000A43C             push    ebp
.text:0000A43D             mov     ebp, esp
.text:0000A43F             sub     esp, 54h
.text:0000A442             push    edi
.text:0000A443             mov     ecx, [ebp+arg_8]
.text:0000A446             xor    edi, edi
.text:0000A448             test   ecx, ecx
.text:0000A44A             push    esi
.text:0000A44B             jle    short loc_A466
.text:0000A44D             mov     esi, [ebp+arg_C] ; key
.text:0000A450             mov     edx, [ebp+arg_4] ; string
.text:0000A453
.text:0000A453 loc_A453:    ; CODE XREF: err_warn+28j
.text:0000A453             xor    eax, eax
.text:0000A455             mov     al, [edx+edi]
.text:0000A458             xor    eax, esi
.text:0000A45A             add    esi, 3
.text:0000A45D             inc    edi
.text:0000A45E             cmp    edi, ecx
.text:0000A460             mov     [ebp+edi+var_55], al
.text:0000A464             jl    short loc_A453
.text:0000A466
.text:0000A466 loc_A466:    ; CODE XREF: err_warn+Fj
.text:0000A466             mov     [ebp+edi+var_54], 0
.text:0000A46B             mov     eax, [ebp+arg_0]
.text:0000A46E             cmp    eax, 18h
.text:0000A473             jnz    short loc_A49C
.text:0000A475             lea    eax, [ebp+var_54]
.text:0000A478             push   eax
.text:0000A479             call   status_line
.text:0000A47E             add    esp, 4
.text:0000A481
.text:0000A481 loc_A481:    ; CODE XREF: err_warn+72j
.text:0000A481             push   50h
.text:0000A483             push   0
.text:0000A485             lea    eax, [ebp+var_54]
.text:0000A488             push   eax
.text:0000A489             call   memset
.text:0000A48E             call   pcv_refresh
.text:0000A493             add    esp, 0Ch
.text:0000A496             pop    esi
.text:0000A497             pop    edi
.text:0000A498             mov    esp, ebp
.text:0000A49A             pop    ebp
.text:0000A49B             retn
.text:0000A49C
.text:0000A49C loc_A49C:    ; CODE XREF: err_warn+37j
.text:0000A49C             push   0
.text:0000A49E             lea    eax, [ebp+var_54]
.text:0000A4A1             mov    edx, [ebp+arg_0]
.text:0000A4A4             push   edx
.text:0000A4A5             push   eax
.text:0000A4A6             call   pcv_lputs
.text:0000A4AB             add    esp, 0Ch
.text:0000A4AE             jmp    short loc_A481
.text:0000A4AE err_warn    endp

```

That's why we were unable to find the error messages in the executable files, because they are encrypted (which is popular practice).

## 8.5. DONGLES

Another call to the `SSQ()` hashing function passes the “offln” string to it and compares the result with `0xFE81` and `0x12A9`.

If they don’t match, it works with some `timer()` function (maybe waiting for a poorly connected dongle to be reconnected and check again?) and then decrypts another error message to dump.

```
.text:0000DA55 loc_DA55:                                ; CODE XREF: sync_sys+24Cj
.text:0000DA55          push    offset aOffln      ; "offln"
.text:0000DA5A          call    SSQ
.text:0000DA5F          add     esp, 4
.text:0000DA62          mov     dl, [ebx]
.text:0000DA64          mov     esi, eax
.text:0000DA66          cmp     dl, 0Bh
.text:0000DA69          jnz    short loc_DA83
.text:0000DA6B          cmp     esi, 0FE81h
.text:0000DA71          jz     OK
.text:0000DA77          cmp     esi, 0FFFFF8EFh
.text:0000DA7D          jz     OK
.text:0000DA83
.text:0000DA83 loc_DA83:                                ; CODE XREF: sync_sys+201j
.text:0000DA83          mov     cl, [ebx]
.text:0000DA85          cmp     cl, 0Ch
.text:0000DA88          jnz    short loc_DA9F
.text:0000DA8A          cmp     esi, 12A9h
.text:0000DA90          jz     OK
.text:0000DA96          cmp     esi, 0FFFFFFF5h
.text:0000DA99          jz     OK
.text:0000DA9F
.text:0000DA9F loc_DA9F:                                ; CODE XREF: sync_sys+220j
.text:0000DA9F          mov     eax, [ebp+var_18]
.text:0000DAA2          test   eax, eax
.text:0000DAA4          jz     short loc_DAB0
.text:0000DAA6          push   24h
.text:0000DAA8          call   timer
.text:0000DAAD          add    esp, 4
.text:0000DAB0
.text:0000DAB0 loc_DAB0:                                ; CODE XREF: sync_sys+23Cj
.text:0000DAB0          inc    edi
.text:0000DAB1          cmp    edi, 3
.text:0000DAB4          jle    short loc_DA55
.text:0000DAB6          mov    eax, ds:net_env
.text:0000DABB          test   eax, eax
.text:0000DABD          jz     short error
...
.text:0000DAF7 error:                                    ; CODE XREF: sync_sys+255j
.text:0000DAF7          ; sync_sys+274j ...
.text:0000DAF7          mov    [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE          mov    [ebp+var_C], 17h ; decrypting key
.text:0000DB05          jmp    decrypt_end_print_message
...
; this name we gave to label:
.text:0000D9B6 decrypt_end_print_message:                ; CODE XREF: sync_sys+29Dj
.text:0000D9B6          ; sync_sys+2ABj
.text:0000D9B6          mov    eax, [ebp+var_18]
.text:0000D9B9          test   eax, eax
.text:0000D9BB          jnz    short loc_D9FB
.text:0000D9BD          mov    edx, [ebp+var_C] ; key
.text:0000D9C0          mov    ecx, [ebp+var_8] ; string
.text:0000D9C3          push   edx
.text:0000D9C4          push   20h
.text:0000D9C6          push   ecx
.text:0000D9C7          push   18h
.text:0000D9C9          call   err_warn
.text:0000D9CE          push   0Fh
.text:0000D9D0          push   190h
.text:0000D9D5          call   sound
```

## 8.5. DONGLES

```
.text:0000D9DA          mov    [ebp+var_18], 1
.text:0000D9E1          add    esp, 18h
.text:0000D9E4          call   pcv_kbhit
.text:0000D9E9          test   eax, eax
.text:0000D9EB          jz    short loc_D9FB

...
; this name we gave to label:
.data:00401736 encrypted_error_message2 db 74h, 72h, 78h, 43h, 48h, 6, 5Ah, 49h, 4Ch, 2 dup(47h,
                                         )
.data:00401736           db 51h, 4Fh, 47h, 61h, 20h, 22h, 3Ch, 24h, 33h, 36h, 76h
.data:00401736           db 3Ah, 33h, 31h, 0Ch, 0, 0Bh, 1Fh, 7, 1Eh, 1Ah
```

Bypassing the dongle is pretty straightforward: just patch all jumps after the relevant `CMP` instructions.

Another option is to write our own SCO OpenServer driver, containing a table of questions and answers, all of those which present in the program.

## Decrypting error messages

By the way, we can also try to decrypt all error messages. The algorithm that is located in the `err_warn()` function is very simple, indeed:

Listing 8.3: Decryption function

```
.text:0000A44D          mov    esi, [ebp+arg_C] ; key
.text:0000A450          mov    edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453          xor    eax, eax
.text:0000A455          mov    al, [edx+edi] ; load encrypted byte
.text:0000A458          xor    eax, esi      ; decrypt it
.text:0000A45A          add    esi, 3       ; change key for the next byte
.text:0000A45D          inc    edi
.text:0000A45E          cmp    edi, ecx
.text:0000A460          mov    [ebp+edi+var_55], al
.text:0000A464          jl    short loc_A453
```

As we can see, not just string is supplied to the decryption function, but also the key:

```
.text:0000DAF7 error:                                ; CODE XREF: sync_sys+255j
.text:0000DAF7                                     ; sync_sys+274j ...
.text:0000DAF7          mov    [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE          mov    [ebp+var_C], 17h ; decrypting key
.text:0000DB05          jmp    decrypt_end_print_message

...
; this name we gave to label manually:
.text:0000D9B6 decrypt_end_print_message:           ; CODE XREF: sync_sys+29Dj
.text:0000D9B6                                     ; sync_sys+2ABj
.text:0000D9B6          mov    eax, [ebp+var_18]
.text:0000D9B9          test   eax, eax
.text:0000D9BB          jnz    short loc_D9FB
.text:0000D9BD          mov    edx, [ebp+var_C] ; key
.text:0000D9C0          mov    ecx, [ebp+var_8] ; string
.text:0000D9C3          push   edx
.text:0000D9C4          push   20h
.text:0000D9C6          push   ecx
.text:0000D9C7          push   18h
.text:0000D9C9          call   err_warn
```

The algorithm is a simple [xoring](#): each byte is xored with a key, but the key is increased by 3 after the processing of each byte.

We can write a simple Python script to check our hypothesis:

## 8.5. DONGLES

Listing 8.4: Python 3.x

```
#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
    sys.stdout.write ("%c" % (i^tmp))
    tmp=tmp+3
sys.stdout.flush()
```

And it prints: “check security device connection”. So yes, this is the decrypted message.

There are also other encrypted messages with their corresponding keys. But needless to say, it is possible to decrypt them without their keys. First, we can see that the key is in fact a byte. It is because the core decryption instruction (`XOR`) works on byte level. The key is located in the `ESI` register, but only one byte part of `ESI` is used. Hence, a key may be greater than 255, but its value is always to be rounded.

As a consequence, we can just try brute-force, trying all possible keys in the 0..255 range. We are also going to skip the messages that has unprintable characters.

Listing 8.5: Python 3.x

```
#!/usr/bin/python
import sys, curses.ascii

msgs=[
[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A], 

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44], 

[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B,
0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C], 

[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x0, 0x4C, 0x40, 0x9, 0x4D, 0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23,
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10], 

[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]] 

def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
    print ("message #%d" % cnt)
    for key in range(0,256):
        result=[]
        tmp=key
        for i in msg:
            result.append (i^tmp)
            tmp=tmp+3
        if is_string_printable (result):
            print ("key=", key, "value=", "".join(list(map(chr, result))))
    cnt=cnt+1
```

## 8.5. DONGLES

And we get:

Listing 8.6: Results

```
message #1
key= 20 value= `eb^h%|``hudw|_af{n~f%ljmSbnwlpk
key= 21 value= ajc]i"}cawtgv{^bgto}g"millcmvkqh
key= 22 value= bkd\j#rbbvsfuz!cduh|d#bhomdlujni
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhsx#ejwjbb!`nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cduhld#għtme?!#!'!#
message #3
key= 7 value= Bk<waoqNUpu$`yreoa\wpmpusj,bkIjh
key= 8 value= Mj?vfnr0jqv%gxqd`_vwlstlk/clHii
key= 9 value= Lm>ugasLkvw&fgpgag^uvcrwml.`mwhj
key= 10 value= Ol!td`tMhwx'efwfbf!tubuvnm!anvok
key= 11 value= No security device station found
key= 12 value= In#rjbvsnuz!{duhdd#r{\`whho#gPtme
message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= 0vlmdq!hg#`juknuhydk!vrbsp!Zy`dbe
message #5
key= 17 value= check security device station
key= 18 value= `ijbh!td`tmhwx'efwfbf!tubuVnm!'!
```

There is some garbage, but we can quickly find the English-language messages!

By the way, since the algorithm is a simple xoring encryption, the very same function can be used to encrypt messages. If needed, we can encrypt our own messages, and patch the program by inserting them.

### 8.5.3 Example #3: MS-DOS

Another very old software for MS-DOS from 1995 also developed by a company that disappeared a long time ago.

In the pre-DOS extenders era, all the software for MS-DOS mostly relied on 16-bit 8086 or 80286 CPUs, so the code was 16-bit en masse.

The 16-bit code is mostly same as you already saw in this book, but all registers are 16-bit and there are less instructions available.

The MS-DOS environment has no system drivers, and any program can deal with the bare hardware via ports, so here you can see the OUT / IN instructions, which are present in mostly in drivers in our times (it is impossible to access ports directly in user mode on all modern OSes).

Given that, the MS-DOS program which works with a dongle has to access the LPT printer port directly.

So we can just search for such instructions. And yes, here they are:

```
seg030:0034          out_port proc far ; CODE XREF: sent_pro+22p
seg030:0034              ; sent_pro+2Ap ...
seg030:0034
seg030:0034          arg_0      = byte ptr  6
seg030:0034
seg030:0034 55          push      bp
seg030:0035 8B EC        mov       bp, sp
seg030:0037 8B 16 7E E7    mov       dx, _out_port ; 0x378
seg030:003B 8A 46 06    mov       al, [bp+arg_0]
seg030:003E EE          out      dx, al
seg030:003F 5D          pop       bp
seg030:0040 CB          retf
seg030:0040          out_port endp
```

(All label names in this example were given by me).

out\_port() is referenced only in one function:

## 8.5. DONGLES

```

seg030:0041          sent_pro proc far ; CODE XREF: check_dongle+34p
seg030:0041
seg030:0041          var_3     = byte ptr -3
seg030:0041          var_2     = word ptr -2
seg030:0041          arg_0     = dword ptr 6
seg030:0041
seg030:0041 C8 04 00 00      enter    4, 0
seg030:0045 56          push     si
seg030:0046 57          push     di
seg030:0047 8B 16 82 E7      mov      dx, _in_port_1 ; 0x37A
seg030:004B EC          in       al, dx
seg030:004C 8A D8          mov      bl, al
seg030:004E 80 E3 FE          and     bl, 0FEh
seg030:0051 80 CB 04          or      bl, 4
seg030:0054 8A C3          mov      al, bl
seg030:0056 88 46 FD          mov      [bp+var_3], al
seg030:0059 80 E3 1F          and     bl, 1Fh
seg030:005C 8A C3          mov      al, bl
seg030:005E EE          out     dx, al
seg030:005F 68 FF 00          push    0FFh
seg030:0062 0E          push    cs
seg030:0063 E8 CE FF          call    near ptr out_port
seg030:0066 59          pop     cx
seg030:0067 68 D3 00          push    0D3h
seg030:006A 0E          push    cs
seg030:006B E8 C6 FF          call    near ptr out_port
seg030:006E 59          pop     cx
seg030:006F 33 F6          xor     si, si
seg030:0071 EB 01          jmp     short loc_359D4
seg030:0073
seg030:0073          loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073 46          inc     si
seg030:0074
seg030:0074          loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074 81 FE 96 00      cmp     si, 96h
seg030:0078 7C F9          jl      short loc_359D3
seg030:007A 68 C3 00          push    0C3h
seg030:007D 0E          push    cs
seg030:007E E8 B3 FF          call    near ptr out_port
seg030:0081 59          pop     cx
seg030:0082 68 C7 00          push    0C7h
seg030:0085 0E          push    cs
seg030:0086 E8 AB FF          call    near ptr out_port
seg030:0089 59          pop     cx
seg030:008A 68 D3 00          push    0D3h
seg030:008D 0E          push    cs
seg030:008E E8 A3 FF          call    near ptr out_port
seg030:0091 59          pop     cx
seg030:0092 68 C3 00          push    0C3h
seg030:0095 0E          push    cs
seg030:0096 E8 9B FF          call    near ptr out_port
seg030:0099 59          pop     cx
seg030:009A 68 C7 00          push    0C7h
seg030:009D 0E          push    cs
seg030:009E E8 93 FF          call    near ptr out_port
seg030:00A1 59          pop     cx
seg030:00A2 68 D3 00          push    0D3h
seg030:00A5 0E          push    cs
seg030:00A6 E8 8B FF          call    near ptr out_port
seg030:00A9 59          pop     cx
seg030:00AA BF FF FF          mov     di, 0FFFFh
seg030:00AD EB 40          jmp     short loc_35A4F
seg030:00AF
seg030:00AF          loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00          mov     si, 4
seg030:00B2
seg030:00B2          loc_35A12: ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7          shl     di, 1
seg030:00B4 8B 16 80 E7          mov     dx, _in_port_2 ; 0x379

```

## 8.5. DONGLES

```

seg030:00B8 EC          in     al, dx
seg030:00B9 A8 80        test    al, 80h
seg030:00BB 75 03        jnz    short loc_35A20
seg030:00BD 83 CF 01        or     di, 1
seg030:00C0
seg030:00C0 loc_35A20: ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+      test    [bp+var_2], 8
seg030:00C5 74 05        jz     short loc_35A2C
seg030:00C7 68 D7 00        push    0D7h ; '+'
seg030:00CA EB 0B        jmp    short loc_35A37
seg030:00CC
seg030:00CC loc_35A2C: ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00        push    0C3h
seg030:00CF 0E          push    cs
seg030:00D0 E8 61 FF        call    near ptr out_port
seg030:00D3 59          pop     cx
seg030:00D4 68 C7 00        push    0C7h
seg030:00D7
seg030:00D7 loc_35A37: ; CODE XREF: sent_pro+89j
seg030:00D7 0E          push    cs
seg030:00D8 E8 59 FF        call    near ptr out_port
seg030:00DB 59          pop     cx
seg030:00DC 68 D3 00        push    0D3h
seg030:00DF 0E          push    cs
seg030:00E0 E8 51 FF        call    near ptr out_port
seg030:00E3 59          pop     cx
seg030:00E4 8B 46 FE        mov     ax, [bp+var_2]
seg030:00E7 D1 E0        shl     ax, 1
seg030:00E9 89 46 FE        mov     [bp+var_2], ax
seg030:00EC 4E          dec     si
seg030:00ED 75 C3        jnz    short loc_35A12
seg030:00EF
seg030:00EF loc_35A4F: ; CODE XREF: sent_pro+6Cj
seg030:00EF C4 5E 06        les     bx, [bp+arg_0]
seg030:00F2 FF 46 06        inc     word ptr [bp+arg_0]
seg030:00F5 26 8A 07        mov     al, es:[bx]
seg030:00F8 98          cbw
seg030:00F9 89 46 FE        mov     [bp+var_2], ax
seg030:00FC 0B C0        or     ax, ax
seg030:00FE 75 AF        jnz    short loc_35A0F
seg030:0100 68 FF 00        push    0FFh
seg030:0103 0E          push    cs
seg030:0104 E8 2D FF        call    near ptr out_port
seg030:0107 59          pop     cx
seg030:0108 8B 16 82 E7        mov     dx, _in_port_1 ; 0x37A
seg030:010C EC          in     al, dx
seg030:010D 8A C8        mov     cl, al
seg030:010F 80 E1 5F        and    cl, 5Fh
seg030:0112 8A C1        mov     al, cl
seg030:0114 EE          out    dx, al
seg030:0115 EC          in     al, dx
seg030:0116 8A C8        mov     cl, al
seg030:0118 F6 C1 20        test   cl, 20h
seg030:011B 74 08        jz     short loc_35A85
seg030:011D 8A 5E FD        mov     bl, [bp+var_3]
seg030:0120 80 E3 DF        and    bl, 0DFh
seg030:0123 EB 03        jmp    short loc_35A88
seg030:0125
seg030:0125 loc_35A85: ; CODE XREF: sent_pro+DAj
seg030:0125 8A 5E FD        mov     bl, [bp+var_3]
seg030:0128
seg030:0128 loc_35A88: ; CODE XREF: sent_pro+E2j
seg030:0128 F6 C1 80        test   cl, 80h
seg030:012B 74 03        jz     short loc_35A90
seg030:012D 80 E3 7F        and    bl, 7Fh
seg030:0130
seg030:0130 loc_35A90: ; CODE XREF: sent_pro+EAj
seg030:0130 8B 16 82 E7        mov     dx, _in_port_1 ; 0x37A
seg030:0134 8A C3        mov     al, bl
seg030:0136 EE          out    dx, al

```

## 8.5. DONGLES

```

seg030:0137 8B C7          mov      ax, di
seg030:0139 5F          pop      di
seg030:013A 5E          pop      si
seg030:013B C9          leave
seg030:013C CB          retf
seg030:013C           sent_pro endp

```

This is again a Sentinel Pro “hashing” dongle as in the previous example. It is noticeably because text strings are passed here, too, and 16 bit values are returned and compared with others.

So that is how Sentinel Pro is accessed via ports.

The output port address is usually 0x378, i.e., the printer port, where the data to the old printers in pre-USB era was passed to.

The port is uni-directional, because when it was developed, no one imagined that someone will need to transfer information from the printer <sup>15</sup>.

The only way to get information from the printer is a status register on port 0x379, which contains such bits as “paper out”, “ack”, “busy”—thus the printer may signal to the host computer if it is ready or not and if paper is present in it.

So the dongle returns information from one of these bits, one bit at each iteration.

`_in_port_2` contains the address of the status word (0x379) and `_in_port_1` contains the control register address (0x37A).

It seems that the dongle returns information via the “busy” flag at `seg030:00B9`: each bit is stored in the `DI` register, which is returned at the end of the function.

What do all these bytes sent to output port mean? Hard to say. Perhaps, commands to the dongle.

But generally speaking, it is not necessary to know: it is easy to solve our task without that knowledge.

Here is the dongle checking routine:

```

00000000 struct_0      struct ; (sizeof=0x1B)
00000000 field_0        db 25 dup(?)      ; string(C)
00000019 _A             dw ?
0000001B struct_0       ends

dseg:3CBC 61 63 72 75+_0  struct_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+    ; DATA XREF: check_dongle+2Eo

... skipped ...

dseg:3E00 63 6F 66 66+  struct_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+  struct_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+  struct_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+  struct_0 <'paper', 0FFDFh>
dseg:3E6C 63 6F 6B 65+  struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+  struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+  struct_0 <'dir', 0FFAAh>
dseg:3EBD 63 6F 70 79+  struct_0 <'copy', 0F557h>

seg030:0145            check_dongle proc far ; CODE XREF: sub_3771D+3EP
seg030:0145
seg030:0145            var_6 = dword ptr -6
seg030:0145            var_2 = word ptr -2
seg030:0145
seg030:0145 C8 06 00 00  enter   6, 0
seg030:0149 56          push     si
seg030:014A 66 6A 00    push     large 0      ; newtime
seg030:014D 6A 00        push     0           ; cmd
seg030:014F 9A C1 18 00+ call     _biostime
seg030:0154 52          push     dx
seg030:0155 50          push     ax
seg030:0156 66 58        pop      eax
seg030:0158 83 C4 06    add     sp, 6

```

<sup>15</sup>If we consider Centronics only. The following IEEE 1284 standard allows the transfer of information from the printer.

## 8.5. DONGLES

```

seg030:015B 66 89 46 FA    mov    [bp+var_6], eax
seg030:015F 66 3B 06 D8+   cmp    eax, _expiration
seg030:0164 7E 44         jle    short loc_35B0A
seg030:0166 6A 14         push   14h
seg030:0168 90           nop
seg030:0169 0E           push   cs
seg030:016A E8 52 00     call   near ptr get_rand
seg030:016D 59           pop    cx
seg030:016E 8B F0         mov    si, ax
seg030:0170 6B C0 1B     imul  ax, 1Bh
seg030:0173 05 BC 3C     add   ax, offset _0
seg030:0176 1E           push   ds
seg030:0177 50           push   ax
seg030:0178 0E           push   cs
seg030:0179 E8 C5 FE     call   near ptr sent_pro
seg030:017C 83 C4 04     add   sp, 4
seg030:017F 89 46 FE     mov    [bp+var_2], ax
seg030:0182 8B C6         mov    ax, si
seg030:0184 6B C0 12     imul  ax, 18
seg030:0187 66 0F BF C0   movsx eax, ax
seg030:018B 66 8B 56 FA   mov    edx, [bp+var_6]
seg030:018F 66 03 D0     add   edx, eax
seg030:0192 66 89 16 D8+  mov    _expiration, edx
seg030:0197 8B DE         mov    bx, si
seg030:0199 6B DB 1B     imul  bx, 27
seg030:019C 8B 87 D5 3C   mov    ax, _Q._A[bx]
seg030:01A0 3B 46 FE     cmp    ax, [bp+var_2]
seg030:01A3 74 05         jz    short loc_35B0A
seg030:01A5 B8 01 00     mov    ax, 1
seg030:01A8 EB 02         jmp   short loc_35B0C
seg030:01AA
seg030:01AA loc_35B0A: ; CODE XREF: check_dongle+1Fj
seg030:01AA                 ; check_dongle+5Ej
seg030:01AA 33 C0         xor   ax, ax
seg030:01AC
seg030:01AC loc_35B0C: ; CODE XREF: check_dongle+63j
seg030:01AC 5E           pop   si
seg030:01AD C9           leave
seg030:01AE CB           retf
seg030:01AE check_dongle endp

```

Since the routine can be called very frequently, e.g., before the execution of each important software feature, and accessing the dongle is generally slow (because of the slow printer port and also slow [MCU](#) in the dongle), they probably added a way to skip some dongle checks, by checking the current time in the `biostime()` function.

The `get_rand()` function uses the standard C function:

```

seg030:01BF      get_rand proc far ; CODE XREF: check_dongle+25p
seg030:01BF
seg030:01BF      arg_0    = word ptr 6
seg030:01BF
seg030:01BF 55      push   bp
seg030:01C0 8B EC   mov    bp, sp
seg030:01C2 9A 3D 21 00+  call   _rand
seg030:01C7 66 0F BF C0   movsx eax, ax
seg030:01CB 66 0F BF 56+  movsx edx, [bp+arg_0]
seg030:01D0 66 0F AF C2   imul  eax, edx
seg030:01D4 66 BB 00 80+  mov    ebx, 8000h
seg030:01DA 66 99         cdq
seg030:01DC 66 F7 FB     idiv  ebx
seg030:01DF 5D           pop   bp
seg030:01E0 CB           retf
seg030:01E0 get_rand endp

```

So the text string is selected randomly, passed into the dongle, and then the result of the hashing is compared with the correct value.

The text strings seem to be constructed randomly as well, during software development.

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

And this is how the main dongle checking function is called:

```
seg033:087B 9A 45 01 96+    call    check_dongle
seg033:0880 0B C0            or     ax, ax
seg033:0882 74 62            jz     short OK
seg033:0884 83 3E 60 42+    cmp    word_620E0, 0
seg033:0889 75 5B            jnz    short OK
seg033:088B FF 06 60 42    inc    word_620E0
seg033:088F 1E              push   ds
seg033:0890 68 22 44        push   offset aTrupcRequiresA ; "This Software Requires a Software ↴
                                ↴ Lock\n"
seg033:0893 1E              push   ds
seg033:0894 68 60 E9        push   offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+    call   _strcpy
seg033:089C 83 C4 08        add    sp, 8
seg033:089F 1E              push   ds
seg033:08A0 68 42 44        push   offset aPleaseContactA ; "Please Contact ..."
seg033:08A3 1E              push   ds
seg033:08A4 68 60 E9        push   offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+    call   _strcat
```

Bypassing the dongle is easy, just force the `check_dongle()` function to always return 0.

For example, by inserting this code at its beginning:

```
mov ax,0
retf
```

The observant reader might recall that the `strcpy()` C function usually requires two pointers in its arguments, but we see that 4 values are passed:

```
seg033:088F 1E              push   ds
seg033:0890 68 22 44        push   offset aTrupcRequiresA ; "This Software ↴
                                ↴ Requires a Software Lock\n"
seg033:0893 1E              push   ds
seg033:0894 68 60 E9        push   offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+    call   _strcpy
seg033:089C 83 C4 08        add    sp, 8
```

This is related to MS-DOS’ memory model. You can read more about it here: [10.6 on page 992](#).

So as you may see, `strcpy()` and any other function that take pointer(s) in arguments work with 16-bit pairs.

Let’s get back to our example. `DS` is currently set to the data segment located in the executable, that is where the text string is stored.

In the `sent_pro()` function, each byte of the string is loaded at

`seg030:00EF`: the `LES` instruction loads the `ES:BX` pair simultaneously from the passed argument.

The `MOV` at `seg030:00F5` loads the byte from the memory at which the `ES:BX` pair points.

## 8.6 “QR9”: Rubik’s cube inspired amateur crypto-algorithm

Sometimes amateur cryptosystems appear to be pretty bizarre.

The author of this book was once asked to reverse engineer an amateur cryptoalgorithm of some data encryption utility, the source code for which was lost<sup>16</sup>.

Here is the listing exported from `IDA` for the original encryption utility:

```
.text:00541000 set_bit          proc near ; CODE XREF: rotate1+42
.text:00541000                  ; rotate2+42 ...
.text:00541000
.text:00541000 arg_0           = dword ptr 4
```

<sup>16</sup>He also got permission from the customer to publish the algorithm’s details

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:00541000 arg_4          = dword ptr  8
.text:00541000 arg_8          = dword ptr  0Ch
.text:00541000 arg_C          = byte ptr  10h
.text:00541000
.text:00541000                 mov     al, [esp+arg_C]
.text:00541004                 mov     ecx, [esp+arg_8]
.text:00541008                 push    esi
.text:00541009                 mov     esi, [esp+4+arg_0]
.text:0054100D                 test    al, al
.text:0054100F                 mov     eax, [esp+4+arg_4]
.text:00541013                 mov     dl, 1
.text:00541015                 jz    short loc_54102B
.text:00541017                 shl     dl, cl
.text:00541019                 mov     cl, cube64[eax+esi*8]
.text:00541020                 or      cl, dl
.text:00541022                 mov     cube64[eax+esi*8], cl
.text:00541029                 pop    esi
.text:0054102A                 retn
.text:0054102B
.text:0054102B loc_54102B:           ; CODE XREF: set_bit+15
.text:0054102B                 shl     dl, cl
.text:0054102D                 mov     cl, cube64[eax+esi*8]
.text:00541034                 not    dl
.text:00541036                 and    cl, dl
.text:00541038                 mov     cube64[eax+esi*8], cl
.text:0054103F                 pop    esi
.text:00541040                 retn
.text:00541040 set_bit            endp
.text:00541040
.text:00541041                 align 10h
.text:00541050
.text:00541050 ; ===== S U B R O U T I N E =====
.text:00541050
.text:00541050
.text:00541050 get_bit           proc near ; CODE XREF: rotate1+16
.text:00541050                 ; rotate2+16 ...
.text:00541050
.text:00541050 arg_0          = dword ptr  4
.text:00541050 arg_4          = dword ptr  8
.text:00541050 arg_8          = byte ptr  0Ch
.text:00541050
.text:00541050                 mov     eax, [esp+arg_4]
.text:00541054                 mov     ecx, [esp+arg_0]
.text:00541058                 mov     al, cube64[eax+ecx*8]
.text:0054105F                 mov     cl, [esp+arg_8]
.text:00541063                 shr    al, cl
.text:00541065                 and    al, 1
.text:00541067                 retn
.text:00541067 get_bit           endp
.text:00541067
.text:00541068                 align 10h
.text:00541070
.text:00541070 ; ===== S U B R O U T I N E =====
.text:00541070
.text:00541070
.text:00541070 rotate1            proc near ; CODE XREF: rotate_all_with_password+8E
.text:00541070
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0          = dword ptr  4
.text:00541070
.text:00541070                 sub    esp, 40h
.text:00541073                 push   ebx
.text:00541074                 push   ebp
.text:00541075                 mov    ebp, [esp+48h+arg_0]
.text:00541079                 push   esi
.text:0054107A                 push   edi
.text:0054107B                 xor    edi, edi ; EDI is loop1 counter
.text:0054107D                 lea    ebx, [esp+50h+internal_array_64]
.text:00541081 first_loop1_begin:        ; CODE XREF: rotate1+2E

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:00541081          xor    esi, esi ; ESI is loop2 counter
.text:00541083
.text:00541083 first_loop2_begin:           ; CODE XREF: rotate1+25
.text:00541083         push   ebp      ; arg_0
.text:00541084         push   esi
.text:00541085         push   edi
.text:00541086         call   get_bit
.text:0054108B         add    esp, 0Ch
.text:0054108E         mov    [ebx+esi], al ; store to internal array
.text:00541091         inc    esi
.text:00541092         cmp    esi, 8
.text:00541095         jl    short first_loop2_begin
.text:00541097         inc    edi
.text:00541098         add    ebx, 8
.text:0054109B         cmp    edi, 8
.text:0054109E         jl    short first_loop1_begin
.text:005410A0         lea    ebx, [esp+50h+internal_array_64]
.text:005410A4         mov    edi, 7 ; EDI is loop1 counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:           ; CODE XREF: rotate1+57
.text:005410A9         xor    esi, esi ; ESI is loop2 counter
.text:005410AB
.text:005410AB second_loop2_begin:           ; CODE XREF: rotate1+4E
.text:005410AB         mov    al, [ebx+esi] ; value from internal array
.text:005410AE         push   eax
.text:005410AF         push   ebp      ; arg_0
.text:005410B0         push   edi
.text:005410B1         push   esi
.text:005410B2         call   set_bit
.text:005410B7         add    esp, 10h
.text:005410BA         inc    esi      ; increment loop2 counter
.text:005410BB         cmp    esi, 8
.text:005410BE         jl    short second_loop2_begin
.text:005410C0         dec    edi      ; decrement loop2 counter
.text:005410C1         add    ebx, 8
.text:005410C4         cmp    edi, 0FFFFFFFh
.text:005410C7         jg    short second_loop1_begin
.text:005410C9         pop    edi
.text:005410CA         pop    esi
.text:005410CB         pop    ebp
.text:005410CC         pop    ebx
.text:005410CD         add    esp, 40h
.text:005410D0         retn
.text:005410D0 rotate1:                   endp
.text:005410D0
.text:005410D1         align 10h
.text:005410E0
.text:005410E0 ; ===== S U B R O U T I N E =====
.text:005410E0
.text:005410E0
.text:005410E0 rotate2 proc near           ; CODE XREF: rotate_all_with_password+7A
.text:005410E0
.text:005410E0 internal_array_64= byte ptr -40h
.text:005410E0 arg_0      = dword ptr 4
.text:005410E0
.text:005410E0         sub    esp, 40h
.text:005410E3         push   ebx
.text:005410E4         push   ebp
.text:005410E5         mov    ebp, [esp+48h+arg_0]
.text:005410E9         push   esi
.text:005410EA         push   edi
.text:005410EB         xor    edi, edi ; loop1 counter
.text:005410ED         lea    ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:           ; CODE XREF: rotate2+2E
.text:005410F1         xor    esi, esi ; loop2 counter
.text:005410F3
.text:005410F3 loc_5410F3:           ; CODE XREF: rotate2+25
.text:005410F3         push   esi      ; loop2
.text:005410F3         push   edi      ; loop1

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:005410F5          push    ebp      ; arg_0
.text:005410F6          call    get_bit
.text:005410FB          add     esp, 0Ch
.text:005410FE          mov     [ebx+esi], al ; store to internal array
.text:00541101          inc     esi      ; increment loop1 counter
.text:00541102          cmp     esi, 8
.text:00541105          jl    short loc_5410F3
.text:00541107          inc     edi      ; increment loop2 counter
.text:00541108          add     ebx, 8
.text:0054110B          cmp     edi, 8
.text:0054110E          jl    short loc_5410F1
.text:00541110          lea     ebx, [esp+50h+internal_array_64]
.text:00541114          mov     edi, 7 ; loop1 counter is initial state 7
.text:00541119          xor     esi, esi ; loop2 counter
.text:0054111B          xor     esi, esi ; CODE XREF: rotate2+57
.text:0054111B loc_54111B:
.text:0054111B          mov     al, [ebx+esi] ; get byte from internal array
.text:0054111E          push    eax
.text:0054111F          push    edi      ; loop1 counter
.text:00541120          push    esi      ; loop2 counter
.text:00541121          push    ebp      ; arg_0
.text:00541122          call    set_bit
.text:00541127          add     esp, 10h
.text:0054112A          inc     esi      ; increment loop2 counter
.text:0054112B          cmp     esi, 8
.text:0054112E          jl    short loc_54111B
.text:00541130          dec     edi      ; decrement loop2 counter
.text:00541131          add     ebx, 8
.text:00541134          cmp     edi, 0FFFFFFFh
.text:00541137          jg     short loc_541119
.text:00541139          pop     edi
.text:0054113A          pop     esi
.text:0054113B          pop     ebp
.text:0054113C          pop     ebx
.text:0054113D          add     esp, 40h
.text:00541140          retn
.text:00541140 rotate2
.text:00541140          endp
.text:00541140
.text:00541141          align 10h
.text:00541150
.text:00541150 ; ===== S U B R O U T I N E =====
.text:00541150
.text:00541150
.text:00541150 rotate3          proc near      ; CODE XREF: rotate_all_with_password+66
.text:00541150
.text:00541150 var_40          = byte ptr -40h
.text:00541150 arg_0           = dword ptr 4
.text:00541150
.text:00541150          sub     esp, 40h
.text:00541153          push    ebx
.text:00541154          push    ebp
.text:00541155          mov     ebp, [esp+48h+arg_0]
.text:00541159          push    esi
.text:0054115A          push    edi
.text:0054115B          xor     edi, edi
.text:0054115D          lea     ebx, [esp+50h+var_40]
.text:00541161
.text:00541161 loc_541161: ; CODE XREF: rotate3+2E
.text:00541161          xor     esi, esi
.text:00541163
.text:00541163 loc_541163: ; CODE XREF: rotate3+25
.text:00541163          push    esi
.text:00541164          push    ebp
.text:00541165          push    edi
.text:00541166          call    get_bit
.text:0054116B          add     esp, 0Ch
.text:0054116E          mov     [ebx+esi], al
.text:00541171          inc     esi

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:00541172          cmp    esi, 8
.text:00541175          jl     short loc_541163
.text:00541177          inc    edi
.text:00541178          add    ebx, 8
.text:0054117B          cmp    edi, 8
.text:0054117E          jl     short loc_541161
.text:00541180          xor    ebx, ebx
.text:00541182          lea    edi, [esp+50h+var_40]
.text:00541186
.text:00541186 loc_541186:      ; CODE XREF: rotate3+54
.text:00541186          mov    esi, 7
.text:0054118B loc_54118B:      ; CODE XREF: rotate3+4E
.text:0054118B          mov    al, [edi]
.text:0054118D          push   eax
.text:0054118E          push   ebx
.text:0054118F          push   ebp
.text:00541190          push   esi
.text:00541191          call   set_bit
.text:00541196          add    esp, 10h
.text:00541199          inc    edi
.text:0054119A          dec    esi
.text:0054119B          cmp    esi, 0FFFFFFFh
.text:0054119E          jg    short loc_54118B
.text:005411A0          inc    ebx
.text:005411A1          cmp    ebx, 8
.text:005411A4          jl     short loc_541186
.text:005411A6          pop    edi
.text:005411A7          pop    esi
.text:005411A8          pop    ebp
.text:005411A9          pop    ebx
.text:005411AA          add    esp, 40h
.text:005411AD          retn
.text:005411AD rotate3      endp
.text:005411AD
.align 10h
.text:005411B0
.text:005411B0 ; ===== S U B R O U T I N E =====
.text:005411B0
.text:005411B0
.text:005411B0 rotate_all_with_password proc near ; CODE XREF: crypt+1F
.text:005411B0                                     ; decrypt+36
.text:005411B0
.text:005411B0 arg_0           = dword ptr 4
.text:005411B0 arg_4           = dword ptr 8
.text:005411B0
.text:005411B0          mov    eax, [esp+arg_0]
.text:005411B4          push   ebp
.text:005411B5          mov    ebp, eax
.text:005411B7          cmp    byte ptr [eax], 0
.text:005411BA          jz    exit
.text:005411C0          push   ebx
.text:005411C1          mov    ebx, [esp+8+arg_4]
.text:005411C5          push   esi
.text:005411C6          push   edi
.text:005411C7
.text:005411C7 loop_begin:    ; CODE XREF: rotate_all_with_password+9F
.text:005411C7          movsx eax, byte ptr [ebp+0]
.text:005411CB          push   eax           ; C
.text:005411CC          call   _tolower
.text:005411D1          add    esp, 4
.text:005411D4          cmp    al, 'a'
.text:005411D6          jl    short next_character_in_password
.text:005411D8          cmp    al, 'z'
.text:005411DA          jg    short next_character_in_password
.text:005411DC          movsx ecx, al
.text:005411DF          sub    ecx, 'a'
.text:005411E2          cmp    ecx, 24
.text:005411E5          jle    short skip_subtracting
.text:005411E7          sub    ecx, 24

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:005411EA
.text:005411EA skip_subtracting:           ; CODE XREF: rotate_all_with_password+35
.text:005411EA                         mov    eax, 55555556h
.text:005411EF                         imul   ecx
.text:005411F1                         mov    eax, edx
.text:005411F3                         shr    eax, 1Fh
.text:005411F6                         add    edx, eax
.text:005411F8                         mov    eax, ecx
.text:005411FA                         mov    esi, edx
.text:005411FC                         mov    ecx, 3
.text:00541201                         cdq
.text:00541202                         idiv   ecx
.text:00541204                         sub    edx, 0
.text:00541207                         jz    short call_rotate1
.text:00541209                         dec    edx
.text:0054120A                         jz    short call_rotate2
.text:0054120C                         dec    edx
.text:0054120D                         jnz   short next_character_in_password
.text:0054120F                         test   ebx, ebx
.text:00541211                         jle    short next_character_in_password
.text:00541213                         mov    edi, ebx
.text:00541215
.text:00541215 call_rotate3:           ; CODE XREF: rotate_all_with_password+6F
.text:00541215 push    esi
.text:00541216 call    rotate3
.text:0054121B add     esp, 4
.text:0054121E dec     edi
.text:0054121F jnz    short call_rotate3
.text:00541221 jmp    short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:           ; CODE XREF: rotate_all_with_password+5A
.text:00541223 test   ebx, ebx
.text:00541225 jle    short next_character_in_password
.text:00541227 mov    edi, ebx
.text:00541229 loc_541229:           ; CODE XREF: rotate_all_with_password+83
.text:00541229 push    esi
.text:0054122A call    rotate2
.text:0054122F add     esp, 4
.text:00541232 dec     edi
.text:00541233 jnz    short loc_541229
.text:00541235 jmp    short next_character_in_password
.text:00541237 call_rotate1:          ; CODE XREF: rotate_all_with_password+57
.text:00541237 test   ebx, ebx
.text:00541239 jle    short next_character_in_password
.text:0054123B mov    edi, ebx
.text:0054123D loc_54123D:           ; CODE XREF: rotate_all_with_password+97
.text:0054123D push    esi
.text:0054123E call    rotate1
.text:00541243 add     esp, 4
.text:00541246 dec     edi
.text:00541247 jnz    short loc_54123D
.text:00541249 next_character_in_password: ; CODE XREF: rotate_all_with_password+26
.text:00541249                      ; rotate_all_with_password+2A ...
.text:00541249 mov    al, [ebp+1]
.text:0054124C inc    ebp
.text:0054124D test   al, al
.text:0054124F jnz    loop_begin
.text:00541255 pop    edi
.text:00541256 pop    esi
.text:00541257 pop    ebx
.text:00541258 exit:                 ; CODE XREF: rotate_all_with_password+A
.text:00541258 pop    ebp
.text:00541259 retn
.text:00541259 rotate_all_with_password endp
.text:00541259

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:0054125A          align 10h
.text:00541260
.text:00541260 ; ====== S U B R O U T I N E ======
.text:00541260
.text:00541260
.text:00541260 crypt      proc near    ; CODE XREF: crypt_file+8A
.text:00541260
.text:00541260 arg_0       = dword ptr  4
.text:00541260 arg_4       = dword ptr  8
.text:00541260 arg_8       = dword ptr  0Ch
.text:00541260
.text:00541260           push    ebx
.text:00541261           mov     ebx, [esp+4+arg_0]
.text:00541265           push    ebp
.text:00541266           push    esi
.text:00541267           push    edi
.text:00541268           xor    ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:    ; CODE XREF: crypt+41
.text:0054126A           mov     eax, [esp+10h+arg_8]
.text:0054126E           mov     ecx, 10h
.text:00541273           mov     esi, ebx
.text:00541275           mov     edi, offset cube64
.text:0054127A           push    1
.text:0054127C           push    eax
.text:0054127D           rep    movsd
.text:0054127F           call    rotate_all_with_password
.text:00541284           mov     eax, [esp+18h+arg_4]
.text:00541288           mov     edi, ebx
.text:0054128A           add    ebp, 40h
.text:0054128D           add    esp, 8
.text:00541290           mov     ecx, 10h
.text:00541295           mov     esi, offset cube64
.text:0054129A           add    ebx, 40h
.text:0054129D           cmp    ebp, eax
.text:0054129F           rep    movsd
.text:005412A1           jl    short loc_54126A
.text:005412A3           pop    edi
.text:005412A4           pop    esi
.text:005412A5           pop    ebp
.text:005412A6           pop    ebx
.text:005412A7           retn
.text:005412A7 crypt      endp
.text:005412A7
.text:005412A8          align 10h
.text:005412B0
.text:005412B0 ; ====== S U B R O U T I N E ======
.text:005412B0
.text:005412B0
.text:005412B0 ; int __cdecl decrypt(int, int, void *Src)
.text:005412B0 decrypt      proc near    ; CODE XREF: decrypt_file+99
.text:005412B0
.text:005412B0 arg_0       = dword ptr  4
.text:005412B0 arg_4       = dword ptr  8
.text:005412B0 Src         = dword ptr  0Ch
.text:005412B0
.text:005412B0           mov     eax, [esp+Src]
.text:005412B4           push    ebx
.text:005412B5           push    ebp
.text:005412B6           push    esi
.text:005412B7           push    edi
.text:005412B8           push    eax           ; Src
.text:005412B9           call    __strupd
.text:005412BE           push    eax           ; Str
.text:005412BF           mov    [esp+18h+Src], eax
.text:005412C3           call    __strrev
.text:005412C8           mov    ebx, [esp+18h+arg_0]
.text:005412CC           add    esp, 8
.text:005412CF           xor    ebp, ebp
.text:005412D1

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:005412D1 loc_5412D1:          ; CODE XREF: decrypt+58
.text:005412D1                 mov     ecx, 10h
.text:005412D6                 mov     esi, ebx
.text:005412D8                 mov     edi, offset cube64
.text:005412DD                 push    3
.text:005412DF                 rep     movsd
.text:005412E1                 mov     ecx, [esp+14h+Src]
.text:005412E5                 push    ecx
.text:005412E6                 call    rotate_all_with_password
.text:005412EB                 mov     eax, [esp+18h+arg_4]
.text:005412EF                 mov     edi, ebx
.text:005412F1                 add    ebp, 40h
.text:005412F4                 add    esp, 8
.text:005412F7                 mov     ecx, 10h
.text:005412FC                 mov     esi, offset cube64
.text:00541301                 add    ebx, 40h
.text:00541304                 cmp    ebp, eax
.text:00541306                 rep     movsd
.text:00541308                 jl    short loc_5412D1
.text:0054130A                 mov     edx, [esp+10h+Src]
.text:0054130E                 push    edx  ; Memory
.text:0054130F                 call    _free
.text:00541314                 add    esp, 4
.text:00541317                 pop    edi
.text:00541318                 pop    esi
.text:00541319                 pop    ebp
.text:0054131A                 pop    ebx
.text:0054131B                 retn
.text:0054131B decrypt        endp
.text:0054131B
.text:0054131C                 align 10h
.text:00541320
.text:00541320 ; ===== S U B R O U T I N E =====
.text:00541320
.text:00541320
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near    ; CODE XREF: _main+42
.text:00541320
.text:00541320 Str           = dword ptr 4
.text:00541320 Filename       = dword ptr 8
.text:00541320 password       = dword ptr 0Ch
.text:00541320
.text:00541320                 mov     eax, [esp+Str]
.text:00541324                 push   ebp
.text:00541325                 push   offset Mode ; "rb"
.text:0054132A                 push   eax      ; Filename
.text:0054132B                 call   _fopen      ; open file
.text:00541330                 mov     ebp, eax
.text:00541332                 add    esp, 8
.text:00541335                 test   ebp, ebp
.text:00541337                 jnz   short loc_541348
.text:00541339                 push   offset Format ; "Cannot open input file!\n"
.text:0054133E                 call   _printf
.text:00541343                 add    esp, 4
.text:00541346                 pop    ebp
.text:00541347                 retn
.text:00541348 loc_541348:          ; CODE XREF: crypt_file+17
.text:00541348                 push   ebx
.text:00541349                 push   esi
.text:0054134A                 push   edi
.text:0054134B                 push   2      ; Origin
.text:0054134D                 push   0      ; Offset
.text:0054134F                 push   ebp      ; File
.text:00541350                 call   _fseek
.text:00541355                 push   ebp      ; File
.text:00541356                 call   _ftell      ; get file size
.text:0054135B                 push   0      ; Origin
.text:0054135D                 push   0      ; Offset
.text:0054135F                 push   ebp      ; File

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:00541360          mov    [esp+2Ch+Str], eax
.text:00541364          call   _fseek      ; rewind to start
.text:00541369          mov    esi, [esp+2Ch+Str]
.text:0054136D          and   esi, 0FFFFFFC0h ; reset all lowest 6 bits
.text:00541370          add    esi, 40h     ; align size to 64-byte border
.text:00541373          push   esi        ; Size
.text:00541374          call   _malloc
.text:00541379          mov    ecx, esi
.text:0054137B          mov    ebx, eax     ; allocated buffer pointer -> to EBX
.text:0054137D          mov    edx, ecx
.text:0054137F          xor    eax, eax
.text:00541381          mov    edi, ebx
.text:00541383          push   ebp        ; File
.text:00541384          shr    ecx, 2
.text:00541387          rep    stosd
.text:00541389          mov    ecx, edx
.text:0054138B          push   1         ; Count
.text:0054138D          and    ecx, 3
.text:00541390          rep    stosb      ; memset (buffer, 0, aligned_size)
.text:00541392          mov    eax, [esp+38h+Str]
.text:00541396          push   eax        ; ElementSize
.text:00541397          push   ebx        ; DstBuf
.text:00541398          call   _fread      ; read file
.text:0054139D          push   ebp        ; File
.text:0054139E          call   _fclose
.text:005413A3          mov    ecx, [esp+44h+password]
.text:005413A7          push   ecx        ; password
.text:005413A8          push   esi        ; aligned size
.text:005413A9          push   ebx        ; buffer
.text:005413AA          call   crypt       ; do crypt
.text:005413AF          mov    edx, [esp+50h+Filename]
.text:005413B3          add    esp, 40h
.text:005413B6          push   offset aWb  ; "wb"
.text:005413BB          push   edx        ; Filename
.text:005413BC          call   _fopen
.text:005413C1          mov    edi, eax
.text:005413C3          push   edi        ; File
.text:005413C4          push   1         ; Count
.text:005413C6          push   3         ; Size
.text:005413C8          push   offset aQr9 ; "QR9"
.text:005413CD          call   _fwrite     ; write file signature
.text:005413D2          push   edi        ; File
.text:005413D3          push   1         ; Count
.text:005413D5          lea    eax, [esp+30h+Str]
.text:005413D9          push   4         ; Size
.text:005413DB          push   eax        ; Str
.text:005413DC          call   _fwrite     ; write original file size
.text:005413E1          push   edi        ; File
.text:005413E2          push   1         ; Count
.text:005413E4          push   esi        ; Size
.text:005413E5          push   ebx        ; Str
.text:005413E6          call   _fwrite     ; write encrypted file
.text:005413EB          push   edi        ; File
.text:005413EC          call   _fclose
.text:005413F1          push   ebx        ; Memory
.text:005413F2          call   _free
.text:005413F7          add    esp, 40h
.text:005413FA          pop    edi
.text:005413FB          pop    esi
.text:005413FC          pop    ebx
.text:005413FD          pop    ebp
.text:005413FE          retn
.text:005413FE crypt_file endp
.text:005413FF          align 10h
.text:00541400          ; ===== S U B R O U T I N E =====
.text:00541400
.text:00541400
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:00541400 decrypt_file    proc near           ; CODE XREF: _main+6E
.text:00541400
.text:00541400 Filename        = dword ptr 4
.text:00541400 arg_4          = dword ptr 8
.text:00541400 Src             = dword ptr 0Ch
.text:00541400
.text:00541400                 mov     eax, [esp+Filename]
.text:00541404                 push    ebx
.text:00541405                 push    ebp
.text:00541406                 push    esi
.text:00541407                 push    edi
.text:00541408                 push    offset aRb  ; "rb"
.text:0054140D                 push    eax      ; Filename
.text:0054140E                 call    _fopen
.text:00541413                 mov     esi, eax
.text:00541415                 add    esp, 8
.text:00541418                 test   esi, esi
.text:0054141A                 jnz    short loc_54142E
.text:0054141C                 push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421                 call    _printf
.text:00541426                 add    esp, 4
.text:00541429                 pop    edi
.text:0054142A                 pop    esi
.text:0054142B                 pop    ebp
.text:0054142C                 pop    ebx
.text:0054142D                 retn
.text:0054142E loc_54142E:          ; CODE XREF: decrypt_file+1A
.text:0054142E                 push    2      ; Origin
.text:00541430                 push    0      ; Offset
.text:00541432                 push    esi    ; File
.text:00541433                 call    _fseek
.text:00541438                 push    esi    ; File
.text:00541439                 call    _ftell
.text:0054143E                 push    0      ; Origin
.text:00541440                 push    0      ; Offset
.text:00541442                 push    esi    ; File
.text:00541443                 mov    ebp, eax
.text:00541445                 call    _fseek
.text:0054144A                 push    ebp    ; Size
.text:0054144B                 call    _malloc
.text:00541450                 push    esi    ; File
.text:00541451                 mov    ebx, eax
.text:00541453                 push    1      ; Count
.text:00541455                 push    ebp    ; ElementSize
.text:00541456                 push    ebx    ; DstBuf
.text:00541457                 call    _fread
.text:0054145C                 push    esi    ; File
.text:0054145D                 call    _fclose
.text:00541462                 add    esp, 34h
.text:00541465                 mov    ecx, 3
.text:0054146A                 mov    edi, offset aQr9_0 ; "QR9"
.text:0054146F                 mov    esi, ebx
.text:00541471                 xor    edx, edx
.text:00541473                 repe   cmpsb
.text:00541475                 jz    short loc_541489
.text:00541477                 push    offset aFileIsNotCrypt ; "File is not encrypted!\n"
.text:0054147C                 call    _printf
.text:00541481                 add    esp, 4
.text:00541484                 pop    edi
.text:00541485                 pop    esi
.text:00541486                 pop    ebp
.text:00541487                 pop    ebx
.text:00541488                 retn
.text:00541489 loc_541489:          ; CODE XREF: decrypt_file+75
.text:00541489                 mov    eax, [esp+10h+Src]
.text:0054148D                 mov    edi, [ebx+3]
.text:00541490                 add    ebp, 0FFFFFFF9h
.text:00541493                 lea    esi, [ebx+7]

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541496      push    eax      ; Src
.text:00541497      push    ebp      ; int
.text:00541498      push    esi      ; int
.text:00541499      call    decrypt
.text:0054149E      mov     ecx, [esp+1Ch+arg_4]
.text:005414A2      push    offset awb_0 ; "wb"
.text:005414A7      push    ecx      ; Filename
.text:005414A8      call    _fopen
.text:005414AD      mov     ebp, eax
.text:005414AF      push    ebp      ; File
.text:005414B0      push    1       ; Count
.text:005414B2      push    edi      ; Size
.text:005414B3      push    esi      ; Str
.text:005414B4      call    _fwrite
.text:005414B9      push    ebp      ; File
.text:005414BA      call    _fclose
.text:005414BF      push    ebx      ; Memory
.text:005414C0      call    _free
.text:005414C5      add    esp, 2Ch
.text:005414C8      pop    edi
.text:005414C9      pop    esi
.text:005414CA      pop    ebp
.text:005414CB      pop    ebx
.text:005414CC      retn
.text:005414CC decrypt_file    endp
```

All function and label names were given by me during the analysis.

Let's start from the top. Here is a function that takes two file names and password.

```
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near
.text:00541320
.text:00541320 Str           = dword ptr 4
.text:00541320 Filename      = dword ptr 8
.text:00541320 password      = dword ptr 0Ch
.text:00541320
```

Open the file and report if an error occurs:

```
.text:00541320      mov     eax, [esp+Str]
.text:00541324      push    ebp
.text:00541325      push    offset Mode      ; "rb"
.text:0054132A      push    eax      ; Filename
.text:0054132B      call    _fopen      ; open file
.text:00541330      mov     ebp, eax
.text:00541332      add    esp, 8
.text:00541335      test   ebp, ebp
.text:00541337      jnz    short loc_541348
.text:00541339      push    offset Format    ; "Cannot open input file!\n"
.text:0054133E      call    _printf
.text:00541343      add    esp, 4
.text:00541346      pop    ebp
.text:00541347      retn
.text:00541348      loc_541348:
```

Get the file size via `fseek()` / `ftell()`:

```
.text:00541348 push    ebx
.text:00541349 push    esi
.text:0054134A push    edi
.text:0054134B push    2       ; Origin
.text:0054134D push    0       ; Offset
.text:0054134F push    ebp      ; File

; move current file position to the end
.text:00541350 call    _fseek
.text:00541355 push    ebp      ; File
.text:00541356 call    _ftell      ; get current file position
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:0054135B push    0          ; Origin
.text:0054135D push    0          ; Offset
.text:0054135F push    ebp        ; File
.text:00541360 mov     [esp+2Ch+Str], eax

; move current file position to the start
.text:00541364 call    _fseek
```

This fragment of code calculates the file size aligned on a 64-byte boundary. This is because this cryptographic algorithm works with only 64-byte blocks. The operation is pretty straightforward: divide the file size by 64, forget about the remainder and add 1, then multiply by 64. The following code removes the remainder as if the value has already been divided by 64 and adds 64. It is almost the same.

```
.text:00541369 mov     esi, [esp+2Ch+Str]
; reset all lowest 6 bits
.text:0054136D and    esi, 0FFFFFFC0h
; align size to 64-byte border
.text:00541370 add    esi, 40h
```

Allocate buffer with aligned size:

```
.text:00541373          push    esi          ; Size
.text:00541374          call    _malloc
```

Call `memset()`, e.g., clear the allocated buffer<sup>17</sup>.

```
.text:00541379 mov     ecx, esi
.text:0054137B mov     ebx, eax      ; allocated buffer pointer -> to EBX
.text:0054137D mov     edx, ecx
.text:0054137F xor    eax, eax
.text:00541381 mov     edi, ebx
.text:00541383 push   ebp        ; File
.text:00541384 shr    ecx, 2
.text:00541387 rep    stosd
.text:00541389 mov     ecx, edx
.text:0054138B push   1          ; Count
.text:0054138D and    ecx, 3
.text:00541390 rep    stosb      ; memset (buffer, 0, aligned_size)
```

Read file via the standard C function `fread()`.

```
.text:00541392          mov    eax, [esp+38h+Str]
.text:00541396          push   eax          ; ElementSize
.text:00541397          push   ebx          ; DstBuf
.text:00541398          call   _fread        ; read file
.text:0054139D          push   ebp          ; File
.text:0054139E          call   _fclose
```

Call `crypt()`. This function takes a buffer, buffer size (aligned) and a password string.

```
.text:005413A3          mov    ecx, [esp+44h+password]
.text:005413A7          push   ecx          ; password
.text:005413A8          push   esi          ; aligned size
.text:005413A9          push   ebx          ; buffer
.text:005413AA          call   crypt         ; do crypt
```

Create the output file. By the way, the developer forgot to check if it has been created correctly! The file opening result is being checked, though.

```
.text:005413AF          mov    edx, [esp+50h+Filename]
.text:005413B3          add    esp, 40h
.text:005413B6          push   offset awb       ; "wb"
.text:005413BB          push   edx          ; Filename
.text:005413BC          call   _fopen
.text:005413C1          mov    edi, eax
```

The newly created file handle is in the `EDI` register now. Write signature “QR9”.

<sup>17</sup>`malloc() + memset()` could be replaced by `calloc()`

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:005413C3          push    edi          ; File
.text:005413C4          push    1           ; Count
.text:005413C6          push    3           ; Size
.text:005413C8          push    offset a0r9   ; "QR9"
.text:005413CD          call    _fwrite      ; write file signature
```

Write the actual file size (not aligned):

```
.text:005413D2          push    edi          ; File
.text:005413D3          push    1           ; Count
.text:005413D5          lea     eax, [esp+30h+Str]
.text:005413D9          push    4           ; Size
.text:005413DB          push    eax          ; Str
.text:005413DC          call    _fwrite      ; write original file size
```

Write the encrypted buffer:

```
.text:005413E1          push    edi          ; File
.text:005413E2          push    1           ; Count
.text:005413E4          push    esi          ; Size
.text:005413E5          push    ebx          ; Str
.text:005413E6          call    _fwrite      ; write encrypted file
```

Close the file and free the allocated buffer:

```
.text:005413EB          push    edi          ; File
.text:005413EC          call    _fclose      ; Memory
.text:005413F1          push    ebx          ; Memory
.text:005413F2          call    _free        ; Memory
.text:005413F7          add     esp, 40h
.text:005413FA          pop    edi
.text:005413FB          pop    esi
.text:005413FC          pop    ebx
.text:005413FD          pop    ebp
.text:005413FE          retn
.text:005413FE crypt_file endp
```

Here is the reconstructed C code:

```
void crypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int flen, flen_aligned;
    BYTE *buf;

    f=fopen(fin, "rb");

    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    };

    fseek (f, 0, SEEK_END);
    flen=f.tell (f);
    fseek (f, 0, SEEK_SET);

    flen_aligned=(flen&0xFFFFFC0)+0x40;

    buf=(BYTE*)malloc (flen_aligned);
    memset (buf, 0, flen_aligned);

    fread (buf, flen, 1, f);

    fclose (f);

    crypt (buf, flen_aligned, pw);

    f=fopen(fout, "wb");
```

```

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);
};

```

The decryption procedure is almost the same:

```

.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file    proc near
.text:00541400
.text:00541400 Filename      = dword ptr  4
.text:00541400 arg_4        = dword ptr  8
.text:00541400 Src          = dword ptr  0Ch
.text:00541400
.text:00541400             mov     eax, [esp+Filename]
.text:00541404             push    ebx
.text:00541405             push    ebp
.text:00541406             push    esi
.text:00541407             push    edi
.text:00541408             push    offset aRb      ; "rb"
.text:0054140D             push    eax           ; Filename
.text:0054140E             call    _fopen
.text:00541413             mov     esi, eax
.text:00541415             add    esp, 8
.text:00541418             test   esi, esi
.text:0054141A             jnz    short loc_54142E
.text:0054141C             push   offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421             call    _printf
.text:00541426             add    esp, 4
.text:00541429             pop    edi
.text:0054142A             pop    esi
.text:0054142B             pop    ebp
.text:0054142C             pop    ebx
.text:0054142D             retn
.text:0054142E
.text:0054142E loc_54142E:
.text:0054142E             push   2          ; Origin
.text:00541430             push   0          ; Offset
.text:00541432             push   esi         ; File
.text:00541433             call    _fseek
.text:00541438             push   esi         ; File
.text:00541439             call    _ftell
.text:0054143E             push   0          ; Origin
.text:00541440             push   0          ; Offset
.text:00541442             push   esi         ; File
.text:00541443             mov    ebp, eax
.text:00541445             call    _fseek
.text:0054144A             push   ebp         ; Size
.text:0054144B             call    _malloc
.text:00541450             push   esi         ; File
.text:00541451             mov    ebx, eax
.text:00541453             push   1          ; Count
.text:00541455             push   ebp         ; ElementSize
.text:00541456             push   ebx         ; DstBuf
.text:00541457             call    _fread
.text:0054145C             push   esi         ; File
.text:0054145D             call    _fclose

```

Check signature (first 3 bytes):

```

.text:00541462             add    esp, 34h
.text:00541465             mov    ecx, 3
.text:0054146A             mov    edi, offset aQr9_0 ; "QR9"
.text:0054146F             mov    esi, ebx
.text:00541471             xor    edx, edx

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541473        repe  cmpsb  
.text:00541475        jz     short loc_541489
```

Report an error if the signature is absent:

```
.text:00541477        push   offset aFileIsNotCrypt ; "File is not encrypted!\n"  
.text:0054147C        call    _printf  
.text:00541481        add    esp, 4  
.text:00541484        pop    edi  
.text:00541485        pop    esi  
.text:00541486        pop    ebp  
.text:00541487        pop    ebx  
.text:00541488        retn  
.text:00541489 loc_541489:
```

Call decrypt().

```
.text:00541489        mov    eax, [esp+10h+Src]  
.text:0054148D        mov    edi, [ebx+3]  
.text:00541490        add    ebp, 0FFFFFFF9h  
.text:00541493        lea    esi, [ebx+7]  
.text:00541496        push   eax          ; Src  
.text:00541497        push   ebp          ; int  
.text:00541498        push   esi          ; int  
.text:00541499        call   decrypt  
.text:0054149E        mov    ecx, [esp+1Ch+arg_4]  
.text:005414A2        push   offset aWb_0    ; "wb"  
.text:005414A7        push   ecx          ; Filename  
.text:005414A8        call   _fopen  
.text:005414AD        mov    ebp, eax  
.text:005414AF        push   ebp          ; File  
.text:005414B0        push   1           ; Count  
.text:005414B2        push   edi          ; Size  
.text:005414B3        push   esi          ; Str  
.text:005414B4        call   _fwrite  
.text:005414B9        push   ebp          ; File  
.text:005414BA        call   _fclose  
.text:005414BF        push   ebx          ; Memory  
.text:005414C0        call   _free  
.text:005414C5        add    esp, 2Ch  
.text:005414C8        pop    edi  
.text:005414C9        pop    esi  
.text:005414CA        pop    ebp  
.text:005414CB        pop    ebx  
.text:005414CC        retn  
.text:005414CC decrypt_file endp
```

Here is the reconstructed C code:

```
void decrypt_file(char *fin, char* fout, char *pw)  
{  
    FILE *f;  
    int real_flen,flen;  
    BYTE *buf;  
  
    f=fopen(fin, "rb");  
  
    if (f==NULL)  
    {  
        printf ("Cannot open input file!\n");  
        return;  
    };  
  
    fseek (f, 0, SEEK_END);  
   flen=f.tell (f);  
    fseek (f, 0, SEEK_SET);  
  
    buf=(BYTE*)malloc (flen);
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

fread (buf, flen, 1, f);

fclose (f);

if (memcmp (buf, "QR9", 3)!=0)
{
    printf ("File is not encrypted!\n");
    return;
};

memcpy (&real_flen, buf+3, 4);

decrypt (buf+(3+4), flen-(3+4), pw);

f=fopen(fout, "wb");

fwrite (buf+(3+4), real_flen, 1, f);

fclose (f);

free (buf);
};

```

OK, now let's go deeper.

Function `crypt()`:

```

.text:00541260 crypt          proc near
.text:00541260
.text:00541260 arg_0          = dword ptr 4
.text:00541260 arg_4          = dword ptr 8
.text:00541260 arg_8          = dword ptr 0Ch
.text:00541260
.text:00541260                 push    ebx
.text:00541261                 mov     ebx, [esp+4+arg_0]
.text:00541265                 push    ebp
.text:00541266                 push    esi
.text:00541267                 push    edi
.text:00541268                 xor    ebp, ebp
.text:0054126A loc_54126A:

```

This fragment of code copies a part of the input buffer to an internal array we later name “cube64”. The size is in the `ECX` register. `MOVSD` stands for *move 32-bit dword*, so, 16 32-bit dwds are exactly 64 bytes.

```

.text:0054126A                 mov    eax, [esp+10h+arg_8]
.text:0054126E                 mov    ecx, 10h
.text:00541273                 mov    esi, ebx ; EBX is pointer within input buffer
.text:00541275                 mov    edi, offset cube64
.text:0054127A                 push   1
.text:0054127C                 push   eax
.text:0054127D                 rep    movsd

```

Call `rotate_all_with_password()`:

```
.text:0054127F                 call   rotate_all_with_password
```

Copy encrypted contents back from “cube64” to buffer:

```

.text:00541284                 mov    eax, [esp+18h+arg_4]
.text:00541288                 mov    edi, ebx
.text:0054128A                 add    ebp, 40h
.text:0054128D                 add    esp, 8
.text:00541290                 mov    ecx, 10h
.text:00541295                 mov    esi, offset cube64
.text:0054129A                 add    ebx, 40h ; add 64 to input buffer pointer
.text:0054129D                 cmp    ebp, eax ; EBP = amount of encrypted data.
.text:0054129F                 rep    movsd

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

If EBP is not bigger than the size input argument, then continue to the next block.

```
.text:005412A1          jl    short loc_54126A
.text:005412A3          pop   edi
.text:005412A4          pop   esi
.text:005412A5          pop   ebp
.text:005412A6          pop   ebx
.text:005412A7          retn
.text:005412A7 crypt    endp
```

Reconstructed `crypt()` function:

```
void crypt (BYTE *buf, int sz, char *pw)
{
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (pw, 1);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);
}
```

OK, now let's go deeper in function `rotate_all_with_password()`. It takes two arguments: password string and a number.

In `crypt()`, the number 1 is used, and in the `decrypt()` function (where `rotate_all_with_password()` function is called too), the number is 3.

```
.text:005411B0 rotate_all_with_password proc near
.text:005411B0
.text:005411B0 arg_0          = dword ptr 4
.text:005411B0 arg_4          = dword ptr 8
.text:005411B0
.text:005411B0             mov    eax, [esp+arg_0]
.text:005411B4             push   ebp
.text:005411B5             mov    ebp, eax
```

Check the current character in the password. If it is zero, exit:

```
.text:005411B7          cmp    byte ptr [eax], 0
.text:005411BA          jz    exit
.text:005411C0          push   ebx
.text:005411C1          mov    ebx, [esp+8+arg_4]
.text:005411C5          push   esi
.text:005411C6          push   edi
.text:005411C7          loop_begin:
```

Call `tolower()`, a standard C function.

```
.text:005411C7          movsx  eax, byte ptr [ebp+0]
.text:005411CB          push   eax           ; C
.text:005411CC          call   _tolower
.text:005411D1          add    esp, 4
```

Hmm, if the password has non-Latin character, it is skipped! Indeed, when we run the encryption utility and try non-Latin characters in the password, they seem to be ignored.

```
.text:005411D4          cmp    al, 'a'
.text:005411D6          jl    short next_character_in_password
.text:005411D8          cmp    al, 'z'
.text:005411DA          jg    short next_character_in_password
.text:005411DC          movsx  ecx, al
```

Subtract the value of “a” (97) from the character.

```
.text:005411DF          sub     ecx, 'a' ; 97
```

After subtracting, we’ll get 0 for “a” here, 1 for “b”, etc. And 25 for “z”.

```
.text:005411E2          cmp     ecx, 24
.text:005411E5          jle     short skip_subtracting
.text:005411E7          sub     ecx, 24
```

It seems, “y” and “z” are exceptional characters too. After that fragment of code, “y” becomes 0 and “z”—1. This implies that the 26 Latin alphabet symbols become values in the range of 0..23, (24 in total).

```
.text:005411EA          ; CODE XREF: rotate_all_with_password+35
.text:005411EA skip_subtracting:
```

This is actually division via multiplication. You can read more about it in the “Division using multiplication” section ([3.9 on page 496](#)).

The code actually divides the password character’s value by 3.

```
.text:005411EA          mov     eax, 55555556h
.text:005411EF          imul    ecx
.text:005411F1          mov     eax, edx
.text:005411F3          shr     eax, 1Fh
.text:005411F6          add     edx, eax
.text:005411F8          mov     eax, ecx
.text:005411FA          mov     esi, edx
.text:005411FC          mov     ecx, 3
.text:00541201          cdq
.text:00541202          idiv    ecx
```

**EDX** is the remainder of the division.

```
.text:00541204 sub     edx, 0
.text:00541207 jz     short call_rotate1 ; if remainder is zero, go to rotate1
.text:00541209 dec    edx
.text:0054120A jz     short call_rotate2 ; .. if it is 1, go to rotate2
.text:0054120C dec    edx
.text:0054120D jnz   short next_character_in_password
.text:0054120F test   ebx, ebx
.text:00541211 jle   short next_character_in_password
.text:00541213 mov    edi, ebx
```

If the remainder is 2, call `rotate3()`. **EDI** is the second argument of the `rotate_all_with_password()` function. As we already noted, 1 is for the encryption operations and 3 is for the decryption. So, here is a loop. When encrypting, `rotate1/2/3` are to be called the same number of times as given in the first argument.

```
.text:00541215 call_rotate3:
.text:00541215          push    esi
.text:00541216          call    rotate3
.text:0054121B          add    esp, 4
.text:0054121E          dec    edi
.text:0054121F          jnz   short call_rotate3
.text:00541221          jmp   short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:
.text:00541223          test   ebx, ebx
.text:00541225          jle   short next_character_in_password
.text:00541227          mov    edi, ebx
.text:00541229
.text:00541229 loc_541229:
.text:00541229          push    esi
.text:0054122A          call    rotate2
.text:0054122F          add    esp, 4
.text:00541232          dec    edi
.text:00541233          jnz   short loc_541229
.text:00541235          jmp   short next_character_in_password
.text:00541237
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541237 call_rotate1:  
.text:00541237          test    ebx, ebx  
.text:00541239          jle     short next_character_in_password  
.text:0054123B          mov     edi, ebx  
.text:0054123D  
.text:0054123D loc_54123D:  
.text:0054123D          push    esi  
.text:0054123E          call    rotate1  
.text:00541243          add    esp, 4  
.text:00541246          dec    edi  
.text:00541247          jnz    short loc_54123D  
.text:00541249
```

Fetch the next character from the password string.

```
.text:00541249 next_character_in_password:  
.text:00541249          mov     al, [ebp+1]
```

Increment the character pointer in the password string:

```
.text:0054124C          inc    ebp  
.text:0054124D          test   al, al  
.text:0054124F          jnz    loop_begin  
.text:00541255          pop    edi  
.text:00541256          pop    esi  
.text:00541257          pop    ebx  
.text:00541258  
.text:00541258 exit:  
.text:00541258          pop    ebp  
.text:00541259          retn  
.text:00541259 rotate_all_with_password endp
```

Here is the reconstructed C code:

```
void rotate_all (char *pwd, int v)  
{  
    char *p=pwd;  
  
    while (*p)  
    {  
        char c=*p;  
        int q;  
  
        c=tolower (c);  
  
        if (c>='a' && c<='z')  
        {  
            q=c-'a';  
            if (q>24)  
                q-=24;  
  
            int quotient=q/3;  
            int remainder=q % 3;  
  
            switch (remainder)  
            {  
                case 0: for (int i=0; i<v; i++) rotate1 (quotient); break;  
                case 1: for (int i=0; i<v; i++) rotate2 (quotient); break;  
                case 2: for (int i=0; i<v; i++) rotate3 (quotient); break;  
            };  
        };  
        p++;  
    };
```

Now let’s go deeper and investigate the rotate1/2/3 functions. Each function calls another two functions. We eventually will name them `set_bit()` and `get_bit()`.

Let’s start with `get_bit()`:

```
.text:00541050 get_bit      proc near
.text:00541050
.text:00541050 arg_0        = dword ptr 4
.text:00541050 arg_4        = dword ptr 8
.text:00541050 arg_8        = byte ptr 0Ch
.text:00541050
.text:00541050             mov    eax, [esp+arg_4]
.text:00541054             mov    ecx, [esp+arg_0]
.text:00541058             mov    al, cube64[eax+ecx*8]
.text:0054105F             mov    cl, [esp+arg_8]
.text:00541063             shr    al, cl
.text:00541065             and    al, 1
.text:00541067             retn
.text:00541067 get_bit      endp
```

...in other words: calculate an index in the cube64 array: *arg\_4* + *arg\_0* \* 8. Then shift a byte from the array by *arg\_8* bits right. Isolate the lowest bit and return it.

Let's see another function, *set\_bit()*:

```
.text:00541000 set_bit      proc near
.text:00541000
.text:00541000 arg_0        = dword ptr 4
.text:00541000 arg_4        = dword ptr 8
.text:00541000 arg_8        = dword ptr 0Ch
.text:00541000 arg_C        = byte ptr 10h
.text:00541000
.text:00541000             mov    al, [esp+arg_C]
.text:00541004             mov    ecx, [esp+arg_8]
.text:00541008             push   esi
.text:00541009             mov    esi, [esp+4+arg_0]
.text:0054100D             test   al, al
.text:0054100F             mov    eax, [esp+4+arg_4]
.text:00541013             mov    dl, 1
.text:00541015             jz    short loc_54102B
```

The value in the **DL** is 1 here. It gets shifted left by *arg\_8*. For example, if *arg\_8* is 4, the value in the **DL** register is to be 0x10 or 1000b in binary form.

```
.text:00541017             shl    dl, cl
.text:00541019             mov    cl, cube64[eax+esi*8]
```

Get a bit from array and explicitly set it.

```
.text:00541020             or     cl, dl
```

Store it back:

```
.text:00541022             mov    cube64[eax+esi*8], cl
.text:00541029             pop    esi
.text:0054102A             retn
.text:0054102B
.text:0054102B loc_54102B:
.text:0054102B             shl    dl, cl
```

If *arg\_C* is not zero...

```
.text:0054102D             mov    cl, cube64[eax+esi*8]
```

...invert DL. For example, if DL's state after the shift is 0x10 or 0b1000, there is 0xEF to be after the **NOT** instruction (or 0b11101111b).

```
.text:00541034             not    dl
```

This instruction clears the bit, in other words, it saves all bits in **CL** which are also set in **DL** except those in **DL** which are cleared. This implies that if **DL** is 11101111b in binary form, all bits are to be saved except the 5th (counting from lowest bit).

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541036          and     cl, dl
```

Store it back:

```
.text:00541038          mov     cube64[eax+esi*8], cl
.text:0054103F          pop    esi
.text:00541040          retn
.text:00541040  set_bit  endp
```

It is almost the same as `get_bit()`, except, if `arg_C` is zero, the function clears the specific bit in the array, or sets it otherwise.

We also know that the array’s size is 64. The first two arguments both in the `set_bit()` and `get_bit()` functions could be seen as 2D coordinates. Then the array is to be an 8\*8 matrix.

Here is a C representation of what we know up to now:

```
#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int shift, int bit)
{
    if (bit)
        SET_BIT (cube[x][y], 1<<shift);
    else
        REMOVE_BIT (cube[x][y], 1<<shift);
};

bool get_bit (int x, int y, int shift)
{
    if ((cube[x][y]>>shift)&1==1)
        return 1;
    return 0;
};
```

Now let’s get back to the `rotate1/2/3` functions.

```
.text:00541070 rotate1      proc near
.text:00541070
```

Internal array allocation in the local stack, with size of 64 bytes:

```
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0         = dword ptr 4
.text:00541070
.text:00541070          sub     esp, 40h
.text:00541073          push    ebx
.text:00541074          push    ebp
.text:00541075          mov     ebp, [esp+48h+arg_0]
.text:00541079          push    esi
.text:0054107A          push    edi
.text:0054107B          xor     edi, edi      ; EDI is loop1 counter
```

`EBX` is a pointer to the internal array:

```
.text:0054107D          lea     ebx, [esp+50h+internal_array_64]
.text:00541081
```

Here we have two nested loops:

```
.text:00541081 first_loop1_begin:
.text:00541081      xor     esi, esi      ; ESI is loop 2 counter
.text:00541083
.text:00541083 first_loop2_begin:
.text:00541083      push    ebp      ; arg_0
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541084    push    esi          ; loop 1 counter
.text:00541085    push    edi          ; loop 2 counter
.text:00541086    call    get_bit
.text:0054108B    add     esp, 0Ch
.text:0054108E    mov     [ebx+esi], al ; store to internal array
.text:00541091    inc     esi          ; increment loop 1 counter
.text:00541092    cmp     esi, 8
.text:00541095    jl     short first_loop2_begin
.text:00541097    inc     edi          ; increment loop 2 counter

; increment internal array pointer by 8 at each loop 1 iteration
.text:00541098    add     ebx, 8
.text:0054109B    cmp     edi, 8
.text:0054109E    jl     short first_loop1_begin
```

...we see that both loops' counters are in the range of 0..7. Also they are used as the first and second argument for the `get_bit()` function. The third argument to `get_bit()` is the only argument of `rotate1()`. The return value from `get_bit()` is placed in the internal array.

Prepare a pointer to the internal array again:

```
.text:005410A0    lea     ebx, [esp+50h+internal_array_64]
.text:005410A4    mov     edi, 7      ; EDI is loop 1 counter, initial state is 7
.text:005410A9
.text:005410A9    second_loop1_begin:
.text:005410A9    xor     esi, esi  ; ESI is loop 2 counter
.text:005410AB
.text:005410AB    second_loop2_begin:
.text:005410AB    mov     al, [ebx+esi] ; value from internal array
.text:005410AE    push    eax
.text:005410AF    push    ebp          ; arg_0
.text:005410B0    push    edi          ; loop 1 counter
.text:005410B1    push    esi          ; loop 2 counter
.text:005410B2    call    set_bit
.text:005410B7    add     esp, 10h
.text:005410BA    inc     esi          ; increment loop 2 counter
.text:005410BB    cmp     esi, 8
.text:005410BE    jl     short second_loop2_begin
.text:005410C0    dec     edi          ; decrement loop 2 counter
.text:005410C1    add     ebx, 8      ; increment pointer in internal array
.text:005410C4    cmp     edi, 0FFFFFFFh
.text:005410C7    jg     short second_loop1_begin
.text:005410C9    pop     edi
.text:005410CA    pop     esi
.text:005410CB    pop     ebp
.text:005410CC    pop     ebx
.text:005410CD    add     esp, 40h
.text:005410D0    retn
.text:005410D0    rotate1           endp
```

...this code is placing the contents of the internal array to the cube global array via the `set_bit()` function, *but* in a different order! Now the counter of the first loop is in the range of 7 to 0, **decrementing** at each iteration!

The C code representation looks like:

```
void rotate1 (int v)
{
    bool tmp[8][8]; // internal array
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (i, j, v);

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            set_bit (j, 7-i, v, tmp[x][y]);
};
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

Not very understandable, but if we take a look at `rotate2()` function:

```
.text:005410E0 rotate2 proc near
.text:005410E0
.text:005410E0 internal_array_64 = byte ptr -40h
.text:005410E0 arg_0 = dword ptr 4
.text:005410E0
.text:005410E0     sub    esp, 40h
.text:005410E3     push   ebx
.text:005410E4     push   ebp
.text:005410E5     mov    ebp, [esp+48h+arg_0]
.text:005410E9     push   esi
.text:005410EA     push   edi
.text:005410EB     xor    edi, edi ; loop 1 counter
.text:005410ED     lea    ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:
.text:005410F1     xor    esi, esi ; loop 2 counter
.text:005410F3
.text:005410F3 loc_5410F3:
.text:005410F3     push   esi ; loop 2 counter
.text:005410F4     push   edi ; loop 1 counter
.text:005410F5     push   ebp ; arg_0
.text:005410F6     call   get_bit
.text:005410FB     add    esp, 0Ch
.text:005410FE     mov    [ebx+esi], al ; store to internal array
.text:00541101     inc    esi ; increment loop 1 counter
.text:00541102     cmp    esi, 8
.text:00541105     jl    short loc_5410F3
.text:00541107     inc    edi ; increment loop 2 counter
.text:00541108     add    ebx, 8
.text:0054110B     cmp    edi, 8
.text:0054110E     jl    short loc_5410F1
.text:00541110     lea    ebx, [esp+50h+internal_array_64]
.text:00541114     mov    edi, 7 ; loop 1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:
.text:00541119     xor    esi, esi ; loop 2 counter
.text:0054111B
.text:0054111B loc_54111B:
.text:0054111B     mov    al, [ebx+esi] ; get byte from internal array
.text:0054111E     push   eax
.text:0054111F     push   edi ; loop 1 counter
.text:00541120     push   esi ; loop 2 counter
.text:00541121     push   ebp ; arg_0
.text:00541122     call   set_bit
.text:00541127     add    esp, 10h
.text:0054112A     inc    esi ; increment loop 2 counter
.text:0054112B     cmp    esi, 8
.text:0054112E     jl    short loc_54111B
.text:00541130     dec    edi ; decrement loop 2 counter
.text:00541131     add    ebx, 8
.text:00541134     cmp    edi, 0xFFFFFFFFh
.text:00541137     jg    short loc_541119
.text:00541139     pop    edi
.text:0054113A     pop    esi
.text:0054113B     pop    ebp
.text:0054113C     pop    ebx
.text:0054113D     add    esp, 40h
.text:00541140     retn
.text:00541140 rotate2 endp
```

It is *almost* the same, except the order of the arguments of the `get_bit()` and `set_bit()` is different.  
Let's rewrite it in C-like code:

```
void rotate2 (int v)
{
    bool tmp[8][8]; // internal array
    int i, j;
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

for (i=0; i<8; i++)
    for (j=0; j<8; j++)
        tmp[i][j]=get_bit (v, i, j);

for (i=0; i<8; i++)
    for (j=0; j<8; j++)
        set_bit (v, j, 7-i, tmp[i][j]);
};

}

```

Let's also rewrite the `rotate3()` function:

```

void rotate3 (int v)
{
    bool tmp[8][8];
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (i, v, j);

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            set_bit (7-j, v, i, tmp[i][j]);
};

```

Well, now things are simpler. If we consider cube64 as a 3D cube of size 8\*8\*8, where each element is a bit, `get_bit()` and `set_bit()` take just the coordinates of a bit as input.

The `rotate1/2/3` functions are in fact rotating all bits in a specific plane. These three functions are one for each cube side and the `v` argument sets the plane in the range of 0..7.

Maybe the algorithm’s author was thinking of a 8\*8\*8 Rubik’s cube <sup>18</sup>?

Yes, indeed.

Let's look closer into the `decrypt()` function, here is its rewritten version:

```

void decrypt (BYTE *buf, int sz, char *pw)
{
    char *p=strdup (pw);
    strrev (p);
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (p, 3);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);

    free (p);
};

```

It is almost the same as for `crypt()`, but the password string is reversed by the `strrev()` standard C function and `rotate_all()` is called with argument 3.

This implies that in case of decryption, each corresponding `rotate1/2/3` call is to be performed thrice.

This is almost as in Rubik’s cube! If you want to get back, do the same in reverse order and direction! If you want to undo the effect of rotating one place in clockwise direction, rotate it once in counter-clockwise direction, or thrice in clockwise direction.

`rotate1()` is apparently for rotating the “front” plane. `rotate2()` is apparently for rotating the “top” plane. `rotate3()` is apparently for rotating the “left” plane.

<sup>18</sup>[wikipedia](#)

<sup>19</sup>[MSDN](#)

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

Let's get back to the core of the `rotate_all()` function:

```
q=c-'a';
if (q>24)
    q-=24;

int quotient=q/3; // in range 0..7
int remainder=q % 3;

switch (remainder)
{
    case 0: for (int i=0; i<v; i++) rotate1 (quotient); break; // front
    case 1: for (int i=0; i<v; i++) rotate2 (quotient); break; // top
    case 2: for (int i=0; i<v; i++) rotate3 (quotient); break; // left
};
```

Now it is much simpler to understand: each password character defines a side (one of three) and a plane (one of 8).  $3 \times 8 = 24$ , that is why two the last two characters of the Latin alphabet are remapped to fit an alphabet of exactly 24 elements.

The algorithm is clearly weak: in case of short passwords you can see that in the encrypted file there are the original bytes of the original file in a binary file editor.

Here is the whole source code reconstructed:

```
#include <windows.h>

#include <stdio.h>
#include <assert.h>

#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)      ((var) |= (bit))
#define REMOVE_BIT(var, bit)   ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int z, bool bit)
{
    if (bit)
        SET_BIT (cube[x][y], 1<<z);
    else
        REMOVE_BIT (cube[x][y], 1<<z);
};

bool get_bit (int x, int y, int z)
{
    if ((cube[x][y]>>z)&1==1)
        return true;
    return false;
};

void rotate_f (int row)
{
    bool tmp[8][8];
    int x, y;

    for (x=0; x<8; x++)
        for (y=0; y<8; y++)
            tmp[x][y]=get_bit (x, y, row);

    for (x=0; x<8; x++)
        for (y=0; y<8; y++)
            set_bit (y, 7-x, row, tmp[x][y]);
};

void rotate_t (int row)
{
    bool tmp[8][8];
    int y, z;

    for (y=0; y<8; y++)
        for (z=0; z<8; z++)
            tmp[y][z]=get_bit (y, z, row);
}
```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

        for (z=0; z<8; z++)
            tmp[y][z]=get_bit (row, y, z);

        for (y=0; y<8; y++)
            for (z=0; z<8; z++)
                set_bit (row, z, 7-y, tmp[y][z]);
};

void rotate_l (int row)
{
    bool tmp[8][8];
    int x, z;

    for (x=0; x<8; x++)
        for (z=0; z<8; z++)
            tmp[x][z]=get_bit (x, row, z);

    for (x=0; x<8; x++)
        for (z=0; z<8; z++)
            set_bit (7-z, row, x, tmp[x][z]);
};

void rotate_all (char *pwd, int v)
{
    char *p=pwd;

    while (*p)
    {
        char c=*p;
        int q;

        c=tolower (c);

        if (c>='a' && c<='z')
        {
            q=c-'a';
            if (q>24)
                q-=24;

            int quotient=q/3;
            int remainder=q % 3;

            switch (remainder)
            {
                case 0: for (int i=0; i<v; i++) rotate_f (quotient); break;
                case 1: for (int i=0; i<v; i++) rotate_t (quotient); break;
                case 2: for (int i=0; i<v; i++) rotate_l (quotient); break;
            };
        };

        p++;
    };
};

void crypt (BYTE *buf, int sz, char *pw)
{
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (pw, 1);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);
};

void decrypt (BYTE *buf, int sz, char *pw)
{

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
char *p=strdup (pw);
strrev (p);
int i=0;

do
{
    memcpy (cube, buf+i, 8*8);
    rotate_all (p, 3);
    memcpy (buf+i, cube, 8*8);
    i+=64;
}
while (i<sz);

free (p);
};

void crypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int flen, flen_aligned;
    BYTE *buf;

    f=fopen(fin, "rb");

    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    };

    fseek (f, 0, SEEK_END);
    flen=f.tell (f);
    fseek (f, 0, SEEK_SET);

    flen_aligned=(flen&0xFFFFFC0)+0x40;

    buf=(BYTE*)malloc (flen_aligned);
    memset (buf, 0, flen_aligned);

    fread (buf, flen, 1, f);

    fclose (f);

    crypt (buf, flen_aligned, pw);

    f=fopen(fout, "wb");

    fwrite ("QR9", 3, 1, f);
    fwrite (&flen, 4, 1, f);
    fwrite (buf, flen_aligned, 1, f);

    fclose (f);

    free (buf);
};

void decrypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int real_flen, flen;
    BYTE *buf;

    f=fopen(fin, "rb");

    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    };
}
```

```

fseek (f, 0, SEEK_END);
flen=f.tell (f);
fseek (f, 0, SEEK_SET);

buf=(BYTE*)malloc (flen);

fread (buf,flen,1,f);

fclose (f);

if (memcmp (buf, "QR9", 3)!=0)
{
    printf ("File is not encrypted!\n");
    return;
};

memcpy (&real_flen, buf+3, 4);

decrypt (buf+(3+4), flen-(3+4), pw);

f=fopen(fout, "wb");

fwrite (buf+(3+4), real_flen, 1, f);

fclose (f);

free (buf);
};

// run: input output 0/1 password
// 0 for encrypt, 1 for decrypt

int main(int argc, char *argv[])
{
    if (argc!=5)
    {
        printf ("Incorrect parameters!\n");
        return 1;
    };

    if (strcmp (argv[3], "0")==0)
        crypt_file (argv[1], argv[2], argv[4]);
    else
        if (strcmp (argv[3], "1")==0)
            decrypt_file (argv[1], argv[2], argv[4]);
        else
            printf ("Wrong param %s\n", argv[3]);
    };

    return 0;
};

```

## 8.7 Encrypted database case #1

(This part has been first appeared in my blog at 26-Aug-2015. Some discussion: <https://news.ycombinator.com/item?id=10128684>.)

### 8.7.1 Base64 and entropy

I've got the XML file containing some encrypted data. Perhaps, it's related to some orders and/or customers information.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Orders>
    <Order>
```

## 8.7. ENCRYPTED DATABASE CASE #1

```
<OrderID>1</OrderID>
<Data>yjmxhXUhB/5MV45chPsXZWAJwIh1S0aD9lFn3XuJMSxJ3/E+UE3hsnH</Data>
</Order>
<Order>
    <OrderID>2</OrderID>
    <Data>0KGe/wnypFBjsy+U0C2P9fC5nDZP3XDZLMPCRaiBw90jIk6Tu5U=</Data>
</Order>
<Order>
    <OrderID>3</OrderID>
    <Data>mqkXfdzvQKvEArdzh+zD9oETVGBFvcTBLs2ph1b5bYddExzp</Data>
</Order>
<Order>
    <OrderID>4</OrderID>
    <Data>FCx6JhIDqnESyT3HAepyE1BJ3cJd7wCk+APCRUeuNtZdpCvQ2MR/7kLXtfUHuA==</Data>
</Order>
...
...
```

The file is available [here](#).

This is clearly base64-encoded data, because all strings consisting of Latin characters, digits, plus (+) and slash (/) symbols. There can be 1 or 2 padding symbols (=), but they are never occurred in the middle of string. Keeping in mind these base64 properties, it's very easy to recognize them.

Let's decode them and calculate entropies ([9.2 on page 947](#)) of these blocks in Wolfram Mathematica:

```
In[]:= ListOfBase64Strings =
  Map[First#[[3]]] &, Cases[Import["encrypted.xml"], XMLElement["Data", _, _], Infinity]];

In[]:= BinaryStrings =
  Map[ImportString[#, {"Base64", "String"}]] &, ListOfBase64Strings];

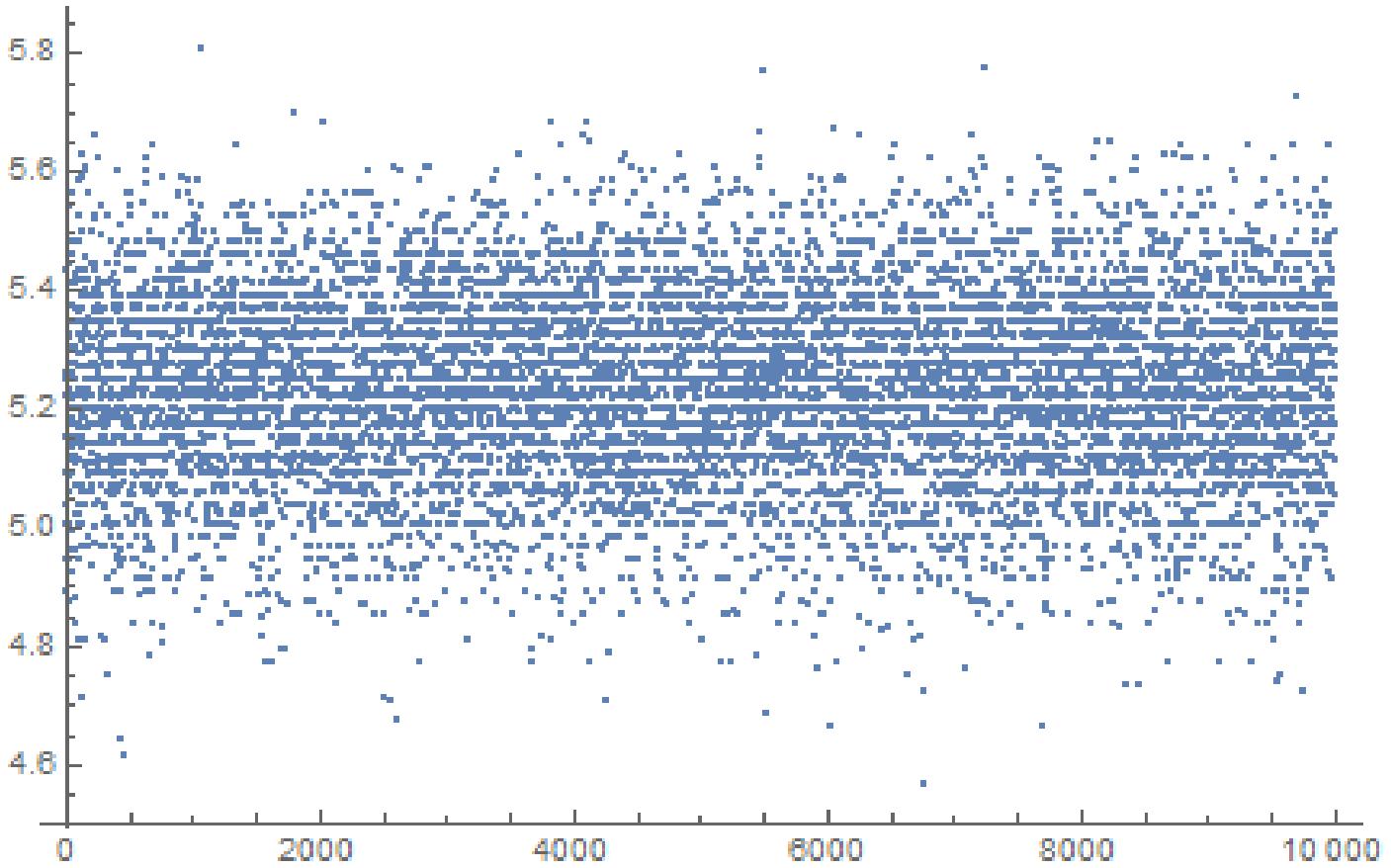
In[]:= Entropies = Map[N[Entropy[2, #]]] &, BinaryStrings];

In[]:= Variance[Entropies]
Out[] = 0.0238614
```

Variance is low. This means the entropy values are not very different from each other. This is visible on graph:

```
In[]:= ListPlot[Entropies]
```

## 8.7. ENCRYPTED DATABASE CASE #1

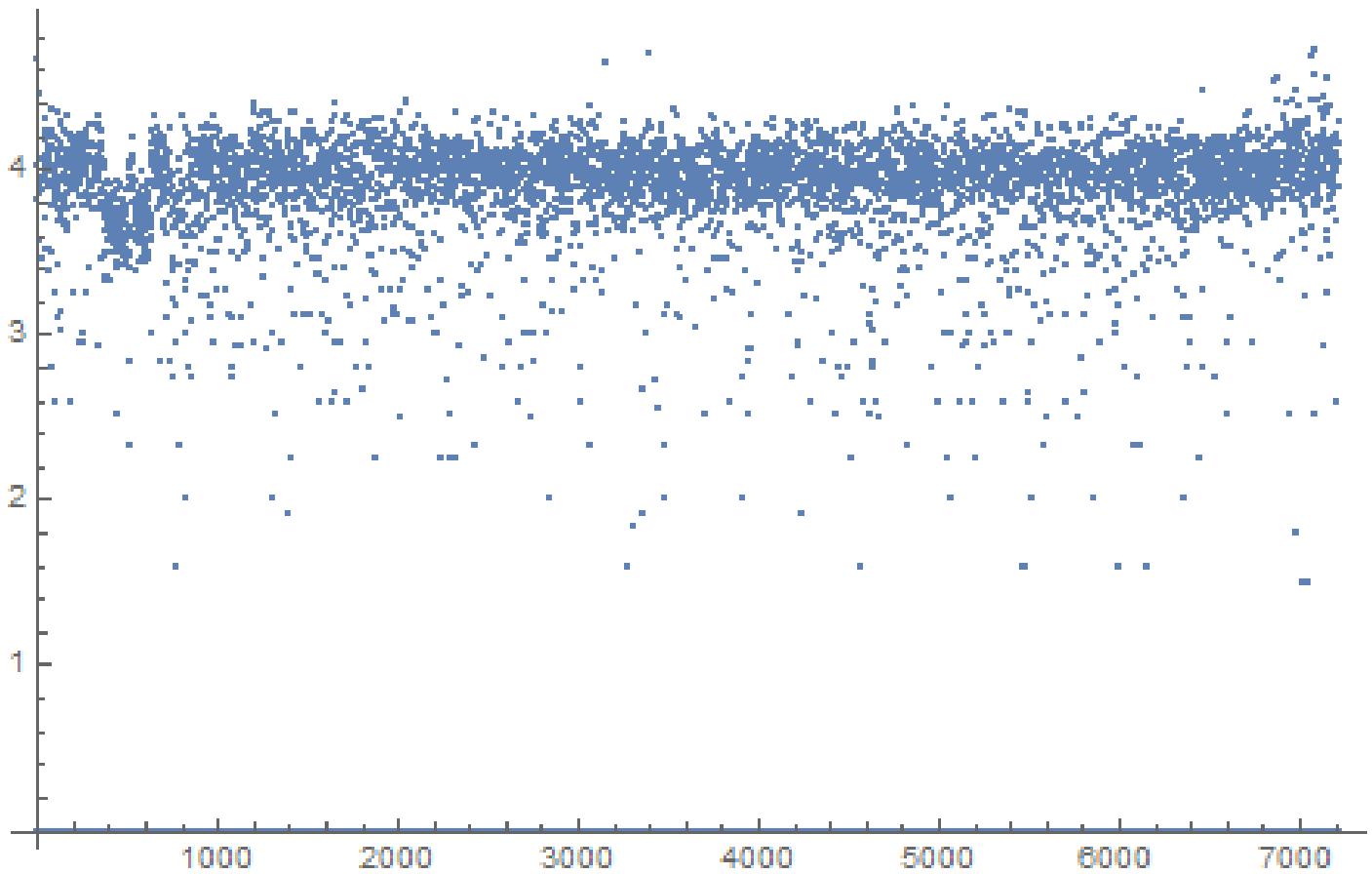


Most values are between 5.0 and 5.4. This is a sign that the data is compressed and/or encrypted.

To understand variance, let's calculate entropies of all lines in Conan Doyle's *The Hound of the Baskervilles* book:

```
In[]:= BaskervillesLines = Import["http://www.gutenberg.org/cache/epub/2852/pg2852.txt", "List"];
In[]:= EntropiesT = Map[N[Entropy[2, #]] &, BaskervillesLines];
In[]:= Variance[EntropiesT]
Out[] = 2.73883
In[]:= ListPlot[EntropiesT]
```

## 8.7. ENCRYPTED DATABASE CASE #1



Most values are gathered around value of 4, but there are also values which are smaller, and they are influenced final variance value.

Perhaps, shortest strings has smaller entropy, let's take short string from the Conan Doyle's book:

```
In[]:= Entropy[2, "Yes, sir."] // N  
Out[] = 2.9477
```

Let's try even shorter:

```
In[]:= Entropy[2, "Yes"] // N  
Out[] = 1.58496  
  
In[]:= Entropy[2, "No"] // N  
Out[] = 1.
```

### 8.7.2 Is it compressed?

OK, so our data is compressed and/or encrypted. Is it compressed? Almost all data compressors put some header at the start, signature, or something like that. As we can see, there are no consistent data at the start. It's still possible that this is DIY handmade data compressor, but they are very rare. Handmade cryptoalgorithm is very easy implement because it's very easy to make it work. Even primitive keyless cryptosystems like `memfrob()`<sup>20</sup> and ROT13 working fine without errors. It's a challenge to write data compressor from scratch using only fantasy and imagination in a way so it will have no evident bugs. Some programmers implements data compression functions by reading textbooks, but this is also rare. The most popular two ways are: 1) just take open-source library like zlib; 2) copy&paste something from somewhere. Open-source data compressions algorithms usually puts some kind of header, and so are popular code snippets from the sites like <http://www.codeproject.com/>.

<sup>20</sup><http://linux.die.net/man/3/memfrob>

### 8.7.3 Is it encrypted?

All major data encryption algorithms process data in blocks. DES—8 bytes, AES—16 bytes. If the input buffer is not divided evenly by block, it's padded, so encrypted data is aligned by cryptoalgorithm's block size. This is not our case.

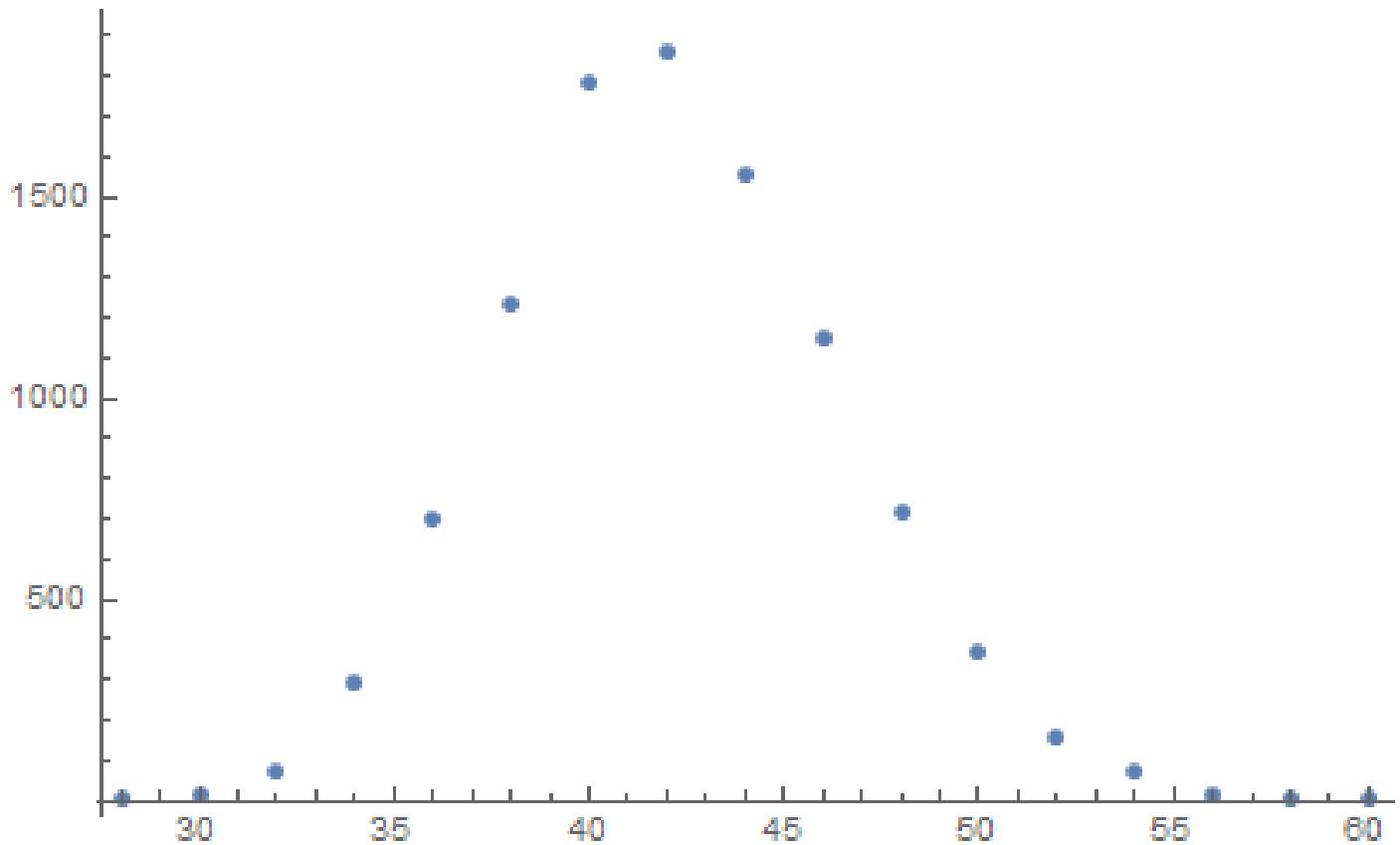
Using Wolfram Mathematica, I analyzed data block's lengths:

```
In[]:= Counts[Map[StringLength[#] &, BinaryStrings]]
Out[]= <|42 -> 1858, 38 -> 1235, 36 -> 699, 46 -> 1151, 40 -> 1784,
44 -> 1558, 50 -> 366, 34 -> 291, 32 -> 74, 56 -> 15, 48 -> 716,
30 -> 13, 52 -> 156, 54 -> 71, 60 -> 3, 58 -> 6, 28 -> 4|>
```

1858 blocks has size of 42 bytes, 1235 blocks has size of 38 bytes, etc.

I made a graph:

```
ListPlot[Counts[Map[StringLength[#] &, BinaryStrings]]]
```



So, most blocks has size between 36 and 48. There is also another thing to notice: all block sizes are even. No single block with odd size.

There are, however, stream ciphers which can operate on byte or bit-level.

### 8.7.4 CryptoPP

The program which can browse this encrypted database is written C# and the .NET code is heavily obfuscated. Nevertheless, there is DLL with x86 code, which, after brief examination, has parts of the CryptoPP popular open-source library! (I just spotted "CryptoPP" strings inside.) Now it's very easy to find all functions inside of DLL because CryptoPP library is open-source.

CryptoPP library has a lot of crypto-functions, including AES (AKA Rijndael). Newer x86 CPUs has AES helper instructions like AESENC, AESDEC and AESKEYGENASSIST<sup>21</sup>. They are not performing encryption/decryption completely, but they do significant amount of job. And newer CryptoPP versions use them. For example, here: 1, 2. To my surprise, during decryption, AESENC is executed, while AESDEC

<sup>21</sup>[https://en.wikipedia.org/wiki/AES\\_instruction\\_set](https://en.wikipedia.org/wiki/AES_instruction_set)

## 8.7. ENCRYPTED DATABASE CASE #1

is not (I just checked with my tracer utility, but any debugger can be used). I checked, if my CPU really supports AES instructions. Some Intel i3 CPUs are not. And if not, CryptoPP library falling back to AES functions implemented in old way <sup>22</sup>. But my CPU supports them. Why AESDEC is still not executed? Why the program use AES encryption in order to decrypt database?

OK, it's not a problem to find a function which encrypts block. It is called *CryptoPP::Rijndael::Enc::ProcessAndXorBlock()*. <https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L349>, and it has references to another function: *Rijndael::Enc::AdvancedProcessBlocks()* <https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L1179>, which, in turn, has references to two functions (*AESNI\_Enc\_Block* and *AESNI\_Enc\_4\_Blocks*) which has AESENC instructions.

So, judging by CryptoPP internals,

*CryptoPP::Rijndael::Enc::ProcessAndXorBlock()* encrypts one 16-byte block. Let's set breakpoint on it and see, what happens during decryption. I use my simple tracer tool again. The software must decrypt first data block now. Oh, by the way, here is the first data block converted from base64 encoding to hexadecimal data, let's have it at hand:

```
00000000: CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D .9..u....1^9r..]
00000010: 95 80 27 02 21 D5 2D 1A 0F D9 45 9F 75 EE 24 C4 ..'!.!....E.u.$.
00000020: B1 27 7F 84 FE 41 37 86 C9 C0 .'.!....A7....
```

This is also arguments of the function from CryptoPP source files:

```
size_t Rijndael::Enc::AdvancedProcessBlocks(const byte *inBlocks, const byte *xorBlocks, byte *outBlocks, size_t length, word32 flags);
```

So it has 5 arguments. Possible flags are:

```
enum {BT_InBlockIsCounter=1, BT_DontIncrementInOutPointers=2, BT_XorInput=4, 
      BT_ReverseDirection=8, BT_AllowParallel=16} FlagsForAdvancedProcessBlocks;
```

OK, run tracer on *ProcessAndXorBlock()* function:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x4339a0,args:5,dump_args:0x10

Warning: no tracer.cfg file.
PID=1984|New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: 01 00 00 00 FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g.."
Argument 3/5
0038B978: CD CD CD CD CD CD CD-CD CD CD CD CD CD CD "....."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..]"
(0) software.exe!0x4339a0(0x38a828, 0x38a838, 0x38bb40, 0x0, 0x8) (called from software.exe!.text+0x3a407 (0x13eb407))
Argument 1/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
Argument 2/5
0038A838: B1 27 7F 84 FE 41 37 86-C9 C0 00 CD CD CD CD CD CD CD ".'....A7....."
Argument 3/5
0038BB40: CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD "....."
(0) software.exe!0x4339a0() -> 0x0
(0) software.exe!0x4339a0(0x38b920, 0x38a828, 0x38bb30, 0x10, 0x0) (called from software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: CA 39 B1 85 75 1B 84 1F-F9 31 5E 39 72 13 EC 5D ".9..u....1^9r..]"
Argument 2/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
```

<sup>22</sup><https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L355>

## 8.7. ENCRYPTED DATABASE CASE #1

```
Argument 3/5
0038BB30: CD "....."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S..."
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from software.exe!.text+0x38b920)
    ↳ x33c0d (0x13e4c0d))
Argument 1/5
0038B920: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
Argument 3/5
0038B978: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ".'.....|..."
PID=1984|Process software.exe exited. ExitCode=0 (0x0)
```

Here we can see inputs and outputs to the *ProcessAndXorBlock()* function.

This is output from the first call of encryption operation:

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..]"
```

Then the *ProcessAndXorBlock()* is called with 0-length block, but with 8 flag (*BT\_ReverseDirection*).

Second:

```
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S..."
```

Wow, there is some string familiar to us!

Third:

```
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ".'.....|..."
```

The first output is very similar to the first 16 bytes of the encrypted buffer.

Output of the first call of *ProcessAndXorBlock()*:

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..]"
```

First 16 bytes of encrypted buffer:

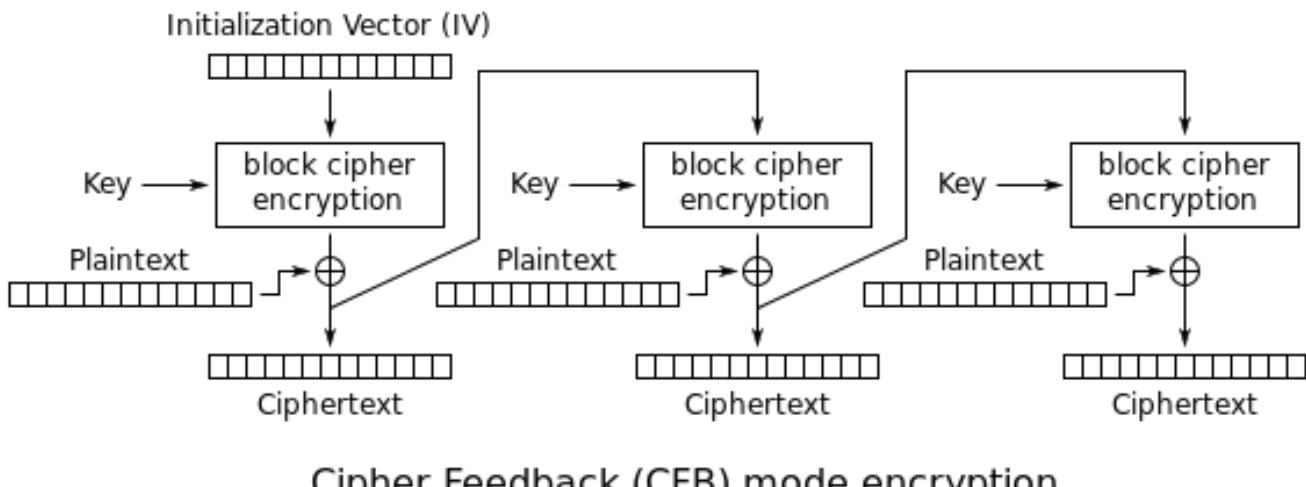
```
00000000: CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D .9..u....1^9r..]
```

There are too much equal bytes! How AES encryption result can be very similar to the encrypted buffer while this is not encryption but rather decryption?!

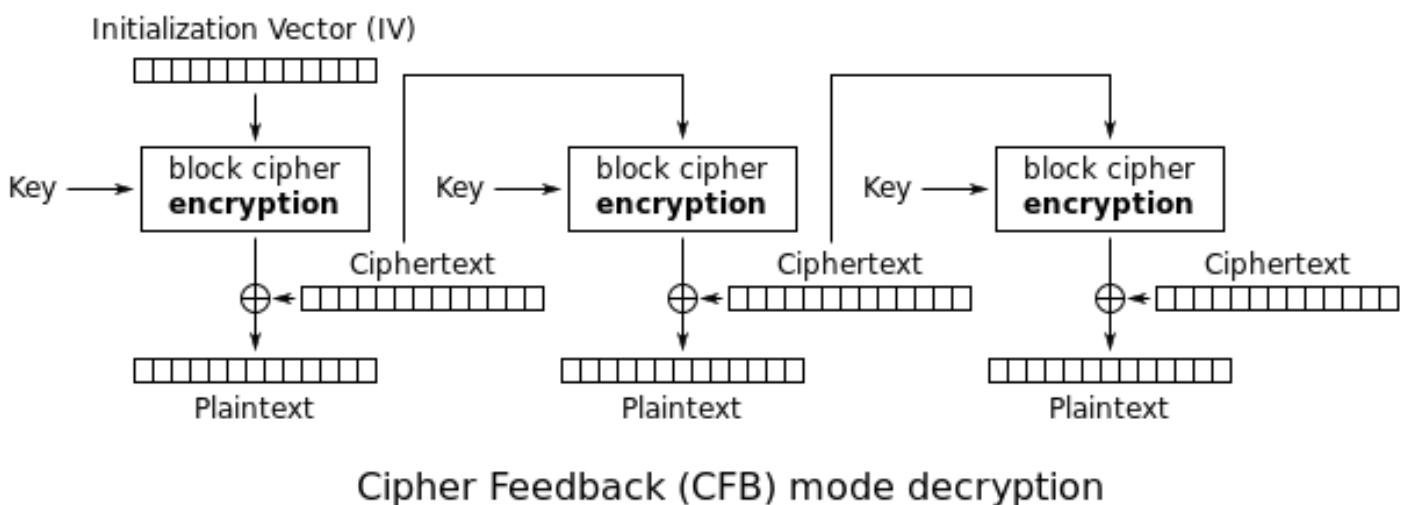
### 8.7.5 Cipher Feedback mode

The answer is CFB (Cipher Feedback mode): In this mode, AES algorithms used not as encryption algorithm, but as a device which generates cryptographically secure random data. The actual encryption is happens using simple XOR operation.

Here is encryption algorithm (images are taken from Wikipedia):



And decryption:



Now let's see: AES encryption operation generates 16 bytes (or 128 bits) or *random* data to be used while XOR-ing, who forces us to use all 16 bytes? If at the last stage we've got 1 byte of data, let's xor 1 byte of data with 1 byte of generated *random* data? This leads to important property of CFB mode: data must not be padded, data of arbitrary size can be encrypted and decrypted.

Oh, that's why all encrypted blocks are not padded. And that's why AESDEC instruction is never called.

Let's try to decrypt first block manually, using Python. CFB mode also use IV (initialization vector), as a seed to *random generator*. In our case, IV is the block which is encrypted at first stage:

```
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g..}"
```

Oh, and we also have to recover encryption key. There is AESKEYGENASSIST is DLL, and it is called, and it is used in the

*Rijndael::Base::UncheckedSetKey()* function:

<https://github.com/mross/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L198> It's easy to find it in IDA and set breakpoint. Let's see:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x435c30,args:3,dump_args:0x10
```

Warning: no tracer.cfg file.

PID=2068|New process software.exe

no module registered with image base 0x77320000

no module registered with image base 0x76e20000

no module registered with image base 0x77320000

## 8.7. ENCRYPTED DATABASE CASE #1

```
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x435c30(0x15e8000, 0x10, 0x14f808) (called from software.exe!.text+0x22fa1 (0x13d3fa1))
Argument 1/3
015E8000: CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E "...~.(_m....).!.."
Argument 3/3
0014F808: 38 82 58 01 C8 B9 46 00-01 D1 3C 01 00 F8 14 00 "8.X...F...<...."
Argument 3/3 +0x0: software.exe!.rdata+0x5238
Argument 3/3 +0x8: software.exe!.text+0x1c101
(0) software.exe!0x435c30() -> 0x13c2801
PID=2068|Process software.exe exited. ExitCode=0 (0x0)
```

So this is the key: *CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E*.

During manual decryption we've got this:

```
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66 E..J.O.H.N.S.ff
00000020: 66 66 66 9E 61 40 D4 07 06 01 fff.a@....
```

Now this is something readable! And now we can see why there were so many equal bytes at the first decryption stage: because plaintext has so many zero bytes!

Let's decrypt the second block:

```
00000000: 17 98 D0 84 3A E9 72 4F DB 82 3F AD E9 3E 2A A8 .....r0..?..>*.
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC A.R.R.O.N.....
00000020: 1B 40 D4 07 06 01 .@....
```

Third, fourth and fifth:

```
00000000: 5D 90 59 06 EF F4 96 B4 7C 33 A7 4A BE FF 66 AB ].Y.....|3.J..f.
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40 I.G.G.S.....e@
00000020: D4 07 06 01 ....
```

```
00000000: D3 15 34 5D 21 18 7C 6E AA F8 2D FE 38 F9 D7 4E ..4]!.|n...8..N
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00 A..D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02 Y.H.z...h@....
```

```
00000000: 1E 8B 90 0A 17 7B C5 52 31 6C 4E 2F DE 1B 27 19 .....{.R1lN...'.
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 00 60 A.R.C.U.S.....
00000020: 66 40 D4 07 06 03 f@....
```

All blocks decrypted seems correctly except of first 16 byte part.

### 8.7.6 Initializing Vector

What can affect first 16 bytes?

Let's back to CFB decryption algorithm again: [8.7.5 on the preceding page](#).

We can see that IV can affect to first block decryption operation, but not the second, because the second stage used ciphertext from the first stage, and in case of decryption, it's the same, no matter what IV has!

So probably, IV is different each time. Using my tracer, I'll take a look at the first input during decryption of the second block of XML file:

```
0038B920: 02 00 00 00 FE FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g..}"
```

... third:

```
0038B920: 03 00 00 00 FD FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g..}"
```

## 8.7. ENCRYPTED DATABASE CASE #1

It seems, first and fifth byte are changed each time. I finally concluded that the first 32-bit integer is just OrderID from the XML file, and the second is also OrderID, but negated. All other 8 bytes are static. Now I have decrypted the whole database: [https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted\\_DB\\_case\\_1/decrypted.full.txt](https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted_DB_case_1/decrypted.full.txt).

The Python script used for this is: [https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted\\_DB\\_case\\_1/decrypt\\_blocks.py](https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted_DB_case_1/decrypt_blocks.py).

Perhaps, author wanted each block encrypted differently, so he/she used OrderID as part of key. It would be also possible to make different AES key instead of IV.

So now we know that IV only affects first block during decryption in CFB mode, this is feature of it. All other blocks can be decrypted without knowledge IV, but using the key.

OK, so why CFB mode? Apparently, because the very first AES example on CryptoPP wiki uses CFB mode: [http://www.cryptopp.com/wiki/Advanced\\_Encryption\\_Standard#Encrypting\\_and\\_Decrypting\\_Using\\_AES](http://www.cryptopp.com/wiki/Advanced_Encryption_Standard#Encrypting_and_Decrypting_Using_AES). Supposedly, CryptoPP developers choose it for simplicity: the example can encrypt/decrypt text strings with arbitrary lengths, without padding.

It is very likely, my program's programmer(s) just copypasted the example from CryptoPP wiki page. Many programmers do so.

The only difference that IV is chosen randomly in CryptoPP wiki example, while this indeterminism wasn't allowable to programmers of the software we are dissecting now, so they choose to initialize IV using Order ID.

Now we can proceed to analyzing matter of each byte in the decrypted block.

### 8.7.7 Structure of the buffer

Let's take first four decrypted blocks:

00000000: 0D 00 FF FE 46 00 52 00	41 00 4E 00 4B 00 49 00	....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00	48 00 4E 00 53 00 66 66	E. .J.O.H.N.S.ff
00000020: 66 66 66 9E 61 40 D4 07	06 01	fff.a@....
00000000: 0B 00 FF FE 4C 00 4F 00	52 00 49 00 20 00 42 00	....L.O.R.I. .B.
00000010: 41 00 52 00 52 00 4F 00	4E 00 CD CC CC CC CC	A.R.R.O.N. ....
00000020: 1B 40 D4 07 06 01		.@....
00000000: 0A 00 FF FE 47 00 41 00	52 00 59 00 20 00 42 00	....G.A.R.Y. .B.
00000010: 49 00 47 00 47 00 53 00	00 00 00 00 00 00 C0 65 40	I.G.G.S. ....e@
00000020: D4 07 06 01		....
00000000: 0F 00 FF FE 4D 00 45 00	4C 00 49 00 4E 00 44 00	....M.E.L.I.N.D.
00000010: 41 00 20 00 44 00 4F 00	48 00 45 00 52 00 54 00	A. .D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF	68 40 D4 07 06 02	Y.H.z...h@....

UTF-16 encoded text strings are clearly visible, these are names and surnames. The first byte (or 16-bit word) is seems string length, we can visually check it. FF FE is seems Unicode BOM.

There are 12 more bytes after each string.

Using this script ([https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted\\_DB\\_case\\_1/dump\\_buffer\\_rest.py](https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted_DB_case_1/dump_buffer_rest.py)) I've got random selection of the block tails:

dennis@...:\$ python decrypt.py encrypted.xml   shuf   head -20		
00000000: 48 E1 7A 14 AE 5F 62 40	DD 07 05 08	H.z..._b@....
00000000: 00 00 00 00 00 40 5A 40	DC 07 08 18	.....@Z@....
00000000: 00 00 00 00 00 80 56 40	D7 07 0B 04	.....V@....
00000000: 00 00 00 00 00 60 61 40	D7 07 0C 1C	.....a@....
00000000: 00 00 00 00 00 20 63 40	D9 07 05 18	..... c@....
00000000: 3D 0A D7 A3 70 FD 34 40	D7 07 07 11	=....p.4@....
00000000: 00 00 00 00 A0 63 40	D5 07 05 19	.....c@....
00000000: CD CC CC CC CC 3C 5C 40	D7 07 08 11	.....@....
00000000: 66 66 66 66 FE 62 40	D4 07 06 05	fffff.b@....
00000000: 1F 85 EB 51 B8 FE 40 40	D6 07 09 1E	...Q...@....
00000000: 00 00 00 00 40 5F 40	DC 07 02 18	.....@_@....
00000000: 48 E1 7A 14 AE 9F 67 40	D8 07 05 12	H.z...g@....
00000000: CD CC CC CC CC 3C 5E 40	DC 07 01 07	.....^@....

## 8.7. ENCRYPTED DATABASE CASE #1

00000000: 00 00 00 00 00 00 67 40	D4 07 0B 0E	.....g@....
00000000: 00 00 00 00 00 40 51 40	DC 07 04 0B	.....@Q@....
00000000: 00 00 00 00 00 40 56 40	D7 07 07 0A	.....@V@....
00000000: 8F C2 F5 28 5C 7F 55 40	DB 07 01 16	....(.U@....
00000000: 00 00 00 00 00 00 32 40	DB 07 06 09	.....2@....
00000000: 66 66 66 66 7E 66 40	D9 07 0A 06	fffff~f@....
00000000: 48 E1 7A 14 AE DF 68 40	D5 07 07 16	H.z...h@....

We first see the 0x40 and 0x07 bytes present in each *tail*. The very last byte is always in 1..0x1F (1..31) range, as I checked. The penultimate byte is always in 1..0xC (1..12) range. Wow, that looks like a date! Year can be represented as 16-bit value, and maybe last 4 bytes is date (16 bits for year, 8 bits for month and day)? 0x7DD is 2013, 0x7D5 is 2005, etc. Seems fine. This is a date. There are 8 more bytes. Judging by the fact this is database named *orders*, maybe some kind of sum is present here? I made attempt to interpret it as double-precision IEEE 754 floating point and dump all values!

Some are:

```
71.0
134.0
51.95
53.0
121.99
96.95
98.95
15.95
85.95
184.99
94.95
29.95
85.0
36.0
130.99
115.95
87.99
127.95
114.0
150.95
```

Looks like real!

Now we can dump names, sums and dates.

```
plain:
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66 E. .J.O.H.N.S.ff
00000020: 66 66 66 9E 61 40 D4 07 06 01 fff.a@....
OrderID= 1 name= FRANKIE JOHNS sum= 140.95 date= 2004 / 6 / 1

plain:
00000000: 0B 00 FF FE 4C 00 4F 00 52 00 49 00 20 00 42 00 ....L.O.R.I. .B.
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC CC A.R.R.O.N. .....
00000020: 1B 40 D4 07 06 01 .@....
OrderID= 2 name= LORI BARRON sum= 6.95 date= 2004 / 6 / 1

plain:
00000000: 0A 00 FF FE 47 00 41 00 52 00 59 00 20 00 42 00 ....G.A.R.Y. .B.
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40 I.G.G.S. ....e@
00000020: D4 07 06 01 .....
OrderID= 3 name= GARY BIGGS sum= 174.0 date= 2004 / 6 / 1

plain:
00000000: 0F 00 FF FE 4D 00 45 00 4C 00 49 00 4E 00 44 00 ....M.E.L.I.N.D.
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00 A. .D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02 Y.H.z...h@....
OrderID= 4 name= MELINDA DOHERTY sum= 199.99 date= 2004 / 6 / 2

plain:
00000000: 0B 00 FF FE 4C 00 45 00 4E 00 41 00 20 00 4D 00 ....L.E.N.A. .M.
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 00 60 A.R.C.U.S. .....
00000020: 66 40 D4 07 06 03 f@....
```

## 8.7. ENCRYPTED DATABASE CASE #1

```
OrderID= 5 name= LENA MARCUS sum= 179.0 date= 2004 / 6 / 3
```

See more: [https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted\\_DB\\_case\\_1/decrypted.full.with\\_data.txt](https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted_DB_case_1/decrypted.full.with_data.txt). Or filtered: [https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted\\_DB\\_case\\_1/decrypted.short.txt](https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted_DB_case_1/decrypted.short.txt). Seems correct.

This is some kind of OOP serialization, i.e., packing differently typed values into binary buffer for storing and/or transmitting.

### 8.7.8 Noise at the end

The only thing is that sometimes, tail is bigger:

```
00000000: 0E 00 FF FE 54 00 48 00 45 00 52 00 45 00 53 00 ....T.H.E.R.E.S.  
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00 E. .T.U.T.T.L.E.  
00000020: 66 66 66 66 1E 63 40 D4 07 07 1A 00 07 07 19 fffff.c@.....  
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(00 07 07 19 bytes are not used and is ballast).

```
00000000: 0C 00 FF FE 4D 00 45 00 4C 00 41 00 4E 00 49 00 ....M.E.L.A.N.I.  
00000010: 45 00 20 00 4B 00 49 00 52 00 4B 00 00 00 00 00 E. .K.I.R.K.....  
00000020: 00 20 64 40 D4 07 09 02 00 02 . d@.....  
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2
```

(00 02 are not used).

After close examination, we can see, that the noise at the end of tail is just left from previous encryption!

Here are two subsequent buffers:

```
00000000: 10 00 FF FE 42 00 4F 00 4E 00 4E 00 49 00 45 00 ....B.O.N.N.I.E.  
00000010: 20 00 47 00 4F 00 4C 00 44 00 53 00 54 00 45 00 .G.O.L.D.S.T.E.  
00000020: 49 00 4E 00 9A 99 99 99 99 79 46 40 D4 07 07 19 I.N.....yF@....  
OrderID= 171 name= BONNIE GOLDSTEIN sum= 44.95 date= 2004 / 7 / 25  
  
00000000: 0E 00 FF FE 54 00 48 00 45 00 52 00 45 00 53 00 ....T.H.E.R.E.S.  
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00 E. .T.U.T.T.L.E.  
00000020: 66 66 66 66 1E 63 40 D4 07 07 1A 00 07 07 19 fffff.c@.....  
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(The last 07 07 19 bytes are copied from the previous plaintext buffer).

Another two subsequent buffers:

```
00000000: 0D 00 FF FE 4C 00 4F 00 52 00 45 00 4E 00 45 00 ....L.O.R.E.N.E.  
00000010: 20 00 4F 00 54 00 4F 00 4F 00 4C 00 45 00 CD CC .O.T.O.O.L.E...  
00000020: CC CC CC 3C 5E 40 D4 07 09 02 ...<^@....  
OrderID= 285 name= LORENE OTOOLE sum= 120.95 date= 2004 / 9 / 2  
  
00000000: 0C 00 FF FE 4D 00 45 00 4C 00 41 00 4E 00 49 00 ....M.E.L.A.N.I.  
00000010: 45 00 20 00 4B 00 49 00 52 00 4B 00 00 00 00 00 E. .K.I.R.K.....  
00000020: 00 20 64 40 D4 07 09 02 00 02 . d@.....  
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2
```

The last 02 byte is copied from the previous plaintext buffer.

It's possible if the buffer used while encrypting is global and/or isn't cleared before each encryption. The final buffer size value is also biased somehow, nevertheless, the bug left uncaught because it doesn't affect decrypting software, it's just ignores noise at the end. This is common mistake. It even affected OpenSSL (Heartbleed bug).

### 8.7.9 Conclusion

Summary: every practicing reverse engineer should be familiar with major crypto algorithms and also major cryptographical modes. Some books about it: [11.1.10 on page 997](#).

## **8.8. OVERCLOCKING COINTERRA BITCOIN MINER**

Encrypted database contents has been artificially constructed by me for the sake of demonstration. I've got most popular USA names and surnames from there: <http://stackoverflow.com/questions/1803628/raw-list-of-person-names>, and combined them randomly. Dates and sums were also generated randomly.

All files used in this article are here: [https://github.com/dennis714/yurichev.com/tree/master/blog/encrypted\\_DB\\_case\\_1](https://github.com/dennis714/yurichev.com/tree/master/blog/encrypted_DB_case_1).

Nevertheless, many features like these I've observed in real-world software applications. This example is based on them.

### **8.7.10 Post Scriptum: brute-forcing IV**

The case you saw is artificially constructed, but based on some real applications I've observed. When I've been working on one of such real applications, I first noticed that Initializing Vector (IV) is generating using some 32-bit number, I wasn't able to find a link between this value and OrderID. So I prepared to use brute-force, which is indeed possible here. It's not a problem to enumerate all 32-bit values and try each IV based on each. Then you decrypt 16-byte block and check for zero bytes, which are always at fixed places.

## **8.8 Overclocking Cointerra Bitcoin miner**

There was Cointerra Bitcoin miner, looking like that:

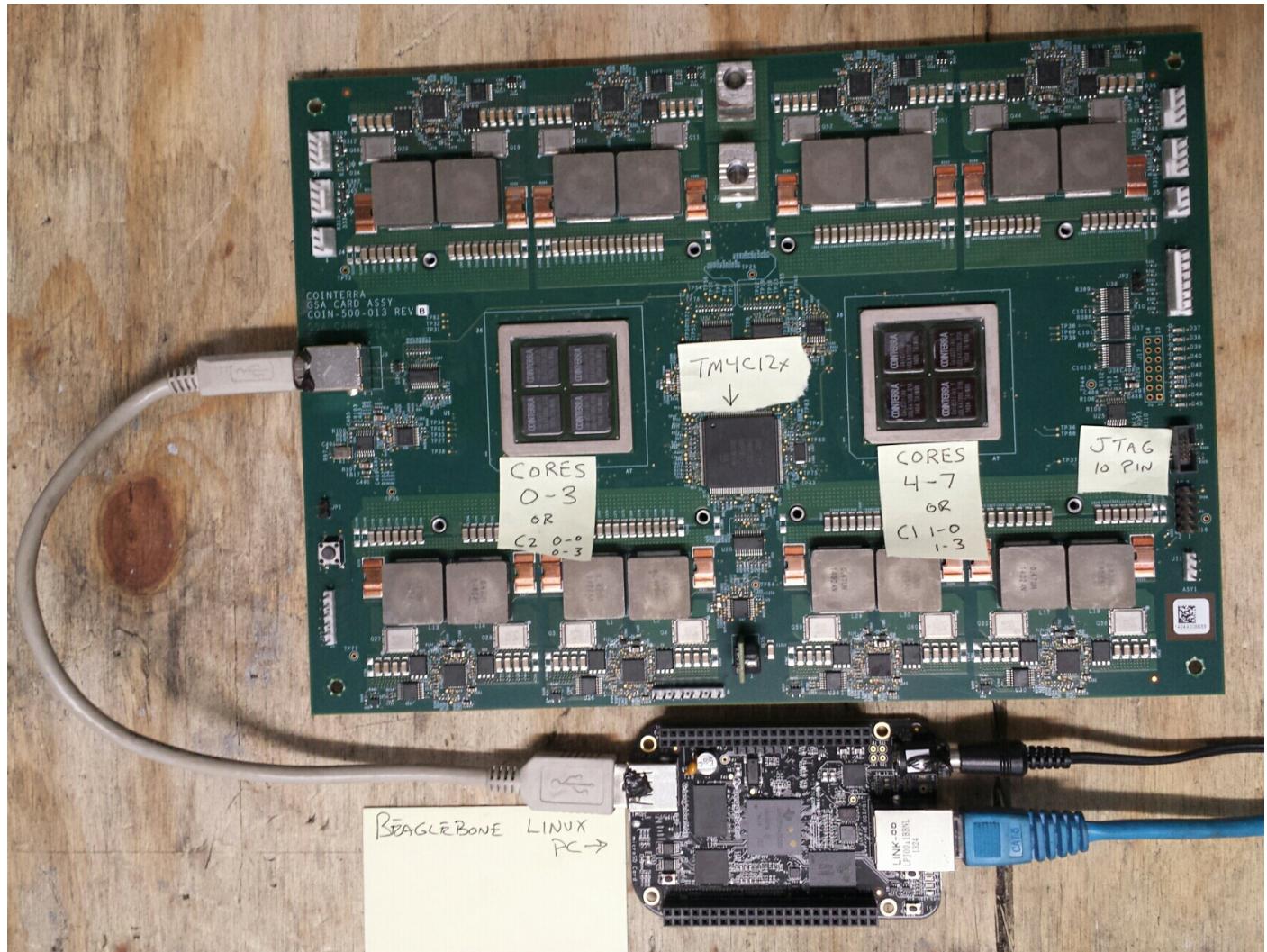


Figure 8.14: Board

## 8.8. OVERCLOCKING COINTERRA BITCOIN MINER

And there was also (possibly leaked) utility<sup>23</sup> which can set clock rate for the board. It runs on additional BeagleBone Linux ARM board (small board at bottom of the picture).

And the author was once asked, is it possible to hack this utility to see, which frequency can be set and which are not. And it is possible to tweak it?

The utility must be executed like that: `./cointool-overclock 0 0 900`, where 900 is frequency in MHz. If the frequency is too high, utility will print “Error with arguments” and exit.

This is a fragment of code around reference to “Error with arguments” text string:

```
...  
.text:0000ABC4      STR      R3, [R11,#var_28]  
.text:0000ABC8      MOV      R3, #optind  
.text:0000ABD0      LDR      R3, [R3]  
.text:0000ABD4      ADD      R3, R3, #1  
.text:0000ABD8      MOV      R3, R3,LSL#2  
.text:0000ABDC      LDR      R2, [R11,#argv]  
.text:0000ABE0      ADD      R3, R2, R3  
.text:0000ABE4      LDR      R3, [R3]  
.text:0000ABE8      MOV      R0, R3 ; nptr  
.text:0000ABEC      MOV      R1, #0 ; endptr  
.text:0000ABF0      MOV      R2, #0 ; base  
.text:0000ABF4      BL       strtoll  
.text:0000ABF8      MOV      R2, R0  
.text:0000ABFC      MOV      R3, R1  
.text:0000AC00      MOV      R3, R2  
.text:0000AC04      STR      R3, [R11,#var_2C]  
.text:0000AC08      MOV      R3, #optind  
.text:0000AC10      LDR      R3, [R3]  
.text:0000AC14      ADD      R3, R3, #2  
.text:0000AC18      MOV      R3, R3,LSL#2  
.text:0000AC1C      LDR      R2, [R11,#argv]  
.text:0000AC20      ADD      R3, R2, R3  
.text:0000AC24      LDR      R3, [R3]  
.text:0000AC28      MOV      R0, R3 ; nptr  
.text:0000AC2C      MOV      R1, #0 ; endptr  
.text:0000AC30      MOV      R2, #0 ; base  
.text:0000AC34      BL       strtoll  
.text:0000AC38      MOV      R2, R0  
.text:0000AC3C      MOV      R3, R1  
.text:0000AC40      MOV      R3, R2  
.text:0000AC44      STR      R3, [R11,#third_argument]  
.text:0000AC48      LDR      R3, [R11,#var_28]  
.text:0000AC4C      CMP      R3, #0  
.text:0000AC50      BLT     errors_with_arguments  
.text:0000AC54      LDR      R3, [R11,#var_28]  
.text:0000AC58      CMP      R3, #1  
.text:0000AC5C      BGT     errors_with_arguments  
.text:0000AC60      LDR      R3, [R11,#var_2C]  
.text:0000AC64      CMP      R3, #0  
.text:0000AC68      BLT     errors_with_arguments  
.text:0000AC6C      LDR      R3, [R11,#var_2C]  
.text:0000AC70      CMP      R3, #3  
.text:0000AC74      BGT     errors_with_arguments  
.text:0000AC78      LDR      R3, [R11,#third_argument]  
.text:0000AC7C      CMP      R3, #0x31  
.text:0000AC80      BLE     errors_with_arguments  
.text:0000AC84      LDR      R2, [R11,#third_argument]  
.text:0000AC88      MOV      R3, #950  
.text:0000AC8C      CMP      R2, R3  
.text:0000AC90      BGT     errors_with_arguments  
.text:0000AC94      LDR      R2, [R11,#third_argument]  
.text:0000AC98      MOV      R3, #0x51EB851F  
.text:0000ACA0      SMULL   R1, R3, R3, R2  
.text:0000ACA4      MOV      R1, R3,ASR#4  
.text:0000ACA8      MOV      R3, R2,ASR#31
```

<sup>23</sup>Can be downloaded here: [http://yurichev.com/blog/bitcoin\\_miner/files/cointool-overclock](http://yurichev.com/blog/bitcoin_miner/files/cointool-overclock)

## 8.8. OVERCLOCKING COINTERRA BITCOIN MINER

```

.text:0000ACAC      RSB      R3, R3, R1
.text:0000ACB0      MOV      R1, #50
.text:0000ACB4      MUL      R3, R1, R3
.text:0000ACB8      RSB      R3, R3, R2
.text:0000ACBC      CMP      R3, #0
.text:0000ACC0      BEQ      loc_ACEC
.text:0000ACC4
.text:0000ACC4 errors_with_arguments
.text:0000ACC4
.text:0000ACC4      LDR      R3, [R11,#argv]
.text:0000ACC8      LDR      R3, [R3]
.text:0000ACCC      MOV      R0, R3 ; path
.text:0000ACD0      BL       __xpg_basename
.text:0000ACD4      MOV      R3, R0
.text:0000ACD8      MOV      R0, #aSErrorWithArgu ; format
.text:0000ACE0      MOV      R1, R3
.text:0000ACE4      BL       printf
.text:0000ACE8      B       loc_ADD4
.text:0000ACEC ; -----
.text:0000ACEC loc_ACEC           ; CODE XREF: main+66C
.text:0000ACEC      LDR      R2, [R11,#third_argument]
.text:0000ACF0      MOV      R3, #499
.text:0000ACF4      CMP      R2, R3
.text:0000ACF8      BGT      loc_AD08
.text:0000ACFC      MOV      R3, #0x64
.text:0000AD00      STR      R3, [R11,#unk_constant]
.text:0000AD04      B       jump_to_write_power
.text:0000AD08 ; -----
.text:0000AD08
.text:0000AD08 loc_AD08           ; CODE XREF: main+6A4
.text:0000AD08      LDR      R2, [R11,#third_argument]
.text:0000AD0C      MOV      R3, #799
.text:0000AD10      CMP      R2, R3
.text:0000AD14      BGT      loc_AD24
.text:0000AD18      MOV      R3, #0x5F
.text:0000AD1C      STR      R3, [R11,#unk_constant]
.text:0000AD20      B       jump_to_write_power
.text:0000AD24 ; -----
.text:0000AD24
.text:0000AD24 loc_AD24           ; CODE XREF: main+6C0
.text:0000AD24      LDR      R2, [R11,#third_argument]
.text:0000AD28      MOV      R3, #899
.text:0000AD2C      CMP      R2, R3
.text:0000AD30      BGT      loc_AD40
.text:0000AD34      MOV      R3, #0x5A
.text:0000AD38      STR      R3, [R11,#unk_constant]
.text:0000AD3C      B       jump_to_write_power
.text:0000AD40 ; -----
.text:0000AD40
.text:0000AD40 loc_AD40           ; CODE XREF: main+6DC
.text:0000AD40      LDR      R2, [R11,#third_argument]
.text:0000AD44      MOV      R3, #999
.text:0000AD48      CMP      R2, R3
.text:0000AD4C      BGT      loc_AD5C
.text:0000AD50      MOV      R3, #0x55
.text:0000AD54      STR      R3, [R11,#unk_constant]
.text:0000AD58      B       jump_to_write_power
.text:0000AD5C ; -----
.text:0000AD5C
.text:0000AD5C loc_AD5C           ; CODE XREF: main+6F8
.text:0000AD5C      LDR      R2, [R11,#third_argument]
.text:0000AD60      MOV      R3, #1099
.text:0000AD64      CMP      R2, R3
.text:0000AD68      BGT      jump_to_write_power
.text:0000AD6C      MOV      R3, #0x50
.text:0000AD70      STR      R3, [R11,#unk_constant]
.text:0000AD74      jump_to_write_power          ; CODE XREF: main+6B0
.text:0000AD74          ; main+6CC ...

```

## 8.8. OVERCLOCKING COINTERRA BITCOIN MINER

```

.text:0000AD74      LDR    R3, [R11,#var_28]
.text:0000AD78      UXTB   R1, R3
.text:0000AD7C      LDR    R3, [R11,#var_2C]
.text:0000AD80      UXTB   R2, R3
.text:0000AD84      LDR    R3, [R11,#unk_constant]
.text:0000AD88      UXTB   R3, R3
.text:0000AD8C      LDR    R0, [R11,#third_argument]
.text:0000AD90      UXTH   R0, R0
.text:0000AD94      STR    R0, [SP,#0x44+var_44]
.text:0000AD98      LDR    R0, [R11,#var_24]
.text:0000AD9C      BL     write_power
.text:0000ADA0      LDR    R0, [R11,#var_24]
.text:0000ADA4      MOV    R1, #0x5A
.text:0000ADA8      BL     read_loop
.text:0000ADAC      B      loc_ADD4

...
.rodata:0000B378 aSErrorWithArgu DCB "%s: Error with arguments",0xA,0 ; DATA XREF: main+684
...

```

Function names were present in debugging information of the original binary, like `write_power`, `read_loop`. But labels inside functions were named by me.

`optind` name looks familiar. It is from `getopt` \*NIX library intended for command-line parsing—well, this is exactly what happens in this utility. Then, the 3rd argument (where frequency value is to be passed) is converted from string to number using call to `strtol()` function.

The value is then checked against various constants. At `0xACEC`, it's checked, if it is lesser or equal to `499`, `0x64` is to be passed finally to `write_power()` function (which sends a command through USB using `send_msg()`). If it is greater than `499`, jump to `0xAD08` is occurred.

At `0xAD08` it's checked, if it's lesser or equal to `799`. `0x5F` is then passed to `write_power()` function in case of success.

There are more checks: for `899` at `0xAD24`, for `0x999` at `0xAD40` and finally, for `1099` at `0xAD5C`. If the input frequency is lesser or equal to `1099`, `0x50` will be passed (at `0xAD6C`) to `write_power()` function. And there is some kind of bug. If the value is still greater than `1099`, the value itself is passed into `write_power()` function. Oh, it's not a bug, because we can't get here: value is checked first against `950` at `0xAC88`, and if it is greater, error message will be displayed and the utility will finish.

Now the table between frequency in MHz and value passed to `write_power()` function:

MHz	hexadecimal	decimal
499MHz	0x64	100
799MHz	0x5f	95
899MHz	0x5a	90
999MHz	0x55	85
1099MHz	0x50	80

As it seems, value passed to the board is gradually decreasing during frequency increasing.

Now we see that value of `950MHz` is a hardcoded limit, at least in this utility. Can we trick it?

Let's back to this piece of code:

```

.text:0000AC84      LDR    R2, [R11,#third_argument]
.text:0000AC88      MOV    R3, #950
.text:0000AC8C      CMP    R2, R3
.text:0000AC90      BGT   errors_with_arguments ; I've patched here to 00 00 00 00

```

We must disable `BGT` branch instruction at `0xAC90` somehow. And this is ARM in ARM mode, because, as we see, all addresses are increasing by 4, i.e., each instruction has size of 4 bytes. `NOP` (no operation)

## 8.9. BREAKING SIMPLE EXECUTABLE CRYPTOR

instruction in ARM mode is just four zero bytes: `00 00 00 00`. So by writing four zeros at 0xAC90 address (or physical offset in file 0x2C90) will disable this check.

Not it's possible to set frequencies up to 1050MHz. Even more is possible, but due to the bug, if input value is greater than 1099, raw value in MHz will be passed to the board, which is incorrect.

I didn't go further, but if I had to, I would try to decrease a value which is passed to `write_power()` function.

Now the scary piece of code which I skipped at first:

```
.text:0000AC94    LDR      R2, [R11,#third_argument]
.text:0000AC98    MOV      R3, #0x51EB851F
.text:0000ACA0    SMULL   R1, R3, R3, R2 ; R3=3rg_arg/3.125
.text:0000ACA4    MOV      R1, R3, ASR#4 ; R1=R3/16=3rg_arg/50
.text:0000ACA8    MOV      R3, R2, ASR#31 ; R3=MSB(3rg_arg)
.text:0000ACAC    RSB      R3, R3, R1 ; R3=3rd_arg/50
.text:0000ACB0    MOV      R1, #50
.text:0000ACB4    MUL      R3, R1, R3 ; R3=50*(3rd_arg/50)
.text:0000ACB8    RSB      R3, R3, R2
.text:0000ACBC    CMP      R3, #0
.text:0000ACC0    BEQ      loc_ACEC
.text:0000ACC4    errors_with_arguments
```

Multiplication via division is used here, and constant is 0x51EB851F. I wrote a simple programmer's calculator<sup>24</sup> for myself. And I have there a feature to calculate modulo inverse.</p>

```
modinv32(0x51EB851F)
Warning, result is not integer: 3.125000
(unsigned) dec: 3 hex: 0x3 bin: 11
```

That means that `SMULL` instruction at 0xACA0 is basically divides 3rd argument by 3.125. In fact, all `modinv32()` function in my calculator does, is this:

$$\frac{1}{\frac{\text{input}}{2^{32}}} = \frac{2^{32}}{\text{input}}$$

Then there are additional shifts and now we see than 3rg argument is just divided by 50. And then it's multiplied by 50 again. Why? This is simplest check, if the input value is can be divided by 50 evenly. If the value of this expression is non-zero,  $x$  can't be divided by 50 evenly:

$$x - \left(\left(\frac{x}{50}\right) \cdot 50\right)$$

This is in fact simple way to calculate remainder of division.

And then, if the remainder is non-zero, error message is displayed. So this utility takes frequency values in form like 850, 900, 950, 1000, etc., but not 855 or 911.

That's it! If you do something like that, please be warned that you may damage your board, just as in case of overclocking other devices like CPUs, GPUs, etc. If you have a Cointerra board, do this on your own risk!

## 8.9 Breaking simple executable cryptor

I've got an executable file which is encrypted by relatively simple encryption. [Here is it](#) (only executable section is left here).

First, all encryption function does is just adds number of position in buffer to the byte. Here is how this can be encoded in Python:

<sup>24</sup><https://github.com/dennis714/progcalc>

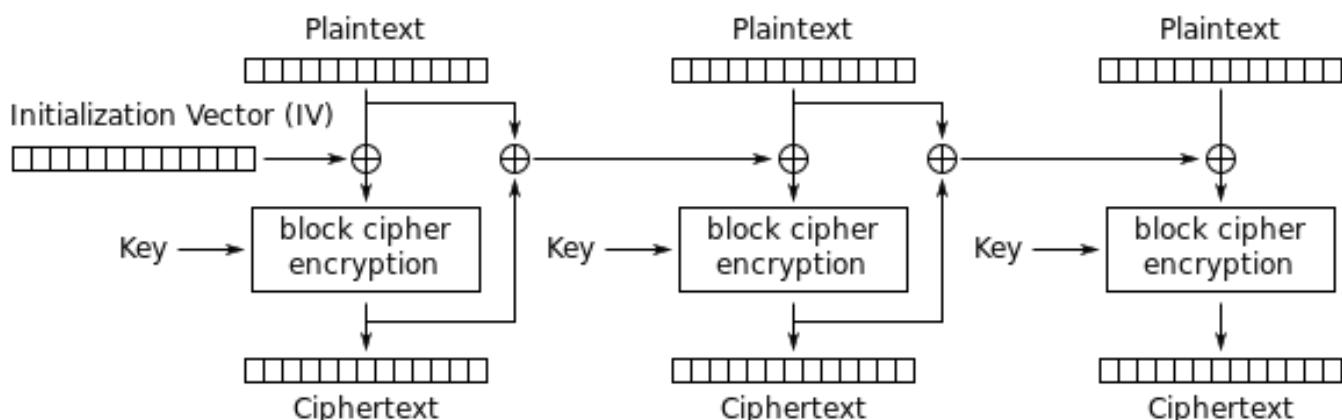
Listing 8.7: Python script

```
#!/usr/bin/env python
def e(i, k):
    return chr ((ord(i)+k) % 256)

def encrypt(buf):
    return e(buf[0], 0)+ e(buf[1], 1)+ e(buf[2], 2) + e(buf[3], 3)+ e(buf[4], 4)+ e(buf[5], 5)+
    ↴ e(buf[6], 6)+ e(buf[7], 7)+
    ↴ e(buf[8], 8)+ e(buf[9], 9)+ e(buf[10], 10)+ e(buf[11], 11)+ e(buf[12], 12)+ e(buf[13], 13)+ e(buf[14], 14)+ e(buf[15], 15)
```

Hence, if you encrypt buffer with 16 zeros, you'll get 0, 1, 2, 3 ... 12, 13, 14, 15.

Propagating Cipher Block Chaining (PCBC) is also used, here is how it works:



Propagating Cipher Block Chaining (PCBC) mode encryption

Figure 8.15: Propagating Cipher Block Chaining encryption (image is taken from Wikipedia article)

The problem is that it's too boring to recover IV (Initialization Vector) each time. Brute-force is also not an option, because IV is too long (16 bytes). Let's see, if it's possible to recover IV for arbitrary encrypted executable file?

Let's try simple frequency analysis. This is 32-bit x86 executable code, so let's gather statistics about most frequent bytes and opcodes. I tried huge oracle.exe file from Oracle RDBMS version 11.2 for windows x86 and I've found that the most frequent byte (no surprise) is zero ( 10%). The next most frequent byte is (again, no surprise) 0xFF ( 5%). The next is 0x8B ( 5%).

0x8B is opcode for `MOV`, this is indeed one of the most busy x86 instructions. Now what about popularity of zero byte? If compiler needs to encode value bigger than 127, it has to use 32-bit displacement instead of 8-bit one, but large values are very rare, so it is padded by zeros. This is at least in `LEA`, `MOV`, `PUSH`, `CALL`.

For example:

8D B0 28 01 00 00	lea      esi, [eax+128h]
8D BF 40 38 00 00	lea      edi, [edi+3840h]

Displacements bigger than 127 are very popular, but they are rarely exceeds 0x10000 (indeed, such large memory buffers/structures are also rare).

Same story with `MOV`, large constants are rare, the most heavily used are 0, 1, 10, 100,  $2^n$ , and so on. Compiler has to pad small constants by zeros to represent them as 32-bit values:

BF 02 00 00 00	mov      edi, 2
BF 01 00 00 00	mov      edi, 1

Now about 00 and FF bytes combined: jumps (including conditional) and calls can pass execution flow forward or backwards, but very often, within the limits of the current executable module. If forward,

## 8.9. BREAKING SIMPLE EXECUTABLE CRYPTOR

displacement is not very big and also padded with zeros. If backwards, displacement is represented as negative value, so padded with FF bytes. For example, transfer execution flow forward:

E8 43 0C 00 00	call _function1
E8 5C 00 00 00	call _function2
0F 84 F0 0A 00 00	jz loc_4F09A0
0F 84 EB 00 00 00	jz loc_4EFBB8

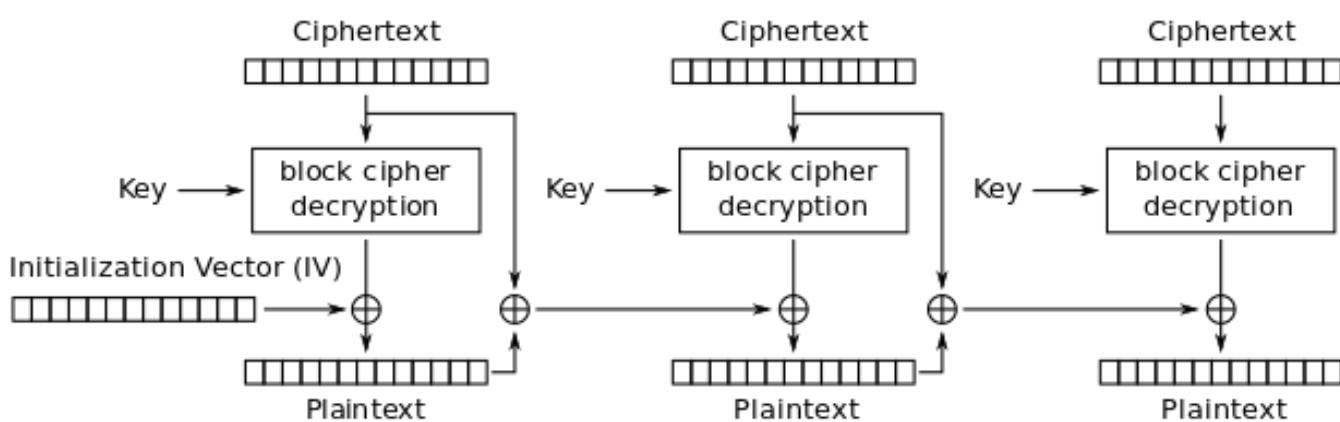
Backwards:

E8 79 0C FE FF	call _function1
E8 F4 16 FF FF	call _function2
0F 84 F8 FB FF FF	jz loc_8212BC
0F 84 06 FD FF FF	jz loc_FF1E7D

FF byte is also very often occurred in negative displacements like these:

8D 85 1E FF FF FF	lea eax, [ebp-0E2h]
8D 95 F8 5C FF FF	lea edx, [ebp-0A308h]

So far so good. Now we have to try various 16-byte keys, decrypt executable section and measure how often 00, FF ad 8B bytes are occurred. Let's also keep in sight how PCBC decryption works:



Propagating Cipher Block Chaining (PCBC) mode decryption

Figure 8.16: Propagating Cipher Block Chaining decryption (image is taken from Wikipedia article)

The good news is that we don't really have to decrypt whole piece of data, but only slice by slice, this is exactly how I did in my previous example: [9.1.4 on page 941](#).

Now I'm trying all possible bytes (0..255) for each byte in key and just pick the byte producing maximal amount of 00/FF/8B bytes in a decrypted slice:

```

#!/usr/bin/env python
import sys, hexdump, array, string, operator

KEY_LEN=16

def chunks(l, n):
    # split n by l-byte chunks
    # http://stackoverflow.com/questions/312443/how-do-you-split-a-list-into-evenly-sized-chunks-in-python
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
    file.close()
    return content

```

```

def decrypt_byte (c, key):
    return chr((ord(c)-key) % 256)

def XOR_PCBC_step (IV, buf, k):
    prev=IV
    rt=""
    for c in buf:
        new_c=decrypt_byte(c, k)
        plain=chr(ord(new_c)^ord(prev))
        prev=chr(ord(c)^ord(plain))
        rt=rt+plain
    return rt

each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 16-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    stat={}
    for i in range(256):
        tmp_key=chr(i)
        tmp=XOR_PCBC_step(tmp_key,each_Nth_byte[N], N)
        # count 0, FFs and 8Bs in decrypted buffer:
        important_bytes=tmp.count('\x00')+tmp.count('\xFF')+tmp.count('\x8B')
        stat[i]=important_bytes
    sorted_stat = sorted(stat.iteritems(), key=operator.itemgetter(1), reverse=True)
    print sorted_stat[0]

```

(Source code can downloaded [here](#).)

I run it and here is a key for which 00/FF/8B bytes presence in decrypted buffer is maximal:

```

N= 0
(147, 1224)
N= 1
(94, 1327)
N= 2
(252, 1223)
N= 3
(218, 1266)
N= 4
(38, 1209)
N= 5
(192, 1378)
N= 6
(199, 1204)
N= 7
(213, 1332)
N= 8
(225, 1251)
N= 9
(112, 1223)
N= 10
(143, 1177)
N= 11
(108, 1286)
N= 12
(10, 1164)
N= 13
(3, 1271)
N= 14
(128, 1253)

```

## 8.9. BREAKING SIMPLE EXECUTABLE CRYPTOR

```
N= 15  
(232, 1330)
```

Let's write decryption utility with the key we got:

```
#!/usr/bin/env python  
import sys, hexdump, array  
  
def xor_strings(s,t):  
    # https://en.wikipedia.org/wiki/XOR_cipher#Example_implementation  
    """xor two strings together"""  
    return "".join(chr(ord(a)^ord(b)) for a,b in zip(s,t))  
  
IV=array.array('B', [147, 94, 252, 218, 38, 192, 199, 213, 225, 112, 143, 108, 10, 3, 128, ↴  
↳ 232]).tostring()  
  
def chunks(l, n):  
    n = max(1, n)  
    return [l[i:i + n] for i in range(0, len(l), n)]  
  
def read_file(fname):  
    file=open(fname, mode='rb')  
    content=file.read()  
    file.close()  
    return content  
  
def decrypt_byte(i, k):  
    return chr ((ord(i)-k) % 256)  
  
def decrypt(buf):  
    return "".join(decrypt_byte(buf[i], i) for i in range(16))  
  
fout=open(sys.argv[2], mode='wb')  
  
prev=IV  
content=read_file(sys.argv[1])  
tmp=chunks(content, 16)  
for c in tmp:  
    new_c=decrypt(c)  
    p=xor_strings (new_c, prev)  
    prev=xor_strings(c, p)  
    fout.write(p)  
fout.close()
```

(Source code can downloaded [here](#).)

Let's check resulting file:

```
$ objdump -b binary -m i386 -D decrypted.bin  
  
...  
  
5: 8b ff          mov    %edi,%edi  
7: 55             push   %ebp  
8: 8b ec          mov    %esp,%ebp  
a: 51             push   %ecx  
b: 53             push   %ebx  
c: 33 db          xor    %ebx,%ebx  
e: 43             inc    %ebx  
f: 84 1d a0 e2 05 01 test   %bl,0x105e2a0  
15: 75 09          jne    0x20  
17: ff 75 08          pushl  0x8(%ebp)  
1a: ff 15 b0 13 00 01 call   *0x10013b0  
20: 6a 6c          push   $0x6c  
22: ff 35 54 d0 01 01 pushl  0x101d054  
28: ff 15 b4 13 00 01 call   *0x10013b4  
2e: 89 45 fc          mov    %eax,-0x4(%ebp)  
31: 85 c0          test   %eax,%eax  
33: 0f 84 d9 00 00 00 je    0x112  
39: 56             push   %esi
```

## 8.10. SAP

```
3a: 57          push %edi
3b: 6a 00       push $0x0
3d: 50          push %eax
3e: ff 15 b8 13 00 01 call *0x10013b8
44: 8b 35 bc 13 00 01 mov 0x10013bc,%esi
4a: 8b f8       mov %eax,%edi
4c: a1 e0 e2 05 01 mov 0x105e2e0,%eax
51: 3b 05 e4 e2 05 01 cmp 0x105e2e4,%eax
57: 75 12       jne 0x6b
59: 53          push %ebx
5a: 6a 03       push $0x3
5c: 57          push %edi
5d: ff d6       call *%esi
```

...

Yes, this is seems correctly disassembled piece of x86 code. The whole dectypted file can be downloaded [here](#).

In fact, this is text section from regedit.exe from Windows 7. But this example is based on a real case I encountered, so just executable is different (and key), algorithm is the same.

### 8.9.1 Other ideas to consider

What if I would fail with such simple frequency analysis? There are other ideas on how to measure correctness of decrypted/decompressed x86 code:

- Many modern compilers aligns functions on 0x10 border. So the space left before is filled with NOPs (0x90) or other NOP instructions with known opcodes: [1.7 on page 1021](#).
- Perhaps, the most frequent pattern in any assembly language is function call:  
PUSH chain / CALL / ADD ESP, X. This sequence can easily detected and found. I've even gathered statistics about average number of function arguments: [10.2 on page 988](#). (Hence, this is average length of PUSH chain.)

By the way, it is interesting to know that the fact that function calls ( PUSH / CALL / ADD ) and MOV instructions are the most frequently executed pieces of code in almost all programs we use. In other words, CPU is very busy passing information between levels of abstractions, or, it can be said, it's very busy switching between these levels. This is a cost of splitting problems into several levels of abstractions.

## 8.10 SAP

### 8.10.1 About SAP client network traffic compression

(Tracing the connection between the TDW\_NOCOMPRESS SAPGUI<sup>25</sup> environment variable and the pesky annoying pop-up window and the actual data compression routine.)

It is known that the network traffic between SAPGUI and SAP is not encrypted by default, but compressed (see here<sup>26</sup> and here<sup>27</sup>).

It is also known that by setting the environment variable TDW\_NOCOMPRESS to 1, it is possible to turn the network packet compression off.

But you will see an annoying pop-up window that cannot be closed:

<sup>25</sup>SAP GUI client

<sup>26</sup><http://go.yurichev.com/17221>

<sup>27</sup>[blog.yurichev.com](http://blog.yurichev.com)

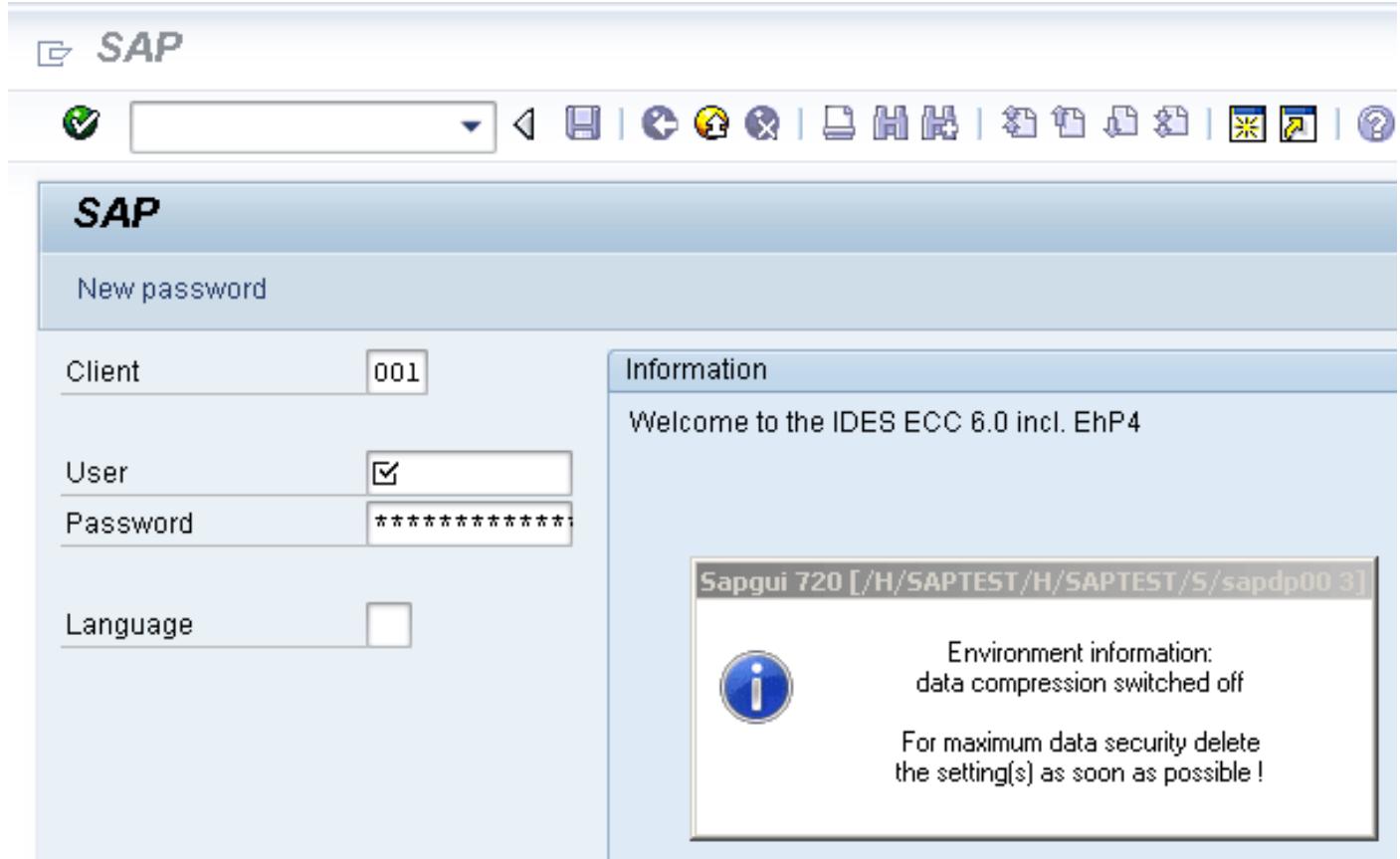


Figure 8.17: Screenshot

Let's see if we can remove the window somehow.

But before this, let's see what we already know.

First: we know that the environment variable *TDW\_NOCOMPRESS* is checked somewhere inside the SAPGUI client.

Second: a string like "data compression switched off" must be present somewhere in it.

With the help of the FAR file manager<sup>28</sup> we can find that both of these strings are stored in the SAPguilib.dll file.

So let's open SAPguilib.dll in [IDA](#) and search for the "*TDW\_NOCOMPRESS*" string. Yes, it is present and there is only one reference to it.

We see the following fragment of code (all file offsets are valid for SAPGUI 720 win32, SAPguilib.dll file version 7200,1,0,9009):

```
.text:6440D51B      lea    eax, [ebp+2108h+var_211C]
.text:6440D51E      push   eax          ; int
.text:6440D51F      push   offset aTdw_nocompress ; "TDW_NOCOMPRESS"
.text:6440D524      mov    byte ptr [edi+15h], 0
.text:6440D528      call   chk_env
.text:6440D52D      pop    ecx
.text:6440D52E      pop    ecx
.text:6440D52F      push   offset byte_64443AF8
.text:6440D534      lea    ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537      call   ds:mfc90_1603
.text:6440D53D      test   eax, eax
.text:6440D53F      jz    short loc_6440D55A
.text:6440D541      lea    ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:6440D544      call   ds:mfc90_910
```

<sup>28</sup><http://go.yurichev.com/17347>

## 8.10. SAP

```
.text:6440D54A          push    eax           ; Str
.text:6440D54B          call    ds:atoi
.text:6440D551          test    eax, eax
.text:6440D553          setnz   al
.text:6440D556          pop    ecx
.text:6440D557          mov     [edi+15h], al
```

The string returned by `chk_env()` via its second argument is then handled by the MFC string functions and then `atoi()`<sup>29</sup> is called. After that, the numerical value is stored in `edi+15h`.

Also take a look at the `chk_env()` function (we gave this name to it manually):

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.text:64413F20 chk_env      proc near
.text:64413F20
.text:64413F20 DstSize      = dword ptr -0Ch
.text:64413F20 var_8        = dword ptr -8
.text:64413F20 DstBuf       = dword ptr -4
.text:64413F20 VarName      = dword ptr 8
.text:64413F20 arg_4        = dword ptr 0Ch
.text:64413F20
.text:64413F20             push    ebp
.text:64413F21             mov     ebp, esp
.text:64413F23             sub    esp, 0Ch
.text:64413F26             mov     [ebp+DstSize], 0
.text:64413F2D             mov     [ebp+DstBuf], 0
.text:64413F34             push    offset unk_6444C88C
.text:64413F39             mov     ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C             call    ds:mfc90_820
.text:64413F42             mov     eax, [ebp+VarName]
.text:64413F45             push    eax           ; VarName
.text:64413F46             mov     ecx, [ebp+DstSize]
.text:64413F49             push    ecx           ; DstSize
.text:64413F4A             mov     edx, [ebp+DstBuf]
.text:64413F4D             push    edx           ; DstBuf
.text:64413F4E             lea    eax, [ebp+DstSize]
.text:64413F51             push    eax           ; ReturnSize
.text:64413F52             call    ds:getenv_s
.text:64413F58             add    esp, 10h
.text:64413F5B             mov     [ebp+var_8], eax
.text:64413F5E             cmp    [ebp+var_8], 0
.text:64413F62             jz    short loc_64413F68
.text:64413F64             xor    eax, eax
.text:64413F66             jmp    short loc_64413FBC
.text:64413F68
.text:64413F68 loc_64413F68:
.text:64413F68             cmp    [ebp+DstSize], 0
.text:64413F6C             jnz    short loc_64413F72
.text:64413F6E             xor    eax, eax
.text:64413F70             jmp    short loc_64413FBC
.text:64413F72
.text:64413F72 loc_64413F72:
.text:64413F72             mov    ecx, [ebp+DstSize]
.text:64413F75             push   ecx
.text:64413F76             mov    ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)
.text:64413F79             call    ds:mfc90_2691
.text:64413F7F             mov     [ebp+DstBuf], eax
.text:64413F82             mov     edx, [ebp+VarName]
.text:64413F85             push    edx           ; VarName
.text:64413F86             mov     eax, [ebp+DstSize]
.text:64413F89             push    eax           ; DstSize
.text:64413F8A             mov     ecx, [ebp+DstBuf]
.text:64413F8D             push    ecx           ; DstBuf
```

<sup>29</sup>standard C library function that converts the digits in a string to a number

## 8.10. SAP

```

.text:64413F8E          lea    edx, [ebp+DstSize]
.text:64413F91          push   edx           ; ReturnSize
.text:64413F92          call   ds:getenv_s
.text:64413F98          add    esp, 10h
.text:64413F9B          mov    [ebp+var_8], eax
.text:64413F9E          push   0xFFFFFFFFh
.text:64413FA0          mov    ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT::ReleaseBuffer(int)
.text:64413FA3          call   ds:mfc90_5835
.text:64413FA9          cmp    [ebp+var_8], 0
.text:64413FAD          jz    short loc_64413FB3
.text:64413FAF          xor    eax, eax
.text:64413FB1          jmp    short loc_64413FBC
.text:64413FB3
.text:64413FB3 loc_64413FB3:
.text:64413FB3          mov    ecx, [ebp+arg_4]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64413FB6          call   ds:mfc90_910
.text:64413FBC
.text:64413FBC loc_64413FBC:
.text:64413FBC
.text:64413FBC          mov    esp, ebp
.text:64413FBE          pop    ebp
.text:64413FBF          retn
.text:64413FBF chk_env  endp

```

Yes. The `getenv_s()`<sup>30</sup>

function is a Microsoft security-enhanced version of `getenv()`<sup>31</sup>.

There are also some MFC string manipulations.

Lots of other environment variables are checked as well. Here is a list of all variables that are being checked and what SAPGUI would write to its trace log when logging is turned on:

DPTRACE	"GUI-OPTION: Trace set to %d"
TDW_HEXDUMP	"GUI-OPTION: Hexdump enabled"
TDW_WORKDIR	"GUI-OPTION: working directory '%s'"
TDW_SPLASHSRCEENOFF	"GUI-OPTION: Splash Screen Off"
	"GUI-OPTION: Splash Screen On"
TDW_REPLYTIMEOUT	"GUI-OPTION: reply timeout %d milliseconds"
TDW_PLAYBACKTIMEOUT	"GUI-OPTION: PlaybackTimeout set to %d milliseconds"
TDW_NOCOMPRESS	"GUI-OPTION: no compression read"
TDW_EXPERT	"GUI-OPTION: expert mode"
TDW_PLAYBACKPROGRESS	"GUI-OPTION: PlaybackProgress"
TDW_PLAYBACKNETTRAFFIC	"GUI-OPTION: PlaybackNetTraffic"
TDW_PLAYLOG	"GUI-OPTION: /PlayLog is YES, file %s"
TDW_PLAYTIME	"GUI-OPTION: /PlayTime set to %d milliseconds"
TDW_LOGFILE	"GUI-OPTION: TDW_LOGFILE '%s'"
TDW_WAN	"GUI-OPTION: WAN - low speed connection enabled"
TDW_FULLSCREEN	"GUI-OPTION: FullMenu enabled"
SAP_CP / SAP_CODEPAGE	"GUI-OPTION: SAP_CODEPAGE '%d'"
UPDOWNLOAD_CP	"GUI-OPTION: UPDOWNLOAD_CP '%d'"
SNC_PARTNERNAME	"GUI-OPTION: SNC name '%s'"
SNC_QOP	"GUI-OPTION: SNC_QOP '%s'"
SNC_LIB	"GUI-OPTION: SNC is set to: %s"
SAPGUI_INPLACE	"GUI-OPTION: environment variable SAPGUI_INPLACE is on"

The settings for each variable are written in the array via a pointer in the `EDI` register. `EDI` is set before the function call:

```

.text:6440EE00          lea    edi, [ebp+2884h+var_2884] ; options here like +0x15...
.text:6440EE03          lea    ecx, [esi+24h]
.text:6440EE06          call   load_command_line
.text:6440EE0B          mov    edi, eax
.text:6440EE0D          xor    ebx, ebx

```

<sup>30</sup>[MSDN](#)

<sup>31</sup>Standard C library returning environment variable

## 8.10. SAP

```
.text:6440EE0F          cmp    edi, ebx
.text:6440EE11          jz     short loc_6440EE42
.text:6440EE13          push   edi
.text:6440EE14          push   offset aSapguiStoppedA ; "Sapgui stopped after ↴
    ↴ commandline interp"...
.text:6440EE19          push   dword_644F93E8
.text:6440EE1F          call   FEWTraceError
```

Now, can we find the “*data record mode switched on*” string?

Yes, and the only reference is in

```
CDwsGui::PrepareInfoWindow().
```

How do we get know the class/method names? There are a lot of special debugging calls that write to the log files, like:

```
.text:64405160          push   dword ptr [esi+2854h]
.text:64405166          push   offset aCdwsGuiPrepare ; "\nCDwsGui::PrepareInfoWindow: ↴
    ↴ sapgui env"...
.text:6440516B          push   dword ptr [esi+2848h]
.text:64405171          call   dbg
.text:64405176          add    esp, 0Ch
```

...or:

```
.text:6440237A          push   eax
.text:6440237B          push   offset aCClientStart_6 ; "CClient::Start: set shortcut ↴
    ↴ user to '\%'"...
.text:64402380          push   dword ptr [edi+4]
.text:64402383          call   dbg
.text:64402388          add    esp, 0Ch
```

It is very useful.

So let's see the contents of this pesky annoying pop-up window's function:

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam      = byte ptr -3Ch
.text:64404F4F var_38       = dword ptr -38h
.text:64404F4F var_34       = dword ptr -34h
.text:64404F4F rc           = tagRECT ptr -2Ch
.text:64404F4F cy           = dword ptr -1Ch
.text:64404F4F h            = dword ptr -18h
.text:64404F4F var_14       = dword ptr -14h
.text:64404F4F var_10       = dword ptr -10h
.text:64404F4F var_4        = dword ptr -4
.text:64404F4F
.text:64404F4F push    30h
.text:64404F51 mov     eax, offset loc_64438E00
.text:64404F56 call   __EH_prolog3
.text:64404F5B mov     esi, ecx      ; ECX is pointer to object
.text:64404F5D xor    ebx, ebx
.text:64404F5F lea     ecx, [ebp+var_14]
.text:64404F62 mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65 call   ds:mfc90_316
.text:64404F6B mov     [ebp+var_4], ebx
.text:64404F6E lea     edi, [esi+2854h]
.text:64404F74 push   offset aEnvironmentInf ; "Environment information:\n"
.text:64404F79 mov     ecx, edi

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B call   ds:mfc90_820
.text:64404F81 cmp    [esi+38h], ebx
.text:64404F84 mov     ebx, ds:mfc90_2539
.text:64404F8A jbe   short loc_64404FA9
.text:64404F8C push   dword ptr [esi+34h]
.text:64404F8F lea     eax, [ebp+var_14]
```

## 8.10. SAP

```
.text:64404F92          push    offset aWorkingDirecto ; "working directory: '\%s'\n"
.text:64404F97          push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98          call    ebx ; mfc90_2539
.text:64404F9A          add    esp, 0Ch
.text:64404F9D          lea    eax, [ebp+var_14]
.text:64404FA0          push   eax
.text:64404FA1          mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FA3          call    ds:mfc90_941
.text:64404FA9 loc_64404FA9:
.text:64404FA9          mov    eax, [esi+38h]
.text:64404FAC          test   eax, eax
.text:64404FAE          jbe    short loc_64404FD3
.text:64404FB0          push   eax
.text:64404FB1          lea    eax, [ebp+var_14]
.text:64404FB4          push   offset aTraceLevelDAct ; "trace level \%d activated\n"
.text:64404FB9          push   eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA          call    ebx ; mfc90_2539
.text:64404FBC          add    esp, 0Ch
.text:64404FBF          lea    eax, [ebp+var_14]
.text:64404FC2          push   eax
.text:64404FC3          mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FC5          call    ds:mfc90_941
.text:64404FCB          xor    ebx, ebx
.text:64404FCD          inc    ebx
.text:64404FCE          mov    [ebp+var_10], ebx
.text:64404FD1          jmp    short loc_64404FD6
.text:64404FD3 loc_64404FD3:
.text:64404FD3          xor    ebx, ebx
.text:64404FD5          inc    ebx
.text:64404FD6 loc_64404FD6:
.text:64404FD6          cmp    [esi+38h], ebx
.text:64404FD9          jbe    short loc_64404FF1
.text:64404FDB          cmp    dword ptr [esi+2978h], 0
.text:64404FE2          jz     short loc_64404FF1
.text:64404FE4          push   offset aHexdumpInTrace ; "hexdump in trace activated\n"
.text:64404FE9          mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB          call    ds:mfc90_945
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1          cmp    byte ptr [esi+78h], 0
.text:64404FF5          jz     short loc_64405007
.text:64404FF7          push   offset aLoggingActivat ; "logging activated\n"
.text:64404FFC          mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE          call    ds:mfc90_945
.text:64405004          mov    [ebp+var_10], ebx
.text:64405007 loc_64405007:
.text:64405007          cmp    byte ptr [esi+3Dh], 0
.text:6440500B          jz     short bypass
.text:6440500D          push   offset aDataCompressio ; "data compression switched off\n"
     ↴ n"
.text:64405012          mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
```

## 8.10. SAP

```
.text:64405014          call   ds:mfc90_945
.text:6440501A          mov    [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D          mov    eax, [esi+20h]
.text:64405020          test   eax, eax
.text:64405022          jz    short loc_6440503A
.text:64405024          cmp    dword ptr [eax+28h], 0
.text:64405028          jz    short loc_6440503A
.text:6440502A          push   offset aDataRecordMode ; "data record mode switched on\n\n"
                           \
.text:6440502F          mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031          call   ds:mfc90_945
.text:64405037          mov    [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A          mov    ecx, edi
.text:6440503C          cmp    [ebp+var_10], ebx
.text:6440503F          jnz   loc_64405142
.text:64405045          push   offset aForMaximumData ; "\nFor maximum data security\n
                           \ delete\nthe s"...

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A          call   ds:mfc90_945
.text:64405050          xor    edi, edi
.text:64405052          push   edi      ; fWinIni
.text:64405053          lea     eax, [ebp+pvParam]
.text:64405056          push   eax      ; pvParam
.text:64405057          push   edi      ; uiParam
.text:64405058          push   30h     ; uiAction
.text:6440505A          call   ds:SystemParametersInfoA
.text:64405060          mov    eax, [ebp+var_34]
.text:64405063          cmp    eax, 1600
.text:64405068          jle   short loc_64405072
.text:6440506A          cdq
.text:6440506B          sub    eax, edx
.text:6440506D          sar    eax, 1
.text:6440506F          mov    [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:
.text:64405072          push   edi      ; hWnd
.text:64405073          mov    [ebp+cy], 0A0h
.text:6440507A          call   ds:GetDC
.text:64405080          mov    [ebp+var_10], eax
.text:64405083          mov    ebx, 12Ch
.text:64405088          cmp    eax, edi
.text:6440508A          jz    loc_64405113
.text:64405090          push   11h     ; i
.text:64405092          call   ds:GetStockObject
.text:64405098          mov    edi, ds:SelectObject
.text:6440509E          push   eax      ; h
.text:6440509F          push   [ebp+var_10] ; hdc
.text:644050A2          call   edi ; SelectObject
.text:644050A4          and    [ebp+rc.left], 0
.text:644050A8          and    [ebp+rc.top], 0
.text:644050AC          mov    [ebp+h], eax
.text:644050AF          push   401h     ; format
.text:644050B4          lea    eax, [ebp+rc]
.text:644050B7          push   eax      ; lprc
.text:644050B8          lea    ecx, [esi+2854h]
.text:644050BE          mov    [ebp+rc.right], ebx
.text:644050C1          mov    [ebp+rc.bottom], 0B4h

; demangled name: ATL::CSimpleStringT::GetLength(void)
.text:644050C8          call   ds:mfc90_3178
.text:644050CE          push   eax      ; cchText
.text:644050CF          lea    ecx, [esi+2854h]
```

```

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:644050D5          call    ds:mfc90_910
.text:644050DB          push    eax      ; lpchText
.text:644050DC          push    [ebp+var_10] ; hdc
.text:644050DF          call    ds:DrawTextA
.text:644050E5          push    4       ; nIndex
.text:644050E7          call    ds:GetSystemMetrics
.text:644050ED          mov     ecx, [ebp+rc.bottom]
.text:644050F0          sub     ecx, [ebp+rc.top]
.text:644050F3          cmp     [ebp+h], 0
.text:644050F7          lea     eax, [eax+ecx+28h]
.text:644050FB          mov     [ebp+cy], eax
.text:644050FE          jz    short loc_64405108
.text:64405100          push    [ebp+h] ; h
.text:64405103          push    [ebp+var_10] ; hdc
.text:64405106          call    edi ; SelectObject
.text:64405108
.text:64405108 loc_64405108:
.text:64405108          push    [ebp+var_10] ; hDC
.text:6440510B          push    0       ; hWnd
.text:6440510D          call    ds:ReleaseDC
.text:64405113
.text:64405113 loc_64405113:
.text:64405113          mov     eax, [ebp+var_38]
.text:64405116          push    80h      ; uFlags
.text:6440511B          push    [ebp+cy] ; cy
.text:6440511E          inc     eax
.text:6440511F          push    ebx      ; cx
.text:64405120          push    eax      ; Y
.text:64405121          mov     eax, [ebp+var_34]
.text:64405124          add     eax, 0FFFFFED4h
.text:64405129          cdq
.text:6440512A          sub     eax, edx
.text:6440512C          sar     eax, 1
.text:6440512E          push    eax      ; X
.text:6440512F          push    0       ; hWndInsertAfter
.text:64405131          push    dword ptr [esi+285Ch] ; hWnd
.text:64405137          call    ds:SetWindowPos
.text:6440513D          xor     ebx, ebx
.text:6440513F          inc     ebx
.text:64405140          jmp    short loc_6440514D
.text:64405142
.text:64405142 loc_64405142:
.text:64405142          push    offset byte_64443AF8

; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147          call    ds:mfc90_820
.text:6440514D
.text:6440514D loc_6440514D:
.text:6440514D          cmp     dword_6450B970, ebx
.text:64405153          jl    short loc_64405188
.text:64405155          call    sub_6441C910
.text:6440515A          mov     dword_644F858C, ebx
.text:64405160          push    dword ptr [esi+2854h]
.text:64405166          push    offset aCdwsGuiPrepare ; "\nCdwsGui::PrepareInfoWindow: ↵
   ↴ sapgui env"...
.text:6440516B          push    dword ptr [esi+2848h]
.text:64405171          call    dbg
.text:64405176          add     esp, 0Ch
.text:64405179          mov     dword_644F858C, 2
.text:64405183          call    sub_6441C920
.text:64405188
.text:64405188 loc_64405188:
.text:64405188          or     [ebp+var_4], 0FFFFFFFh
.text:6440518C          lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT::~CStringT()
.text:6440518F          call    ds:mfc90_601
.text:64405195          call    __EH_eptLog3

```

## 8.10. SAP

```
.text:6440519A          retn  
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

At the start of the function `ECX` has a pointer to the object (since it is a `thiscall` ([3.19.1 on page 546](#))-type of function). In our case, the object obviously has class type of `CDwsGui`. Depending on the option turned on in the object, a specific message part is to be concatenated with the resulting message.

If the value at address `this+0x3D` is not zero, the compression is off:

```
.text:64405007 loc_64405007:  
.text:64405007          cmp     byte ptr [esi+3Dh], 0  
.text:6440500B          jz      short bypass  
.text:6440500D          push    offset aDataCompression ; "data compression switched off\n"  
    ↳ n"  
.text:64405012          mov     ecx, edi  
  
; demangled name: ATL::CStringT::operator+=(char const *)  
.text:64405014          call    ds:mfc90_945  
.text:6440501A          mov     [ebp+var_10], ebx  
.text:6440501D          mov     [ebp+var_10], ebx  
.text:6440501D bypass:
```

It is interesting that finally the `var_10` variable state defines whether the message is to be shown at all:

```
.text:6440503C          cmp     [ebp+var_10], ebx  
.text:6440503F          jnz    exit ; bypass drawing  
  
; add strings "For maximum data security delete" / "the setting(s) as soon as possible !":  
  
.text:64405045          push    offset aForMaximumData ; "\nFor maximum data security  
    ↳ delete\nthe s..."  
.text:6440504A          call    ds:mfc90_945 ; ATL::CStringT::operator+=(char const *)  
.text:64405050          xor    edi, edi  
.text:64405052          push    edi ; fWinIni  
.text:64405053          lea     eax, [ebp+pvParam]  
.text:64405056          push    eax ; pvParam  
.text:64405057          push    edi ; uiParam  
.text:64405058          push    30h ; uiAction  
.text:6440505A          call    ds:SystemParametersInfoA  
.text:64405060          mov     eax, [ebp+var_34]  
.text:64405063          cmp     eax, 1600  
.text:64405068          jle    short loc_64405072  
.text:6440506A          cdq  
.text:6440506B          sub    eax, edx  
.text:6440506D          sar    eax, 1  
.text:6440506F          mov     [ebp+var_34], eax  
.text:64405072          loc_64405072:  
  
start drawing:  
  
.text:64405072          push    edi ; hWnd  
.text:64405073          mov     [ebp+cy], 0A0h  
.text:6440507A          call    ds:GetDC
```

Let's check our theory on practice.

JNZ at this line ...

```
.text:6440503F          jnz    exit ; bypass drawing
```

...replace it with just `JMP`, and we get SAPGUI working without the pesky annoying pop-up window appearing!

Now let's dig deeper and find a connection between the `0x15` offset in the `load_command_line()` (we gave it this name) function and the `this+0x3D` variable in `CDwsGui::PrepareInfoWindow`. Are we sure the value is the same?

## 8.10. SAP

We are starting to search for all occurrences of the `0x15` value in code. For a small programs like SAPGUI, it sometimes works. Here is the first occurrence we've got:

```
.text:64404C19 sub_64404C19    proc near
.text:64404C19                = dword ptr 4
.text:64404C19 arg_0          push    ebx
.text:64404C19                push    ebp
.text:64404C1B                push    esi
.text:64404C1C                push    edi
.text:64404C1D                mov     edi, [esp+10h+arg_0]
.text:64404C21                mov     eax, [edi]
.text:64404C23                mov     esi, ecx ; ESI/ECX are pointers to some unknown object.
.text:64404C25                mov     [esi], eax
.text:64404C27                mov     eax, [edi+4]
.text:64404C2A                mov     [esi+4], eax
.text:64404C2D                mov     eax, [edi+8]
.text:64404C30                mov     [esi+8], eax
.text:64404C33                lea     eax, [edi+0Ch]
.text:64404C36                push    eax
.text:64404C37                lea     ecx, [esi+0Ch]

; demangled name: ATL::CStringT::operator=(class ATL::CStringT ... &
.text:64404C3A                call    ds:mfc90_817
.text:64404C40                mov     eax, [edi+10h]
.text:64404C43                mov     [esi+10h], eax
.text:64404C46                mov     al, [edi+14h]
.text:64404C49                mov     [esi+14h], al
.text:64404C4C                mov     al, [edi+15h] ; copy byte from 0x15 offset
.text:64404C4F                mov     [esi+15h], al ; to 0x15 offset in CDwsGui object
```

The function has been called from the function named `CDwsGui::CopyOptions!` And thanks again for debugging information.

But the real answer is in `CDwsGui::Init()`:

```
.text:6440B0BF loc_6440B0BF:
.text:6440B0BF                mov     eax, [ebp+arg_0]
.text:6440B0C2                push   [ebp+arg_4]
.text:6440B0C5                mov     [esi+2844h], eax
.text:6440B0CB                lea     eax, [esi+28h] ; ESI is pointer to CDwsGui object
.text:6440B0CE                push   eax
.text:6440B0CF                call   CDwsGui__CopyOptions
```

Finally, we understand: the array filled in the `load_command_line()` function is actually placed in the `CDwsGui` class, but at address `this+0x28`. `0x15 + 0x28` is exactly `0x3D`. OK, we found the point where the value is copied to.

Let's also find the rest of the places where the `0x3D` offset is used. Here is one of them in the `CDwsGui::SapguiRun` function (again, thanks to the debugging calls):

```
.text:64409D58                cmp     [esi+3Dh], bl ; ESI is pointer to CDwsGui object
.text:64409D5B                lea     ecx, [esi+2B8h]
.text:64409D61                setz   al
.text:64409D64                push   eax           ; arg_10 of CConnectionContext::CreateNetwork
.text:64409D65                push   dword ptr [esi+64h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D68                call   ds:mfc90_910
.text:64409D68                ; no arguments
.text:64409D6E                push   eax
.text:64409D6F                lea     ecx, [esi+2BCh]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D75                call   ds:mfc90_910
.text:64409D75                ; no arguments
.text:64409D7B                push   eax
.text:64409D7C                push   esi
```

## 8.10. SAP

.text:64409D7D	lea	ecx, [esi+8]
.text:64409D80	call	CConnectionContext__CreateNetwork

Let's check our findings.

Replace the `setz al` here with the `xor eax, eax / nop` instructions, clear the `TDW_NOCOMPRESS` environment variable and run SAPGUI. Wow! There pesky annoying window is no more (just as expected, because the variable is not set) but in Wireshark we can see that the network packets are not compressed anymore! Obviously, this is the point where the compression flag is to be set in the `CConnectionContext` object.

So, the compression flag is passed in the 5th argument of `CConnectionContext::CreateNetwork`. Inside the function, another one is called:

```
...
.text:64403476          push    [ebp+compression]
.text:64403479          push    [ebp+arg_C]
.text:6440347C          push    [ebp+arg_8]
.text:6440347F          push    [ebp+arg_4]
.text:64403482          push    [ebp+arg_0]
.text:64403485          call    CNetwork__CNetwork
```

The compression flag is passed here in the 5th argument to the `CNetwork::CNetwork` constructor.

And here is how the `CNetwork` constructor sets the flag in the `CNetwork` object according to its 5th argument *and* another variable which probably could also affect network packets compression.

```
.text:64411DF1          cmp     [ebp+compression], esi
.text:64411DF7          jz      short set_EAX_to_0
.text:64411DF9          mov     al, [ebx+78h] ; another value may affect compression?
.text:64411DFC          cmp     al, '3'
.text:64411DFE          jz      short set_EAX_to_1
.text:64411E00          cmp     al, '4'
.text:64411E02          jnz    short set_EAX_to_0
.text:64411E04
.text:64411E04 set_EAX_to_1:
.text:64411E04          xor    eax, eax
.text:64411E06          inc    eax           ; EAX -> 1
.text:64411E07          jmp    short loc_64411E0B
.text:64411E09
.text:64411E09 set_EAX_to_0:
.text:64411E09          xor    eax, eax       ; EAX -> 0
.text:64411E0B
.text:64411E0B loc_64411E0B:
.text:64411E0B          mov    [ebx+3A4h], eax ; EBX is pointer to CNetwork object
```

At this point we know the compression flag is stored in the `CNetwork` class at address `this+0x3A4`.

Now let's dig through `SAPguilib.dll` for the `0x3A4` value. And here is the second occurrence in `CDws-Gui::OnClientMessageWrite` (endless thanks for the debugging information):

```
.text:64406F76 loc_64406F76:
.text:64406F76          mov    ecx, [ebp+7728h+var_7794]
.text:64406F79          cmp    dword ptr [ecx+3A4h], 1
.text:64406F80          jnz    compression_flag_is_zero
.text:64406F86          mov    byte ptr [ebx+7], 1
.text:64406F8A          mov    eax, [esi+18h]
.text:64406F8D          mov    ecx, eax
.text:64406F8F          test   eax, eax
.text:64406F91          ja     short loc_64406FFF
.text:64406F93          mov    ecx, [esi+14h]
.text:64406F96          mov    eax, [esi+20h]
.text:64406F99
.text:64406F99 loc_64406F99:
.text:64406F99          push   dword ptr [edi+2868h] ; int
.text:64406F9F          lea    edx, [ebp+7728h+var_77A4]
.text:64406FA2          push   edx           ; int
.text:64406FA3          push   30000          ; int
.text:64406FA8          lea    edx, [ebp+7728h+Dst]
.text:64406FAB          push   edx           ; Dst
```

## 8.10. SAP

```
.text:64406FAC      push    ecx          ; int
.text:64406FAD      push    eax          ; Src
.text:64406FAE      push    dword ptr [edi+28C0h] ; int
.text:64406FB4      call    sub_644055C5      ; actual compression routine
.text:64406FB9      add     esp, 1Ch
.text:64406FBC      cmp     eax, 0FFFFFFF6h
.text:64406FBF      jz     short loc_64407004
.text:64406FC1      cmp     eax, 1
.text:64406FC4      jz     loc_6440708C
.text:64406FCA      cmp     eax, 2
.text:64406FCD      jz     short loc_64407004
.text:64406FCF      push    eax
.text:64406FD0      push    offset aCompressionErr ; "compression error [rc = \%d]- ↵
    ↪ program wi"...
.text:64406FD5      push    offset aGui_err_compre ; "GUI_ERR_COMPRESS"
.text:64406FDA      push    dword ptr [edi+28D0h]
.text:64406FE0      call    SapPcTxtRead
```

Let's take a look in `sub_644055C5`. In it we can only see the call to `memcpy()` and another function named (by IDA) `sub_64417440`.

And, let's take a look inside `sub_64417440`. What we see is:

```
.text:6441747C      push    offset aErrorCsSrcCompre ; "\nERROR: CsRCompress: invalid ↵
    ↪ handle"
.text:64417481      call    eax ; dword_644F94C8
.text:64417483      add     esp, 4
```

Voilà! We've found the function that actually compresses the data. As it was shown in past <sup>32</sup>, this function is used in SAP and also the open-source MaxDB project. So it is available in source form.

Doing the last check here:

```
.text:64406F79      cmp     dword ptr [ecx+3A4h], 1
.text:64406F80      jnz    compression_flag_is_zero
```

Replace `JNZ` here for an unconditional `JMP`. Remove the environment variable `TDW_NOCOMPRESS`. Voilà!

In Wireshark we see that the client messages are not compressed. The server responses, however, are compressed.

So we found exact connection between the environment variable and the point where data compression routine can be called or bypassed.

## 8.10.2 SAP 6.0 password checking functions

One time when the author of this book have returned again to his SAP 6.0 IDES installed in a VMware box, he figured out that he forgot the password for the SAP\* account, then he have recalled it, but then he got this error message «*Password logon no longer possible - too many failed attempts*», since he've made all these attempts in attempt to recall it.

The first extremely good news was that the full `disp+work.pdb` PDB file is supplied with SAP, and it contain almost everything: function names, structures, types, local variable and argument names, etc. What a lavish gift!

There is `TYPEINFODUMP`<sup>33</sup> utility for converting PDB files into something readable and grepable.

Here is an example of a function information + its arguments + its local variables:

```
FUNCTION ThVmcsEvent
  Address: 10143190  Size: 675 bytes Index: 60483 TypeIndex: 60484
  Type: int NEAR_C ThVmcsEvent (unsigned int, unsigned char, unsigned short*)
Flags: 0
PARAMETER events
  Address: Reg335+288  Size: 4 bytes Index: 60488 TypeIndex: 60489
```

<sup>32</sup><http://go.yurichev.com/17312>

<sup>33</sup><http://go.yurichev.com/17038>

## 8.10. SAP

```
Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296  Size:      1 bytes  Index:    60490  TypeIndex:    60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304  Size:      8 bytes  Index:    60492  TypeIndex:    60493
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address: 12274af0  Size:      8 bytes  Index:    60495  TypeIndex:    60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304  Size:      8 bytes  Index:    60498  TypeIndex:    60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64   Size:    204 bytes  Index:    60501  TypeIndex:    60502
  Type: AD_RECORD
Flags: 90
LOCAL_VAR adlen
  Address: Reg335+296  Size:      4 bytes  Index:    60508  TypeIndex:    60509
  Type: int
Flags: 90
```

And here is an example of some structure:

```
STRUCT DBSL_STMTID
Size: 120  Variables: 4  Functions: 0  Base classes: 0
MEMBER moduletype
  Type: DBSL_MODULETYPE
  Offset: 0  Index: 3  TypeIndex: 38653
MEMBER module
  Type: wchar_t module[40]
  Offset: 4  Index: 3  TypeIndex: 831
MEMBER stmtnum
  Type: long
  Offset: 84  Index: 3  TypeIndex: 440
MEMBER timestamp
  Type: wchar_t timestamp[15]
  Offset: 88  Index: 3  TypeIndex: 6612
```

Wow!

Another good news: *debugging* calls (there are plenty of them) are very useful.

Here you can also notice the *ct\_level* global variable<sup>34</sup>, that reflects the current trace level.

There are a lot of debugging inserts in the *disp+work.exe* file:

```
cmp    cs:ct_level, 1
jl     short loc_1400375DA
call   DpLock
lea    rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov    edx, 4Eh           ; line
call   CTrcSaveLocation
mov    r8, cs:func_48
mov    rcx, cs:hd1         ; hd1
lea    rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov    r9d, ebx
call   DpTrcErr
call   DpUnlock
```

If the current trace level is bigger or equal to threshold defined in the code here, a debugging message is to be written to the log files like *dev\_w0*, *dev\_disp*, and other *dev\** files.

Let's try grepping in the file that we have got with the help of the TYPEINFODUMP utility:

<sup>34</sup>More about trace level: <http://go.yurichev.com/17039>

## 8.10. SAP

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

We have got:

```
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeName
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor'':`1'::dtor$1
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor'':`1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password
```

Let's also try to search for debug messages which contain the words «password» and «locked». One of them is the string «*user was locked by subsequently failed password logon attempts*», referenced in function *password\_attempt\_limit\_exceeded()*.

Other strings that this function can write to a log file are: «*password logon attempt will be rejected immediately (preventing dictionary attacks)*», «*failed-logon lock: expired (but not removed due to 'read-only' operation)*», «*failed-logon lock: expired => removed*».

After playing for a little with this function, we noticed that the problem is exactly in it. It is called from the *chckpass()* function —one of the password checking functions.

First, we would like to make sure that we are at the correct point:

Run [tracer](#):

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode
```

```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1
    ↴      ", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

The call path is: *syssigni() -> DylSigni() -> dychkusr() -> usrexist() -> chckpass()*.

The number 0x35 is an error returned in *chckpass()* at that point:

## 8.10. SAP

```
.text:00000001402ED567 loc_1402ED567: ; CODE XREF: chckpass+B4
.text:00000001402ED567          mov    rcx, rbx      ; usr02
.text:00000001402ED56A          call   password_idle_check
.text:00000001402ED56F          cmp    eax, 33h
.text:00000001402ED572          jz    loc_1402EDB4E
.text:00000001402ED578          cmp    eax, 36h
.text:00000001402ED57B          jz    loc_1402EDB3D
.text:00000001402ED581          xor    edx, edx      ; usr02_READONLY
.text:00000001402ED583          mov    rcx, rbx      ; usr02
.text:00000001402ED586          call   password_attempt_limit_exceeded
.text:00000001402ED58B          test  al, al
.text:00000001402ED58D          jz    short loc_1402ED5A0
.text:00000001402ED58F          mov    eax, 35h
.text:00000001402ED594          add    rsp, 60h
.text:00000001402ED598          pop    r14
.text:00000001402ED59A          pop    r12
.text:00000001402ED59C          pop    rdi
.text:00000001402ED59D          pop    rsi
.text:00000001402ED59E          pop    rbx
.text:00000001402ED59F          retn
```

Fine, let's check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!password_attempt_limit_exceeded,args:4,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0x257758, 0) ↴
  ↴ (called from 0x1402ed58b (disp+work.exe!chckpass+0xeb))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0, 0) (called ↴
  ↴ from 0x1402e9794 (disp+work.exe!chngpass+0xe4))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Excellent! We can successfully login now.

By the way, we can pretend we forgot the password, fixing the *chckpass()* function to return a value of 0 is enough to bypass the check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus
  ↴ ", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

What also can be said while analyzing the *password\_attempt\_limit\_exceeded()* function is that at the very beginning of it, this call can be seen:

```
lea    rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call  sapgparam
test  rax, rax
jz    short loc_1402E19DE
movzx eax, word ptr [rax]
cmp   ax, 'N'
jz    short loc_1402E19D4
cmp   ax, 'n'
jz    short loc_1402E19D4
cmp   ax, '0'
jnz   short loc_1402E19DE
```

Obviously, function *sapgparam()* is used to query the value of some configuration parameter. This function can be called from 1768 different places. It seems that with the help of this information, we can easily find the places in code, the control flow of which can be affected by specific configuration parameters.

It is really sweet. The function names are very clear, much clearer than in the Oracle RDBMS.

It seems that the *disp+work* process is written in C++. Has it been rewritten some time ago?

## 8.11 Oracle RDBMS

### 8.11.1 V\$VERSION table in the Oracle RDBMS

Oracle RDBMS 11.2 is a huge program, its main module `oracle.exe` contain approx. 124,000 functions. For comparison, the Windows 7 x86 kernel (`ntoskrnl.exe`) contains approx. 11,000 functions and the Linux 3.9.8 kernel (with default drivers compiled)—31,000 functions.

Let's start with an easy question. Where does Oracle RDBMS get all this information, when we execute this simple statement in SQL\*Plus:

```
SQL> select * from V$VERSION;
```

And we get:

```
BANNER
-----
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE    11.2.0.1.0      Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

Let's start. Where in the Oracle RDBMS can we find the string `V$VERSION`?

In the win32-version, `oracle.exe` file contains the string, it's easy to see. But we can also use the object (.o) files from the Linux version of Oracle RDBMS since, unlike the win32 version `oracle.exe`, the function names (and global variables as well) are preserved there.

So, the `kqf.o` file contains the `V$VERSION` string. The object file is in the main Oracle-library `libserver11.a`.

A reference to this text string can find in the `kqfviw` table stored in the same file, `kqf.o`:

Listing 8.8: `kqf.o`

```
.rodata:0800C4A0 kqfviw dd 0Bh      ; DATA XREF: kqfchk:loc_8003A6D
.rodata:0800C4A0                      ; kqfgbn+34
.rodata:0800C4A4      dd offset _2__STRING_10102_0 ; "GV$WAITSTAT"
.rodata:0800C4A8      dd 4
.rodata:0800C4AC      dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4B0      dd 3
.rodata:0800C4B4      dd 0
.rodata:0800C4B8      dd 195h
.rodata:0800C4BC      dd 4
.rodata:0800C4C0      dd 0
.rodata:0800C4C4      dd 0FFFC1CBh
.rodata:0800C4C8      dd 3
.rodata:0800C4CC      dd 0
.rodata:0800C4D0      dd 0Ah
.rodata:0800C4D4      dd offset _2__STRING_10104_0 ; "V$WAITSTAT"
.rodata:0800C4D8      dd 4
.rodata:0800C4DC      dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4E0      dd 3
.rodata:0800C4E4      dd 0
.rodata:0800C4E8      dd 4Eh
.rodata:0800C4EC      dd 3
.rodata:0800C4F0      dd 0
.rodata:0800C4F4      dd 0FFFC003h
.rodata:0800C4F8      dd 4
.rodata:0800C4FC      dd 0
.rodata:0800C500      dd 5
.rodata:0800C504      dd offset _2__STRING_10105_0 ; "GV$BH"
.rodata:0800C508      dd 4
.rodata:0800C50C      dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C510      dd 3
.rodata:0800C514      dd 0
.rodata:0800C518      dd 269h
```

## 8.11. ORACLE RDBMS

```
.rodata:0800C51C      dd 15h
.rodata:0800C520      dd 0
.rodata:0800C524      dd 0FFFFC1EDh
.rodata:0800C528      dd 8
.rodata:0800C52C      dd 0
.rodata:0800C530      dd 4
.rodata:0800C534      dd offset _2__STRING_10106_0 ; "V$BH"
.rodata:0800C538      dd 4
.rodata:0800C53C      dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C540      dd 3
.rodata:0800C544      dd 0
.rodata:0800C548      dd 0F5h
.rodata:0800C54C      dd 14h
.rodata:0800C550      dd 0
.rodata:0800C554      dd 0FFFFC1EEh
.rodata:0800C558      dd 5
.rodata:0800C55C      dd 0
```

By the way, often, while analyzing Oracle RDBMS's internals, you may ask yourself, why are the names of the functions and global variable so weird.

Probably, because Oracle RDBMS is a very old product and was developed in C in the 1980s.

And that was a time when the C standard guaranteed that the function names/variables can support only up to 6 characters inclusive: «6 significant initial characters in an external identifier»<sup>35</sup>

Probably, the table `kqfviw` contains most (maybe even all) views prefixed with V\$, these are *fixed views*, present all the time. Superficially, by noticing the cyclic recurrence of data, we can easily see that each `kqfviw` table element has 12 32-bit fields. It is very simple to create a 12-elements structure in [IDA](#) and apply it to all table elements. As of Oracle RDBMS version 11.2, there are 1023 table elements, i.e., in it are described 1023 of all possible *fixed views*.

We are going to return to this number later.

As we can see, there is not much information in these numbers in the fields. The first number is always equals to the name of the view (without the terminating zero). This is correct for each element. But this information is not very useful.

We also know that the information about all fixed views can be retrieved from a *fixed view* named `V$FIXED_VIEW_DEFINITION` (by the way, the information for this view is also taken from the `kqfviw` and `kqfvip` tables.) By the way, there are 1023 elements in those too. Coincidence? No.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
-----
VIEW_DEFINITION
-----

V$VERSION
select BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

So, `V$VERSION` is some kind of a *thunk view* for another view, named `GV$VERSION`, which is, in turn:

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';

VIEW_NAME
-----
VIEW_DEFINITION
-----

GV$VERSION
select inst_id, banner from x$version
```

The tables prefixed with X\$ in the Oracle RDBMS are service tables too, undocumented, cannot be changed by the user and are refreshed dynamically.

<sup>35</sup>Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988) ([yurichev.com](http://yurichev.com))

## 8.11. ORACLE RDBMS

If we search for the text

```
select BANNER from GV$VERSION where inst_id =  
USERENV('Instance')
```

... in the `kqf.o` file, we find it in the `kqfvip` table:

Listing 8.9: `kqf.o`

```
.rodata:080185A0 kqfvip dd offset _2__STRING_11126_0 ; DATA XREF: kqfgvcn+18  
.rodata:080185A0 ; kqfgvt+F  
.rodata:080185A0 ; "select inst_id,decode(index,1,'data bloc"..."  
.rodata:080185A4 dd offset kqfv459_c_0  
.rodata:080185A8 dd 0  
.rodata:080185AC dd 0  
  
...  
.rodata:08019570 dd offset _2__STRING_11378_0 ; "select BANNER from GV$VERSION where in  
↳ "...  
.rodata:08019574 dd offset kqfv133_c_0  
.rodata:08019578 dd 0  
.rodata:0801957C dd 0  
.rodata:08019580 dd offset _2__STRING_11379_0 ; "select inst_id,decode(bitandcfflg,1) <  
↳ ,0"..."  
.rodata:08019584 dd offset kqfv403_c_0  
.rodata:08019588 dd 0  
.rodata:0801958C dd 0  
.rodata:08019590 dd offset _2__STRING_11380_0 ; "select STATUS , NAME, IS_RECOVERY_DEST <  
↳ "...  
.rodata:08019594 dd offset kqfv199_c_0
```

The table appear to have 4 fields in each element. By the way, there are 1023 elements in it, again, the number we already know.

The second field points to another table that contains the table fields for this *fixed view*. As for `V$VERSION`, this table has only two elements, the first is 6 and the second is the `BANNER` string (the number 6 is this string's length) and after, a *terminating* element that contains 0 and a *null* C string:

Listing 8.10: `kqf.o`

```
.rodata:080BBAC4 kqfv133_c_0 dd 6 ; DATA XREF: .rodata:08019574  
.rodata:080BBAC8 dd offset _2__STRING_5017_0 ; "BANNER"  
.rodata:080BBACC dd 0  
.rodata:080BBAD0 dd offset _2__STRING_0_0
```

By joining data from both `kqfviw` and `kqfvip` tables, we can get the SQL statements which are executed when the user wants to query information from a specific *fixed view*.

So we can write an oracle tables<sup>36</sup> program, to gather all this information from Oracle RDBMS for Linux's object files. For `V$VERSION`, we find this:

Listing 8.11: Result of oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xfffffc085 0x4  
kqfvip_element.statement: [select BANNER from GV$VERSION where inst_id = USERENV('Instance')]  
kqfvip_element.params:  
[BANNER]
```

And:

Listing 8.12: Result of oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xfffffc192 0x1  
kqfvip_element.statement: [select inst_id, banner from x$version]  
kqfvip_element.params:  
[INST_ID] [BANNER]
```

<sup>36</sup>[yurichev.com](http://yurichev.com)

## 8.11. ORACLE RDBMS

The `GV$VERSION` fixed view is different from `V$VERSION` only in that it has one more field with the identifier *instance*.

Anyway, we are going to stick with the `X$VERSION` table. Just like any other X\$-table, it is undocumented, however, we can query it:

```
SQL> select * from x$version;

ADDR          INDX      INST_ID
-----
BANNER
-----
0DBAF574          0           1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
...
...
```

This table has some additional fields, like `ADDR` and `INDX`.

While scrolling `kqf.o` in [IDA](#) we can spot another table that contains a pointer to the `X$VERSION` string, this is `kqftab`:

Listing 8.13: kqf.o

```
.rodata:0803CAC0      dd 9           ; element number 0x1f6
.rodata:0803CAC4      dd offset _2__STRING_13113_0 ; "X$VERSION"
.rodata:0803CAC8      dd 4
.rodata:0803CACC      dd offset _2__STRING_13114_0 ; "kqvt"
.rodata:0803CAD0      dd 4
.rodata:0803CAD4      dd 4
.rodata:0803CAD8      dd 0
.rodata:0803CADC      dd 4
.rodata:0803CAE0      dd 0Ch
.rodata:0803CAE4      dd 0FFFFC075h
.rodata:0803CAE8      dd 3
.rodata:0803CAEC      dd 0
.rodata:0803CAF0      dd 7
.rodata:0803CAF4      dd offset _2__STRING_13115_0 ; "X$KQFSZ"
.rodata:0803CAF8      dd 5
.rodata:0803CAFC      dd offset _2__STRING_13116_0 ; "kqfsz"
.rodata:0803CB00      dd 1
.rodata:0803CB04      dd 38h
.rodata:0803CB08      dd 0
.rodata:0803CB0C      dd 7
.rodata:0803CB10      dd 0
.rodata:0803CB14      dd 0FFFFC09Dh
.rodata:0803CB18      dd 2
.rodata:0803CB1C      dd 0
```

There are a lot of references to the X\$-table names, apparently, to all Oracle RDBMS 11.2 X\$-tables. But again, we don't have enough information.

It's not clear what does the `kqvt` string stands for.

The `kq` prefix may mean *kernel* or *query*.

`v` apparently stands for *version* and `t` —*type*? Hard to say.

A table with a similar name can be found in `kqf.o`:

Listing 8.14: kqf.o

```
.rodata:0808C360 kqvt_c_0 kqftap_param <4, offset _2__STRING_19_0, 917h, 0, 0, 0, 4, 0, 0>
.rodata:0808C360                                     ; DATA XREF: .rodata:08042680
.rodata:0808C360                                     ; "ADDR"
.rodata:0808C384                                     kqftap_param <4, offset _2__STRING_20_0, 0B02h, 0, 0, 0, 4, 0, 0> ; ^{
    ↳ INDX"
.rodata:0808C3A8                                     kqftap_param <7, offset _2__STRING_21_0, 0B02h, 0, 0, 0, 4, 0, 0> ; ^{
    ↳ INST_ID"
```

## 8.11. ORACLE RDBMS

```
.rodata:0808C3CC          kqftap_param <6, offset _2__STRING_5017_0, 601h, 0, 0, 0, 0, 50h, 0, 0> ↵
    ↴ ; "BANNER"
.rodata:0808C3F0          kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, 0, 0, 0, 0, 0>
```

It contains information about all fields in the `X$VERSION` table. The only reference to this table is in the `kqftap` table:

Listing 8.15: `kqf.o`

```
.rodata:08042680          kqftap_element <0, offset kqvt_c_0, offset kqvrow, 0> ; ↵
    ↴ element 0x1f6
```

It is interesting that this element here is `0x1f6th` (502nd), just like the pointer to the `X$VERSION` string in the `kqftab` table.

Probably, the `kqftap` and `kqftab` tables complement each other, just like `kqfvip` and `kqfviw`.

We also see a pointer to the `kqvrow()` function. Finally, we got something useful!

So we will add these tables to our oracle tables<sup>37</sup> utility too. For `X$VERSION` we get:

Listing 8.16: Result of oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xfffffc075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

With the help of `tracer`, it is easy to check that this function is called 6 times in row (from the `qerfxFetch()` function) while querying the `X$VERSION` table.

Let's run `tracer` in `cc` mode (it comments each executed instruction):

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
_kqvrow_ proc near

var_7C      = byte ptr -7Ch
var_18      = dword ptr -18h
var_14      = dword ptr -14h
Dest        = dword ptr -10h
var_C       = dword ptr -0Ch
var_8       = dword ptr -8
var_4       = dword ptr -4
arg_8       = dword ptr 10h
arg_C       = dword ptr 14h
arg_14     = dword ptr 1Ch
arg_18     = dword ptr 20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

    push    ebp
    mov     ebp, esp
    sub     esp, 7Ch
    mov     eax, [ebp+arg_14] ; [EBP+1Ch]=1
    mov     ecx, TlsIndex ; [69AEB08h]=0
    mov     edx, large fs:2Ch
    mov     edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
    cmp     eax, 2           ; EAX=1
    mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
    jz      loc_2CE1288
    mov     ecx, [eax]       ; [EAX]=0..5
    mov     [ebp+var_4], edi ; EDI=0xc98c938
```

<sup>37</sup>[yurichev.com](http://yurichev.com)

```

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
    ; _kqvrow_+1A9
    cmp    ecx, 5           ; ECX=0..5
    ja     loc_56C11C7
    mov    edi, [ebp+arg_18] ; [EBP+20h]=0
    mov    [ebp+var_14], edx ; EDX=0xc98c938
    mov    [ebp+var_8], ebx ; EBX=0
    mov    ebx, eax         ; EAX=0xcdfe554
    mov    [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
    mov    edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a
    jmp    edx               ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
    push   offset aXKqvvsnBuffer ; "x$kvvsn buffer"
    mov    ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
    xor    edx, edx
    mov    esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
    push   edx               ; EDX=0
    push   edx               ; EDX=0
    push   50h
    push   ecx               ; ECX=0x8a172b4
    push   dword ptr [esi+10494h] ; [ESI+10494h]=0xc98cd58
    call   _kghalf            ; tracing nested maximum level (1) reached, skipping this CALL
    mov    esi, ds:_imp_vsnum ; [59771A8h]=0x61bc49e0
    mov    [ebp+Dest], eax ; EAX=0xce2ffb0
    mov    [ebx+8], eax ; EAX=0xce2ffb0
    mov    [ebx+4], eax ; EAX=0xce2ffb0
    mov    edi, [esi] ; [ESI]=0xb200100
    mov    esi, ds:_imp_vsnstr ; [597D6D4h]=0x65852148, "- Production"
    push   esi               ; ESI=0x65852148, "- Production"
    mov    ebx, edi           ; EDI=0xb200100
    shr    ebx, 18h           ; EBX=0xb200100
    mov    ecx, edi           ; EDI=0xb200100
    shr    ecx, 14h           ; ECX=0xb200100
    and   ecx, 0Fh           ; ECX=0xb2
    mov    edx, edi           ; EDI=0xb200100
    shr    edx, 0Ch           ; EDX=0xb200100
    movzx  edx, dl           ; DL=0
    mov    eax, edi           ; EDI=0xb200100
    shr    eax, 8             ; EAX=0xb200100
    and   eax, 0Fh           ; EAX=0xb2001
    and   edi, 0FFh          ; EDI=0xb200100
    push   edi               ; EDI=0
    mov    edi, [ebp+arg_18] ; [EBP+20h]=0
    push   eax               ; EAX=1
    mov    eax, ds:_imp_vsnban ; [597D6D8h]=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
    push   edx               ; EDX=0
    push   ecx               ; ECX=2
    push   ebx               ; EBX=0xb
    mov    ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
    push   eax               ; EAX=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
    mov    eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
    push   eax               ; EAX=0xce2ffb0
    call   ds:_imp_sprintf ; op1=MSVCR80.dll!sprintf tracing nested maximum level (1) reached, skipping this CALL
    add    esp, 38h
    mov    dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
    ; _kqvrow_+128 ...
    test   edi, edi           ; EDI=0
    jnz    __VInfreq_kqvrow

```

## 8.11. ORACLE RDBMS

```

mov    esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
mov    edi, [ebp+var_4] ; [EBP-4]=0xc98c938
mov    eax, ebx          ; EBX=0xcdfe554
mov    ebx, [ebp+var_8] ; [EBP-8]=0
lea    eax, [eax+4]      ; [EAX+4]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
        , "Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production", "PL/SQL "
        , "Release 11.2.0.1.0 - Production", "TNS for 32-bit Windows: Version 11.2.0.1.0 - "
        , "Production"

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
    mov    esp, ebp
    pop    ebp
    retn           ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
    mov    edx, [ebx+8]      ; [EBX+8]=0xce2ffb0, "Oracle Database 11g Enterprise Edition "
        , "Release 11.2.0.1.0 - Production"
    mov    dword ptr [ebx], 2
    mov    [ebx+4], edx      ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition "
        , "Release 11.2.0.1.0 - Production"
    push   edx              ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition "
        , "Release 11.2.0.1.0 - Production"
    call   _kkxvsn          ; tracing nested maximum level (1) reached, skipping this
        , CALL
    pop    ecx
    mov    edx, [ebx+4]      ; [EBX+4]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
    movzx  ecx, byte ptr [edx]; [EDX]=0x50
    test  ecx, ecx          ; ECX=0x50
    jnz   short loc_2CE1192
    mov    edx, [ebp+var_14]
    mov    esi, [ebp+var_C]
    mov    eax, ebx
    mov    ebx, [ebp+var_8]
    mov    ecx, [eax]
    jmp   loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
    push  0
    push  50h
    mov   edx, [ebx+8]      ; [EBX+8]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
    mov   [ebx+4], edx      ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
    push  edx              ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
    call  _lmxver          ; tracing nested maximum level (1) reached, skipping this
        , CALL
    add   esp, 0Ch
    mov   dword ptr [ebx], 3
    jmp   short loc_2CE1192

loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
    mov   edx, [ebx+8]      ; [EBX+8]=0xce2ffb0
    mov   [ebp+var_18], 50h
    mov   [ebx+4], edx      ; EDX=0xce2ffb0
    push  0
    call  _npinli          ; tracing nested maximum level (1) reached, skipping this
        , CALL
    pop   ecx
    test  eax, eax          ; EAX=0
    jnz   loc_56C11DA
    mov   ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
    lea   edx, [ebp+var_18] ; [EBP-18h]=0x50
    push  edx              ; EDX=0xd76c93c
    push  dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
    push  dword ptr [ecx+13278h]; [ECX+13278h]=0xacce190
    call  _nrtnsvrs         ; tracing nested maximum level (1) reached, skipping this
        , CALL
    add   esp, 0Ch

loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
    mov   dword ptr [ebx], 4
    jmp   loc_2CE1192

```

## 8.11. ORACLE RDBMS

```

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
    lea      edx, [ebp+var_7C] ; [EBP-7Ch]=1
    push    edx                ; EDX=0xd76c8d8
    push    0
    mov     esi, [ebx+8]       ; [EBX+8]=0xce2ffb0, "TNS for 32-bit Windows: Version ↵
    ↳ 11.2.0.1.0 - Production"
    mov     [ebx+4], esi       ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 ↵
    ↳ - Production"
    mov     ecx, 50h
    mov     [ebp+var_18], ecx ; ECX=0x50
    push    ecx                ; ECX=0x50
    push    esi                ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 ↵
    ↳ - Production"
    call   _lxvers            ; tracing nested maximum level (1) reached, skipping this ↵
    ↳ CALL
    add    esp, 10h
    mov    edx, [ebp+var_18] ; [EBP-18h]=0x50
    mov    dword ptr [ebx], 5
    test   edx, edx          ; EDX=0x50
    jnz   loc_2CE1192
    mov    edx, [ebp+var_14]
    mov    esi, [ebp+var_C]
    mov    eax, ebx
    mov    ebx, [ebp+var_8]
    mov    ecx, 5
    jmp   loc_2CE10F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
    mov    edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
    mov    esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
    mov    edi, [ebp+var_4] ; [EBP-4]=0xc98c938
    mov    eax, ebx          ; EBX=0xcdfe554
    mov    ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow_+1F
    mov    eax, [eax+8]       ; [EAX+8]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
    test   eax, eax          ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
    jz    short loc_2CE12A7
    push   offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
    push   eax                ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
    mov    eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
    push   eax                ; EAX=0x8a172b4
    push   dword ptr [edx+10494h] ; [EDX+10494h]=0xc98cd58
    call   _kgfrf              ; tracing nested maximum level (1) reached, skipping this ↵
    ↳ CALL
    add    esp, 10h

loc_2CE12A7: ; CODE XREF: _kqvrow_+1C1
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    retn
_kqvrow_ endp
; EAX=0

```

Now it is easy to see that the row number is passed from outside. The function returns the string, constructing it as follows:

String 1	Using <code>vsnstr</code> , <code>vsnnum</code> , <code>vsnban</code> global variables. Calls <code>sprintf()</code> .
String 2	Calls <code>kkxvsn()</code> .
String 3	Calls <code>lmxver()</code> .
String 4	Calls <code>npinli()</code> , <code>nrtnsvrs()</code> .
String 5	Calls <code>lxvers()</code> .

That's how the corresponding functions are called for determining each module's version.

## 8.11.2 X\$KSMLRU table in Oracle RDBMS

There is a mention of a special table in the *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* note:

There is a fixed table called X\$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.

If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.

One unusual thing about the X\$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should be carefully kept since it cannot be retrieved back after the query is issued.

However, as it can be easily checked, the contents of this table are cleared each time it's queried. Are we able to find why? Let's get back to tables we already know: `kqftab` and `kqftap` which were generated with oracle tables<sup>38</sup>'s help, that has all information about the X\$-tables. We can see here that the `ksmlrs()` function is called to prepare this table's elements:

Listing 8.17: Result of oracle tables

```

kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xfffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRP00L] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL

```

Indeed, with `tracer`'s help it is easy to see that this function is called each time we query the X\$KSMLRU table.

Here we see a references to the `ksmsplu_sp()` and `ksmsplu_jp()` functions, each of them calls the `ksmsplu()` at the end. At the end of the `ksmsplu()` function we see a call to `memset()`:

Listing 8.18: ksm.o

```

...
.text:00434C50 loc_434C50: ; DATA XREF: .rdata:off_5E50EA8
.text:00434C50     mov    edx, [ebp-4]
.text:00434C53     mov    [eax], esi
.text:00434C55     mov    esi, [edi]
.text:00434C57     mov    [eax+4], esi
.text:00434C5A     mov    [edi], eax
.text:00434C5C     add    edx, 1
.text:00434C5F     mov    [ebp-4], edx

```

<sup>38</sup>[yurichev.com](http://yurichev.com)

## 8.11. ORACLE RDBMS

```
.text:00434C62      jnz    loc_434B7D
.text:00434C68      mov    ecx, [ebp+14h]
.text:00434C6B      mov    ebx, [ebp-10h]
.text:00434C6E      mov    esi, [ebp-0Ch]
.text:00434C71      mov    edi, [ebp-8]
.text:00434C74      lea    eax, [ecx+8Ch]
.text:00434C7A      push   370h          ; Size
.text:00434C7F      push   0             ; Val
.text:00434C81      push   eax           ; Dst
.text:00434C82      call   __intel_fast_memset
.text:00434C87      add    esp, 0Ch
.text:00434C8A      mov    esp, ebp
.text:00434C8C      pop    ebp
.text:00434C8D      retn
.text:00434C8D _ksmsplu endp
```

Constructions like `memset (block, 0, size)` are often used just to zero memory block. What if we take a risk, block the `memset()` call and see what happens?

Let's run `tracer` with the following options: set breakpoint at `0x434C7A` (the point where the arguments to `memset()` are to be passed), so that `tracer` will set program counter `EIP` to the point where the arguments passed to `memset()` are to be cleared (at `0x434C8A`) It can be said that we just simulate an unconditional jump from address `0x434C7A` to `0x434C8A`.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A, set(eip,0x00434C8A)
```

(Important: all these addresses are valid only for the win32 version of Oracle RDBMS 11.2)

Indeed, now we can query the `X$KSMLRU` table as many times as we want and it is not being cleared anymore!

~~Do not try this at home ("MythBusters")~~ Do not try this on your production servers.

It is probably not a very useful or desired system behavior, but as an experiment for locating a piece of code that we need, it perfectly suits our needs!

### 8.11.3 V\$TIMER table in Oracle RDBMS

`V$TIMER` is another *fixed view* that reflects a rapidly changing value:

V\$TIMER displays the elapsed time in hundredths of a second. Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).

(From Oracle RDBMS documentation <sup>39</sup>)

It is interesting that the periods are different for Oracle for win32 and for Linux. Will we be able to find the function that generates this value?

As we can see, this information is finally taken from the `X$KSUTM` table.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME
-----
VIEW_DEFINITION
-----

V$TIMER
select HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME
```

<sup>39</sup><http://go.yurichev.com/17088>

## 8.11. ORACLE RDBMS

```
-----  
VIEW_DEFINITION  
-----  
  
GV$TIMER  
select inst_id,ksutmtim from x$ksutm
```

Now we are stuck in a small problem, there are no references to value generating function(s) in the tables `kqftab` / `kqftap`:

Listing 8.19: Result of oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xfffffc09b 0x3  
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0  
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0  
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0  
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e  
kqftap_element.fn1=NULL  
kqftap_element.fn2=NULL
```

When we try to find the string `KSUTMTIM`, we see it in this function:

```
kqfd_DRN_ksutm_c proc near ; DATA XREF: .rodata:0805B4E8  
  
arg_0 = dword ptr 8  
arg_8 = dword ptr 10h  
arg_C = dword ptr 14h  
  
push    ebp  
mov     ebp, esp  
push    [ebp+arg_C]  
push    offset ksugtm  
push    offset _2_STRING_1263_0 ; "KSUTMTIM"  
push    [ebp+arg_8]  
push    [ebp+arg_0]  
call    kqfd_cfui_drain  
add    esp, 14h  
mov    esp, ebp  
pop    ebp  
retn  
kqfd_DRN_ksutm_c endp
```

The `kqfd_DRN_ksutm_c()` function is mentioned in the

`kqfd_tab_registry_0` table:

```
dd offset _2_STRING_62_0 ; "X$KSUTM"  
dd offset kqfd_OPN_ksutm_c  
dd offset kqfd_tabl_fetch  
dd 0  
dd 0  
dd offset kqfd_DRN_ksutm_c
```

There is a function `ksugtm()` referenced here. Let's see what's in it (Linux x86):

Listing 8.20: ksu.o

```
ksugtm proc near  
  
var_1C = byte ptr -1Ch  
arg_4 = dword ptr 0Ch  
  
push    ebp  
mov     ebp, esp  
sub    esp, 1Ch  
lea     eax, [ebp+var_1C]  
push    eax  
call    slgcs  
pop    ecx  
mov    edx, [ebp+arg_4]
```

## 8.11. ORACLE RDBMS

```
mov    [edx], eax
mov    eax, 4
mov    esp, ebp
pop    ebp
retn
ksugtm endp
```

The code in the win32 version is almost the same.

Is this the function we are looking for? Let's see:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Let's try again:

```
SQL> select * from V$TIMER;
```

```
-----  
HSECS  
-----  
27294929
```

```
SQL> select * from V$TIMER;
```

```
-----  
HSECS  
-----  
27295006
```

```
SQL> select * from V$TIMER;
```

```
-----  
HSECS  
-----  
27295167
```

Listing 8.21: [tracer](#) output

```
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!_VInfreq_qerfxFetch
  ↴ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                      "8.          "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01                  ".|..        "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!_VInfreq_qerfxFetch
  ↴ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                      "8.          "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01                  ".}..        "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!_VInfreq_qerfxFetch
  ↴ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                      "8.          "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01                  ".}..        "
```

Indeed—the value is the same we see in SQL\*Plus and it is returned via the second argument.

Let's see what is in `slgcs()` (Linux x86):

```
slgcs  proc near

var_4  = dword ptr -4
arg_0  = dword ptr  8

    push    ebp
    mov     ebp, esp
    push    esi
    mov     [ebp+var_4], ebx
```

## 8.11. ORACLE RDBMS

```
mov    eax, [ebp+arg_0]
call   $+5
pop    ebx
nop
      ; PIC mode
mov    ebx, offset _GLOBAL_OFFSET_TABLE_
mov    dword ptr [eax], 0
call   sltrgatime64    ; PIC mode
push   0
push   0Ah
push   edx
push   eax
call   __udivdi3     ; PIC mode
mov    ebx, [ebp+var_4]
add    esp, 10h
mov    esp, ebp
pop    ebp
retn
slgcs endp
```

(it is just a call to `sltrgatime64()`

and division of its result by 10 ([3.9 on page 496](#))

And win32-version:

```
_slgcs proc near ; CODE XREF: _dbgefgHtElResetCount+15
           ; _dbgcrunActions+1528
    db    66h
    nop
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+8]
    mov   dword ptr [eax], 0
    call  ds:_imp__GetTickCount@0 ; GetTickCount()
    mov   edx, eax
    mov   eax, 0CCCCCCCCDh
    mul   edx
    shr   edx, 3
    mov   eax, edx
    mov   esp, ebp
    pop   ebp
    retn
_slgcs endp
```

It is just the result of `GetTickCount()` <sup>40</sup> divided by 10 ([3.9 on page 496](#)).

Voilà! That's why the win32 version and the Linux x86 version show different results, because they are generated by different OS functions.

*Drain* apparently implies *connecting* a specific table column to a specific function.

We will add support of the table `kqfd_tabl_registry_0` to oracle tables<sup>41</sup>, now we can see how the table column's variables are *connected* to a specific functions:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksusg_c]
```

*OPN*, apparently stands for, *open*, and *DRN*, apparently, for *drain*.

<sup>40</sup>[MSDN](#)

<sup>41</sup>[yurichev.com](#)

## 8.12 Handwritten assembly code

### 8.12.1 EICAR test file

This .COM-file is intended for testing antivirus software, it is possible to run in MS-DOS and it prints this string: "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!"<sup>42</sup>.

Its important property is that it's consists entirely of printable ASCII-symbols, which, in turn, makes it possible to create it in any text editor:

```
X50!P%@AP[4\PZX54(P^)7CC]{$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Let's decompile it:

```
; initial conditions: SP=0FFFFh, SS:[SP]=0
0100 58          pop     ax
; AX=0, SP=0
0101 35 4F 21    xor     ax, 214Fh
; AX = 214Fh and SP = 0
0104 50          push    ax
; AX = 214Fh, SP = FFFFh and SS:[FFFE] = 214Fh
0105 25 40 41    and     ax, 4140h
; AX = 140h, SP = FFFFh and SS:[FFFE] = 214Fh
0108 50          push    ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h and SS:[FFFE] = 214Fh
0109 5B          pop     bx
; AX = 140h, BX = 140h, SP = FFFFh and SS:[FFFE] = 214Fh
010A 34 5C          xor    al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFFh and SS:[FFFE] = 214Fh
010C 50          push    ax
010D 5A          pop     dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFFh and SS:[FFFE] = 214Fh
010E 58          pop     ax
; AX = 214Fh, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 28    xor     ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0
0112 50          push    ax
0113 5E          pop     si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37          sub    [bx], si
0116 43          inc     bx
0117 43          inc     bx
0118 29 37          sub    [bx], si
011A 7D 24          jge    short near ptr word_10140
011C 45 49 43 ... db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!'
0140 48 2B word_10140 dw 2B48h ; CD 21 (INT 21) will be here
0142 48 2A          dw 2A48h ; CD 20 (INT 20) will be here
0144 0D          db 0Dh
0145 0A          db 0Ah
```

We will add comments about the registers and stack after each instruction.

Essentially, all these instructions are here only to execute this code:

```
B4 09      MOV AH, 9
BA 1C 01    MOV DX, 11Ch
CD 21      INT 21h
CD 20      INT 20h
```

INT 21h with 9th function (passed in AH) just prints a string, the address of which is passed in DS:DX. By the way, the string has to be terminated with the '\$' sign. Apparently, it's inherited from CP/M and this function was left in DOS for compatibility. INT 20h exits to DOS.

But as we can see, these instruction's opcodes are not strictly printable. So the main part of EICAR file is:

- preparing the register (AH and DX) values that we need;
- preparing INT 21 and INT 20 opcodes in memory;

<sup>42</sup>wikipedia

## 8.13. DEMOS

- executing INT 21 and INT 20.

By the way, this technique is widely used in shellcode construction, when one have to pass x86 code in string form.

Here is also a list of all x86 instructions which have printable opcodes: [.1.6 on page 1020](#).

## 8.13 Demos

Demos (or demomaking) were an excellent exercise in mathematics, computer graphics programming and very tight x86 hand coding.

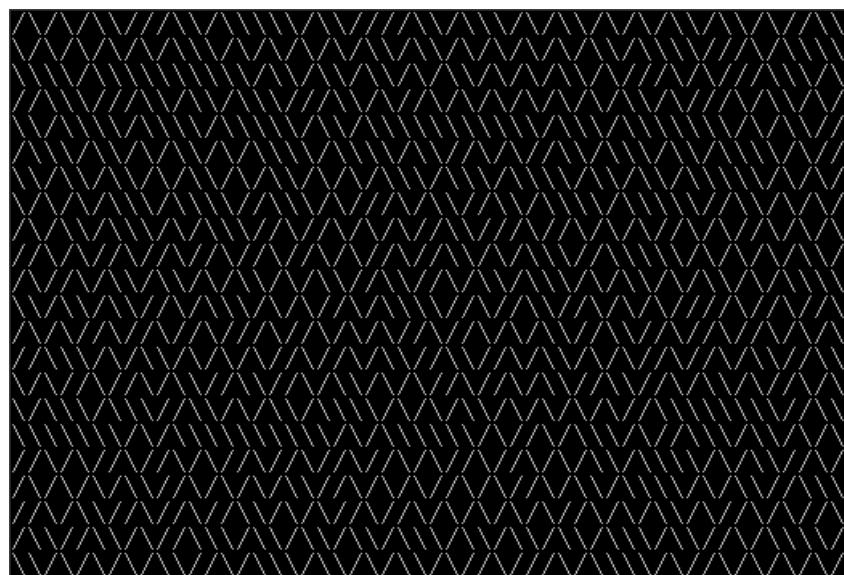
### 8.13.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10

All examples here are MS-DOS .COM files.

In [Nick Montfort et al, *10 PRINT CHR\$(205.5+RND(1)); : GOTO 10*, (The MIT Press:2012)] <sup>43</sup>

we can read about one of the most simple possible random maze generators.

It just prints a slash or backslash characters randomly and endlessly, resulting in something like this:



There are a few known implementations for 16-bit x86.

#### Trixter's 42 byte version

The listing was taken from his website<sup>44</sup>, but the comments are mine.

```
00000000: B001      mov     al,1      ; set 40x25 video mode
00000002: CD10      int     010
00000004: 30FF      xor     bh,bh    ; set video page for int 10h call
00000006: B9D007    mov     cx,007D0   ; 2000 characters to output
00000009: 31C0      xor     ax,ax
0000000B: 9C        pushf   ; push flags
; get random value from timer chip
0000000C: FA        cli     ; disable interrupts
0000000D: E643      out    043,al   ; write 0 to port 43h
; read 16-bit value from port 40h
0000000F: E440      in     al,040
00000011: 88C4      mov     ah,al
00000013: E440      in     al,040
00000015: 9D        popf   ; enable interrupts by restoring IF flag
```

<sup>43</sup>Also available as <http://go.yurichev.com/17286>

<sup>44</sup><http://go.yurichev.com/17305>

### 8.13. DEMOS

```
00000016: 86C4      xchg      ah,al
; here we have 16-bit pseudorandom value
00000018: D1E8      shr       ax,1
0000001A: D1E8      shr       ax,1
; CF currently have second bit from the value
0000001C: B05C      mov       al,05C ;'\'
; if CF=1, skip the next instruction
0000001E: 7202      jc        00000022
; if CF=0, reload AL register with another character
00000020: B02F      mov       al,02F ;'/'
; output character
00000022: B40E      mov       ah,00E
00000024: CD10      int       010
00000026: E2E1      loop     00000009 ; loop 2000 times
00000028: CD20      int       020      ; exit to DOS
```

The pseudo-random value here is in fact the time that has passed from the system's boot, taken from the 8253 time chip, the value increases by one 18.2 times per second.

By writing zero to port `43h`, we send the command "select counter 0", "counter latch", "binary counter" (not a **BCD** value).

The interrupts are enabled back with the `POPF` instruction, which restores the `IF` flag as well.

It is not possible to use the `IN` instruction with registers other than `AL`, hence the shuffling.

#### My attempt to reduce Trixter's version: 27 bytes

We can say that since we use the timer not to get a precise time value, but a pseudo-random one, we do not need to spend time (and code) to disable the interrupts.

Another thing we can say is that we need only one bit from the low 8-bit part, so let's read only it.

We can reduced the code slightly and we've got 27 bytes:

```
00000000: B9D007  mov      cx,007D0 ; limit output to 2000 characters
00000003: 31C0    xor      ax,ax   ; command to timer chip
00000005: E643    out     043,al
00000007: E440    in      al,040  ; read 8-bit of timer
00000009: D1E8    shr     ax,1    ; get second bit to CF flag
0000000B: D1E8    shr     ax,1
0000000D: B05C    mov     al,05C  ; prepare '\'
0000000F: 7202    jc      00000013
00000011: B02F    mov     al,02F  ; prepare '/'
; output character to screen
00000013: B40E    mov     ah,00E
00000015: CD10    int     010
00000017: E2EA    loop    00000003
; exit to DOS
00000019: CD20    int     020
```

#### Taking random memory garbage as a source of randomness

Since it is MS-DOS, there is no memory protection at all, we can read from whatever address we want. Even more than that: a simple `LDSB` instruction reads a byte from the `DS:SI` address, but it's not a problem if the registers' values are not set up, let it read 1) random bytes; 2) from a random place in memory!

It is suggested in Trixter's webpage<sup>45</sup> to use `LDSB` without any setup.

It is also suggested that the `SCASB`

instruction can be used instead, because it sets a flag according to the byte it reads.

Another idea to minimize the code is to use the `INT 29h` DOS syscall, which just prints the character stored in the `AL` register.

<sup>45</sup><http://go.yurichev.com/17305>

## 8.13. DEMOS

That is what Peter Ferrie and Andrey "herm1t" Baranovich did (11 and 10 bytes) <sup>46</sup>:

Listing 8.22: Andrey "herm1t" Baranovich: 11 bytes

```
00000000: B05C      mov     al,05C    ; '\'  
; read AL byte from random place of memory  
00000002: AE        scasb  
; PF = parity (AL - random_memory_byte) = parity (5Ch - random_memory_byte)  
00000003: 7A02      jp      00000007  
00000005: B02F      mov     al,02F    ; '/'  
00000007: CD29      int     029      ; output AL to screen  
00000009: EBF5      jmp     00000000 ; loop endlessly
```

SCASB also uses the value in the AL register, it subtract a random memory byte's value from the 5Ch value in AL. JP is a rare instruction, here it used for checking the parity flag (PF), which is generated by the formulae in the listing. As a consequence, the output character is determined not by some bit in a random memory byte, but by a sum of bits, this (hopefully) makes the result more distributed.

It is possible to make this even shorter by using the undocumented x86 instruction SALC (AKA SETALC) ("Set AL CF"). It was introduced in the NEC V20 CPU and sets AL to 0xFF if CF is 1 or to 0 if otherwise.

Listing 8.23: Peter Ferrie: 10 bytes

```
; AL is random at this point  
00000000: AE        scasb  
; CF is set according subtracting random memory byte from AL.  
; so it is somewhat random at this point  
00000001: D6        setalc  
; AL is set to 0xFF if CF=1 or to 0 if otherwise  
00000002: 242D      and     al,02D ; '-'  
; AL here is 0x2D or 0  
00000004: 042F      add     al,02F ; '/'  
; AL here is 0x5C or 0x2F  
00000006: CD29      int     029      ; output AL to screen  
00000008: EBF6      jmps    00000000 ; loop endlessly
```

So it is possible to get rid of conditional jumps at all. The ASCII code of backslash ("\") is 0x5C and 0x2F for slash ("/"). So we have to convert one (pseudo-random) bit in the CF flag to a value of 0x5C or 0x2F.

This is done easily: by AND-ing all bits in AL (where all 8 bits are set or cleared) with 0x2D we have just 0 or 0x2D.

By adding 0x2F to this value, we get 0x5C or 0x2F.

Then we just output it to the screen.

## Conclusion

It is also worth mentioning that the result may be different in DOSBox, Windows NT and even MS-DOS, due to different conditions: the timer chip can be emulated differently and the initial register contents may be different as well.

<sup>46</sup><http://go.yurichev.com/17087>

## 8.13.2 Mandelbrot set

You know, if you magnify the coastline, it still looks like a coastline, and a lot of other things have this property. Nature has recursive algorithms that it uses to generate clouds and Swiss cheese and things like that.

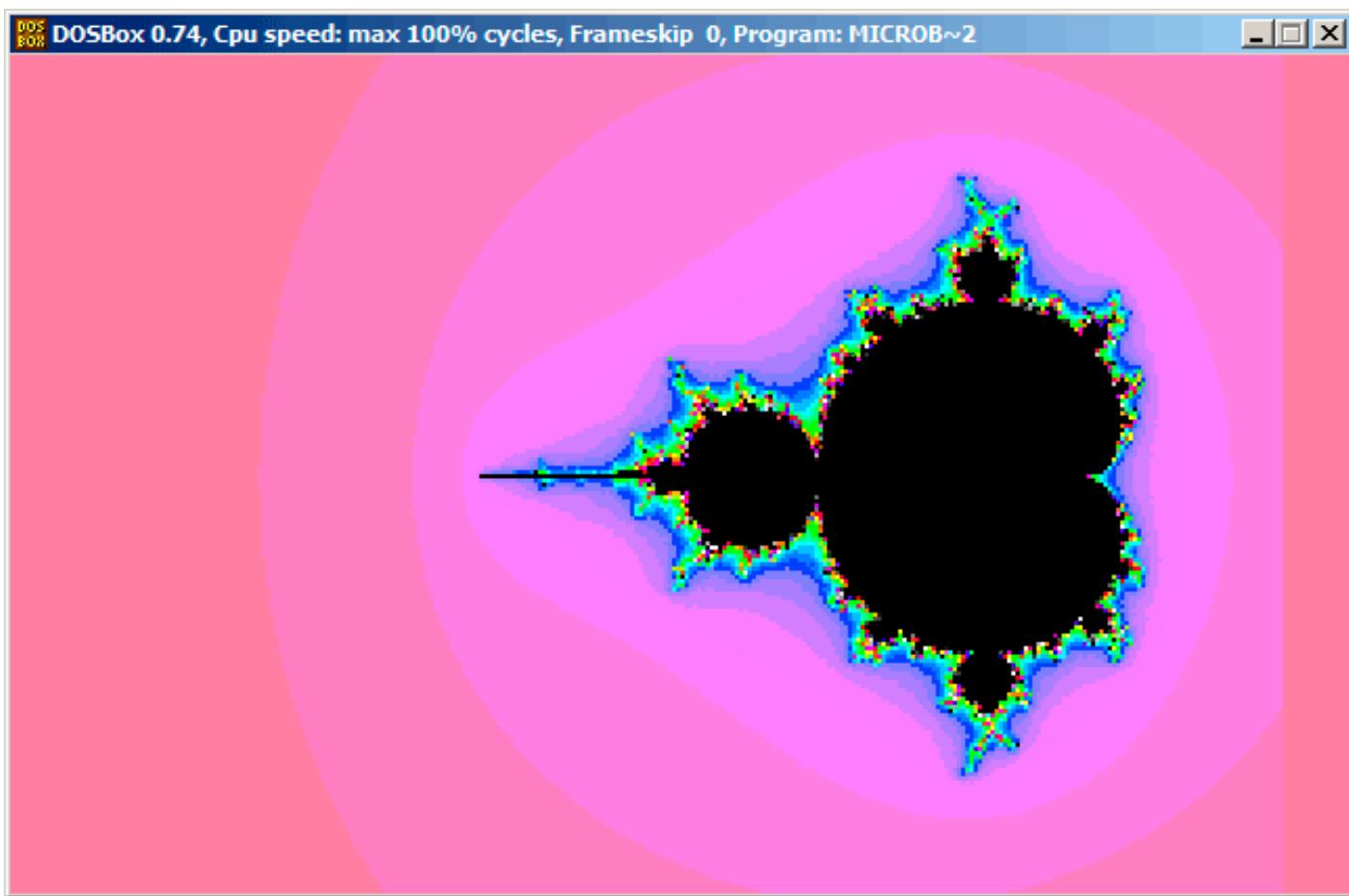
Donald Knuth, interview (1993)

Mandelbrot set is a fractal, which exhibits self-similarity.

When you increase scale, you see that this characteristic pattern repeating infinitely.

Here is a demo<sup>47</sup> written by “Sir\_Lagsalot” in 2009, that draws the Mandelbrot set, which is just a x86 program with executable file size of only 64 bytes. There are only 30 16-bit x86 instructions.

Here it is what it draws:



Let's try to understand how it works.

### Theory

#### A word about complex numbers

A complex number is a number that consists of two parts—real (Re) and imaginary (Im).

The complex plane is a two-dimensional plane where any complex number can be placed: the real part is one coordinate and the imaginary part is the other.

Some basic rules we have to keep in mind:

- Addition:  $(a + bi) + (c + di) = (a + c) + (b + d)i$

In other words:

<sup>47</sup>Download it [here](#),

### 8.13. DEMOS

$$\operatorname{Re}(sum) = \operatorname{Re}(a) + \operatorname{Re}(b)$$

$$\operatorname{Im}(sum) = \operatorname{Im}(a) + \operatorname{Im}(b)$$

- **Multiplication:**  $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$

In other words:

$$\operatorname{Re}(product) = \operatorname{Re}(a) \cdot \operatorname{Re}(c) - \operatorname{Re}(b) \cdot \operatorname{Re}(d)$$

$$\operatorname{Im}(product) = \operatorname{Im}(b) \cdot \operatorname{Im}(c) + \operatorname{Im}(a) \cdot \operatorname{Im}(d)$$

- **Square:**  $(a + bi)^2 = (a + bi)(a + bi) = (a^2 - b^2) + (2ab)i$

In other words:

$$\operatorname{Re}(square) = \operatorname{Re}(a)^2 - \operatorname{Im}(a)^2$$

$$\operatorname{Im}(square) = 2 \cdot \operatorname{Re}(a) \cdot \operatorname{Im}(a)$$

### How to draw the Mandelbrot set

The Mandelbrot set is a set of points for which the  $z_{n+1} = z_n^2 + c$  recursive sequence (where  $z$  and  $c$  are complex numbers and  $c$  is the starting value) does not approach infinity.

In plain English language:

- Enumerate all points on screen.
- Check if the specific point is in the Mandelbrot set.
- Here is how to check it:
  - Represent the point as a complex number.
  - Calculate the square of it.
  - Add the starting value of the point to it.
  - Does it go off limits? If yes, break.
  - Move the point to the new place at the coordinates we just calculated.
  - Repeat all this for some reasonable number of iterations.
- The point is still in limits? Then draw the point.
- The point has eventually gone off limits?
  - (For a black-white image) do not draw anything.
  - (For a colored image) transform the number of iterations to some color. So the color shows the speed with which point has gone off limits.

Here is Pythonesque algorithm for both complex and integer number representations:

Listing 8.24: For complex numbers

```
def check_if_is_in_set(P):
    P_start=P
    iterations=0

    while True:
        if (P>bounds):
            break
        P=P^2+P_start
        if iterations > max_iterations:
            break
        iterations++

    return iterations

# black-white
for each point on screen P:
    if check_if_is_in_set (P) < max_iterations:
        draw point
```

## 8.13. DEMOS

```
# colored
for each point on screen P:
    iterations = if check_if_is_in_set (P)
    map iterations to color
    draw color point
```

The integer version is where the operations on complex numbers are replaced with integer operations according to the rules which were explained above.

Listing 8.25: For integer numbers

```
def check_if_is_in_set(X, Y):
    X_start=X
    Y_start=Y
    iterations=0

    while True:
        if (X^2 + Y^2 > bounds):
            break
        new_X=X^2 - Y^2 + X_start
        new_Y=2*X*Y + Y_start
        if iterations > max_iterations:
            break
        iterations++

    return iterations

# black-white
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        if check_if_is_in_set (X,Y) < max_iterations:
            draw point at X, Y

# colored
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        iterations = if check_if_is_in_set (X,Y)
        map iterations to color
        draw color point at X,Y
```

Here is also a C# source which is present in the Wikipedia article<sup>48</sup>, but we'll modify it so it will print the iteration numbers instead of some symbol <sup>49</sup>:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Mnoj
{
    class Program
    {
        static void Main(string[] args)
        {
            double realCoord, imagCoord;
            double realTemp, imagTemp, realTemp2, arg;
            int iterations;
            for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
            {
                for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
                {
                    iterations = 0;
                    realTemp = realCoord;
                    imagTemp = imagCoord;
                    arg = (realCoord * realCoord) + (imagCoord * imagCoord);
                    while ((arg < 2*2) && (iterations < 40))
                    {

```

<sup>48</sup>[wikipedia](#)

<sup>49</sup>Here is also the executable file: [beginners.re](#)

## 8.13. DEMOS

```
    realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
    imagTemp = (2 * realTemp * imagTemp) - imagCoord;
    realTemp = realTemp2;
    arg = (realTemp * realTemp) + (imagTemp * imagTemp);
    iterations += 1;
}
Console.WriteLine("{0,2:D} ", iterations);
}
Console.WriteLine("\n");
}
Console.ReadKey();
}
}
```

Here is the resulting file, which is too wide to be included here:

[beginners.re](#).

The maximal number of iterations is 40, so when you see 40 in this dump, it means that this point has been wandering for 40 iterations but never got off limits.

A number  $n$  less than 40 means that point remained inside the bounds only for  $n$  iterations, then it went outside them.

### 8.13. DEMOS

There is a cool demo available at <http://go.yurichev.com/17309>, which shows visually how the point moves on the plane at each iteration for some specific point. Here are two screenshots.

First, we've clicked inside the yellow area and saw that the trajectory (green line) eventually swirls at some point inside:

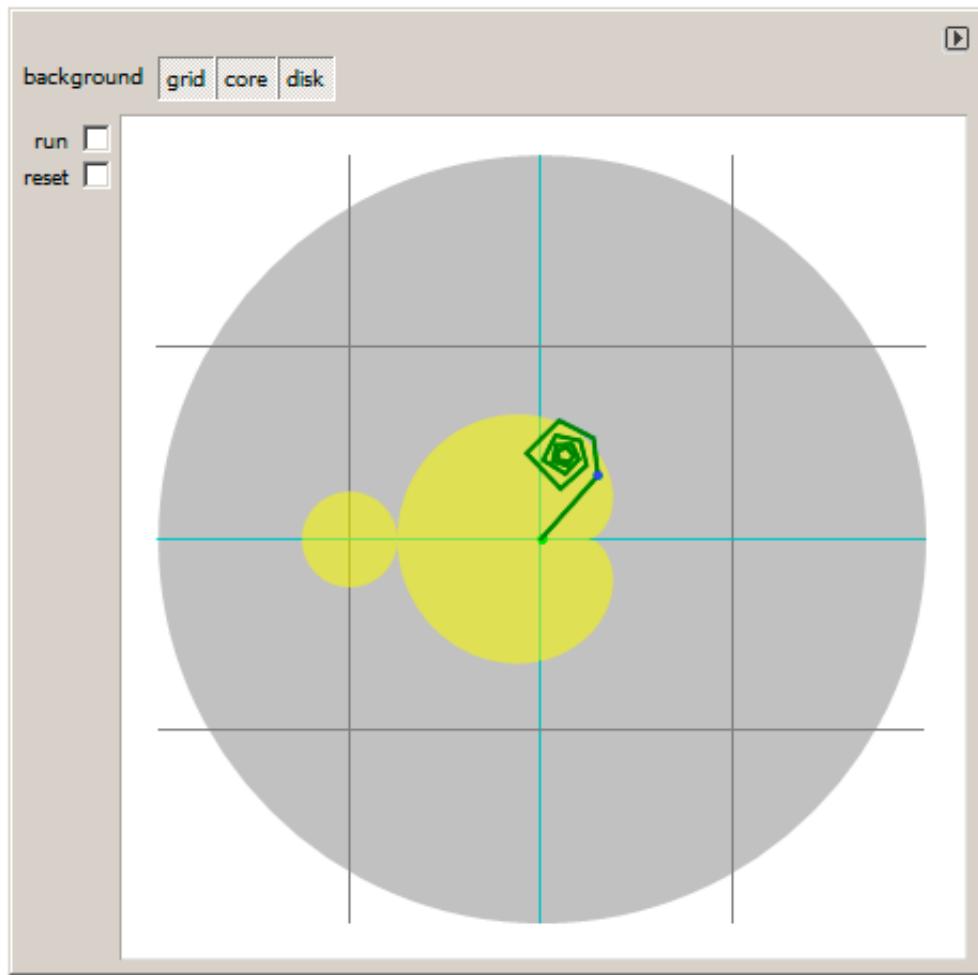


Figure 8.18: Click inside yellow area

This implies that the point we've clicked belongs to the Mandelbrot set.

### 8.13. DEMOS

Then we've clicked outside the yellow area and saw a much more chaotic point movement, which quickly went off bounds:

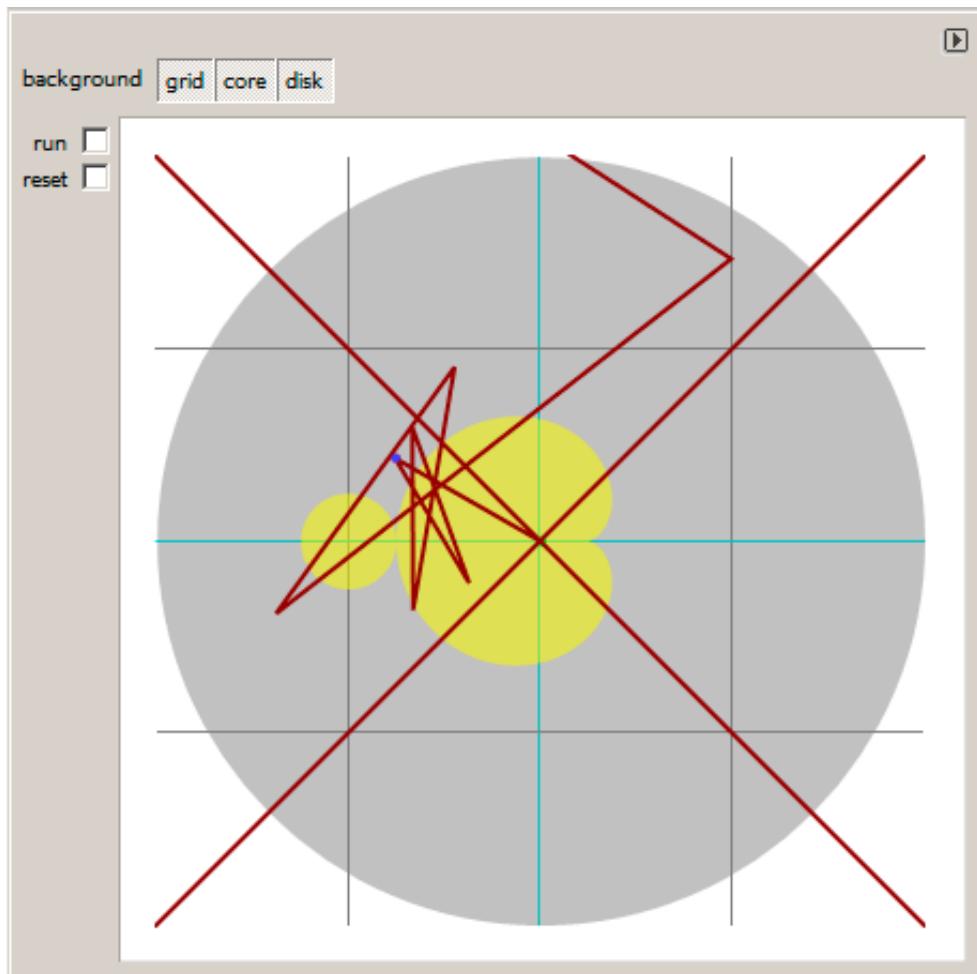


Figure 8.19: Click outside yellow area

This means the point doesn't belong to Mandelbrot set.

Another good demo is available here: <http://go.yurichev.com/17310>.

## 8.13. DEMOS

### Let's get back to the demo

The demo, although very tiny (just 64 bytes or 30 instructions), implements the common algorithm described here, but using some coding tricks.

The source code is easily downloadable, so here is it, but let's also add comments:

Listing 8.26: Commented source code

```
1 ; X is column on screen
2 ; Y is row on screen
3
4
5 ; X=0, Y=0           X=319, Y=0
6 ; +----->
7 ;
8 ;
9 ;
10 ;
11 ;
12 ;
13 ; v
14 ; X=0, Y=199       X=319, Y=199
15
16
17 ; switch to VGA 320*200*256 graphics mode
18 mov al,13h
19 int 10h
20 ; initial BX is 0
21 ; initial DI is 0xFFFFE
22 ; DS:BX (or DS:0) is pointing to Program Segment Prefix at this moment
23 ; ... first 4 bytes of which are CD 20 FF 9F
24 les ax,[bx]
25 ; ES:AX=9FFF:20CD
26
27 FillLoop:
28 ; set DX to 0. CWD works as: DX:AX = sign_extend(AX).
29 ; AX here 0x20CD (at startup) or less than 320 (when getting back after loop),
30 ; so DX will always be 0.
31 cwd
32 mov ax,di
33 ; AX is current pointer within VGA buffer
34 ; divide current pointer by 320
35 mov cx,320
36 div cx
37 ; DX (start_X) - remainder (column: 0..319); AX - result (row: 0..199)
38 sub ax,100
39 ; AX=AX-100, so AX (start_Y) now is in range -100..99
40 ; DX is in range 0..319 or 0x0000..0x013F
41 dec dh
42 ; DX now is in range 0xFF00..0x003F (-256..63)
43
44 xor bx,bx
45 xor si,si
46 ; BX (temp_X)=0; SI (temp_Y)=0
47
48 ; get maximal number of iterations
49 ; CX is still 320 here, so this is also maximal number of iteration
50 MandelLoop:
51 mov bp,si      ; BP = temp_Y
52 imul si,bx    ; SI = temp_X*temp_Y
53 add si,si     ; SI = SI*2 = (temp_X*temp_Y)*2
54 imul bx,bx    ; BX = BX^2 = temp_X^2
55 jo MandelBreak ; overflow?
56 imul bp,bp    ; BP = BP^2 = temp_Y^2
57 jo MandelBreak ; overflow?
58 add bx,bp     ; BX = BX+BP = temp_X^2 + temp_Y^2
59 jo MandelBreak ; overflow?
60 sub bx,bp     ; BX = BX-BP = temp_X^2 + temp_Y^2 - temp_Y^2 = temp_X^2
61 sub bx,bp     ; BX = BX-BP = temp_X^2 - temp_Y^2
```

### 8.13. DEMOS

```
63 ; correct scale:
64 sar bx,6      ; BX=BX/64
65 add bx,dx     ; BX=BX+start_X
66 ; now temp_X = temp_X^2 - temp_Y^2 + start_X
67 sar si,6      ; SI=SI/64
68 add si,ax     ; SI=SI+start_Y
69 ; now temp_Y = (temp_X*temp_Y)*2 + start_Y
70
71 loop MandelLoop
72
73 MandelBreak:
74 ; CX=iterations
75 xchg ax,cx
76 ; AX=iterations. store AL to VGA buffer at ES:[DI]
77 stosb
78 ; stosb also increments DI, so DI now points to the next point in VGA buffer
79 ; jump always, so this is eternal loop here
80 jmp FillLoop
```

#### Algorithm:

- Switch to 320\*200 VGA video mode, 256 colors.  $320 * 200 = 64000$  (0xFA00).

Each pixel is encoded by one byte, so the buffer size is 0xFA00 bytes. It is addressed using the ES:DI registers pair.

ES must be 0xA000 here, because this is the segment address of the VGA video buffer, but storing 0xA000 to ES requires at least 4 bytes (`PUSH 0A000h / POP ES`). You can read more about the 16-bit MS-DOS memory model here: [10.6 on page 992](#).

Assuming that BX is zero here, and the Program Segment Prefix is at the zeroth address, the 2-byte `LES AX, [BX]` instruction stores 0x20CD to AX and 0xFFFF to ES.

So the program starts to draw 16 pixels (or bytes) before the actual video buffer. But this is MS-DOS, there is no memory protection, so a write happens into the very end of conventional memory, and usually, there is nothing important. That's why you see a red strip 16 pixels wide at the right side. The whole picture is shifted left by 16 pixels. This is the price of saving 2 bytes.

- An infinite loop processes each pixel.

Probably, the most common way to enumerate all pixels on the screen is with two loops: one for the X coordinate, another for the Y coordinate. But then you'll need to multiply the coordinates to address a byte in the VGA video buffer.

The author of this demo decided to do it otherwise: enumerate all bytes in the video buffer by using one single loop instead of two, and get the coordinates of the current point using division. The resulting coordinates are: X in the range of -256..63 and Y in the range of -100..99. You can see on the screenshot that the picture is somewhat shifted to the right part of screen.

That's because the biggest heart-shaped black hole usually appears on coordinates 0,0 and these are shifted here to right. Could the author just subtract 160 from the value to get X in the range of -160..159? Yes, but the instruction `SUB DX, 160` takes 4 bytes, while `DEC DH` —2 bytes (which subtracts 0x100 (256) from DX). So the whole picture is shifted for the cost of another 2 bytes of saved space.

- Check, if the current point is inside the Mandelbrot set. The algorithm is the one that has been described here.
- The loop is organized using the `LOOP` instruction, which uses the CX register as counter.

The author could set the number of iterations to some specific number, but he didn't: 320 is already present in CX (has been set at line 35), and this is good maximal iteration number anyway. We save here some space by not the reloading CX register with another value.

- `IMUL` is used here instead of `MUL`, because we work with signed values: keep in mind that the 0,0 coordinates has to be somewhere near the center of the screen.  
It's the same with `SAR` (arithmetic shift for signed values): it's used instead of `SHR`.
- Another idea is to simplify the bounds check. We must check a coordinate pair, i.e., two variables. What the author does is to checks thrice for overflow: two squaring operations and one addition.

## 8.13. DEMOS

Indeed, we use 16-bit registers, which hold signed values in the range of -32768..32767, so if any of the coordinates is greater than 32767 during the signed multiplication, this point is definitely out of bounds: we jump to the `MandelBreak` label.

- There is also a division by 64 (SAR instruction). 64 sets scale.

Try to increase the value and you can get a closer look, or to decrease if for a more distant look.

- We are at the `MandelBreak` label, there are two ways of getting here: the loop ended with CX=0 (the point is inside the Mandelbrot set); or because an overflow has happened (CX still holds some value). Now we write the low 8-bit part of CX (CL) to the video buffer.

The default palette is rough, nevertheless, 0 is black: hence we see black holes in the places where the points are in the Mandelbrot set. The palette can be initialized at the program's start, but keep in mind, this is only a 64 bytes program!

- The program runs in an infinite loop, because an additional check where to stop, or any user interface will result in additional instructions.

Some other optimization tricks:

- The 1-byte CWD is used here for clearing DX instead of the 2-byte `XOR DX, DX` or even the 3-byte `MOV DX, 0`.
- The 1-byte `XCHG AX, CX` is used instead of the 2-byte `MOV AX,CX`. The current value of AX is not needed here anyway.
- DI (position in video buffer) is not initialized, and it is 0xFFFF at the start <sup>50</sup>.

That's OK, because the program works for all DI in the range of 0..0xFFFF eternally, and the user can't notice that it is started off the screen (the last pixel of a 320\*200 video buffer is at address 0xF9FF). So some work is actually done off the limits of the screen.

Otherwise, you'll need an additional instructions to set DI to 0 and check for the video buffer's end.

## My “fixed” version

Listing 8.27: My “fixed” version

```
1 org 100h
2 mov al,13h
3 int 10h
4
5 ; set palette
6 mov dx, 3c8h
7 mov al, 0
8 out dx, al
9 mov cx, 100h
10 inc dx
11 l00:
12 mov al, cl
13 shl ax, 2
14 out dx, al ; red
15 out dx, al ; green
16 out dx, al ; blue
17 loop l00
18
19 push 0a000h
20 pop es
21
22 xor di, di
23
24 FillLoop:
25 cwd
26 mov ax,di
27 mov cx,320
28 div cx
29 sub ax,100
30 sub dx,160
```

<sup>50</sup>More information about initial register values: <http://go.yurichev.com/17004>

## 8.13. DEMOS

```
31
32 xor bx,bx
33 xor si,si
34
35 MandelLoop:
36 mov bp,si
37 imul si,bx
38 add si,si
39 imul bx,bx
40 jo MandelBreak
41 imul bp,bp
42 jo MandelBreak
43 add bx,bp
44 jo MandelBreak
45 sub bx,bp
46 sub bx,bp
47
48 sar bx,6
49 add bx,dx
50 sar si,6
51 add si,ax
52
53 loop MandelLoop
54
55 MandelBreak:
56 xchg ax,cx
57 stosb
58 cmp di, 0FA00h
59 jb FillLoop
60
61 ; wait for keypress
62 xor ax,ax
63 int 16h
64 ; set text video mode
65 mov ax, 3
66 int 10h
67 ; exit
68 int 20h
```

The author of these lines made an attempt to fix all these oddities: now the palette is smooth grayscale, the video buffer is at the correct place (lines 19..20), the picture is drawn on center of the screen (line 30), the program eventually ends and waits for the user's keypress (lines 58..68).

But now it's much bigger: 105 bytes (or 54 instructions) <sup>51</sup>.

<sup>51</sup>You can experiment by yourself: get DosBox and NASM and compile it as: `nasm fiole.asm -fbin -o file.com`

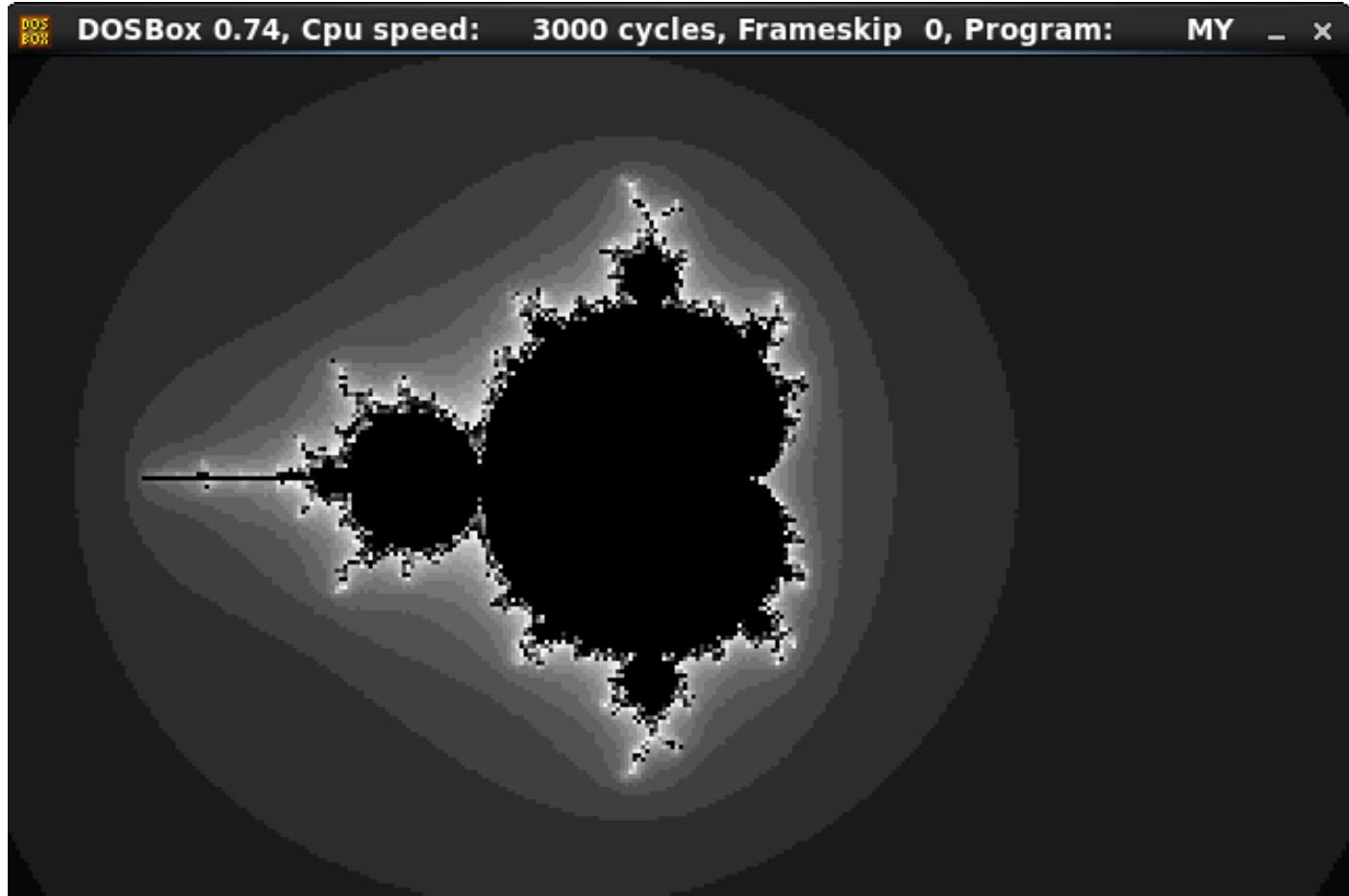


Figure 8.20: My “fixed” version

## 8.14 Other examples

An example about Z3 and manual decompilation was here. It is (temporarily) moved there: [http://yurichev.com/tmp/SAT\\_SMT\\_DRAFT.pdf](http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf).

# **Chapter 9**

## **Examples of reversing proprietary file formats**

### **9.1 Primitive XOR-encryption**

### 9.1.1 Norton Guide: simplest possible 1-byte XOR encryption

Norton Guide<sup>1</sup> was popular in the epoch of MS-DOS, it was a resident program that worked as a hypertext reference manual.

Norton Guide's databases are files with the extension .ng, the contents of which look encrypted:

Figure 9.1: Very typical look

Why did we think that it's encrypted but not compressed?

We see that the 0x1A byte (looking like “→”) occurs often, it would not be possible in a compressed file.

We also see long parts that consist only of Latin letters, and they look like strings in an unknown language.

<sup>1</sup>wikipedia

## 9.1. PRIMITIVE XOR-ENCRYPTION

Since the 0x1A byte occurs so often, we can try to decrypt the file, assuming that it's encrypted by the simplest XOR-encryption.

If we apply XOR with the 0x1A constant to each byte in Hiew, we can see familiar English text strings:

The screenshot shows the Hiew X86.NG editor interface. The title bar says "Hiew: X86.NG". The status bar at the bottom shows memory addresses from 1 to 11. The main window displays memory dump and assembly code. The assembly code is in Intel syntax. The memory dump shows many 0x1A bytes, which are replaced by readable ASCII characters in the assembly code. The assembly code includes CPU instructions like "CPU Instruction set Registers, privilege Exceptions, Addressing modes, Opcodes", FPU instructions, MMX instructions, and other assembly language constructs. The assembly code is mostly in Russian, with some English words like "privilege Exceptions" and "Addressing modes". The memory dump shows the raw binary data, with many 0x1A bytes appearing where they were replaced by characters in the assembly code.

Figure 9.2: Hiew XORing with 0x1A

XOR encryption with one single constant byte is the simplest possible encryption method, which is, nevertheless, encountered sometimes.

Now we understand why the 0x1A byte is occurring so often: because there are so many zero bytes and they were replaced by 0x1A in encrypted form.

But the constant might be different. In this case, we could try every constant in the 0..255 range and look for something familiar in the decrypted file. 256 is not so much.

More about Norton Guide's file format: <http://go.yurichev.com/17317>.

## Entropy

A very important property of such primitive encryption systems is that the information entropy of the encrypted/decrypted block is the same.

Here is my analysis in Wolfram Mathematica 10.

## 9.1. PRIMITIVE XOR-ENCRYPTION

Listing 9.1: Wolfram Mathematica 10

```
In[1]:= input = BinaryReadList["X86.NG"];  
In[2]:= Entropy[2, input] // N  
Out[2]= 5.62724  
  
In[3]:= decrypted = Map[BitXor[#, 16^^1A] &, input];  
  
In[4]:= Export["X86_decrypted.NG", decrypted, "Binary"];  
  
In[5]:= Entropy[2, decrypted] // N  
Out[5]= 5.62724  
  
In[6]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]] // N  
Out[6]= 4.42366
```

What we do here is load the file, get its entropy, decrypt it, save it and get the entropy again (the same!).

Mathematica also offers some well-known English language texts for analysis.

So we also get the entropy of Shakespeare's sonnets, and it is close to the entropy of the file we just analyzed.

The file we analyzed consists of English language sentences, which are close to the language of Shakespeare.

And the XOR-ed bitwise English language text has the same entropy.

However, this is not true when the file is XOR-ed with a pattern larger than one byte.

The file we analyzed can be downloaded here: <http://go.yurichev.com/17350>.

### One more word about base of entropy

Wolfram Mathematica calculates entropy with base of  $e$  (base of the natural logarithm), and the UNIX `ent` utility<sup>2</sup> uses base 2.

So we set base 2 explicitly in `Entropy` command, so Mathematica will give us the same results as the `ent` utility.

<sup>2</sup><http://www.fourmilab.ch/random/>

## 9.1.2 Simplest possible 4-byte XOR encryption

If a longer pattern was used for XOR-encryption, for example a 4 byte pattern, it's easy to spot as well. For example, here is the beginning of the kernel32.dll file (32-bit version from Windows Server 2008):

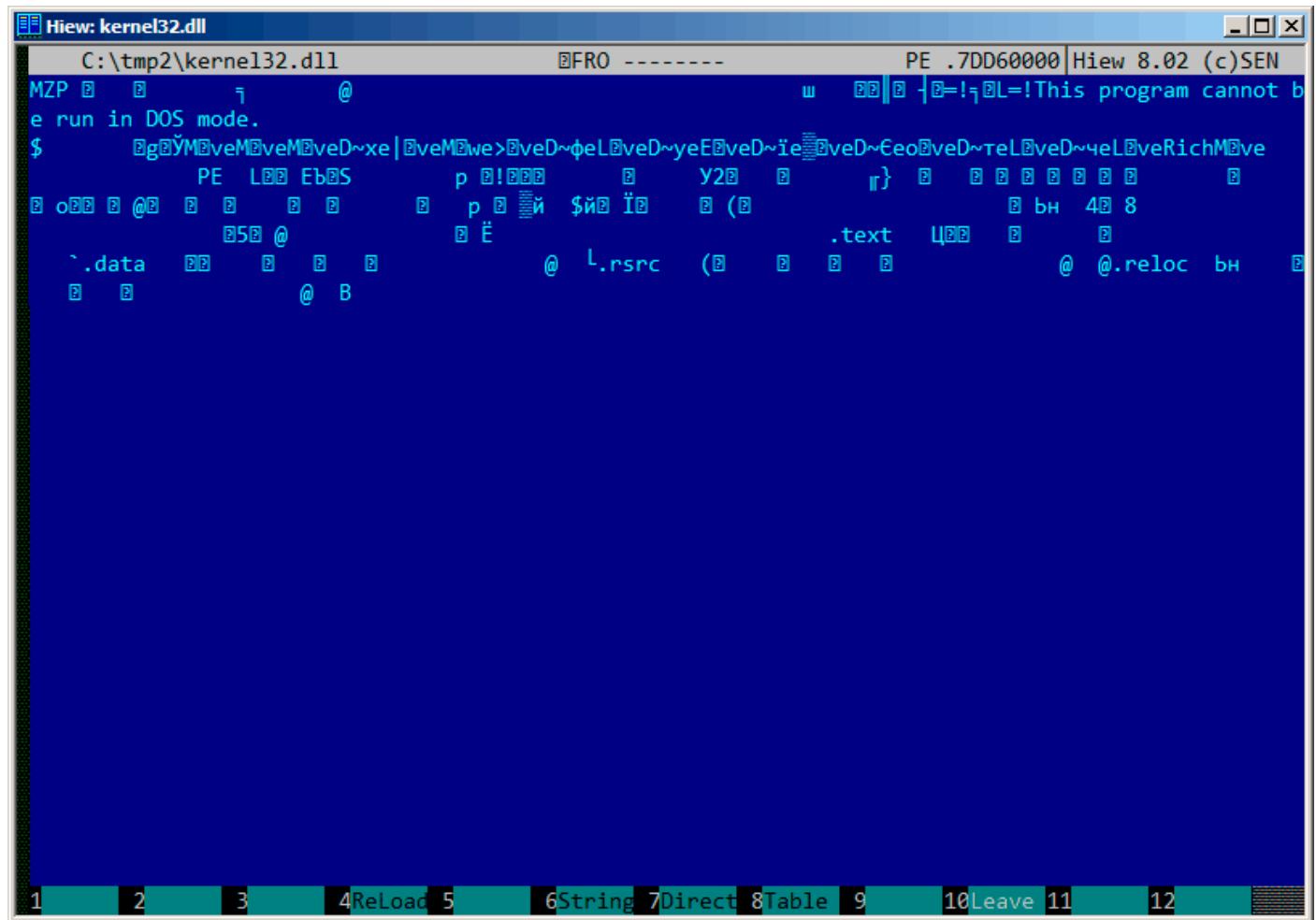


Figure 9.3: Original file

## 9.1. PRIMITIVE XOR-ENCRYPTION

Here it is “encrypted” with a 4-byte key:

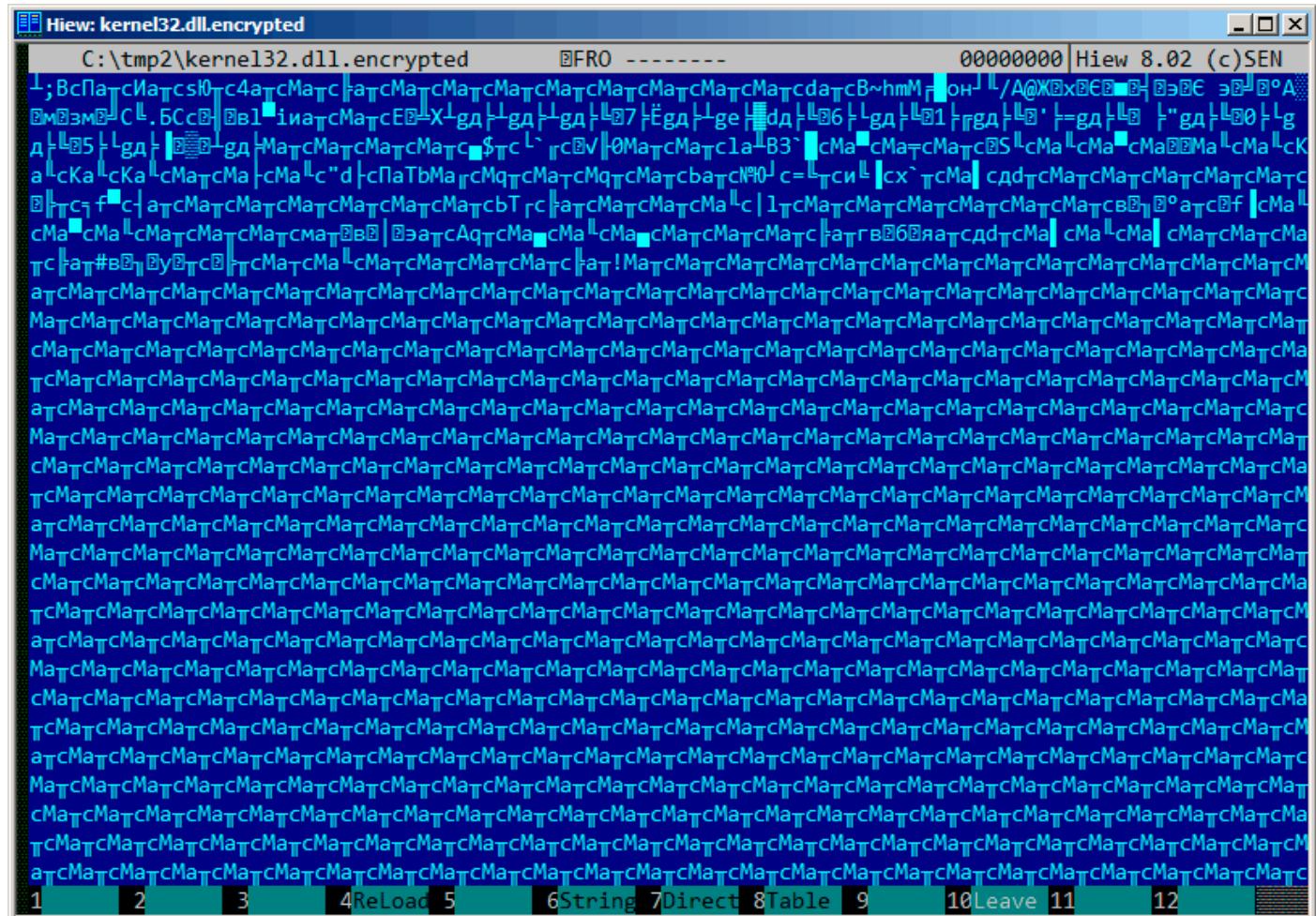


Figure 9.4: “Encrypted” file

It's very easy to spot the recurring 4 symbols.

Indeed, the header of a PE-file has a lot of long zero areas, which are the reason for the key to become visible.

## 9.1. PRIMITIVE XOR-ENCRYPTION

Here is the beginning of a PE-header in hexadecimal form:

```

Hiew: kernel32.dll
C:\tmp2\kernel32.dll          FRO -----  PE .7DD60290
.7DD600E0: 00 00 00 00-00 00 00 00-50 45 00 00-4C 01 04 00  PE L@@
.7DD600F0: 85 9A 15 53-00 00 00 00-00 00 00 00-E0 00 02 21  E@@S p @!
.7DD60100: 0B 01 09 00-00 00 0D 00-00 00 03 00-00 00 00 00  @@  @
.7DD60110: 93 32 01 00-00 00 01 00-00 00 0D 00-00 00 D6 7D  Y2@  @  } 
.7DD60120: 00 00 01 00-00 00 01 00-06 00 01 00-06 00 01 00  @  @  @  @  @  @
.7DD60130: 06 00 01 00-00 00 00 00-00 00 11 00-00 00 01 00  @  @  @  @  @
.7DD60140: AE 05 11 00-03 00 40 01-00 00 04 00-00 10 00 00  @@ @ @  @  @
.7DD60150: 00 00 10 00-00 10 00 00-00 00 00 00-10 00 00 00  @  @  @
.7DD60160: 70 FF 0B 00-B1 A9 00 00-24 A9 0C 00-F4 01 00 00  p @@@ $@@ @@@
.7DD60170: 00 00 0F 00-28 05 00 00-00 00 00 00-00 00 00 00  @ ( @
.7DD60180: 00 00 00 00-00 00 00 00-00 00 10 00-9C AD 00 00  @ @n
.7DD60190: 34 07 0D 00-38 00 00 00-00 00 00 00-00 00 00 00 4@ 8
.7DD601A0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.7DD601B0: 10 35 08 00-40 00 00 00-00 00 00 00-00 00 00 00  @@ @
.7DD601C0: 00 00 01 00-F0 0D 00 00-00 00 00 00-00 00 00 00  @ @
.7DD601D0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.7DD601E0: 2E 74 65 78-74 00 00 00-96 07 0C 00-00 00 01 00  .text  @@  @
.7DD601F0: 00 00 0D 00-00 00 01 00-00 00 00 00-00 00 00 00  @
.7DD60200: 00 00 00 00-20 00 00 60-2E 64 61 74-61 00 00 00  ` .data
.7DD60210: 0C 10 00 00-00 00 0E 00-00 00 01 00-00 00 0E 00  @@  @  @  @
.7DD60220: 00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0  @ L
.7DD60230: 2E 72 73 72-63 00 00 00-28 05 00 00-00 00 0F 00  .rsrc  (@  @
.7DD60240: 00 00 01 00-00 00 0F 00-00 00 00 00-00 00 00 00  @  @
.7DD60250: 00 00 00 00-40 00 00 40-2E 72 65 6C-6F 63 00 00  @ @.reloc
.7DD60260: 9C AD 00 00-00 00 10 00-00 00 01 00-00 00 10 00  @n  @  @  @
.7DD60270: 00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 42  @ B
.7DD60280: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 00
.7DD60290: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 00

```

1Global 2FileBlk 3CryBlk 4ReLoad 5String 6Direct 7Table 8Table 9Leave 10Leave 11

Figure 9.5: PE-header

## 9.1. PRIMITIVE XOR-ENCRYPTION

Here it is “encrypted”:

Address	Value	Content
000000E0:	8C 61 D2 63-8C 61 D2 63-DC 24 D2 63-C0 60 D6 63	Ma[cMaTc\$cTcLc
000000F0:	09 FB C7 30-8C 61 D2 63-8C 61 D2 63-6C 61 D0 42	0MaTcMatclaB
00000100:	87 60 DB 63-8C 61 DF 63-8C 61 D1 63-8C 61 D2 63	3`cMa[cMaTcMaTc
00000110:	1F 53 D3 63-8C 61 D3 63-8C 61 DF 63-8C 61 04 1E	BS[cMa[cMa[cMa@
00000120:	8C 61 D3 63-8C 61 D3 63-8A 61 D3 63-8A 61 D3 63	Ma[cMa[cKa[cKaLc
00000130:	8A 61 D3 63-8C 61 D2 63-8C 61 C3 63-8C 61 D3 63	KaLcMaTcMa[cMaLc
00000140:	22 64 C3 63-8F 61 92 62-8C 61 D6 63-8C 71 D2 63	"d[cPlaTbMa[cMqTc
00000150:	8C 61 C2 63-8C 71 D2 63-8C 61 D2 63-9C 61 D2 63	Ma[cMqTcMaTcbaTc
00000160:	FC 9E D9 63-3D C8 D2 63-A8 C8 DE 63-78 60 D2 63	N@0c=cTcИLcx`Tc
00000170:	8C 61 DD 63-A4 64 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cddTcMatcMatc
00000180:	8C 61 D2 63-8C 61 D2 63-8C 61 C2 63-10 CC D2 63	Ma[cMatcMatc@Tc
00000190:	B8 66 DF 63-B4 61 D2 63-8C 61 D2 63-8C 61 D2 63	7f[c]atcMatcMatc
000001A0:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc
000001B0:	9C 54 DA 63-CC 61 D2 63-8C 61 D2 63-8C 61 D2 63	bT[c]atcMatcMatc
000001C0:	8C 61 D3 63-7C 6C D2 63-8C 61 D2 63-8C 61 D2 63	Ma[c lTcMatcMatc
000001D0:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc
000001E0:	A2 15 B7 1B-F8 61 D2 63-1A 66 DE 63-8C 61 D3 63	в@0a[c@f[cMaLc
000001F0:	8C 61 DF 63-8C 61 D3 63-8C 61 D2 63-8C 61 D2 63	Ma[cMa[cMa[cMatc
00000200:	8C 61 D2 63-AC 61 D2 03-A2 05 B3 17-ED 61 D2 63	Ma[cma@в@#в@#в@Tc
00000210:	80 71 D2 63-8C 61 DC 63-8C 61 D3 63-8C 61 DC 63	AqTcMa[cMa[cMaLc
00000220:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-CC 61 D2 A3	Ma[cMatcMatc#атг
00000230:	A2 13 A1 11-EF 61 D2 63-A4 64 D2 63-8C 61 DD 63	в@бяаTcddTcMaLc
00000240:	8C 61 D3 63-8C 61 DD 63-8C 61 D2 63-8C 61 D2 63	Ma[cMa[cMatcMatc
00000250:	8C 61 D2 63-CC 61 D2 23-A2 13 B7 0F-E3 02 D2 63	Ma[c#ат#в@#в@#в@Tc
00000260:	10 CC D2 63-8C 61 C2 63-8C 61 D3 63-8C 61 C2 63	БTcMatcMa[cMatc
00000270:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-CC 61 D2 21	Ma[cMatcMatc#ат!
00000280:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc
00000290:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc

Figure 9.6: “Encrypted” PE-header

It's easy to spot that the key is the following 4 bytes: `8C 61 D2 63`.

With this information, it's easy to decrypt the whole file.

So it is important to keep in mind these properties of PE-files: 1) PE-header has many zero-filled areas; 2) all PE-sections are padded with zeros at a page boundary (4096 bytes), so long zero areas are usually present after each section.

Some other file formats may contain long zero areas.

It's typical for files used by scientific and engineering software.

For those who want to inspect these files on their own, they are downloadable here: <http://go.yurichev.com/17352>.

## Exercise

- <http://challenges.re/50>

## 9.1. PRIMITIVE XOR-ENCRYPTION

### 9.1.3 Simple encryption using XOR mask

I've found an old interactive fiction game while diving deep into *if-archive*<sup>3</sup>:

```
The New Castle v3.5 - Text/Adventure Game
in the style of the original Infocom (tm)
type games, Zork, Colossal Cave (Adventure),
etc. Can you solve the mystery of the
abandoned castle?
Shareware from Software Customization.
Software Customization [ASP] Version 3.5 Feb. 2000
```

It's downloadable here: [http://yurichev.com/blog/XOR\\_mask\\_1/files/newcastle.tgz](http://yurichev.com/blog/XOR_mask_1/files/newcastle.tgz).

There is a file inside (named *castle.dbf*) which is clearly encrypted, but not by a real crypto algorithm, nor it's compressed, this is something rather simpler. I wouldn't even measure entropy level ([9.2 on page 947](#)) of the file. Here is how it looks like in Midnight Commander:

```
/home/dennis/P/RE-book/decrypt_dat_file/castle.dbf
Pg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J1q13'\.\Qt>9P.(r$K8!L 78;QA-.<7]'Z.lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J-q-8V4[.0<<?.&;*..5&MB&q&K.+T?e@+0@(9,[.wE(Tu a
c.w.iubgv.^az..rn..}c~wf.z.uP.J1q.>X'GD.??N0lrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP..<003.7@V.<8*K7m=.8s\R=<+.ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP..?4#&.*.^.^:)*.$!35ly9[u!>.I&.> [%.lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.G"449Wg...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J1##Vm.M.d<</V4m>/K*). uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.N8q69J*\Zx:4%^c$f!f!..~)...,.
.E1Xz+G:s..x..mc.j a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J=f?JcI.\8(/~m&).42TLu%"LW.0.0X'J,("Y2/w_"H!.a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.G80>z. 2255$kP0m=(B.s..yqz..s.iq.nm'8)[..]K&IjH6L:.w.iubgv.^az..rn..}c~wf.z.uPg.tqf
c.w.iubgv.^az..rn..}c~wf.z.uP.G6$!1P-0.g&2,K"!fG*sY\;po
w0.36.<[\2#^Sh?M.^g0[.0[w!]!-g457?*zFN>"P.Itc~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.L0q $V..D^ 5" We9:#.-<RKu>).P7Yz0F*K80f.B):X$H/Za0&H62i!*?"S~[$;B[r/P,])c~wf.z.uP" M&%3
v.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.W05#8U:.R.'4%P0974.y$MH<%'IM4Yz#A)(@9jQD82I?Ijk$K,J2.0:7kvR;X,3_Hr:L,))c~wf.z.uP0N;02
\{.k.&%
"J?<Z,))c~wf.z.uPBC $43.,N.C<8kN,?>".12L 69.KC:XveI J,("U.,"F*_%GaN"R9[=u "vG1H/>..r.0I))c~wf.z.uP;W%f8V4.CV"
J!nJK)2c~wf.z.uP"J1q5&K&IW^:;k]"?9(K*,.u!.L60z5D/MWI+@D-6Z>,+ 5L0[2R<9.>vM;I5?CJ6nP;3c~wf.z.uP$G55/8^y...t}k.cmrf.
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cm...g
.e...qm.rz.i.bq...hw.m.j a
```

Figure 9.7: Encrypted file in Midnight Commander

The encrypted file can be downloaded [here](#).

Will it be possible to decrypt it without accessing to the program, using just this file?

There is a clearly visible pattern of repeating string. If a simple encryption by XOR mask was applied, such repeating strings is a prominent signature, because, probably, there were a long lacunas of zero bytes, which, in turn, are present in many executable files as well as in binary data files.

Here I'll dump the file's beginning using *xxd* UNIX utility:

```
...
0000030: 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d 61 .a.c.w.iubgv.^.
0000040: 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a 02 z..rn..}c~wf.z.
0000050: 75 50 02 4a 31 71 31 33 5c 27 08 5c 51 74 3e 39 uP.J1q13'\.\Qt>9
0000060: 50 2e 28 72 24 4b 38 21 4c 09 37 38 3b 51 41 2d P.(r$K8!L.78;QA-
0000070: 1c 3c 37 5d 27 5a 1c 7c 6a 10 14 68 77 08 6d 1a .<7]'Z.|j..hw.m.

0000080: 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d j.a.c.w.iubgv.^.
0000090: 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a az..rn..}c~wf.z
00000a0: 02 75 50 64 02 74 71 66 76 19 63 08 13 17 74 7d .uPd.tqfv.c...t}
```

<sup>3</sup><http://www.ifarchive.org/>

## 9.1. PRIMITIVE XOR-ENCRYPTION

```

00000b0: 6b 19 63 6d 72 66 0e 79 73 1f 09 75 71 6f 05 04 k.cmrf.y..uqo..
00000c0: 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77 08 6d ..ze.n..|j..hw.m

00000d0: 1a 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e .j.a.c.w.iubgv.~
00000e0: 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e .az..rn..}c~wf.
00000f0: 7a 02 75 50 01 4a 3b 71 2d 38 56 34 5b 13 40 3c z.uP.J;q-8V4[.@<
0000100: 3c 3f 19 26 3b 3b 2a 0e 35 26 4d 42 26 71 26 4b <?.&;*.*.5&MB&q&K
0000110: 04 2b 54 3f 65 40 2b 4f 40 28 39 10 5b 2e 77 45 .+T?e@+0@(9.[.wE

0000120: 28 54 75 09 61 0d 63 0f 77 14 69 75 62 67 76 01 (Tu.a.c.w.iubgv.
0000130: 7e 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 ~.az..rn..}c~wf
0000140: 1e 7a 02 75 50 02 4a 31 71 15 3e 58 27 47 44 17 .z.uP.J1q.>X'GD.
0000150: 3f 33 24 4e 30 6c 72 66 0e 79 73 1f 09 75 71 6f ?3$N0lrf.y..uqo
0000160: 05 04 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77 ....ze.n..|j..hw

...

```

Let's stick at visible repeating "iubgv" string. By looking at this dump, we can clearly see that the period of the string occurrence is 0x51 or 81. Probably, 81 is size of block? Size of the file is 1658961, and it can be divided evenly by 81 (and there are 20481 blocks then).

Now I'll use Mathematica to analyze, are there repeating 81-byte blocks in the file? I'll split input file by 81-byte blocks and then I'll use *Tally[]*<sup>4</sup> function which just calculates, how many times some item has been occurred in the input list. Tally's output is not sorted, so I also add *Sort[]* function to sort it by number of occurrences in descending order.

```

input = BinaryReadList["/home/dennis/.../castle.dbf"];
blocks = Partition[input, 81];
stat = Sort[Tally[blocks], #1[[2]] > #2[[2]] &]

```

And here is output:

```

{{{80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125, 107,
25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4,
127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126,
119, 102, 30, 122, 2, 117}, 1739},
{{80, 100, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
125, 99, 126, 119, 102, 30, 122, 2, 117}, 1422},
{{80, 101, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
125, 99, 126, 119, 102, 30, 122, 2, 117}, 1012},
{{80, 120, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
125, 99, 126, 119, 102, 30, 122, 2, 117}, 377},
...
{{80, 2, 74, 49, 113, 21, 62, 88, 39, 71, 68, 23, 63, 51, 36, 78, 48,
108, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4, 127, 28,
122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8, 109, 26,
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
30, 122, 2, 117}, 1},
{{80, 1, 74, 59, 113, 45, 56, 86, 52, 91, 19, 64, 60, 60, 63,

```

<sup>4</sup><https://reference.wolfram.com/language/ref/Tally.html>

## 9.1. PRIMITIVE XOR-ENCRYPTION

```

25, 38, 59, 59, 42, 14, 53, 38, 77, 66, 38, 113, 38, 75, 4, 43, 84,
63, 101, 64, 43, 79, 64, 40, 57, 16, 91, 46, 119, 69, 40, 84, 117,
9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29,
97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30,
122, 2, 117}, 1},
{{80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, 57,
80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, 28,
60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26,
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
30, 122, 2, 117}, 1}}

```

Tally's output is list pairs, each pair has 81-byte block and number of times it has been occurred in the file. We see that the most frequent block is the first, it has been occurred 1739 times. The second one has been occurred 1422 times. There are others: 1012 times, 377 times, etc. 81-byte blocks which has been occurred just once are at the end of output.

Let's try to compare these blocks? The first and the second? Is there a function in Mathematica which compares lists/arrays? Certainly is, but for educational purposes, I'll use XOR operation for comparison. Indeed: if bytes in two input arrays are identical, XOR result is 0. If they are non-equal, result will be non-zero.

Let's compare first block (occurred 1739 times) and the second (occurred 1422 times):

They are differ only in the second byte.

Let's compare second (occurred 1422 times) and third (occurred 1012 times):

They are also differ only in the second byte.

Anyway, let's try to use the most occurred block as a XOR key and try to decrypt four first 81-byte blocks in the file:

## 9.1. PRIMITIVE XOR-ENCRYPTION

(I've replaced unprintable characters by "?".)

So we see that first and third blocks are empty (or almost empty), but second and fourth has clearly visible English language words/phrases. It seems that our assumption about key is correct (at least partially). This means that the most occurred 81-block in the file can be found at places of lacunas of zero bytes or something like that.

Let's try to decrypt the whole file:

```
DecryptBlock[blk_] := BitXor[key, blk]

decrypted = Map[DecryptBlock[#] &, blocks];

BinaryWrite["/home/dennis/.../tmp", Flatten[decrypted]];

Close["/home/dennis/.../tmp"]
```

### 9.1. PRIMITIVE XOR-ENCRYPTION

RE-book/decrypt\_dat\_file/tmp

4011/1620K

eHE.WEED.OF

0%

```

TTER.FRUIT..... fHO.KNOWS.WHAT.EVIL.LURKS.IN.THE.HE
..... eHE.sHADOW.KNOWS.

..... x.HAVE.THE.HEART.OF.A.CHILD.
P.IT.IN.A.GLASS.JAR.ON.MY.DESK..... uEVERON.
..... fHERE.THE.sHADOW.LIES.

..... pLL.POSITIONING.IS.relative.AND.NOT.absolute.

..... eHIS.IS.A.KLUDGE.TO.MAKE.THIS.STUPID.THING.WORK.
..... cELAX

Y..... cLOCK.tICKS.AWAY..... uEBUGGING.pROGRAMS.IS.FUN...s
RD
K.WALLS.

..... pND.FROM.WITHIN.THE.TOMB.OF.THE.UNDEAD..VAMPIRES.BEGAN.THEIR.FEA
.EORTURED.CRIES.RANG.OUT..... tASTES.GREAT..IESS.FILLING.

..... bUDDENL
RAITHLIKE.FIGURE.APPEARS.BEFORE.YOU..SEEMING.TO..... wLOAT.IN.THE.AIR...
WFUL.VOICE.HE.SAYS...aLAS..THE.VERY....._ATURE.OF.THE.WORLD.HAS.CHANGED
ON.CANNOT.BE.FOUND...aLL.....\UST.NOW.PASS.AWAY....rAISING.HIS.DAKEN.STA
HE.FADES.INTO.....eHE.SPREADING.DARKNESS...iN.HIS.PLACE.APPEARS.A.TASTEFU
GN.....CEADING...

```

Figure 9.8: Decrypted file in Midnight Commander, 1st attempt

Looks like some kind of English phrases for some game, but something wrong. First of all, cases are inverted: phrases and some words are started with lowercase characters, while other characters are in upper case. Also, some phrases started with wrong letters. Take a look at the very first phrase: "eHE WEED OF CRIME BEARS BITTER FRUIT". What is "eHE"? Isn't "tHE" have to be here? Is it possible that our decryption key has wrong byte at this place?

Let's look again at the second block in the file, at key and at decryption result:

```

In[]:= blocks[[2]]
Out[]={80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, \
57, 80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, \
28, 60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26, \
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29, \
97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30, \
122, 2, 117}

In[]:= key
Out[]={80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, \
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, \
5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, \
8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, \
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, \
102, 30, 122, 2, 117}

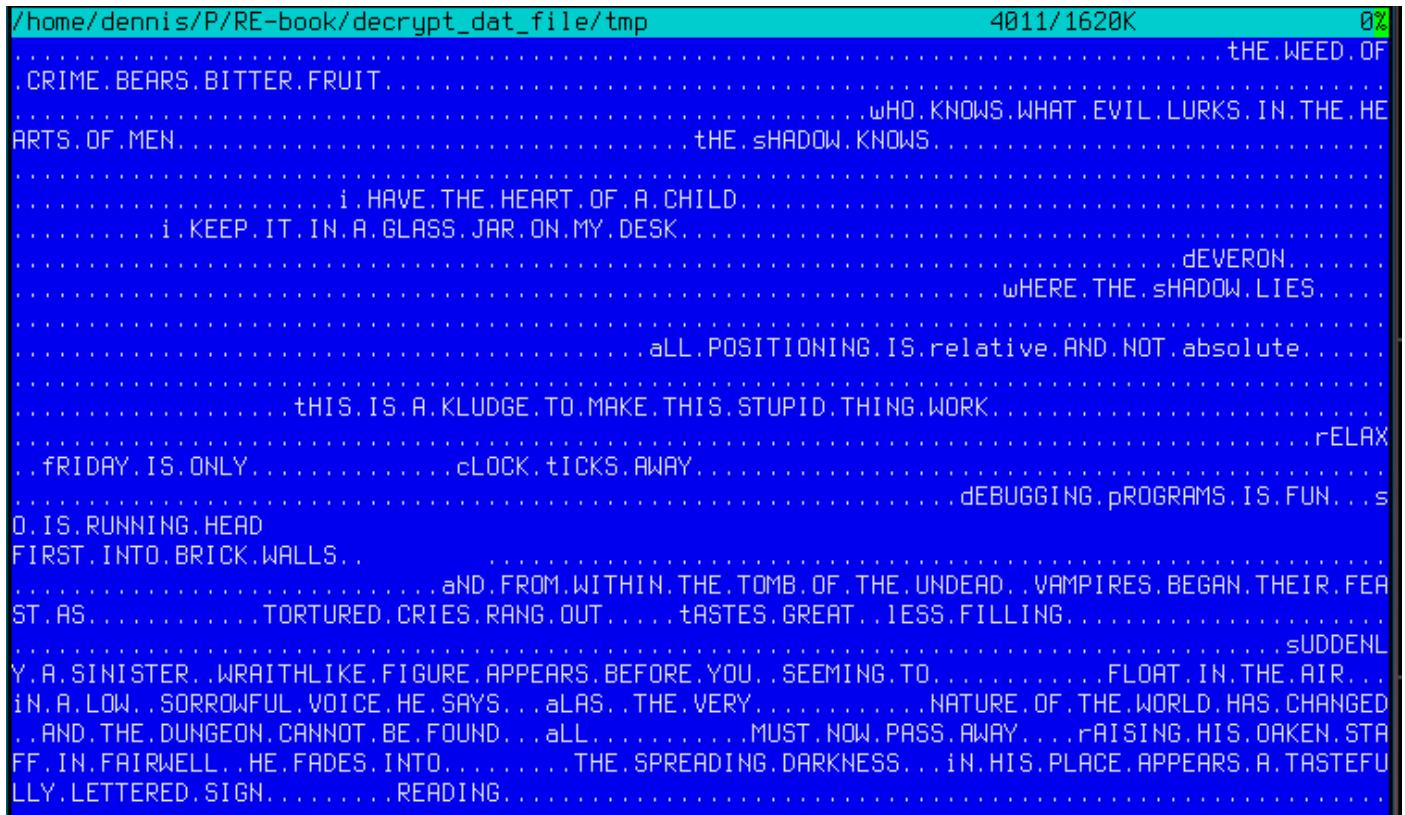
In[]:= BitXor[key, blocks[[2]]]
Out[]={0, 101, 72, 69, 0, 87, 69, 69, 68, 0, 79, 70, 0, 67, 82, \
73, 77, 69, 0, 66, 69, 65, 82, 83, 0, 66, 73, 84, 84, 69, 82, 0, 70, \
82, 85, 73, 84, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0}
```

## 9.1. PRIMITIVE XOR-ENCRYPTION

Encrypted byte is 2, byte from key is  $103$ ,  $2 \oplus 103 = 101$  and  $101$  is ASCII code for "e" character. What byte a key must be equal to, so the resulting ASCII code will be  $116$  (for "t" character)?  $2 \oplus 116 = 118$ , let's put  $118$  in key at the second byte...

```
key = {80, 118, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125,
 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5,
 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119,
 102, 30, 122, 2, 117}
```

... and decrypt the whole file again.



The screenshot shows a terminal window with the command `/home/dennis/P/RE-book/decrypt_dat_file/tmp` running. The output shows a decrypted file with various lines of text. Some lines are correctly formatted, while others have case inversions or punctuation errors. For example, "THE.WEED.OF CRIME.BEARS.BITTER.FRUIT" is present, but "i.HAVE.THE.HEART.OF.A.CHILD" and "i.KEEP.IT.IN.A.GLASS.JAR.ON.MY.DESK" are also visible. The file appears to be a mix of different text snippets, possibly from a game or a story.

Figure 9.9: Decrypted file in Midnight Commander, 2nd attempt

Wow, now the grammar is correct, all phrases started with correct letters. But still, case inversion is suspicious. Why would game's developer write them in such a manner? Maybe our key is still incorrect?

While observing ASCII table in Wikipedia article<sup>5</sup> we can notice that uppercase and lowercase letter's ASCII codes are differ in just one bit (6th bit starting at 1st,  $0b100000$ ). This bit in decimal form is  $32\dots 32$ ? But  $32$  is ASCII code for space!

Indeed, one can switch case just by XOR-ing ASCII character code with  $32$ .

It is possible that the empty lacunas in the file are not zero bytes, but rather spaces? Let's modify XOR key one more time (I'll XOR each byte of key by  $32$ ):

```
(* "32" is scalar and "key" is vector, but that's OK *)
In[]:= key3 = BitXor[32, key]
Out[]={112, 86, 34, 84, 81, 70, 86, 57, 67, 40, 51, 55, 84, 93, 75, \
57, 67, 77, 82, 70, 46, 89, 83, 63, 41, 85, 81, 79, 37, 36, 95, 60, \
90, 69, 40, 78, 46, 50, 92, 74, 48, 52, 72, 87, 40, 77, 58, 74, 41, \
65, 45, 67, 47, 87, 52, 73, 85, 66, 71, 86, 33, 94, 61, 65, 90, 49, \
47, 82, 78, 35, 37, 93, 93, 67, 94, 87, 70, 62, 90, 34, 85}
In[]:= DecryptBlock[blk_] := BitXor[key3, blk]
```

<sup>5</sup><https://en.wikipedia.org/wiki/ASCII>

## 9.1. PRIMITIVE XOR-ENCRYPTION

Let's decrypt the input file again:

```
/home/dennis/P/RE-book/decrypt_dat_file/tmp3
1
2
in the hearts of men?                                The Shadow knows!
2
I keep it in a glass jar on my desk.                  I H
Deveron:
Where the Shadow lies.                               1
All positioning is RELATIVE and not ABSOLUTE.
This is a kludge to make this
1
1
(So is running head-first into brick walls!!)      2
And from within the tomb of the undead, vampires began their feast as      tom
g!"                                              10
hlike figure appears before you, seeming to          float in the air. In a low, s
nature of the world has changed, and the dungeon cannot be found. All      mus
well, he fades into          the spreading darkness. In his place appears a tasteful
INITIALIZATION FAILURE
The darkness becomes all encompassing, and your vision fo
Lick My User Port!!!
1
CRATCH Paper.           1
hem you were playing GAMES all day...                1
Keep it up and we'll both go out for a beer.        1
No, odd addresses don't occur on the South side of the st
Did you really expect me to re
1
1
So
In
```

Figure 9.10: Decrypted file in Midnight Commander, final attempt

(Decrypted file is available [here](#).) This is undoubtedly a correct source file. Oh, and we see numbers at the start of each block. It has to be a source of our erroneous XOR key. As it seems, the most occurred 81-byte block in the file is a block filled with spaces and containing "1" character at the place of second byte. Indeed, somehow, many blocks here are interleaved with this one. Maybe it's some kind of padding for short phrases/messages? Other highly occurred 81-byte blocks are also space-filled blocks, but with different digits, hence, they are differ only at the second byte.

That's all! Now we can write utility to encrypt the file back, and maybe modify it before.

Mathematica notebook file is downloadable [here](#).

Summary: XOR encryption like that is not robust at all. It has been intended by game's developer(s), probably, just to prevent gamer(s) to peek into internals of game, nothing else. Still, encryption like that is extremely popular due to its simplicity and many reverse engineers are usually familiar with it.

### 9.1.4 Simple encryption using XOR mask, case II

I've got another encrypted file, which is clearly encrypted by something simple, like XOR-ing:

## 9.1. PRIMITIVE XOR-ENCRYPTION

0x00000000																
00000000	DD	D2	0F	70	1C	E7	9E	8D	E9	EC	AC	3D	61	5A	15	95
00000010	5C	F5	D3	0D	70	38	E7	94	DF	F2	E2	BC	76	34	61	0F
00000020	98	5D	FC	D9	01	26	2A	FD	82	DF	E9	E2	BB	33	61	7B
00000030	14	D9	45	F8	C5	01	3D	20	FD	95	96	EB	E4	BC	7A	61
00000040	61	1B	8F	54	9D	AA	54	20	20	E1	DB	8B	ED	EC	BC	33
00000050	61	7C	15	8D	11	F9	CE	47	22	2A	FE	8E	9A	EB	F7	EF
00000060	39	22	71	1B	8A	58	FF	CE	52	70	38	E7	9E	91	A5	EB
00000070	AA	76	36	73	09	D9	44	E0	80	40	3C	23	AF	95	96	E2
00000080	EB	BB	7A	61	65	1B	8A	11	E3	C5	40	24	2A	EB	F6	F5
00000090	E4	F7	EF	22	29	77	5A	9B	43	F5	C1	4A	36	2E	FC	8F
000000A0	DF	F1	E2	AD	3A	24	3C	5A	B0	11	E3	D4	4E	3F	2B	AF
000000B0	8E	8F	EA	ED	EF	22	29	77	5A	91	54	F1	D2	55	38	62
000000C0	FD	BE	98	A5	E2	A1	32	61	62	13	9A	5A	F5	C4	01	25
000000D0	3F	AF	8F	97	E0	8E	C5	25	35	7B	19	92	11	E7	C8	48
000000E0	33	27	AF	94	8A	F7	A3	B9	3F	32	7B	0E	96	43	B0	C8
000000F0	40	34	6F	E3	9E	99	F1	A3	AD	33	29	7B	14	9D	11	F8
00000100	C9	4C	70	3B	E7	9E	DF	EB	EA	A8	3E	35	32	18	9C	57
00000110	FF	D2	44	7E	6F	C6	8F	DF	F2	E2	BC	76	20	1F	70	9F
00000120	58	FE	C5	0D	70	3B	E7	92	9C	EE	A3	BF	3F	24	71	1F
00000130	D9	5E	F6	80	56	3F	20	EB	D7	DF	E7	F6	R3	34	2E	67
00000140	09	D4	59	F5	C1	45	35	2B	A3	DB	90	E3	A3	BB	3E	24
00000150	32	09	96	43	E4	80	56	3B	26	EC	93	DF	EC	F0	EF	3D
00000160	2F	7D	0D	97	11	F1	D3	2C	5A	2E	AF	D9	AF	E0	ED	AE
00000170	38	26	32	16	98	46	E9	C5	53	7E	6D	AF	B1	8A	F6	F7
00000180	EF	23	2F	76	1F	8B	11	E4	C8	44	70	27	EA	9A	9B	A5
00000190	F4	AE	25	61	73	5A	9B	43	FF	C1	45	70	3C	E6	97	89
000001A0	E0	F1	EF	34	20	7C	1E	D9	5F	F5	C1	53	3C	36	82	F1
000001B0	9E	EB	A3	A6	38	22	7A	5A	98	52	E2	CF	52	23	61	AF
000001C0	D9	AB	EA	A3	85	37	2C	77	09	D9	7C	FF	D2	55	39	22
000001D0	EA	89	D3	A5	CE	E1	04	6F	51	54	AA	1F	BC	80	47	22
000001E0	20	E2	DB	97	EC	F0	EF	30	33	7B	1F	97	55	E3	80	4E
000001F0	36	6F	FB	93	9A	88	89	8C	78	02	3C	32	D7	1D	B2	80

Figure 9.11: Encrypted file in Midnight Commander

The encrypted file can be downloaded [here](#).

`ent` Linux utility reports about 7.5 bits per byte, and this is high level of entropy ([9.2 on page 947](#)), close to compressed or properly encrypted file. But still, we clearly see some pattern, there are some blocks with size of 17 bytes, hence, you see some kind of ladder, shifting by 1 byte at each 16-byte line.

It's also known that the plain text is just English language text.

Now let's assume that this piece of text is encrypted by simple XOR-ing with 17-byte key.

I tried to find some repeating 17-byte blocks in Mathematica, like I did before in my previous example ([9.1.3 on page 935](#)):

Listing 9.2: Mathematica

```
In[]:=input = BinaryReadList["/home/dennis/tmp/cipher.txt"];
In[]:=blocks = Partition[input, 17];
In[]:=Sort[Tally[blocks], #1[[2]] > #2[[2]] &]
Out[]:={{248,128,88,63,58,175,159,154,232,226,161,50,97,127,3,217,80},1},
{{226,207,67,60,42,226,219,150,246,163,166,56,97,101,18,144,82},1},
{{228,128,79,49,59,250,137,154,165,236,169,118,53,122,31,217,65},1},
{{252,217,1,39,39,238,143,223,241,235,170,91,75,119,2,152,82},1},
{{244,204,88,112,59,234,151,147,165,238,170,118,49,126,27,144,95},1},
{{241,196,78,112,54,224,142,223,242,236,186,58,37,50,17,144,95},1},
{{176,201,71,112,56,230,143,151,234,246,187,118,44,125,8,156,17},1},
```

## 9.1. PRIMITIVE XOR-ENCRYPTION

```
...
{{255, 206, 82, 112, 56, 231, 158, 145, 165, 235, 170, 118, 54, 115, 9, 217, 68}, 1},
{{249, 206, 71, 34, 42, 254, 142, 154, 235, 247, 239, 57, 34, 113, 27, 138, 88}, 1},
{{157, 170, 84, 32, 32, 225, 219, 139, 237, 236, 188, 51, 97, 124, 21, 141, 17}, 1},
{{248, 197, 1, 61, 32, 253, 149, 150, 235, 228, 188, 122, 97, 97, 27, 143, 84}, 1},
{{252, 217, 1, 38, 42, 253, 130, 223, 233, 226, 187, 51, 97, 123, 20, 217, 69}, 1},
{{245, 211, 13, 112, 56, 231, 148, 223, 242, 226, 188, 118, 52, 97, 15, 152, 93}, 1},
{{221, 210, 15, 112, 28, 231, 158, 141, 233, 236, 172, 61, 97, 90, 21, 149, 92}, 1}}
```

No luck, each 17-byte block is unique within the file and occurred only once. Perhaps, there are no 17-byte zero (or space) lacunas. It is possible indeed: such long space indentation and padding may be absent in tightly typeset text.

The first idea is to try all possible 17-byte keys and find those, which will result in printable plain text after decryption. Bruteforce is not an option, because there are  $256^{17}$  possible keys ( $10^{40}$ ), that's too much. But there are good news: who said we have to test 17-byte key as a whole, why can't we test each byte of key separately? It is possible indeed.

Now the algorithm is:

- try all 256 bytes for 1st byte of key;
- decrypt 1st byte of each 17-byte blocks in the file;
- are all decrypted bytes we got are printable? keep tabs on it;
- do so for all 17 bytes of key.

I wrote with the following Python script to check this idea:

Listing 9.3: Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        possible_keys.append(i)
    print possible_keys, "len=", len(possible_keys)
```

(Full version of the source code is [here](#).)

Here is its output:

```
N= 0
[144, 145, 151] len= 3
N= 1
[160, 161] len= 2
N= 2
[32, 33, 38] len= 3
N= 3
[80, 81, 87] len= 3
N= 4
[78, 79] len= 2
N= 5
[142, 143] len= 2
N= 6
[250, 251] len= 2
N= 7
```

## 9.1. PRIMITIVE XOR-ENCRYPTION

```
[254, 255] len= 2
N= 8
[130, 132, 133] len= 3
N= 9
[130, 131] len= 2
N= 10
[206, 207] len= 2
N= 11
[81, 86, 87] len= 3
N= 12
[64, 65] len= 2
N= 13
[18, 19] len= 2
N= 14
[122, 123] len= 2
N= 15
[248, 249] len= 2
N= 16
[48, 49] len= 2
```

So there are 2 or 3 possible bytes for each byte of 17-byte key. This is much better than 256 possible bytes for each byte, but still too much. There are up to 1 million of possible keys:

Listing 9.4: Mathematica

```
In[]:= 3*2*3*3*2*2*2*3*2*2*3*2*2*2*2*2*2
Out[]= 995328
```

It's possible to check all of them, but then we must check visually, if the decrypted text is looks like English language text.

Let's also take into consideration the fact that we deal with 1) natural language; 2) English language. Natural languages has some prominent statistical features. First of all, punctuation and word lengths. What is average word length in English language? Let's just count spaces in some well-known English language texts using Mathematica.

Here is "[The Complete Works of William Shakespeare](#)" text file from Gutenberg Library:

Listing 9.5: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg100.txt"];
In[]:= Tally[input]
Out[]={{239, 1}, {187, 1}, {191, 1}, {84, 39878}, {104,
218875}, {101, 406157}, {32, 1285884}, {80, 12038}, {114,
209907}, {111, 282560}, {106, 2788}, {99, 67194}, {116,
291243}, {71, 11261}, {117, 115225}, {110, 216805}, {98,
46768}, {103, 57328}, {69, 42703}, {66, 15450}, {107, 29345}, {102,
69103}, {67, 21526}, {109, 95890}, {112, 46849}, {108, 146532}, {87,
16508}, {115, 215605}, {105, 199130}, {97, 245509}, {83,
34082}, {44, 83315}, {121, 85549}, {13, 124787}, {10, 124787}, {119,
73155}, {100, 134216}, {118, 34077}, {46, 78216}, {89, 9128}, {45,
8150}, {76, 23919}, {42, 73}, {79, 33268}, {82, 29040}, {73,
55893}, {72, 18486}, {68, 15726}, {58, 1843}, {65, 44560}, {49,
982}, {50, 373}, {48, 325}, {91, 2076}, {35, 3}, {93, 2068}, {74,
2071}, {57, 966}, {52, 107}, {70, 11770}, {85, 14169}, {78,
27393}, {75, 6206}, {77, 15887}, {120, 4681}, {33, 8840}, {60,
468}, {86, 3587}, {51, 343}, {88, 608}, {40, 643}, {41, 644}, {62,
440}, {39, 31077}, {34, 488}, {59, 17199}, {126, 1}, {95, 71}, {113,
2414}, {81, 1179}, {63, 10476}, {47, 48}, {55, 45}, {54, 73}, {64,
3}, {53, 94}, {56, 47}, {122, 1098}, {90, 532}, {124, 33}, {38,
21}, {96, 1}, {125, 2}, {37, 1}, {36, 2}}
```

```
In[]:= Length[input]/1285884 // N
Out[]= 4.34712
```

There are 1285884 spaces in the whole file, and the frequency of space occurrence is 1 space per 4.3 characters.

Now here is [Alice's Adventures in Wonderland, by Lewis Carroll](#) from the same library:

## 9.1. PRIMITIVE XOR-ENCRYPTION

Listing 9.6: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg11.txt"];
In[]:= Tally[input]
Out[]={{239, 1}, {187, 1}, {191, 1}, {80, 172}, {114, 6398}, {111,
9243}, {106, 222}, {101, 15082}, {99, 2815}, {116, 11629}, {32,
27964}, {71, 193}, {117, 3867}, {110, 7869}, {98, 1621}, {103,
2750}, {39, 2885}, {115, 6980}, {65, 721}, {108, 5053}, {105,
7802}, {100, 5227}, {118, 911}, {87, 256}, {97, 9081}, {44,
2566}, {121, 2442}, {76, 158}, {119, 2696}, {67, 185}, {13,
3735}, {10, 3735}, {84, 571}, {104, 7580}, {66, 125}, {107,
1202}, {102, 2248}, {109, 2245}, {46, 1206}, {89, 142}, {112,
1796}, {45, 744}, {58, 255}, {68, 242}, {74, 13}, {50, 12}, {53,
13}, {48, 22}, {56, 10}, {91, 4}, {69, 313}, {35, 1}, {49, 68}, {93,
4}, {82, 212}, {77, 222}, {57, 11}, {52, 10}, {42, 88}, {83,
288}, {79, 234}, {70, 134}, {72, 309}, {73, 831}, {85, 111}, {78,
182}, {75, 88}, {86, 52}, {51, 13}, {63, 202}, {40, 76}, {41,
76}, {59, 194}, {33, 451}, {113, 135}, {120, 170}, {90, 1}, {122,
79}, {34, 135}, {95, 4}, {81, 85}, {88, 6}, {47, 24}, {55, 6}, {54,
7}, {37, 1}, {64, 2}, {36, 2}}
```

```
In[]:= Length[input]/27964 // N
Out[]= 5.99049
```

The result is different probably because of different formatting of these texts (maybe indentation and/or padding).

OK, so let's assume the average frequency of space in English language is 1 space per 4..7 characters.

Now the good news again: we can measure frequency of spaces while decrypting our file gradually. Now I count spaces in each *slice* and throw away 1-byte keys which produce buffers with too small number of spaces (or too large, but this is almost impossible given so short key):

Listing 9.7: Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        # count spaces in decrypted buffer:
        spaces=tmp.count(' ')
        if spaces==0:
            continue
        spaces_ratio=len(tmp)/spaces
        if spaces_ratio<4:
            continue
        if spaces_ratio>7:
            continue
        possible_keys.append(i)
print possible_keys, "len=", len(possible_keys)
```

(Full version of the source code is [here](#).)

This reports just one single possible byte for each byte of key:

N= 0

## 9.1. PRIMITIVE XOR-ENCRYPTION

```
[144] len= 1
N= 1
[160] len= 1
N= 2
[33] len= 1
N= 3
[80] len= 1
N= 4
[79] len= 1
N= 5
[143] len= 1
N= 6
[251] len= 1
N= 7
[255] len= 1
N= 8
[133] len= 1
N= 9
[131] len= 1
N= 10
[207] len= 1
N= 11
[86] len= 1
N= 12
[65] len= 1
N= 13
[18] len= 1
N= 14
[122] len= 1
N= 15
[249] len= 1
N= 16
[49] len= 1
```

Let's check this key in Mathematica:

Listing 9.8: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/cipher.txt"];
In[]:= blocks = Partition[input, 17];
In[]:= key = {144, 160, 33, 80, 79, 143, 251, 255, 133, 131, 207, 86, 65, 18, 122, 249, 49};
In[]:= EncryptBlock[blk_] := BitXor[key, blk]
In[]:= encrypted = Map[EncryptBlock[#] &, blocks];
In[]:= BinaryWrite["/home/dennis/tmp/plain2.txt", Flatten[encrypted]]
In[]:= Close["/home/dennis/tmp/plain2.txt"]
```

And the plain text is:

Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the stick which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer." Just under the head was a broad silver band nearly an inch across. "To James Mortimer, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry--dignified, solid, and reassuring.

"Well, Watson, what do you make of it?"

Holmes was sitting with his back to me, and I had given him no sign of my occupation.

...

(Full version of the text is [here](#).)

The text looks correct. Yes, I made up this example and choose well-known text of Conan Doyle, but it's very close to what I had in my practice some time ago.

### Other ideas to consider

If we would fail with space counting, there are other ideas to try:

- Take into consideration the fact that lowercase letters are much more frequent than uppercase ones.
- Frequency analysis.
- There is also a good technique to detect language of a text: trigrams. Each language has some very frequent letter triplets, these may be "the" and "tha" for English. Read more about it: [N-Gram-Based Text Categorization](#), <http://code.activestate.com/recipes/326576/>. Interestingly enough, trigrams detection can be used when you decrypt a ciphertext gradually, like in this example (you just have to test 3 adjacent decrypted characters).

For non-Latin writing systems encoded in UTF-8, things may be easier. For example, Russian text encoded in UTF-8 has each byte interleaved with 0xD0/0xD1 byte. It is because Cyrillic characters are placed in 4th block of Unicode. Other writing systems has their own blocks.

## 9.2 Analyzing unknown binary files using information entropy

For the sake of simplification, I would say, information entropy is a measure, how tightly some piece of data can be compressed. For example, it is usually not possible to compress already compressed archive file, so it has high entropy. On the other hand, one megabyte of zero bytes can be compressed to a tiny output file. Indeed, in plain English language, one million of zeros can be described just as "resulting file is one million zero bytes". Compressed files are usually a list of instructions to decompressor, like this: "put 1000 zeros, then 0x23 byte, then 0x45 byte, then put a block of size 10 bytes which we've seen 500 bytes back, etc."

Texts written in natural languages are also can be compressed tightly, because natural languages has a lot of redundancy (otherwise, a tiny typo will always lead to misunderstanding, like any toggled bit in compressed archive make decompression nearly impossible), some words are used very often, etc. It's possible to drop some words and text will be still readable.

Code for CPUs is also can be compressed, because some ISA instructions are used much more often than others. In x86, most used instructions are MOV/PUSH/CALL—indeed, most of the time, computer CPU is just shuffling data and switching between levels of abstractions. If to consider data shuffling as moving data between levels of abstractions, this is also part of switching.

Data compressors and encryptors tend to produce very high entropy results. Good pseudorandom number generators also produce data which cannot be compressed (it is possible to measure their quality by this sign).

So, in other words, entropy is a measure which can help to probe unknown data block.

### 9.2.1 Analyzing entropy in Mathematica

(This part has been first appeared in my blog at 13-May-2015. Some discussion: <https://news.ycombinator.com/item?id=9545276>.)

It is possible to slice some file by blocks, probe each and draw a graph a graph. I did this in Wolfram Mathematica for demonstration and here is a source code (Mathematica 10):

```
(* loading the file *)
input=BinaryReadList["file.bin"];

(* setting block sizes *)
BlockSize=4096;BlockSizeToShow=256;
```

## 9.2. ANALYZING USING INFORMATION ENTROPY

```
(* slice blocks by 4k *)
blocks=Partition[input,BlockSize];

(* how many blocks we've got? *)
Length[blocks]

(* calculate entropy for each block. 2 in Entropy[] (base) is set with the intention so Entropy[
   ↴ []
function will produce the same results as Linux ent utility does *)
entropies=Map[N[Entropy[2,#]]&,blocks];

(* helper functions *)
fBlockToShow[input_,offset_]:=Take[input,{1+offset,1+offset+BlockSizeToShow}]
fToASCII[val_]:=FromCharacterCode[val,"PrintableASCII"]
fToHex[val_]:=IntegerString[val,16]
fPutASCIIWindow[data_]:=Framed[Grid[Partition[Map[fToASCII,data],16]]]
fPutHexWindow[data_]:=Framed[Grid[Partition[Map[fToHex,data],16],Alignment->Right]]

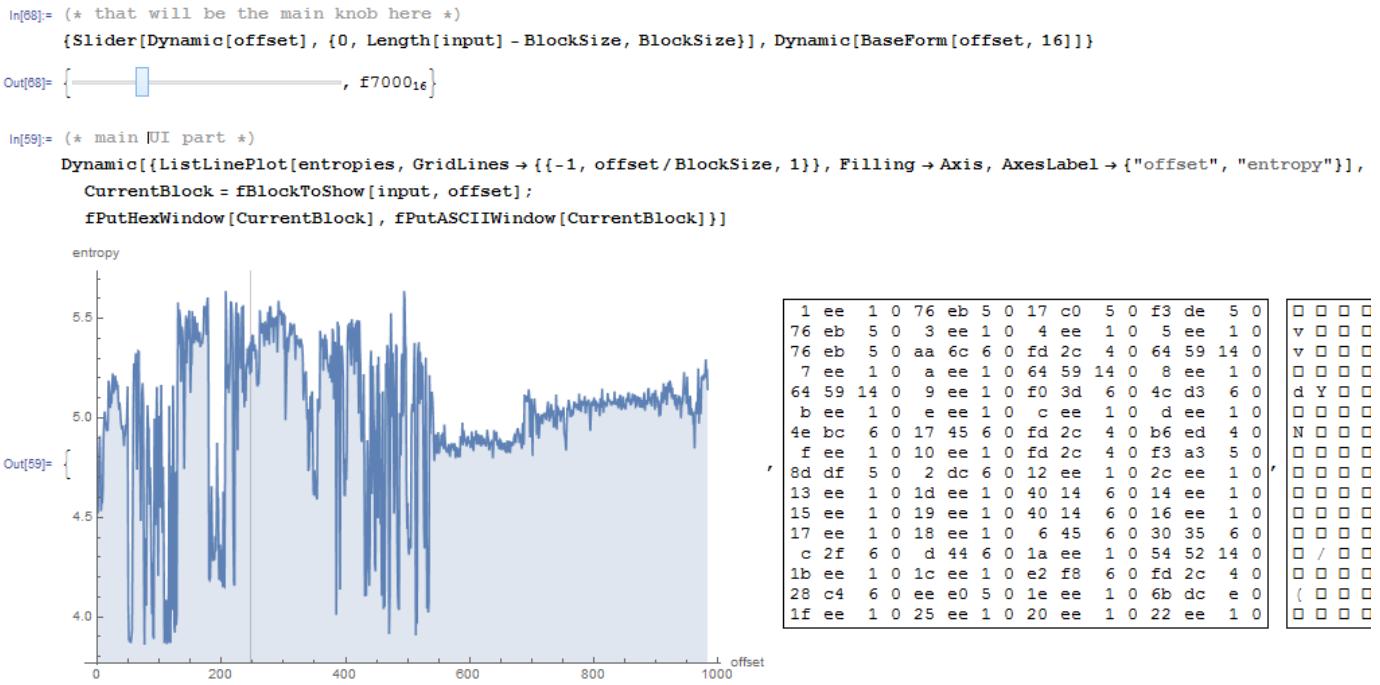
(* that will be the main knob here *)
{Slider[Dynamic[offset],{0,Length[input]-BlockSize,BlockSize}],Dynamic[BaseForm[offset,16]]}

(* main UI part *)
Dynamic[{ListLinePlot[entropies,GridLines->{{-1,offset/BlockSize,1}},Filling->Axis,AxesLabel[
   ↴ ->{"offset","entropy"}],
CurrentBlock=fBlockToShow[input,offset];
fPutHexWindow[CurrentBlock],
fPutASCIIWindow[CurrentBlock]}]
```

## GeoIP ISP database

Let's start with the [GeoIP](#) file (which assigns ISP to the block of IP addresses). This binary file (*GeoIPISP.dat*) has some tables (which are IP address ranges perhaps) plus some text blob at the end of the file (containing ISP names).

When I load it to Mathematica, I see this:

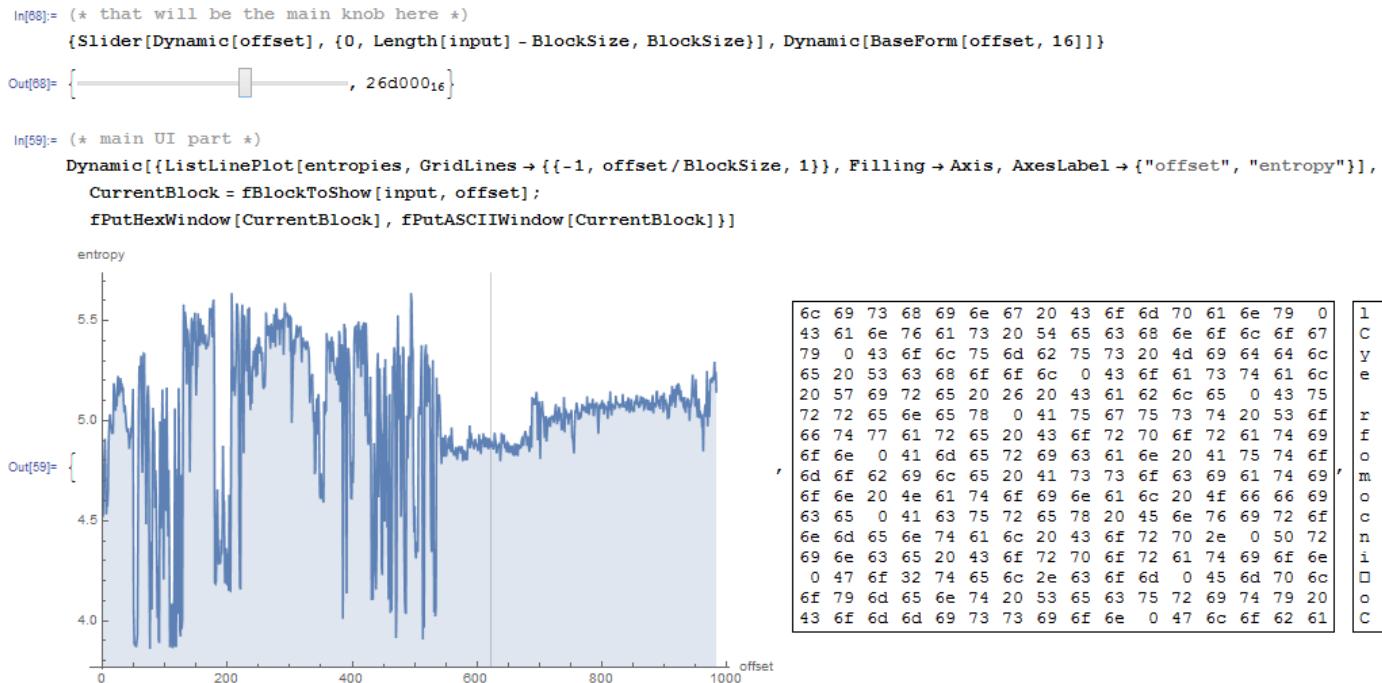


There are two parts in graph: first is somewhat chaotic, second is more steady.

0 in horizontal axis in graph means lowest entropy (the data which can be compressed very tightly, *ordered* in other words) and 8 is highest (cannot be compressed at all, *chaotic* or *random* in other words). Why 0 and 8? 0 means 0 bits per byte (byte slot is not filled at all) and 8 means 8 bits per byte, i.e., the whole byte slot is filled with the information tightly.

## 9.2. ANALYZING USING INFORMATION ENTROPY

So I put slider to point in the middle of the first block, and I clearly see some array of 32-bit integers. Now I put slider in the middle of the second block and I see English text:



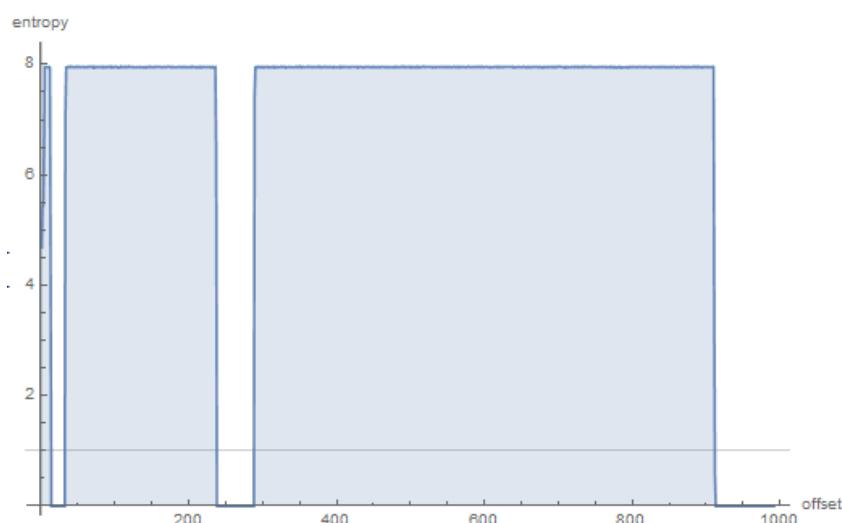
Indeed, this are names of ISPs. So, entropy of English text is 4.5-5.5 bits per byte? Yes, something like this. Wolfram Mathematica has some well-known English literature corpus embedded, and we can see entropy of Shakespeare's sonnets:

```
In[]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]]//N
Out[] = 4.42366
```

4.4 is close to what we've got (4.7-5.3). Of course, classic English literature texts are somewhat different from ISP names and other English texts we can find in binary files (debugging/logging/error messages), but this value is close.

### TP-Link WR941 firmware

Now advanced example. I've got firmware for TP-Link WR941 router:



We see here 3 blocks with empty lacunas. The first block (started at address 0) is small, second (address somewhere at 0x22000) is bigger and third (address 0x123000) is biggest. I can't be sure about exact entropy of the first block, but 2nd and 3rd has very high entropy, meaning that these blocks are either compressed and/or encrypted.

## 9.2. ANALYZING USING INFORMATION ENTROPY

I tried [binwalk](#) for this firmware file:

DECIMAL	HEXADECIMAL	DESCRIPTION
0	0x0	TP-Link firmware header, firmware version: 0..-15221.3, image ↴ ↳ version: "", product ID: 0x0, product version: 155254789, kernel load address: 0x0, ↴ ↳ kernel entry point: 0x-7FFE000, kernel offset: 4063744, kernel length: 512, rootfs ↴ ↳ offset: 837431, rootfs length: 1048576, bootloader offset: 2883584, bootloader length: 0
14832	0x39F0	U-Boot version string, "U-Boot 1.1.4 (Jun 27 2014 - 14:56:49)"
14880	0x3A20	CRC32 polynomial table, big endian
16176	0x3F30	uImage header, header size: 64 bytes, header CRC: 0x3AC66E95, ↴ ↳ created: 2014-06-27 06:56:50, image size: 34587 bytes, Data Address: 0x80010000, Entry ↴ ↳ Point: 0x80010000, data CRC: 0xDF2DBA0B, OS: Linux, CPU: MIPS, image type: Firmware Image ↴ ↳ , compression type: lzma, image name: "u-boot image"
16240	0x3F70	LZMA compressed data, properties: 0x5D, dictionary size: 33554432 ↴ ↳ bytes, uncompressed size: 90000 bytes
131584	0x20200	TP-Link firmware header, firmware version: 0.0.3, image version: ↴ ↳ "", product ID: 0x0, product version: 155254789, kernel load address: 0x0, kernel entry ↴ ↳ point: 0x-7FFE000, kernel offset: 3932160, kernel length: 512, rootfs offset: 837431, ↴ ↳ rootfs length: 1048576, bootloader offset: 2883584, bootloader length: 0
132096	0x20400	LZMA compressed data, properties: 0x5D, dictionary size: 33554432 ↴ ↳ bytes, uncompressed size: 2388212 bytes
1180160	0x120200	Squashfs filesystem, little endian, version 4.0, compression:lzma ↴ ↳ , size: 2548511 bytes, 536 inodes, blocksize: 131072 bytes, created: 2014-06-27 07:06:52

Indeed: there are some stuff at the beginning, but two large LZMA compressed blocks are started at 0x20400 and 0x120200. These are roughly addresses we have seen in Mathematica. Oh, and by the way, binwalk can show entropy information as well (-E option):

DECIMAL	HEXADECIMAL	ENTROPY
0	0x0	Falling entropy edge (0.419187)
16384	0x4000	Rising entropy edge (0.988639)
51200	0xC800	Falling entropy edge (0.000000)
133120	0x20800	Rising entropy edge (0.987596)
968704	0xEC800	Falling entropy edge (0.508720)
1181696	0x120800	Rising entropy edge (0.989615)
3727360	0x38E000	Falling entropy edge (0.732390)

Rising edges are corresponding to rising edges of block on our graph. Falling edges are the points where empty lacunas are started.

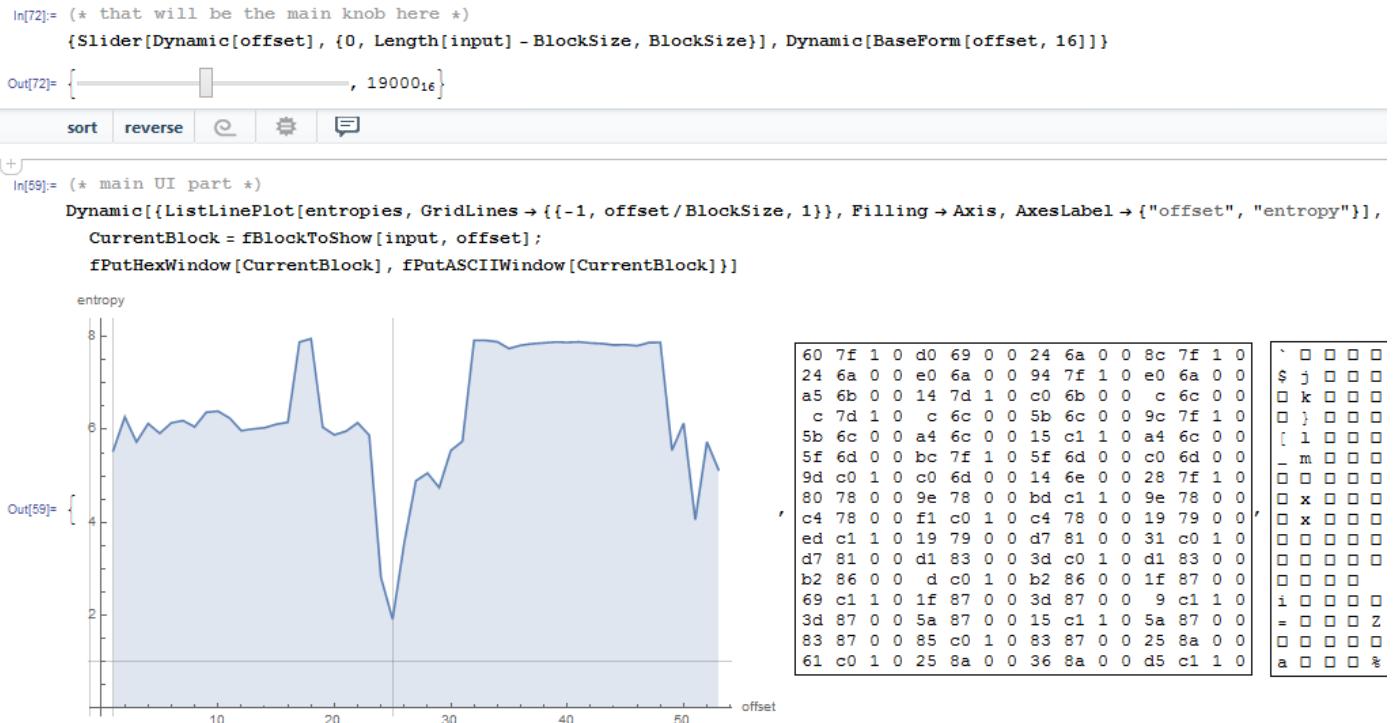
I wasn't able to force binwalk to generate PNG graphs (due to absence of some Python library), but here is an example how binwalk can do them: [http://binwalk.org/wp-content/uploads/2013/12/lg\\_dtv.png](http://binwalk.org/wp-content/uploads/2013/12/lg_dtv.png).

What can we say about lacunas? By looking in hex editor, we see that these are just filled with 0xFF bytes. Why developers put them? Perhaps, because they weren't able to calculate precise compressed blocks sizes, so they allocated space for them with some reserve.

## Notepad

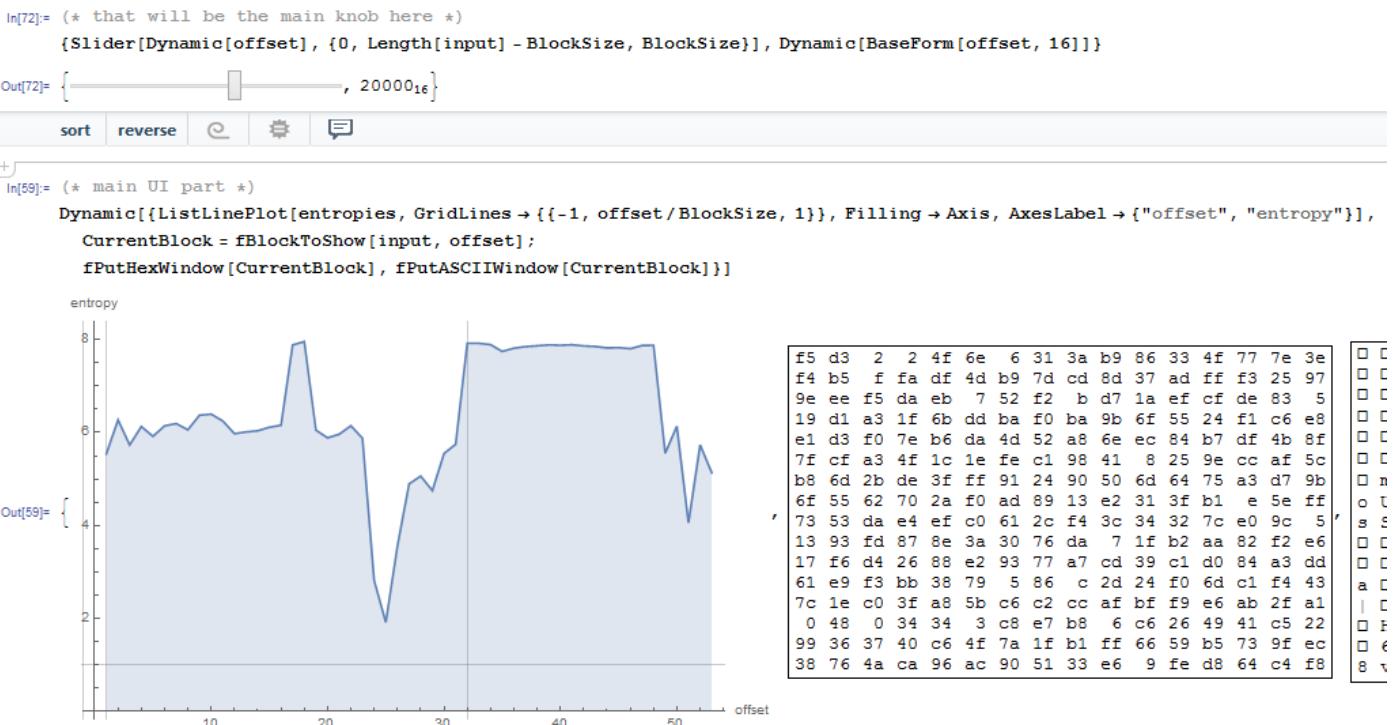
Another example is notepad.exe I've picked in Windows 8.1:

## 9.2. ANALYZING USING INFORMATION ENTROPY



There is cavity at 0x19000 (absolute file offset). I opened the executable file in hex editor and found imports table there (which has lower entropy than x86-64 code in the first half of graph).

There are also high entropy block started at 0x20000:

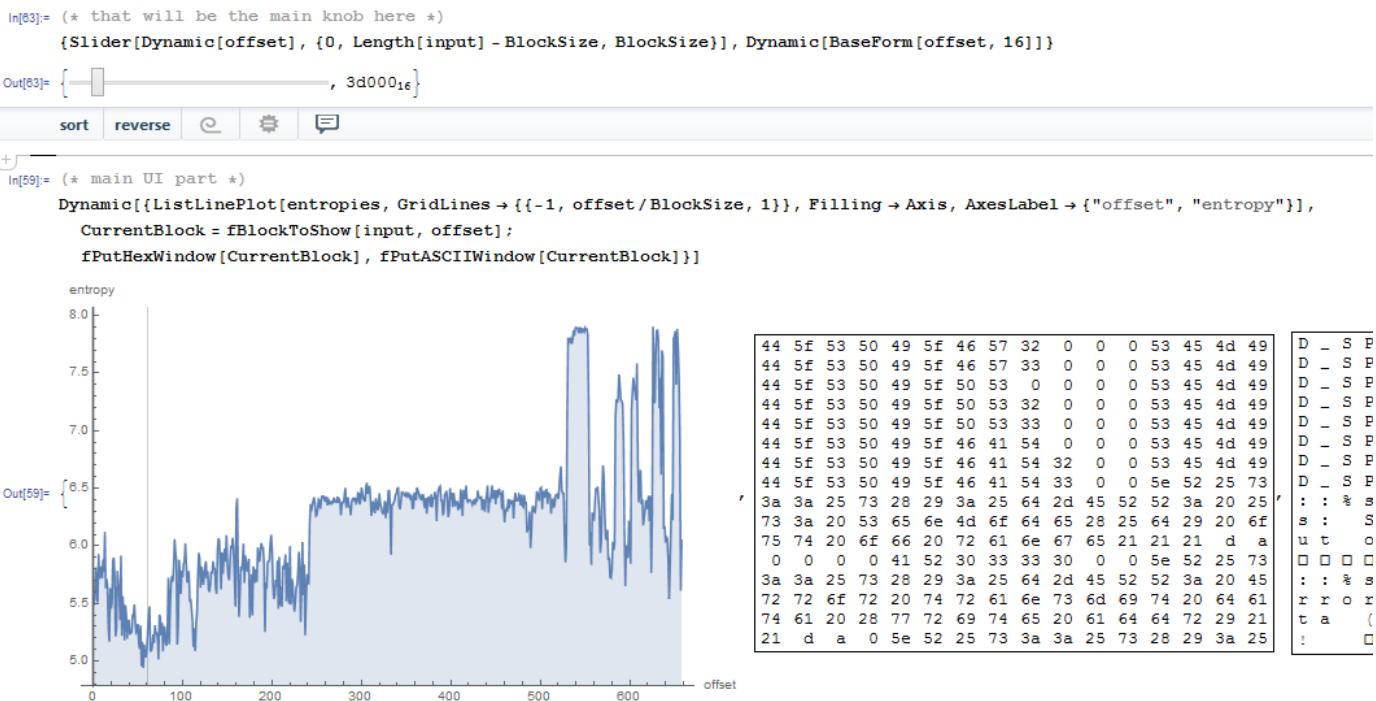


In hex editor I can see PNG file here, embedded in the PE file resource section (it is a large image of notepad icon). PNG files are compressed, indeed.

### Unnamed dashcam

Now the most advanced example in this article is the firmware of some unnamed dashcam I've received from friend:

## 9.2. ANALYZING USING INFORMATION ENTROPY



The cavity at the very beginning is an English text: debugging messages. I checked various ISAs and I found that the first third of the whole file (with the text segment inside) is in fact MIPS (little-endian) code!

For instance, this is very distinctive MIPS function epilogue:

ROM:000013B0	move	\$sp, \$fp
ROM:000013B4	lw	\$ra, 0x1C(\$sp)
ROM:000013B8	lw	\$fp, 0x18(\$sp)
ROM:000013BC	lw	\$s1, 0x14(\$sp)
ROM:000013C0	lw	\$s0, 0x10(\$sp)
ROM:000013C4	jr	\$ra
ROM:000013C8	addiu	\$sp, 0x20

From our graph we can see that MIPS code has entropy of 5-6 bits per byte. Indeed, I once measured various ISAs entropy and I've got these values:

- x86: .text section of ntoskrnl.exe file from Windows 2003: 6.6
- x64: .text section of ntoskrnl.exe file from Windows 7 x64: 6.5
- ARM (thumb mode), Angry Birds Classic: 7.05
- ARM (ARM mode) Linux Kernel 3.8.0: 6.03
- MIPS (little endian), .text section of user32.dll from Windows NT 4: 6.09

So the entropy of executable code is higher than of English text, but still can be compressed.

Now the second third is started at 0xF5000. I don't know what this is. I tried different ISAs but without success. The entropy of the block looks even steadier than for executable one. Maybe some kind of data?

There is also a spike at 0x213000. I checked it in hex editor and I found JPEG file there (which, of course, compressed)! I also don't know what is at the end. Let's try Binwalk for this file:

```
dennis@ubuntu:~/P/entropy$ binwalk FW96650A.bin
```

DECIMAL	HEXADECIMAL	DESCRIPTION
<hr/>		
167698	0x28F12	Unix path: /15/20/24/25/30/60/120/240fps can be served..
280286	0x446DE	Copyright string: "Copyright (c) 2012 Novatek Microelectronic ↴ Corp."
2169199	0x21196F	JPEG image data, JFIF standard 1.01
2300847	0x231BAF	MySQL MISAM compressed data file Version 3

```
dennis@ubuntu:~/P/entropy$ binwalk -E FW96650A.bin
```

## 9.2. ANALYZING USING INFORMATION ENTROPY

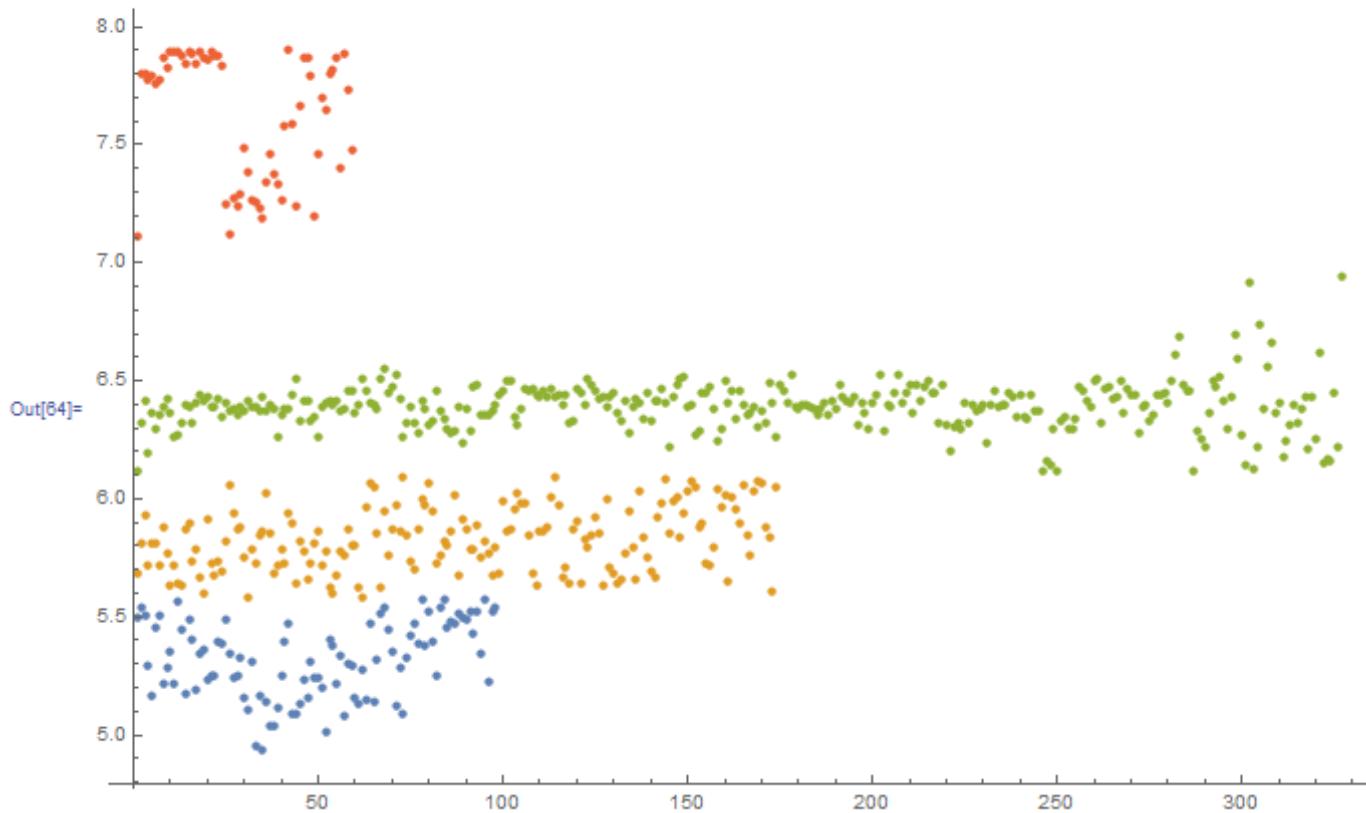
WARNING: pyqtgraph not found, visual entropy graphing will be disabled

DECIMAL	HEXADECIMAL	ENTROPY
0	0x0	Falling entropy edge (0.579792)
2170880	0x212000	Rising entropy edge (0.967373)
2267136	0x229800	Falling entropy edge (0.802974)
2426880	0x250800	Falling entropy edge (0.846639)
2490368	0x260000	Falling entropy edge (0.849804)
2560000	0x271000	Rising entropy edge (0.974340)
2574336	0x274800	Rising entropy edge (0.970958)
2588672	0x278000	Falling entropy edge (0.763507)
2592768	0x279000	Rising entropy edge (0.951883)
2596864	0x27A000	Falling entropy edge (0.712814)
2600960	0x27B000	Rising entropy edge (0.968167)
2607104	0x27C800	Rising entropy edge (0.958582)
2609152	0x27D000	Falling entropy edge (0.760989)
2654208	0x288000	Rising entropy edge (0.954127)
2670592	0x28C000	Rising entropy edge (0.967883)
2676736	0x28D800	Rising entropy edge (0.975779)
2684928	0x28F800	Falling entropy edge (0.744369)

Yes, it found JPEG file and even MySQL data! But I'm not sure if it's true—I didn't check it yet.

It's also interesting to try clusterization in Mathematica:

```
In[64]:= (* let also take a look on clustering attempt of Mathematica *)
ListPlot[FindClusters[entropies]]
```



Here is an example of how Mathematica grouped various entropy values into distinctive groups. Indeed, there is something credible. Blue dots in range of 5.0-5.5 are supposedly related to English text. Yellow dots in 5.5-6 are MIPS code. A lot of green dots in 6.0-6.5 is the unknown second third. Orange dots close to 8.0 are related to compressed JPEG file. Other orange dots are supposedly related to the end of the firmware (unknown to us data).

**Links**

Binary files used while writing: <http://yurichev.com/blog/entropy/files/>. Wolfram Mathematica notebook file: [http://yurichev.com/blog/entropy/files/binary\\_file\\_entropy.nb](http://yurichev.com/blog/entropy/files/binary_file_entropy.nb) (all cells must be evaluated to start things working).

**9.2.2 Conclusion**

Information entropy can be used as a quick-n-dirty method for inspecting unknown binary files. In particular, it is a very quick way to find compressed/encrypted pieces of data. Someone say it's possible to find RSA (and other asymmetric cryptographic algorithms) public/private keys in executable code (which has high entropy as well), but I didn't try this myself.

**9.2.3 Tools**

Handy Linux `ent` utility to measure entropy of a file<sup>6</sup>.

There is a great online entropy visualizer made by Aldo Cortesi, which I tried to mimic using Mathematica: <http://binvis.io>. His articles about entropy visualization are worth reading: <http://corte.si/posts/visualisation/entropy/index.html>, <http://corte.si/posts/visualisation/malware/index.html>, <http://corte.si/posts/visualisation/binvis/index.html>.

`radare2` framework has `#entropy` command for this.

A tool for IDA: `IDAtropy`<sup>7</sup>.

**9.2.4 A word about primitive encryption like XORing**

It's interesting that simple XOR encryption doesn't affect entropy of data. I've shown this in *Norton Guide* example in the book ([9.1.1 on page 928](#)).

Generalizing: encryption by substitution cipher also doesn't affect entropy of data (and XOR can be viewed as substitution cipher). The reason of that is because entropy calculation algorithm view data on byte-level. On the other hand, the data encrypted by 2 or 4-byte XOR pattern will result in another entropy.

Nevertheless, low entropy is usually a good sign of weak amateur cryptography (which is also used in license keys, license files, etc.).

**9.2.5 More about entropy of executable code**

It is quickly noticeable that probably a biggest source of high-entropy in executable code are relative offsets encoded in opcodes. For example, these two consequent instructions will produce different relative offsets in their opcodes, while they are in fact pointing to the same function:

```
function proc
...
function endp
...
CALL function
...
CALL function
```

Ideal executable code compressor would encode information like this: *there is a CALL to a "function" at address X and the same CALL at address Y without necessity to encode address of the function twice*.

To deal with this, executable compressors are sometimes able to reduce entropy here. One example is UPX: <http://sourceforge.net/p/upx/code/ci/default/tree/doc/filter.txt>.

<sup>6</sup><http://www.fourmilab.ch/random/>

<sup>7</sup><https://github.com/danigargu/IDAtropy>

## 9.2.6 Random number generators

When I run GnuPG to generate new secret key, it asking for some entropy...

We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy! (Need 169 more bytes)

This means that good a PRNG produces long high-entropy results, and this is what the secret asymmetrical cryptographical key needs. But [CPRNG<sup>8</sup>](#) is tricky (because computer is highly deterministic device itself), so the GnuPG asking for some additional randomness from the user.

Here is a case where I made attempt to calculate entropy of some unknown blocks: [8.7 on page 863](#).

### 9.2.7 Entropy of various files

Entropy of random bytes is close to 8:

```
$ dd bs=1M count=1 if=/dev/urandom | ent
Entropy = 7.999803 bits per byte.
```

This means, almost all available space inside of byte is filled with information.

256 bytes in range of 0..255 gives exact value of 8:

```
#!/usr/bin/env python
import sys

for i in range(256):
    sys.stdout.write(chr(i))
```

```
python 1.py | ent
Entropy = 8.000000 bits per byte.
```

Order of bytes doesn't matter. This means, all available space inside of byte is filled.

Entropy of all zero bytes is 0:

```
$ dd bs=1M count=1 if=/dev/zero | ent
Entropy = 0.000000 bits per byte.
```

Entropy of a string consisting of a single (any) byte is 0:

```
$ echo -n "aaaaaaaaaaaaaaaaaa" | ent
Entropy = 0.000000 bits per byte.
```

Entropy of base64 string is the same as entropy of source data, but multiplied by  $\frac{3}{4}$ . This is because base64 encoding uses 64 symbols instead of 256.

```
$ dd bs=1M count=1 if=/dev/urandom | base64 | ent
Entropy = 6.022068 bits per byte.
```

Perhaps, 6.02 not that close to 6 because padding symbols (=) spoils our statistics for a little.

Uuencode, also uses 64 symbols:

```
$ dd bs=1M count=1 if=/dev/urandom | uuencode - | ent
Entropy = 6.013162 bits per byte.
```

This means, any base64 and Uuencode strings can be transmitted using 6-bit bytes or characters.

Any random information in hexadecimal form has entropy of 4 bits per byte:

<sup>8</sup>Cryptographically secure Pseudorandom Number Generator

## 9.2. ANALYZING USING INFORMATION ENTROPY

```
$ openssl rand -hex $(( 2**16 )) | ent
Entropy = 4.000013 bits per byte.
```

Entropy of randomly picked English language text from Gutenberg library has entropy 4.5. The reason of this is because English texts uses mostly 26 symbols, and  $\log_2(26) = 4.7$ , i.e., you would need 5-bit bytes to transmit uncompressed English texts, that would be enough (it was indeed so in teletype era).

Randomly chosen Russian language text from <http://lib.ru> library is F.M.Dostoevsky "Idiot"<sup>9</sup>, internally encoded in CP1251 encoding.

And this file has entropy of 4.98. Russian language has 33 characters, and  $\log_2(33) = 5.04$ . But it has unpopular and rare “ë” character<sup>10</sup>. And  $\log_2(32) = 5$  (Russian alphabet without this rare character) — now this close to what we've got.

In fact, the text we studying uses “ë” letter, but, probably, it's still rarely used there.

The very same file transcoded from CP1251 to UTF-8 gave entropy of 4.23. Each Cyrillic character encoded in UTF-8 is usually encoded as a pair, and the first byte is always one of: 0xD0 or 0xD1. Perhaps, this caused bias.

Let's generate random bits and output them as “T” or “F” characters:

```
#!/usr/bin/env python
import random, sys

rt=""
for i in range(102400):
    if random.randint(0,1)==1:
        rt=rt+"T"
    else:
        rt=rt+"F"
print rt
```

Sample: ...TTTFTFTTTFFFFTTTFTTTTTFTTTFFTTFTFTTTFTTTFFFF... .

Entropy is very close to 1 (i.e., 1 bit per byte).

Let's generate random decimal numbers:

```
#!/usr/bin/env python
import random, sys

rt=""
for i in range(102400):
    rt=rt,"%d" % random.randint(0,9)
print rt
```

Sample: ...52203466119390328807552582367031963888032.... .

Entropy will be close to 3.32, indeed, this is  $\log_2(10)$ .

### 9.2.8 Making lower level of entropy

The author of these lines once saw a software which stored each byte of encrypted data in 3 bytes: each has roughly  $\frac{byte}{3}$  value, so reconstructing encrypted byte back involving summing up 3 consecutive bytes. Looks absurdly.

But some people say this was done in order to conceal the very fact the data has something encrypted inside: measuring entropy of such block will show much lower level of it.

<sup>9</sup>[http://az.lib.ru/d/dostoevskij\\_f\\_m/text\\_0070.shtml](http://az.lib.ru/d/dostoevskij_f_m/text_0070.shtml)

<sup>10</sup>When I typed it here in text, I've needed to look down to my keyboard and find it.

## 9.3 Millennium game save file

The “Millenium Return to Earth” is an ancient DOS game (1991), that allows you to mine resources, build ships, equip them on other planets, and so on<sup>11</sup>.

Like many other games, it allows you to save all game state into a file.

Let's see if we can find something in it.

---

<sup>11</sup>It can be downloaded for free [here](#)

### 9.3. MILLENIUM GAME SAVE FILE

So there is a mine in the game. Mines at some planets work faster, or slower on others. The set of resources is also different.

Here we can see what resources are mined at the time:



Figure 9.12: Mine: state 1

Let's save a game state. This is a file of size 9538 bytes.

Let's wait some "days" here in the game, and now we've got more resources from the mine:

### **9.3. MILLENIUM GAME SAVE FILE**



Figure 9.13: Mine: state 2

Let's save game state again.

Now let's try to just do binary comparison of the save files using the simple DOS/Windows FC utility:

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BE8: 99 E8
00000BEC: A1 F3
00000BEE: 83 C7
00000BFB: A8 28
00000BFD: 98 18
00000BFF: A8 28
00000C01: A8 28
00000C07: D8 58
00000C09: E4 A4
00000C0D: 38 B8
00000C0F: E8 68
...

```

The output is incomplete here, there are more differences, but we will cut result to show the most interesting.

In the first state, we have 14 “units” of hydrogen and 102 “units” of oxygen.

### 9.3. MILLENIUM GAME SAVE FILE

We have 22 and 155 “units” respectively in the second state. If these values are saved into the save file, we would see this in the difference. And indeed we do. There is 0x0E (14) at position 0xBDA and this value is 0x16 (22) in the new version of the file. This is probably hydrogen. There is 0x66 (102) at position 0xBDC in the old version and 0x9B (155) in the new version of the file. This seems to be the oxygen.

Both files are available on the website for those who wants to inspect them (or experiment) more: [beginners.re](#).

### 9.3. MILLENIUM GAME SAVE FILE

Here is the new version of file opened in Hiew, we marked the values related to the resources mined in the game:

C:\tmp\2200save.i.v2	FRO	00000BDA
00000BDA: 16 00 9B 00-16 00 16 00-00 00 00 00-4C 01 E8 00	ы ў ў	Лвш
00000BEA: 00 00 F3 00-C7 00 00 00-00 00 00 00-00 00 00 00	€	
00000BFA: 10 28 70 18-10 28 10 28-00 00 00 00-F0 58 A8 A4	Б(рвв(в(	ЁХид
00000C0A: 00 00 B0 B8-90 68 00 00-00 00 00 00-00 00 00 00	Ph	
00000C1A: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00		

Figure 9.14: Hiew: state 1

Let's check each, and these are.

These are clearly 16-bit values: not a strange thing for 16-bit DOS software where the *int* type has 16-bit width.

### 9.3. MILLENIUM GAME SAVE FILE

Let's check our assumptions. We will write the 1234 (0x4D2) value at the first position (this must be hydrogen):

C:\tmp\2200save.i.v2 EDITMODE 00000BDC  
00000BDA: D2 04 9B 00-16 00 16 00-00 00 00 00-4C 01 E8 00  
00000BEA: 00 00 F3 00-C7 00 00 00-00 00 00 00-00 00 00 00  
00000BFA: 10 28 70 18-10 28 10 28-00 00 00 00-F0 58 A8 A4  
00000C0A: 00 00 B0 B8-90 68 00 00-00 00 00 00-00 00 00 00  
00000C1A: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00

Figure 9.15: Hiew: let's write 1234 (0x4D2) there

Then we will load the changed file in the game and took a look at mine statistics:



Figure 9.16: Let's check for hydrogen value

So yes, this is it.

### 9.3. MILLENIUM GAME SAVE FILE

Now let's try to finish the game as soon as possible, set the maximal values everywhere:

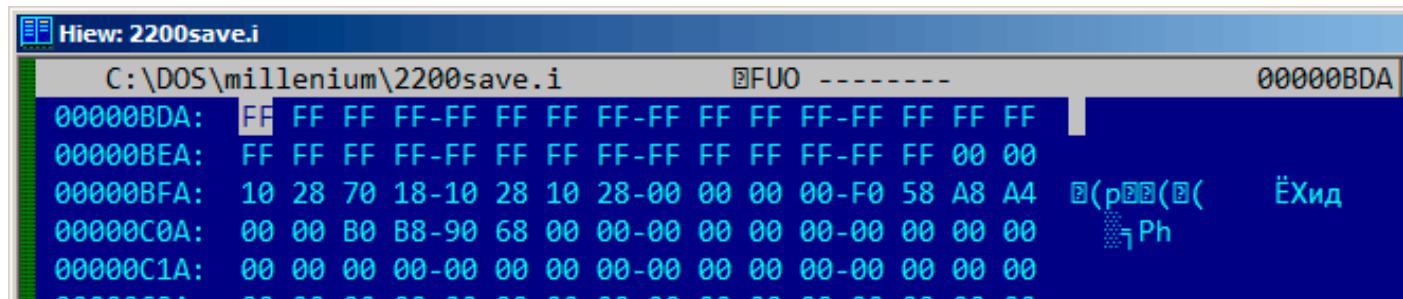


Figure 9.17: Hiew: let's set maximal values

0xFFFF is 65535, so yes, we now have a lot of resources:



Figure 9.18: All resources are 65535 (0xFFFF) indeed

#### 9.4. FORTUNE PROGRAM INDEXING FILE

Let's skip some "days" in the game and oops! We have a lower amount of some resources:



Figure 9.19: Resource variables overflow

That's just overflow.

The game's developer supposedly didn't think about such high amounts of resources, so there are probably no overflow checks, but the mine is "working" in the game, resources are added, hence the overflows. Apparently, it is a bad idea to be that greedy.

There are probably a lot of more values saved in this file.

So this is very simple method of cheating in games. High score files often can be easily patched like that. More about files and memory snapshots comparing: [5.11.6 on page 731](#).

## 9.4 Reverse engineering of simple *fortune* program indexing file

(This part was first appeared in my blog at 25-Apr-2015.)

*fortune* is well-known UNIX program which shows random phrase from a collection. Some geeks are often set up their system in such way, so *fortune* can be called after logging on. *fortune* takes phrases from the text files laying in `/usr/share/games/fortunes` (as of Ubuntu Linux). Here is example ("fortunes" text file):

```
A day for firm decisions!!!!!! Or is it?<%
A few hours grace before the madness begins again.<%
A gift of a flower will soon be made to you.<%
A long-forgotten loved one will appear soon.<%
Buy the negatives at any price.<%
```

#### 9.4. FORTUNE PROGRAM INDEXING FILE

A tall, dark stranger will have more fun than you.

%

...

So it is just phrases, sometimes multiline ones, divided by percent sign. The task of *fortune* program is to find random phrase and to print it. In order to achieve this, it must scan the whole text file, count phrases, choose random and print it. But the text file can get bigger, and even on modern computers, this naive algorithm is a bit uneconomical to computer resources. The straightforward way is to keep binary index file containing offset of each phrase in text file. With index file, *fortune* program can work much faster: just to choose random index element, take offset from there, set offset in text file and read phrase from it. This is actually done in *fortune* file. Let's inspect what is in its index file inside (these are .dat files in the same directory) in hexadecimal editor. This program is open-source of course, but intentionally, I will not peek into its source code.

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 01 af 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 00 2b
000020 00 00 00 60 00 00 00 8f 00 00 00 df 00 00 01 14
000030 00 00 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6
000040 00 00 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5
000050 00 00 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7
000060 00 00 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f
000070 00 00 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b
000080 00 00 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce
000090 00 00 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a
0000a0 00 00 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a
0000b0 00 00 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b
0000c0 00 00 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9
0000d0 00 00 09 27 00 00 09 43 00 00 09 79 00 00 09 a3
0000e0 00 00 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e
0000f0 00 00 0a 8a 00 00 0a a6 00 00 0a bf 00 00 0a ef
000100 00 00 0b 18 00 00 0b 43 00 00 0b 61 00 00 0b 8e
000110 00 00 0b cf 00 00 0b fa 00 00 0c 3b 00 00 0c 66
000120 00 00 0c 85 00 00 0c b9 00 00 0c d2 00 00 0d 02
000130 00 00 0d 3b 00 00 0d 67 00 00 0d ac 00 00 0d e0
000140 00 00 0e 1e 00 00 0e 67 00 00 0e a5 00 00 0e da
000150 00 00 0e ff 00 00 0f 43 00 00 0f 8a 00 00 0f bc
000160 00 00 0f e5 00 00 10 1e 00 00 10 63 00 00 10 9d
000170 00 00 10 e3 00 00 11 10 00 00 11 46 00 00 11 6c
000180 00 00 11 99 00 00 11 cb 00 00 11 f5 00 00 12 32
000190 00 00 12 61 00 00 12 8c 00 00 12 ca 00 00 13 87
0001a0 00 00 13 c4 00 00 13 fc 00 00 14 1a 00 00 14 6f
0001b0 00 00 14 ae 00 00 14 de 00 00 15 1b 00 00 15 55
0001c0 00 00 15 a6 00 00 15 d8 00 00 16 0f 00 00 16 4e
...
...
```

Without any special aid we could see that there are four 4-byte elements on each 16-byte line. Perhaps, it's our index array. I'm trying to load the whole file in Wolfram Mathematica as 32-bit integer array:

```
In[]:= BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32"]
Out[]={33554432, 2936078336, 3137339392, 251658240, 0, 37, 0, \
721420288, 1610612736, 2399141888, 3741319168, 335609856, 1208025088, \
2080440320, 2868969472, 3858825216, 537001984, 989986816, 2046951424, \
3305242624, 67305472, 1023606784, 1745027072, 2801991680, 3775070208, \
419692544, 755236864, 2130968576, 2902720512, 3573809152, 84213760, \
990183424, 1678049280, 2181365760, 2902786048, 3456434176, \
4144300032, 470155264, 1627783168, 2047213568, 3506831360, 168230912, \
1392967680, 2584150016, 4161208320, 654835712, 1493696512, \
2332557312, 2684878848, 3288858624, 3775397888, 4178051072, \
...
...
```

Nope, something wrong. Numbers are suspiciously big. But let's back to *od* output: each common 4-byte element has two zero bytes and two non-zero bytes, so the offsets (at least at the beginning of the file) are 16-bit at maximum. Probably different endianness is used in the file? Default endianness in Mathematica is little-endian, as used in Intel CPUs. Now I'm changing it to big-endian:

```
In[]:= BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32",
ByteOrdering -> 1]
```

## 9.4. FORTUNE PROGRAM INDEXING FILE

```
Out[] = {2, 431, 187, 15, 0, 620756992, 0, 43, 96, 143, 223, 276, \
328, 380, 427, 486, 544, 571, 634, 709, 772, 829, 872, 935, 993, \
1049, 1069, 1151, 1197, 1237, 1285, 1339, 1380, 1410, 1453, 1486, \
1527, 1564, 1633, 1658, 1745, 1802, 1875, 1946, 2040, 2087, 2137, \
2187, 2208, 2244, 2273, 2297, 2343, 2371, 2425, 2467, 2531, 2581, \
2637, 2654, 2698, 2726, 2751, 2799, 2840, 2883, 2913, 2958, 3023, \
3066, 3131, 3174, 3205, 3257, 3282, 3330, 3387, 3431, 3500, 3552, \
...}
```

Yes, this is something readable. I choose random element (3066) which is 0xBFA in hexadecimal form. I'm opening 'fortunes' text file in hex editor, I'm setting 0xBFA as offset and I see this phrase:

```
$ od -t x1 -c --skip-bytes=0xbfa --address-radix=x fortunes
000bfa 44 6f 20 77 68 61 74 20 63 6f 6d 65 73 20 6e 61
      D   o     w   h   a   t     c   o   m   e   s     n   a
000c0a 74 75 72 61 6c 6c 79 2e 20 20 53 65 65 74 68 65
      t   u   r   a   l   l   y   .     S   e   e   t   h   e
000c1a 20 61 6e 64 20 66 75 6d 65 20 61 6e 64 20 74 68
      a   n   d     f   u   m   e     a   n   d     t   h
....
```

Or:

```
Do what comes naturally. Seethe and fume and throw a tantrum.
%
```

Other offset are also can be checked, yes, they are valid offsets.

I can also check in Mathematica that each subsequent element is bigger than previous. I.e., the array is ascending. In mathematics lingo, this is called *strictly increasing monotonic function*.

```
In]:= Differences[input]
```

```
Out[] = {429, -244, -172, -15, 620756992, -620756992, 43, 53, 47, \
80, 53, 52, 52, 47, 59, 58, 27, 63, 75, 63, 57, 43, 63, 58, 56, 20, \
82, 46, 40, 48, 54, 41, 30, 43, 33, 41, 37, 69, 25, 87, 57, 73, 71, \
94, 47, 50, 50, 21, 36, 29, 24, 46, 28, 54, 42, 64, 50, 56, 17, 44, \
28, 25, 48, 41, 43, 30, 45, 65, 43, 65, 43, 31, 52, 25, 48, 57, 44, \
69, 52, 62, 73, 62, 53, 37, 68, 71, 50, 41, 57, 69, 58, 70, 45, 54, \
38, 45, 50, 42, 61, 47, 43, 62, 189, 61, 56, 30, 85, 63, 48, 61, 58, \
81, 50, 55, 63, 83, 80, 49, 42, 94, 54, 67, 81, 52, 57, 68, 43, 28, \
120, 64, 53, 81, 33, 82, 88, 29, 61, 32, 75, 63, 70, 47, 101, 60, 79, \
33, 48, 65, 35, 59, 47, 55, 22, 43, 35, 102, 53, 80, 65, 45, 31, 29, \
69, 32, 25, 38, 34, 35, 49, 59, 39, 41, 18, 43, 41, 83, 37, 31, 34, \
59, 72, 72, 81, 77, 53, 53, 50, 51, 45, 53, 39, 70, 54, 103, 33, 70, \
51, 95, 67, 54, 55, 65, 61, 54, 54, 53, 45, 100, 63, 48, 65, 71, 23, \
28, 43, 51, 61, 101, 65, 39, 78, 66, 43, 36, 56, 40, 67, 92, 65, 61, \
31, 45, 52, 94, 82, 82, 91, 46, 76, 55, 19, 58, 68, 41, 75, 30, 67, \
92, 54, 52, 108, 60, 56, 76, 41, 79, 54, 65, 74, 112, 76, 47, 53, 61, \
66, 53, 28, 41, 81, 75, 69, 89, 63, 60, 18, 18, 50, 79, 92, 37, 63, \
88, 52, 81, 60, 80, 26, 46, 80, 64, 78, 70, 75, 46, 91, 22, 63, 46, \
34, 81, 75, 59, 62, 66, 74, 76, 111, 55, 73, 40, 61, 55, 38, 56, 47, \
78, 81, 62, 37, 41, 60, 68, 40, 33, 54, 34, 41, 36, 49, 44, 68, 51, \
50, 52, 36, 53, 66, 46, 41, 45, 51, 44, 44, 33, 72, 40, 71, 57, 55, \
39, 66, 40, 56, 68, 43, 88, 78, 30, 54, 64, 36, 55, 35, 88, 45, 56, \
76, 61, 66, 29, 76, 53, 96, 36, 46, 54, 28, 51, 82, 53, 60, 77, 21, \
84, 53, 43, 104, 85, 50, 47, 39, 66, 78, 81, 94, 70, 49, 67, 61, 37, \
51, 91, 99, 58, 51, 49, 46, 68, 72, 40, 56, 63, 65, 41, 62, 47, 41, \
43, 30, 43, 67, 78, 80, 101, 61, 73, 70, 41, 82, 69, 45, 65, 38, 41, \
57, 82, 66}
```

As we can see, except of the very first 6 values (which is probably belongs to index file header), all numbers are in fact length of all text phrases (offset of the next phrase minus offset of the current phrase is in fact length of the current phrase).

It's very important to keep in mind that bit-endianness can be confused with incorrect array start. Indeed, from *od* output we see that each element started with two zeros. But when shifted by two bytes in either side, we can interpret this array as little-endian:

#### 9.4. FORTUNE PROGRAM INDEXING FILE

```
$ od -t x1 --address-radix=x --skip-bytes=0x32 fortunes.dat
000032 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6 00 00
000042 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5 00 00
000052 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7 00 00
000062 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f 00 00
000072 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b 00 00
000082 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce 00 00
000092 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a 00 00
0000a2 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a 00 00
0000b2 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b 00 00
0000c2 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9 00 00
0000d2 09 27 00 00 09 43 00 00 09 79 00 00 09 a3 00 00
0000e2 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e 00 00
...
...
```

If we would interpret this array as little-endian, the first element is 0x4801, second is 0x7C01, etc. High 8-bit part of each of these 16-bit values are seems random to us, and the lowest 8-bit part is seems ascending.

But I'm sure that this is big-endian array, because the very last 32-bit element of the file is big-endian (00 00 5f c4 here):

```
$ od -t x1 --address-radix=x fortunes.dat
...
000660 00 00 59 0d 00 00 59 55 00 00 59 7d 00 00 59 b5
000670 00 00 59 f4 00 00 5a 35 00 00 5a 5e 00 00 5a 9c
000680 00 00 5a cb 00 00 5a f4 00 00 5b 1f 00 00 5b 3d
000690 00 00 5b 68 00 00 5b ab 00 00 5b f9 00 00 5c 49
0006a0 00 00 5c ae 00 00 5c eb 00 00 5d 34 00 00 5d 7a
0006b0 00 00 5d a3 00 00 5d f5 00 00 5e 3a 00 00 5e 67
0006c0 00 00 5e a8 00 00 5e ce 00 00 5e f7 00 00 5f 30
0006d0 00 00 5f 82 00 00 5f c4
0006d8
```

Perhaps, *fortune* program developer had big-endian computer or maybe it was ported from something like it.

OK, so the array is big-endian, and, judging by common sense, the very first phrase in the text file must be started at zeroth offset. So zero value should be present in the array somewhere at the very beginning. We've got couple of zero elements at the beginning. But the second is most appealing: 43 is going right after it and 43 is valid offset to valid English phrase in the text file.

The last array element is 0x5FC4, and there are no such byte at this offset in the text file. So the last array element is pointing behind the end of file. It's supposedly done because phrase length is calculated as difference between offset to the current phrase and offset to the next phrase. This can be faster than traversing phrase string for percent character. But this wouldn't work for the last element. So the *dummy* element is also added at the end of array.

So the first 6 32-bit integer values are supposedly some kind of header.

Oh, I forgot to count phrases in text file:

```
$ cat fortunes | grep % | wc -l
432
```

The number of phrases can be present in index, but may be not. In case of very simple index files, number of elements can be easily deduced from index file size. Anyway, there are 432 phrases in the text file. And we see something very familiar at the second element (value 431). I've checked other files (*literature.dat* and *riddles.dat* in Ubuntu Linux) and yes, the second 32-bit element is indeed number of phrases minus 1. Why *minus 1*? Perhaps, this is not number of phrases, but rather index of the last phrase (starting at zero)?

And there are some other elements in the header. In Mathematica, I'm loading each of three available files and I'm taking a look on the header:

## 9.4. FORTUNE PROGRAM INDEXING FILE

```
In[14]:= input = BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32",
    ByteOrdering → 1];
]

In[18]:= BaseForm[Take[input, {1, 6}], 16]
Out[18]/BaseForm=
{216, 1af16, bb16, f16, 016, 2500000016}}
```

```
In[19]:= input = BinaryReadList["c:/tmp1/literature.dat", "UnsignedInteger32",
    ByteOrdering → 1];
]

In[20]:= BaseForm[Take[input, {1, 6}], 16]
Out[20]/BaseForm=
{216, 10616, 98316, 1a16, 016, 2500000016}}
```

```
In[21]:= input = BinaryReadList["c:/tmp1/riddles.dat", "UnsignedInteger32", ByteOrdering → 1];
]

In[22]:= BaseForm[Take[input, {1, 6}], 16]
Out[22]/BaseForm=
{216, 8016, 7f216, 2416, 016, 2500000016}}
```

I have no idea what other values mean, except the size of index file. Some fields are the same for all files, some are not. From my own experience, there could be:

- file signature;
- file version;
- checksum;
- some other flags;
- maybe even text language identifier;
- text file timestamp, so *fortune* program will regenerate index file if user added some new phrase(s) to it.

For example, Oracle .SYM files ([9.5 on the following page](#)) which contain symbols table for DLL files, also contain timestamp of corresponding DLL file, so to be sure it is still valid. But there are still possibility that the index file is regenerated if modification time of index file is older than text file. On the other hand, text file and index file timestamps can gone out of sync after archiving/unarchiving/installing/deploying/etc.

But there are no timestamp, in my opinion. The most compact way of representing date and time is UNIX time value, which is big 32-bit number. We don't see any of such here. Other ways are even less compact.

So here is algorithm, how *fortune* supposedly works:

- take number of last phrase from the second element;
- generate random number in range of 0..number\_of\_last\_phrase;
- find corresponding element in array of offsets, take also following offset;
- output to *stdout* all characters from the text file starting at current offset until the next offset minus 2 (so to ignore terminating percent sign and character of the following phrase).

### 9.4.1 Hacking

Let's try to check some of our assumptions. I will create this text file under the path and name */usr/share/games/fortunes/fortunes*:

```
Phrase one.
%
Phrase two.
%
```

## 9.5. ORACLE RDBMS: .SYM-FILES

Then this fortunes.dat file. I take header from the original fortunes.dat, I changed second field (count of all phrases) to zero and I left two elements in the array: 0 and 0x1c, because the whole length of the text fortunes file is 28 (0x1c) bytes:

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 00 00 00 bb 00 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Now I run it:

```
$ /usr/games/fortune
fortune: no fortune found
```

Something wrong. Let's change the second field to 1:

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Now it works. It's always shows only the first phrase:

```
$ /usr/games/fortune
Phrase one.
```

Hmmm. Let's leave only one element in array (0) without terminating one:

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 00
00001c
```

Fortune program always shows only first phrase.

From this experiment we got to know that percent sign is parsed inside of fortune file and the size is not calculated as I deduced before, perhaps, even terminal array element is not used. However, it still can be used. And probably it was used in past?

### 9.4.2 The files

For the sake of demonstration, I still didn't take a look in *fortune* source code. If you want to try to understand meaning of other values in index file header, you may try to achieve it without looking into source code as well. Files I took from Ubuntu Linux 14.04 are here: <http://beginners.re/examples/fortune/>, hacked files are also here.

Oh, and I took the files from x64 version of Ubuntu, but array elements are still has size of 32 bit. It is because *fortune* text files are probably never exceeds 4GiB!<sup>12</sup> size. But if it will, all elements must have size of 64 bit so to be able to store offset to the text file larger than 4GB.

For impatient readers, the source code of *fortune* is [here](#).

## 9.5 Oracle RDBMS: .SYM-files

When an Oracle RDBMS process experiences some kind of crash, it writes a lot of information into log files, including stack trace, like this:

```
----- Call Stack Trace -----
calling          call   entry           argument values in hex
location        type   point           (? means dubious value)
-----
_kqvrow()          00000000
_opifch2()+2729    CALLptr 00000000      23D4B914 E47F264 1F19AE2
_kpoal8()+2832    CALLrel _opifch2()     EB1C8A8 1
_opiopr()+1248    CALLreg 00000000      89 5 EB1CC74
                                         5E 1C EB1F0A0
```

<sup>12</sup>GiB!

## 9.5. ORACLE RDBMS: .SYM-FILES

_ttcpip() + 1051	CALLreg	00000000	5E 1C EB1F0A0 0
_opitsk() + 1404	CALL???	00000000	C96C040 5E EB1F0A0 0 EB1ED30
			EB1F1CC 53E52E 0 EB1F1F8
_opiino() + 980	CALLrel	_opitsk()	0 0
_opiodr() + 1248	CALLreg	00000000	3C 4 EB1FBF4
_opidrv() + 1201	CALLrel	_opiodr()	3C 4 EB1FBF4 0
_sou2o() + 55	CALLrel	_opidrv()	3C 4 EB1FBF4
_opimai_real() + 124	CALLrel	_sou2o()	EB1FC04 3C 4 EB1FBF4
_opimai() + 125	CALLrel	_opimai_real()	2 EB1FC2C
_OracleThreadStart@	CALLrel	_opimai()	2 EB1FF6C 7C88A7F4 EB1FC34 0
4() + 830			EB1FD04
77E6481C	CALLreg	00000000	E41FF9C 0 0 E41FF9C 0 EB1FFC4
00000000	CALL???	00000000	

But of course, Oracle RDBMS's executables must have some kind of debug information or map files with symbol information included or something like that.

Windows NT Oracle RDBMS has symbol information in files with .SYM extension, but the format is proprietary. (Plain text files are good, but needs additional parsing, hence offer slower access.)

Let's see if we can understand its format.

We will pick the shortest `orawtc8.sym` file that comes with the `orawtc8.dll` file in Oracle 8.1.7 <sup>13</sup>.

---

<sup>13</sup>We can chose an ancient Oracle RDBMS version intentionally due to the smaller size of its modules

## 9.5. ORACLE RDBMS: .SYM-FILES

Here is the file opened in Hiew:

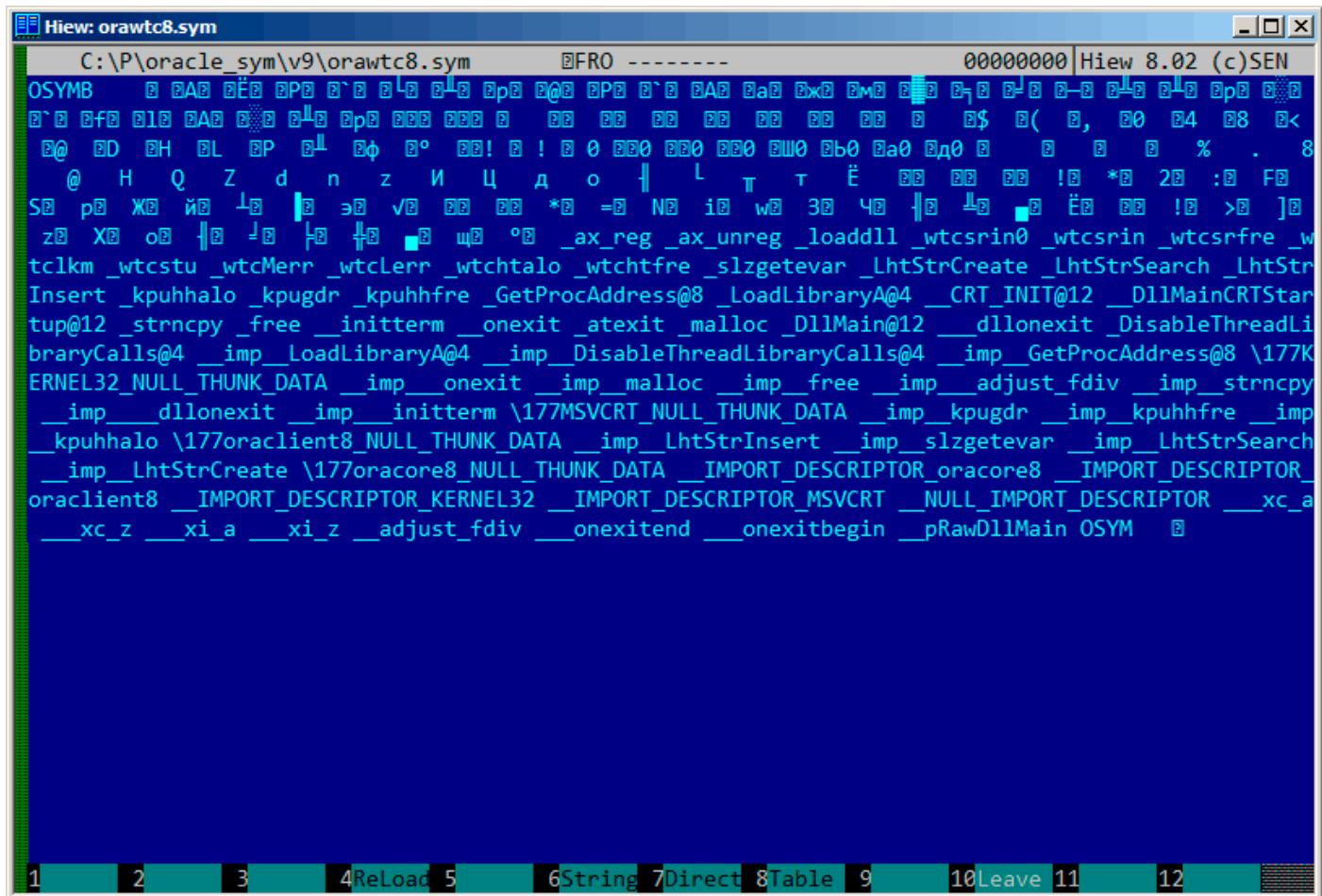


Figure 9.20: The whole file in Hiew

By comparing the file with other .SYM files, we can quickly see that `OSYM` is always header (and footer), so this is maybe the file's signature.

We also see that basically, the file format is: `OSYM` + some binary data + zero delimited text strings + `OSYM`. The strings are, obviously, function and global variable names.

### 9.5. ORACLE RDBMS: .SYM-FILES

We will mark the OSYM signatures and strings here:

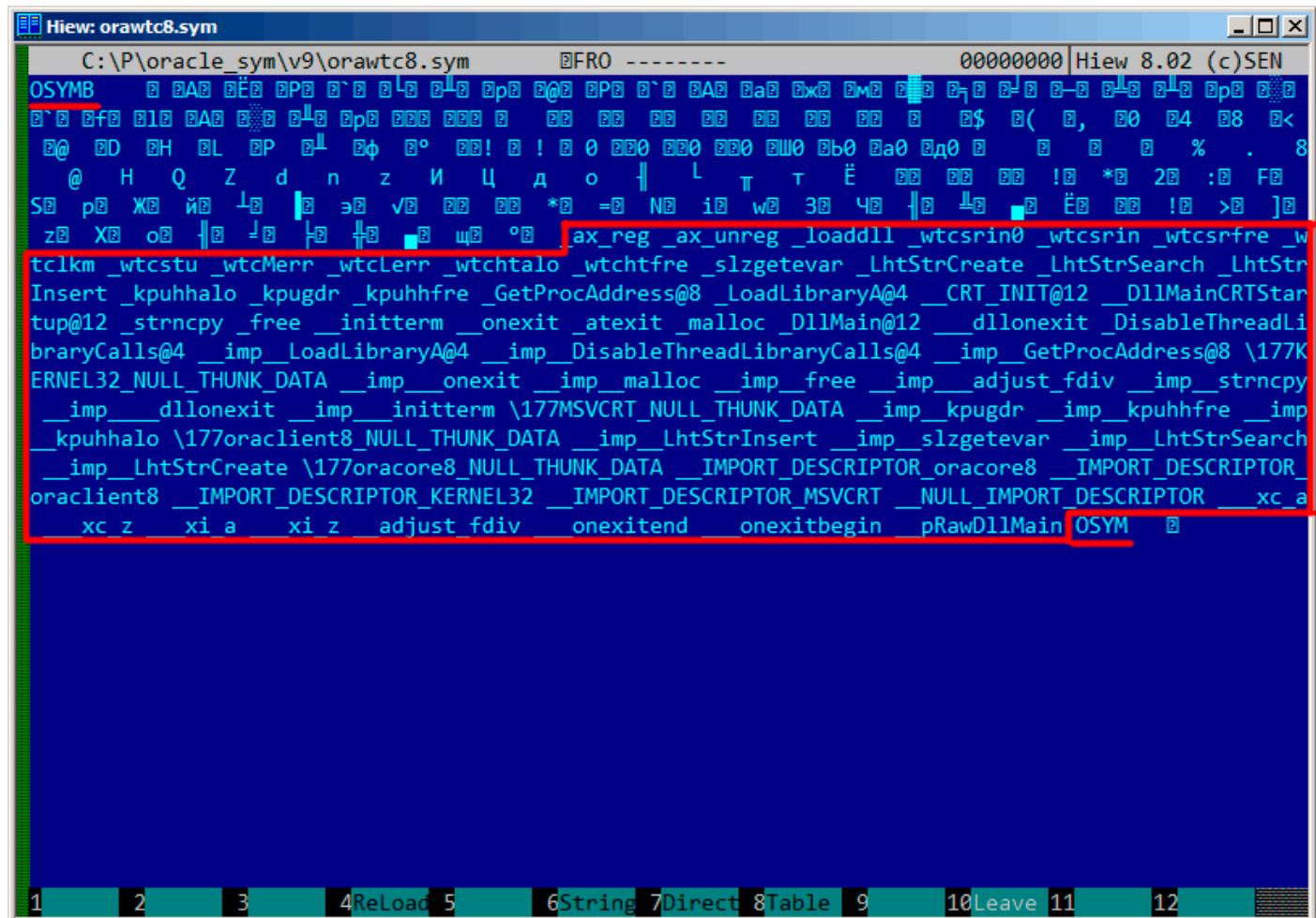


Figure 9.21: OSYM signature and text strings

Well, let's see. In Hiew, we will mark the whole strings block (except the trailing OSYM signatures) and put it into a separate file. Then we run UNIX *strings* and *wc* utilities to count the text strings:

```
strings strings_block | wc -l  
66
```

So there are 66 text strings. Please note that number.

We can say, in general, as a rule, the number of *anything* is often stored separately in binary files.

It's indeed so, we can find the 66 value (0x42) at the file's start, right after the OSYM signature:

```
$ hexdump -C orawtc8.sym
00000000  4f 53 59 4d 42 00 00 00  00 10 00 10 80 10 00 10  |OSYMB.....
00000010  f0 10 00 10 50 11 00 10  60 11 00 10 c0 11 00 10  |....P...
00000020  d0 11 00 10 70 13 00 10  40 15 00 10 50 15 00 10  |....p...@..P...
00000030  60 15 00 10 80 15 00 10  a0 15 00 10 a6 15 00 10  |`.....|
```

Of course, 0x42 here is not a byte, but most likely a 32-bit value packed as little-endian, hence we see 0x42 and then at least 3 zero bytes.

Why do we believe it's 32-bit? Because, Oracle RDBMS's symbol files may be pretty big.

The oracle.sym file for the main oracle.exe (version 10.2.0.4) executable contains 0x3A38E (238478) symbols. A 16-bit value isn't enough here.

We can check other .SYM files like this and it proves our guess: the value after the 32-bit OSYM signature always reflects the number of text strings in the file.

## 9.5. ORACLE RDBMS: .SYM-FILES

It's a general feature of almost all binary files: a header with a signature plus some other information about the file.

Now let's investigate closer what this binary block is.

Using Hiew again, we put the block starting at address 8 (i.e., after the 32-bit *count* value) ending at the strings block, into a separate binary file.

## 9.5. ORACLE RDBMS: .SYM-FILES

Let's see the binary block in Hiew:

Address	Hex Value	ASCII	Binary
00000000:	00 10 00 10-80 10 00 10-F0 10 00 10-50 11 00 10	Ø ÆØ ØÑØ ØPØ Ø	000000000
00000010:	60 11 00 10-C0 11 00 10-D0 11 00 10-70 13 00 10	` Ø ØLØ ØLØ ØpØ Ø	
00000020:	40 15 00 10-50 15 00 10-60 15 00 10-80 15 00 10	@ Ø ØPØ Ø`Ø ØAØ Ø	
00000030:	A0 15 00 10-A6 15 00 10-AC 15 00 10-B2 15 00 10	a Ø ØjØ ØMØ Ø`Ø Ø	
00000040:	B8 15 00 10-BE 15 00 10-C4 15 00 10-CA 15 00 10	í Ø ØJØ Ø-Ø Ø-Ø Ø	
00000050:	D0 15 00 10-E0 15 00 10-B0 16 00 10-60 17 00 10	l Ø ØpØ ØØØ Ø~Ø Ø	
00000060:	66 17 00 10-6C 17 00 10-80 17 00 10-B0 17 00 10	f Ø Ø1Ø ØAØ ØØØ Ø	
00000070:	D0 17 00 10-E0 17 00 10-10 18 00 10-16 18 00 10	l Ø ØpØ ØØØ ØØØ Ø	
00000080:	00 20 00 10-04 20 00 10-08 20 00 10-0C 20 00 10	Ø Ø Ø Ø Ø Ø Ø Ø	
00000090:	10 20 00 10-14 20 00 10-18 20 00 10-1C 20 00 10	Ø Ø Ø Ø Ø Ø Ø Ø	
000000A0:	20 20 00 10-24 20 00 10-28 20 00 10-2C 20 00 10	Ø\$ Ø( Ø, Ø	
000000B0:	30 20 00 10-34 20 00 10-38 20 00 10-3C 20 00 10	Ø Ø4 Ø8 Ø< Ø	
000000C0:	40 20 00 10-44 20 00 10-48 20 00 10-4C 20 00 10	@ ØD ØH ØL Ø	
000000D0:	50 20 00 10-D0 20 00 10-E4 20 00 10-F8 20 00 10	P ØL Øf Øº Ø	
000000E0:	0C 21 00 10-20 21 00 10-00 30 00 10-04 30 00 10	Ø! Ø ! Ø Ø ØØ Ø	
000000F0:	08 30 00 10-0C 30 00 10-98 30 00 10-9C 30 00 10	ØØ ØØØ ØØØ ØØØ Ø	
00000100:	A0 30 00 10-A4 30 00 10-00 00 00 00-08 00 00 00	a0 Ød0 Ø Ø Ø	
00000110:	12 00 00 00-1B 00 00 00-25 00 00 00-2E 00 00 00	Ø Ø % .	
00000120:	38 00 00 00-40 00 00 00-48 00 00 00-51 00 00 00	8 @ H Q	
00000130:	5A 00 00 00-64 00 00 00-6E 00 00 00-7A 00 00 00	Z d n z	
00000140:	88 00 00 00-96 00 00 00-A4 00 00 00-AE 00 00 00	И Ц Д о	
00000150:	B6 00 00 00-C0 00 00 00-D2 00 00 00-E2 00 00 00	l L T	
00000160:	F0 00 00 00-07 01 00 00-10 01 00 00-16 01 00 00	Ø Ø Ø Ø Ø	
00000170:	21 01 00 00-2A 01 00 00-32 01 00 00-3A 01 00 00	!Ø *Ø 2Ø :Ø	
00000180:	46 01 00 00-53 01 00 00-70 01 00 00-86 01 00 00	FØ SØ pØ ЖØ	
00000190:	A9 01 00 00-C1 01 00 00-DE 01 00 00-ED 01 00 00	йØ lØ  Ø эØ	
000001A0:	FB 01 00 00-07 02 00 00-1B 02 00 00-2A 02 00 00	Ø Ø Ø Ø * Ø</td <td></td>	

Figure 9.22: Binary block

There is a clear pattern in it.

## 9.5. ORACLE RDBMS: .SYM-FILES

We will add red lines to divide the block:

C:\P\oracle_sym\v9\asd2											EFRO	-----				00000000			
00000000:	00	10	00	10	80	10	00	10	F0	10	00	10	50	11	00	10	00	00	
00000010:	60	11	00	10	C0	11	00	10	D0	11	00	10	70	13	00	10	00	00	
00000020:	40	15	00	10	50	15	00	10	60	15	00	10	80	15	00	10	00	00	
00000030:	A0	15	00	10	A6	15	00	10	AC	15	00	10	B2	15	00	10	00	00	
00000040:	B8	15	00	10	BE	15	00	10	C4	15	00	10	CA	15	00	10	00	00	
00000050:	D0	15	00	10	E0	15	00	10	B0	16	00	10	60	17	00	10	00	00	
00000060:	66	17	00	10	6C	17	00	10	80	17	00	10	B0	17	00	10	00	00	
00000070:	D0	17	00	10	E0	17	00	10	10	18	00	10	16	18	00	10	00	00	
00000080:	00	20	00	10	04	20	00	10	08	20	00	10	0C	20	00	10	00	00	
00000090:	10	20	00	10	14	20	00	10	18	20	00	10	1C	20	00	10	00	00	
000000A0:	20	20	00	10	24	20	00	10	28	20	00	10	2C	20	00	10	00	00	
000000B0:	30	20	00	10	34	20	00	10	38	20	00	10	3C	20	00	10	00	00	
000000C0:	40	20	00	10	44	20	00	10	48	20	00	10	4C	20	00	10	00	00	
000000D0:	50	20	00	10	D0	20	00	10	E4	20	00	10	F8	20	00	10	Р	В	
000000E0:	0C	21	00	10	20	21	00	10	00	30	00	10	04	30	00	10	0!	0	
000000F0:	08	30	00	10	0C	30	00	10	98	30	00	10	9C	30	00	10	00	00	
00000100:	A0	30	00	10	A4	30	00	10	00	00	00	00	08	00	00	00	а0	0д0	
00000110:	12	00	00	00	1B	00	00	00	25	00	00	00	2E	00	00	00	0	0%	
00000120:	38	00	00	00	40	00	00	00	48	00	00	00	51	00	00	00	8	0Н	
00000130:	5A	00	00	00	64	00	00	00	6E	00	00	00	7A	00	00	00	Z	д	
00000140:	88	00	00	00	96	00	00	00	A4	00	00	00	AE	00	00	00	И	Ц	
00000150:	B6	00	00	00	C0	00	00	00	D2	00	00	00	E2	00	00	00	Л	Т	
00000160:	F0	00	00	00	07	01	00	00	10	01	00	00	16	01	00	00	Ё	00	
00000170:	21	01	00	00	2A	01	00	00	32	01	00	00	3A	01	00	00	!0	*0	
00000180:	46	01	00	00	53	01	00	00	70	01	00	00	86	01	00	00	FB	50	
00000190:	A9	01	00	00	C1	01	00	00	DE	01	00	00	ED	01	00	00	й0	Л	
000001A0:	FB	01	00	00	07	02	00	00	1B	02	00	00	2A	02	00	00	в0	00	
000001B0:	3D	02	00	00	4E	02	00	00	69	02	00	00	77	02	00	00	=0	Н0	

Figure 9.23: Binary block patterns

Hiew, like almost any other hexadecimal editor, shows 16 bytes per line. So the pattern is clearly visible: there are 4 32-bit values per line.

The pattern is visually visible because some values here (till address `0x104`) are always in `0x1000xxxx` form, started with `0x10` and zero bytes.

Other values (starting at 0x108) are in 0x0000xxxx form, so always started with two zero bytes.

Let's dump the block as an array of 32-bit values:

Listing 9.9: first column is address

```
$ od -v -t x4 binary_block
00000000 10001000 10001080 100010f0 10001150
00000020 10001160 100011c0 100011d0 10001370
00000040 10001540 10001550 10001560 10001580
00000060 100015a0 100015a6 100015ac 100015b2
00000100 100015b8 100015be 100015c4 100015ca
00000120 100015d0 100015e0 100016b0 10001760
00000140 10001766 1000176c 10001780 100017b0
00000160 100017d0 100017e0 10001810 10001816
00000200 10002000 10002004 10002008 1000200c
00000220 10002010 10002014 10002018 1000201c
```

## 9.5. ORACLE RDBMS: .SYM-FILES

```
0000240 10002020 10002024 10002028 1000202c
0000260 10002030 10002034 10002038 1000203c
0000300 10002040 10002044 10002048 1000204c
0000320 10002050 100020d0 100020e4 100020f8
0000340 1000210c 10002120 10003000 10003004
0000360 10003008 1000300c 10003098 1000309c
0000400 100030a0 100030a4 00000000 00000008
0000420 00000012 0000001b 00000025 0000002e
0000440 00000038 00000040 00000048 00000051
0000460 0000005a 00000064 0000006e 0000007a
0000500 00000088 00000096 000000a4 000000ae
0000520 000000b6 000000c0 000000d2 000000e2
0000540 000000f0 00000107 00000110 00000116
0000560 00000121 0000012a 00000132 0000013a
0000600 00000146 00000153 00000170 00000186
0000620 000001a9 000001c1 000001de 000001ed
0000640 000001fb 00000207 0000021b 0000022a
0000660 0000023d 0000024e 00000269 00000277
0000700 00000287 00000297 000002b6 000002ca
0000720 000002dc 000002f0 00000304 00000321
0000740 0000033e 0000035d 0000037a 00000395
0000760 000003ae 000003b6 000003be 000003c6
0001000 000003ce 000003dc 000003e9 000003f8
0001020
```

There are 132 values, that's  $66 \times 2$ . Probably, there are two 32-bit values for each symbol, but maybe there are two arrays? Let's see.

Values starting with `0x1000` may be addresses.

This is a .SYM file for a DLL after all, and the default base address of win32 DLLs is `0x10000000`, and the code usually starts at `0x10001000`.

When we open the orawtc8.dll file in [IDA](#), the base address is different, but nevertheless, the first function is:

```
.text:60351000 sub_60351000    proc near
.text:60351000
.text:60351000 arg_0     = dword ptr  8
.text:60351000 arg_4     = dword ptr  0Ch
.text:60351000 arg_8     = dword ptr  10h
.text:60351000
.text:60351000         push   ebp
.text:60351001         mov    ebp, esp
.text:60351003         mov    eax, dword_60353014
.text:60351008         cmp    eax, 0FFFFFFFh
.text:6035100B         jnz    short loc_6035104F
.text:6035100D         mov    ecx, hModule
.text:60351013         xor    eax, eax
.text:60351015         cmp    ecx, 0FFFFFFFh
.text:60351018         mov    dword_60353014, eax
.text:6035101D         jnz    short loc_60351031
.text:6035101F         call   sub_603510F0
.text:60351024         mov    ecx, eax
.text:60351026         mov    eax, dword_60353014
.text:6035102B         mov    hModule, ecx
.text:60351031
.text:60351031 loc_60351031: ; CODE XREF: sub_60351000+1D
.text:60351031         test   ecx, ecx
.text:60351033         jbe    short loc_6035104F
.text:60351035         push   offset ProcName ; "ax_reg"
.text:6035103A         push   ecx           ; hModule
.text:6035103B         call   ds:GetProcAddress
...
```

Wow, "ax\_reg" string sounds familiar.

It's indeed the first string in the strings block! So the name of this function seems to be "ax\_reg".

The second function is:

## 9.5. ORACLE RDBMS: .SYM-FILES

```
.text:60351080 sub_60351080    proc near
.text:60351080
.text:60351080 arg_0      = dword ptr  8
.text:60351080 arg_4      = dword ptr  0Ch
.text:60351080
.text:60351080          push   ebp
.text:60351081          mov    ebp, esp
.text:60351083          mov    eax, dword_60353018
.text:60351088          cmp    eax, 0FFFFFFFh
.text:6035108B          jnz    short loc_603510CF
.text:6035108D          mov    ecx, hModule
.text:60351093          xor    eax, eax
.text:60351095          cmp    ecx, 0FFFFFFFh
.text:60351098          mov    dword_60353018, eax
.text:6035109D          jnz    short loc_603510B1
.text:6035109F          call   sub_603510F0
.text:603510A4          mov    ecx, eax
.text:603510A6          mov    eax, dword_60353018
.text:603510AB          mov    hModule, ecx
.text:603510B1
.text:603510B1 loc_603510B1: ; CODE XREF: sub_60351080+1D
.text:603510B1          test   ecx, ecx
.text:603510B3          jbe   short loc_603510CF
.text:603510B5          push   offset aAx_unreg ; "ax_unreg"
.text:603510BA          push   ecx           ; hModule
.text:603510BB          call   ds:GetProcAddress
...
...
```

The “ax\_unreg” string is also the second string in the strings block!

The starting address of the second function is **0x60351080**, and the second value in the binary block is **10001080**. So this is the address, but for a DLL with the default base address.

We can quickly check and be sure that the first 66 values in the array (i.e., the first half of the array) are just function addresses in the DLL, including some labels, etc. Well, what's the other part of array then? The other 66 values that start with **0x0000**? These seem to be in range **[0...0x3F8]**. And they do not look like bitfields: the series of numbers is increasing.

The last hexadecimal digit seems to be random, so, it's unlikely the address of something (it would be divisible by 4 or maybe 8 or 0x10 otherwise).

Let's ask ourselves: what else Oracle RDBMS's developers would save here, in this file?

Quick wild guess: it could be the address of the text string (function name).

It can be quickly checked, and yes, each number is just the position of the first character in the strings block.

This is it! All done.

We will write an utility to convert these .SYM files into **IDA** script, so we can load the .idc script and it sets the function names:

```
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>

int main (int argc, char *argv[])
{
    uint32_t sig, cnt, offset;
    uint32_t *d1, *d2;
    int     h, i, remain, file_len;
    char   *d3;
    uint32_t array_size_in_bytes;

    assert (argc[1]); // file name
    assert (argc[2]); // additional offset (if needed)
```

```

// additional offset
assert (sscanf (argv[2], "%X", &offset)==1);

// get file length
assert ((h=open (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1);
assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
assert (lseek (h, 0, SEEK_SET)!=-1);

// read signature
assert (read (h, &sig, 4)==4);
// read count
assert (read (h, &cnt, 4)==4);

assert (sig==0x4D59534F); // OSYM

// skip timestamp (for 11g)
//_lseek (h, 4, 1);

array_size_in_bytes=cnt*sizeof(uint32_t);

// load symbol addresses array
d1=(uint32_t*)malloc (array_size_in_bytes);
assert (d1);
assert (read (h, d1, array_size_in_bytes)==array_size_in_bytes);

// load string offsets array
d2=(uint32_t*)malloc (array_size_in_bytes);
assert (d2);
assert (read (h, d2, array_size_in_bytes)==array_size_in_bytes);

// calculate strings block size
remain=file_len-(8+4)-(cnt*8);

// load strings block
assert (d3=(char*)malloc (remain));
assert (read (h, d3, remain)==remain);

printf ("#include <idc.idc>\n\n");
printf ("static main() {\n");

for (i=0; i<cnt; i++)
    printf ("\tMakeName(0x%08X, \"%s\");\n", offset + d1[i], &d3[d2[i]]);

printf ("}\n");

close (h);
free (d1); free (d2); free (d3);
};

}

```

Here is an example of its work:

```

#include <idc.idc>

static main() {
    MakeName(0x60351000, "_ax_reg");
    MakeName(0x60351080, "_ax_unreg");
    MakeName(0x603510F0, "_loaddll");
    MakeName(0x60351150, "_wtcsrin0");
    MakeName(0x60351160, "_wtcsrin");
    MakeName(0x603511C0, "_wtcsrfre");
    MakeName(0x603511D0, "_wtclkm");
    MakeName(0x60351370, "_wtcstu");
...
}

```

The example files were used in this example are here: [beginners.re](#).

## 9.6. ORACLE RDBMS: .MSB-FILES

Oh, let's also try Oracle RDBMS for win64. There has to be 64-bit addresses instead, right?

The 8-byte pattern is visible even easier here:

	FRO	SYMM64	m@
00000000:	4F 53 59 4D-41 4D 36 34-BD 6D 05 00-00 00 00 00 00	OSYMM64	m@
00000010:	CD 21 2A 47-00 00 00 00-00 00 00 00-00 00 00 00 00	=!*G	
00000020:	00 00 00 00-00 00 00 00-00 00 40 00-00 00 00 00 00		@
00000030:	00 10 40 00-00 00 00 00-6C 10 40 00-00 00 00 00 00	Б@	1Б@
00000040:	04 11 40 00-00 00 00 00-80 13 40 00-00 00 00 00 00	ВВ@	AB@
00000050:	E3 13 40 00-00 00 00 00-01 14 40 00-00 00 00 00 00	yВ@	ВВ@
00000060:	1F 14 40 00-00 00 00 00-3E 14 40 00-00 00 00 00 00	ВВ@	>Б@
00000070:	54 14 40 00-00 00 00 00-1E 18 40 00-00 00 00 00 00	ТВ@	ВВ@
00000080:	97 1B 40 00-00 00 00 00-C1 1B 40 00-00 00 00 00 00	ЧВ@	ЛБ@
00000090:	0A 1C 40 00-00 00 00 00-4C 1C 40 00-00 00 00 00 00	ВВ@	ЛБ@
000000A0:	7A 1C 40 00-00 00 00 00-98 1C 40 00-00 00 00 00 00	zВ@	ШВ@
000000B0:	E7 25 40 00-00 00 00 00-11 26 40 00-00 00 00 00 00	Ч%@	В&@
000000C0:	80 26 40 00-00 00 00 00-C4 26 40 00-00 00 00 00 00	A&@	-&@
000000D0:	F4 26 40 00-00 00 00 00-24 27 40 00-00 00 00 00 00	Ї&@	\$'@
000000E0:	50 27 40 00-00 00 00 00-78 27 40 00-00 00 00 00 00	P'@	x'@
000000F0:	A0 27 40 00-00 00 00 00-4E 28 40 00-00 00 00 00 00	a'@	N(@
00000100:	26 29 40 00-00 00 00 00-B4 2C 40 00-00 00 00 00 00	&)@	Л,@
00000110:	66 2D 40 00-00 00 00 00-A6 2D 40 00-00 00 00 00 00	f-@	ж-@
00000120:	30 2E 40 00-00 00 00 00-BA 2E 40 00-00 00 00 00 00	0.@	.@
00000130:	F2 30 40 00-00 00 00 00-84 31 40 00-00 00 00 00 00	€0@	Д1@
00000140:	F0 31 40 00-00 00 00 00-5E 32 40 00-00 00 00 00 00	Ё1@	^2@
00000150:	CC 32 40 00-00 00 00 00-3A 33 40 00-00 00 00 00 00	¶2@	:3@
00000160:	A8 33 40 00-00 00 00 00-16 34 40 00-00 00 00 00 00	и3@	Б4@
00000170:	84 34 40 00-00 00 00 00-F2 34 40 00-00 00 00 00 00	Д4@	€4@
00000180:	60 35 40 00-00 00 00 00-CC 35 40 00-00 00 00 00 00	~5@	Л5@
00000190:	3A 36 40 00-00 00 00 00-A8 36 40 00-00 00 00 00 00	:6@	и6@
000001A0:	16 37 40 00-00 00 00 00-84 37 40 00-00 00 00 00 00	В7@	Д7@

Figure 9.24: .SYM-file example from Oracle RDBMS for win64

So yes, all tables now have 64-bit elements, even string offsets!

The signature is now `OSYMM64`, to distinguish the target platform, apparently.

This is it!

Here is also library which has functions to access Oracle RDBMS.SYM-files: [GitHub](#).

## 9.6 Oracle RDBMS: .MSB-files

When working toward the solution of a problem, it always helps if you know the answer.

Murphy's Laws, Rule of Accuracy

This is a binary file that contains error messages with their corresponding numbers. Let's try to understand its format and find a way to unpack it.

## 9.6. ORACLE RDBMS: .MSB-FILES

There are Oracle RDBMS error message files in text form, so we can compare the text and packed binary files <sup>14</sup>.

This is the beginning of the ORAUS.MSG text file with some irrelevant comments stripped:

Listing 9.10: Beginning of ORAUS.MSG file without comments

```
00000, 00000, "normal, successful completion"
00001, 00000, "unique constraint (%s.%s) violated"
00017, 00000, "session requested to set trace event"
00018, 00000, "maximum number of sessions exceeded"
00019, 00000, "maximum number of session licenses exceeded"
00020, 00000, "maximum number of processes (%s) exceeded"
00021, 00000, "session attached to some other process; cannot switch session"
00022, 00000, "invalid session ID; access denied"
00023, 00000, "session references process private memory; cannot detach session"
00024, 00000, "logins from more than one process not allowed in single-process mode"
00025, 00000, "failed to allocate %s"
00026, 00000, "missing or invalid session ID"
00027, 00000, "cannot kill current session"
00028, 00000, "your session has been killed"
00029, 00000, "session is not a user session"
00030, 00000, "User session ID does not exist."
00031, 00000, "session marked for kill"
...
...
```

The first number is the error code. The second is perhaps maybe some additional flags.

<sup>14</sup>Open-source text files don't exist in Oracle RDBMS for every .MSB file, so that's why we will work on their file format

## 9.6. ORACLE RDBMS: .MSB-FILES

Now let's open the ORAUS.MSB binary file and find these text strings. And there are:

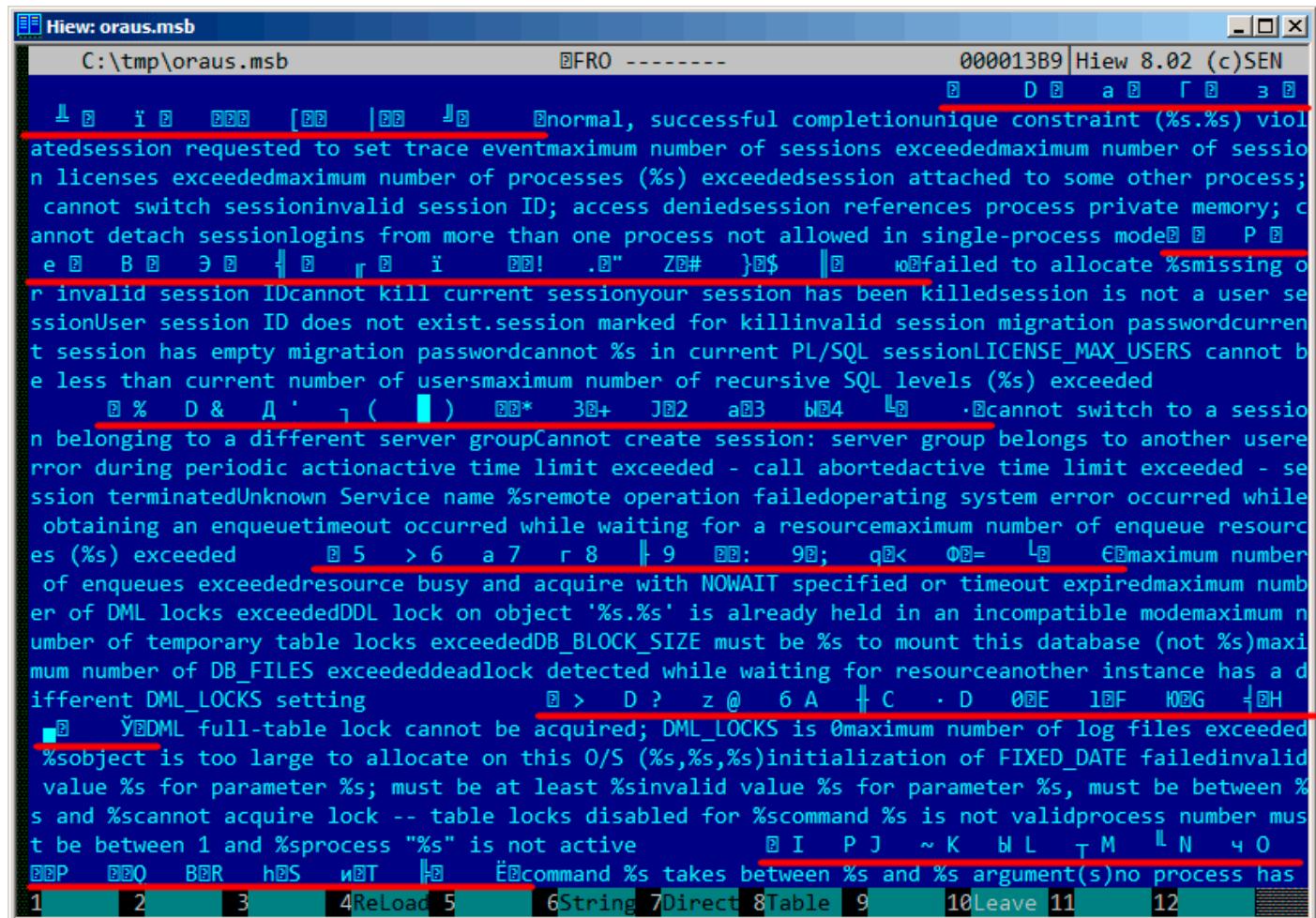


Figure 9.25: Hiew: first block

We see the text strings (including those from the beginning of the ORAUS.MSG file) interleaved with some binary values. By quick investigation, we can see that main part of the binary file is divided by blocks of size 0x200 (512) bytes.

## 9.6. ORACLE RDBMS: .MSB-FILES

Let's see the contents of the first block:

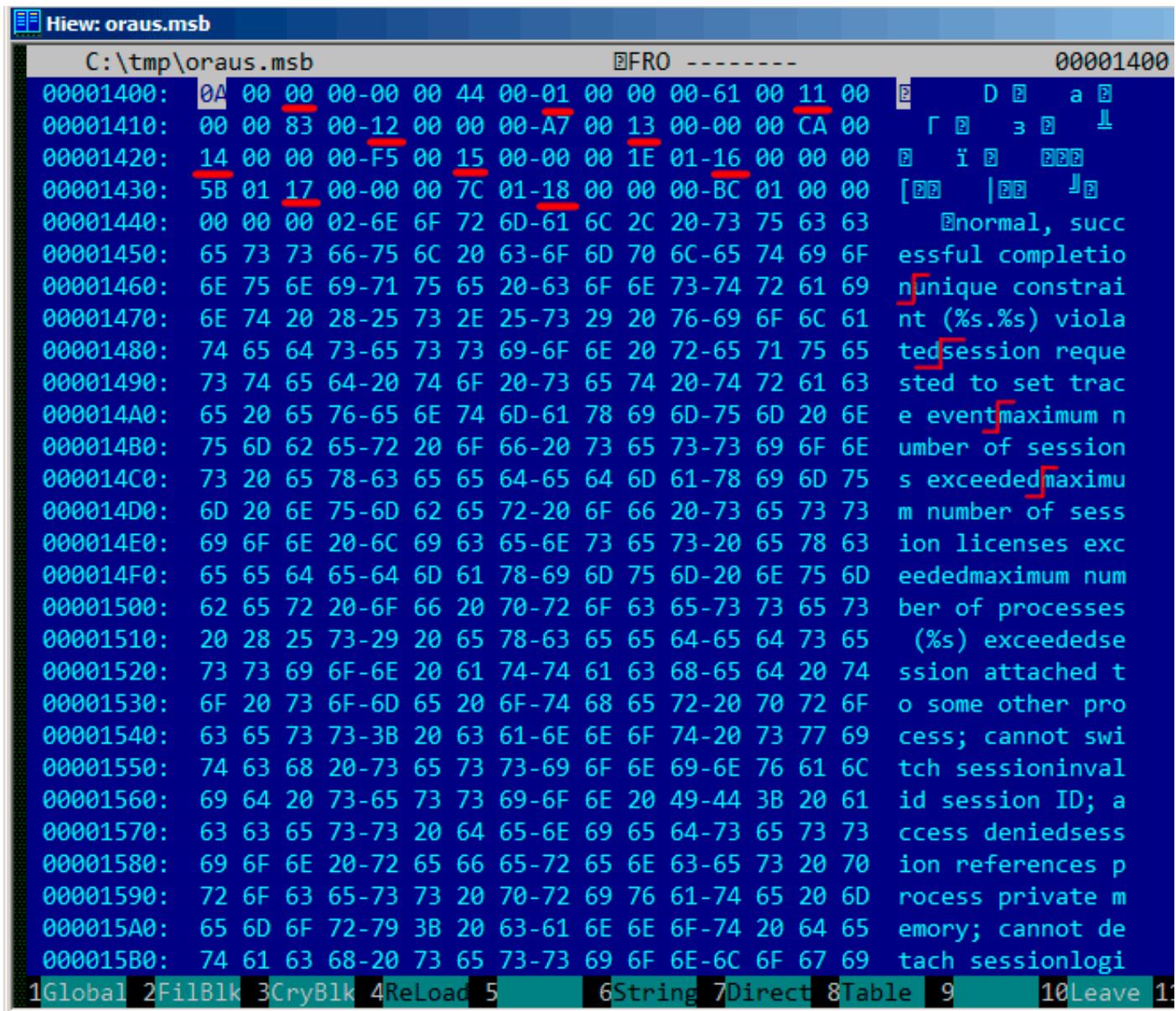


Figure 9.26: Hiew: first block

Here we see the texts of the first messages errors. What we also see is that there are no zero bytes between the error messages. This implies that these are not null-terminated C strings. As a consequence, the length of each error message must be encoded somehow. Let's also try to find the error numbers. The ORAUS.MSG files starts with these: 0, 1, 17 (0x11), 18 (0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)... We will find these numbers at the beginning of the block and mark them with red lines. The period between error codes is 6 bytes.

This implies that there are probably 6 bytes of information allocated for each error message.

The first 16-bit value (0xA here or 10) means the number of messages in each block: this can be checked by investigating other blocks. Indeed: the error messages have arbitrary size. Some are longer, some are shorter. But block size is always fixed, hence, you never know how many text messages can be packed in each block.

As we already noted, since these are not null-terminated C strings, their size must be encoded somewhere. The size of the first string “normal, successful completion” is 29 (0x1D) bytes. The size of the second string “unique constraint (%s.%s) violated” is 34 (0x22) bytes. We can’t find these values (0x1D or/and 0x22) in the block.

There is also another thing. Oracle RDBMS has to determine the position of the string it needs to load in the block, right? The first string “normal, successful completion” starts at position 0x1444 (if we count starting at the beginning of the file) or at 0x44 (from the block’s start). The second string “unique constraint

## 9.6. ORACLE RDBMS: .MSB-FILES

(%s.%s) violated" starts at position 0x1461 (from the file's start) or at 0x61 (from the at the block's start). These numbers (0x44 and 0x61) are familiar somehow! We can clearly see them at the start of the block.

So, each 6-byte block is:

- 16-bit error number;
- 16-bit zero (maybe additional flags);
- 16-bit starting position of the text string within the current block.

We can quickly check the other values and be sure our guess is correct. And there is also the last "dummy" 6-byte block with an error number of zero and starting position beyond the last error message's last character. Probably that's how text message length is determined? We just enumerate 6-byte blocks to find the error number we need, then we get the text string's position, then we get the position of the text string by looking at the next 6-byte block! This way we determine the string's boundaries! This method allows to save some space by not saving the text string's size in the file!

It's not possible to say it saves a lot of space, but it's a clever trick.

## 9.6. ORACLE RDBMS: .MSB-FILES

Let's back to the header of .MSB-file:

		FRO	-----	00000000
00000000:	15 13 22 01-13 03 09 09-00 00 00 00-00 00 00 00			ББ"БББББ
00000010:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000020:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000030:	00 00 00 00-02 00 00 00-01 00 00 00-01 00 00 00			Б Б Б
00000040:	08 00 00 00-84 07 00 00-00 00 00 00-9C EA 00 00			Б ДБ бъ
00000050:	60 40 00 00-14 4D 0C 00-12 04 02 0D-13 00 00 00			^@ БМВ БББ
00000060:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000070:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000080:	00 00 00 00-00 00 00 00-00 00 0F 12-01 00 00 00			БББ
00000090:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000A0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000B0:	00 00 00 00-00 00 00 00-00 00 00 00-15 13 00 00			ББ
000000C0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000D0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000E0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000F0:	03 00 00 00-01 00 00 00-01 00 00 00-01 00 00 00			Б Б Б Б
00000100:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000110:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000120:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000130:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000140:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000150:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000160:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000170:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000180:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000190:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000001A0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000001B0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
1Global 2FilBlk 3CryBlk 4ReLoad 5String 6Direct 7Table 8Table 9Leave 10Leave 11				

Figure 9.27: Hiew: file header

Now we can quickly find the number of blocks in the file (marked by red). We can checked other .MSB-files and we see that it's true for all of them.

There are a lot of other values, but we will not investigate them, since our job (an unpacking utility) is done.

If we have to write a .MSB file packer, we would probably have to understand the meaning of the other values.

## 9.6. ORACLE RDBMS: .MSB-FILES

There is also a table that came after the header which probably contains 16-bit values:

Hiew: oraus.msb												00000800
	FRO											00000800
00000800:	83	34	8F	34	9B	34	AA	34	BE	34	C7	34
00000810:	E3	34	EB	34	24	35	2C	35	32	35	39	35
00000820:	4E	35	56	35	5D	35	84	35	8A	35	8F	35
00000830:	C6	35	CE	35	D8	35	E4	35	04	36	0F	36
00000840:	2C	36	52	36	5B	36	94	36	A2	36	B4	36
00000850:	CE	36	D7	36	DF	36	E7	36	ED	36	F5	36
00000860:	0C	37	13	37	1A	37	21	37	29	37	31	37
00000870:	4E	37	55	37	5E	37	68	37	6E	37	75	37
00000880:	A2	37	AF	37	B7	37	BD	37	C5	37	CC	37
00000890:	E0	37	E8	37	F2	37	F9	37	45	38	73	38
000008A0:	B1	38	B7	38	BC	38	C6	38	0A	39	0F	39
000008B0:	23	39	29	39	2F	39	35	39	3E	39	46	39
000008C0:	AE	39	9A	3A	A5	3A	B1	3A	BC	3A	C7	3A
000008D0:	E5	3A	F4	3A	00	3B	0B	3B	15	3B	2E	3B
000008E0:	51	3B	5E	3B	68	3B	74	3B	84	3B	8E	3B
000008F0:	65	3C	6E	3C	77	3C	8F	3C	96	3C	C0	3C
00000900:	F5	3C	53	3D	88	3E	90	3E	96	3E	9E	3E
00000910:	BA	3E	C4	3E	CF	3E	D9	3E	E1	3E	EA	3E
00000920:	07	3F	12	3F	1B	3F	23	3F	2B	3F	34	3F
00000930:	4D	3F	56	3F	61	3F	6C	3F	78	3F	80	3F
00000940:	99	3F	16	40	1F	40	26	40	2F	40	80	40
00000950:	AA	40	B6	40	C0	40	CA	40	D4	40	DC	40
00000960:	FA	40	02	41	0B	41	15	41	1D	41	44	41
00000970:	5F	41	66	41	6E	41	7B	41	86	41	8D	41
00000980:	A7	41	AF	41	B7	41	BD	41	3B	42	60	44
00000990:	DD	44	55	46	5E	46	42	4A	4E	4A	56	4A
000009A0:	AA	4A	B3	4A	B7	4A	BB	4A	BD	4A	BF	4A
000009B0:	C6	4A	CA	4A	CD	4A	D1	4A	DA	4A	E0	4A

Figure 9.28: Hiew: last\_errnos table

Their size can be determined visually (red lines are drawn here).

While dumping these values, we have found that each 16-bit number is the last error code for each block.

So that's how Oracle RDBMS quickly finds the error message:

- load a table we will call `last_errnos` (that contains the last error number for each block);
- find a block that contains the error code we need, assuming all error codes increase across each block and across the file as well;
- load the specific block;
- enumerate the 6-byte structures until the specific error number is found;
- get the position of the first character from the current 6-byte block;
- get the position of the last character from the next 6-byte block;
- load all characters of the message in this range.

This is C program that we wrote which unpacks .MSB-files: [beginners.re](#).

There are also the two files which were used in the example (Oracle RDBMS 11.1.0.6): [beginners.re](#), [beginners.ms](#).

## 9.6.1 Summary

The method is probably too old-school for modern computers. Supposedly, this file format was developed in the mid-80's by someone who also coded for *big iron* with memory/disk space economy in mind. Nevertheless, it has been an interesting and yet easy task to understand a proprietary file format without looking into Oracle RDBMS's code.

## 9.7 Exercise

Try to reverse engineer of any binary files of your favorite game, including high-score files, resources, etc. There are also binary files with known structure: utmp/wtmp files.

The EXIF header in JPEG file is documented, but you can try to understand its structure without help, just shoot photos at various date/time, places, and try to find date/time and GPS location in EXIF. Try to patch GPS location, upload JPEG file to Facebook and see, how it will put your picture on the map.

Try to patch any information in MP3 file and see how your favorite MP3-player will react.

## 9.8 Further reading

[Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.](#)

# Chapter 10

## Other things

### 10.1 Executable files patching

#### 10.1.1 Text strings

The C strings are the thing that is the easiest to patch (unless they are encrypted) in any hex editor. This technique is available even for those who are not aware of machine code and executable file formats. The new string has not to be bigger than the old one, because there's a risk of overwriting another value or code there.

Using this method, a lot of software was *localized* in the MS-DOS era, at least in the ex-USSR countries in 80's and 90's. It was the reason why some weird abbreviations were present in the *localized* software: there was no room for longer strings.

As for Delphi strings, the string's size must also be corrected, if needed.

#### 10.1.2 x86 code

Frequent patching tasks are:

- One of the most frequent jobs is to disable some instruction. It is often done by filling it using byte `0x90` (`NOP`).
- Conditional jumps, which have an opcode like `74 xx` (`JZ`), can be filled with two `NOPs`.  
It is also possible to disable a conditional jump by writing 0 at the second byte (*jump offset*).
- Another frequent job is to make a conditional jump to always trigger: this can be done by writing `0xEB` instead of the opcode, which stands for `JMP`.
- A function's execution can be disabled by writing `RETN` (0xC3) at its beginning. This is true for all functions excluding `stdcall` ([6.1.2 on page 733](#)). While patching `stdcall` functions, one has to determine the number of arguments (for example, by finding `RETN` in this function), and use `RETN` with a 16-bit argument (0xC2).
- Sometimes, a disabled function has to return 0 or 1. This can be done by `MOV EAX, 0` or `MOV EAX, 1`, but it's slightly verbose.  
A better way is `XOR EAX, EAX` (2 bytes `0x31 0xC0`) or `XOR EAX, EAX / INC EAX` (3 bytes `0x31 0xC0 0x41`).

A software may be protected against modifications.

This protection is often done by reading the executable code and calculating a checksum. Therefore, the code must be read before protection is triggered.

This can be determined by setting a breakpoint on reading memory.

`tracer` has the BPM option for this.

PE executable file relocs ([6.5.2 on page 759](#)) must not to be touched while patching, because the Windows loader may overwrite your new code. (They are grayed in Hiew, for example: [fig.1.21](#)).

## 10.2. FUNCTION ARGUMENTS STATISTICS

As a last resort, it is possible to write jumps that circumvent the relocations, or you will have to edit the relocations table.

## **10.2 Function arguments statistics**

I've always been interesting in what is average number of function arguments.

Just analyzed many Windows 7 32-bit DLLs

(crypt32.dll, mfc71.dll, msrv100.dll, shell32.dll, user32.dll, d3d11.dll, mshtml.dll, msxml6.dll, sqlncli11.dll, wininet.dll, mfc120.dll, msvbvm60.dll, ole32.dll, themeui.dll, wmp.dll) (because they use stdcall convention, and so it is to grep disassembly output just by RETN X ).

- no arguments: 29%
- 1 argument: 23%
- 2 arguments: 20%
- 3 arguments: 11%
- 4 arguments: 7%
- 5 arguments: 3%
- 6 arguments: 2%
- 7 arguments: 1%

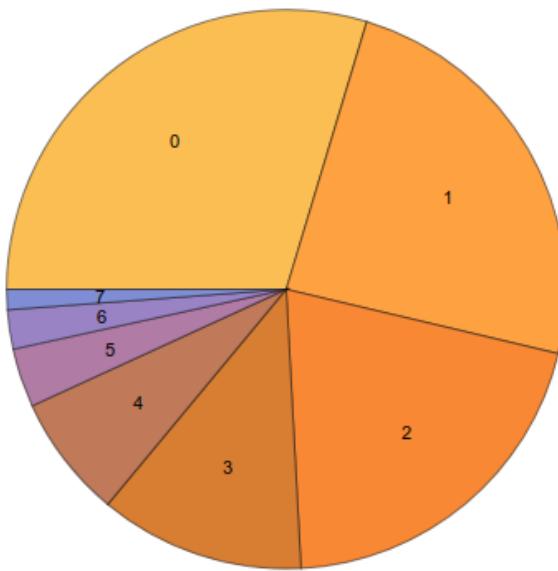


Figure 10.1: Function arguments statistics

This is heavily dependent on programming style and may be very different for other software products.

## **10.3 Compiler intrinsic**

A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a call to it. It is often a pseudofunction for specific [CPU](#) instruction.

For example, there are no cyclic shift operations in C/C++ languages, but they are present in most [CPUs](#).

## 10.4. COMPILER'S ANOMALIES

For programmer's convenience, at least MSVC has pseudofunctions `_rotl()` and `_rotr()`<sup>1</sup> which are translated by the compiler directly to the ROL/ROR x86 instructions.

Another example are functions to generate SSE-instructions right in the code.

Full list of MSVC intrinsics: [MSDN](#).

## 10.4 Compiler's anomalies

### 10.4.1 Oracle RDBMS 11.2 and Intel C++ 10.1

Intel C++ 10.1, which was used for Oracle RDBMS 11.2 Linux86 compilation, may emit two `JZ` in row, and there are no references to the second `JZ`. The second `JZ` is thus meaningless.

Listing 10.1: kdli.o from libserver11.a

```
.text:08114CF1          loc_8114CF1: ; CODE XREF: _PGOSF539_kdlimemSer+89A
.text:08114CF1
.text:08114CF1 8B 45 08      mov    eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14      movzx edx, byte ptr [eax+14h]
.text:08114CF8 F6 C2 01      test   dl, 1
.text:08114CFB 0F 85 17 08 00 00  jnz   loc_8115518
.text:08114D01 85 C9      test   ecx, ecx
.text:08114D03 0F 84 8A 00 00 00  jz    loc_8114D93
.text:08114D09 0F 84 09 08 00 00  jz    loc_8115518
.text:08114D0F 8B 53 08      mov    edx, [ebx+8]
.text:08114D12 89 55 FC      mov    [ebp+var_4], edx
.text:08114D15 31 C0      xor    eax, eax
.text:08114D17 89 45 F4      mov    [ebp+var_C], eax
.text:08114D1A 50      push   eax
.text:08114D1B 52      push   edx
.text:08114D1C E8 03 54 00 00  call   len2nbytes
.text:08114D21 83 C4 08      add    esp, 8
```

Listing 10.2: from the same code

```
.text:0811A2A5          loc_811A2A5: ; CODE XREF: kdliSerLengths+11C
.text:0811A2A5
.text:0811A2A5 8B 7D 08      mov    edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10      mov    edi, [edi+10h]
.text:0811A2AB 0F B6 57 14      movzx edx, byte ptr [edi+14h]
.text:0811A2AF F6 C2 01      test   dl, 1
.text:0811A2B2 75 3E      jnz   short loc_811A2F2
.text:0811A2B4 83 E0 01      and    eax, 1
.text:0811A2B7 74 1F      jz    short loc_811A2D8
.text:0811A2B9 74 37      jz    short loc_811A2F2
.text:0811A2BB 6A 00      push   0
.text:0811A2BD FF 71 08      push   dword ptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF  call   len2nbytes
```

It is supposedly a code generator bug that was not found by tests, because resulting code works correctly anyway.

### 10.4.2 MSVC 6.0

Just found in some old code:

```
fabs
fld   [esp+50h+var_34]
fabs
fxch st(1) ; first instruction
fxch st(1) ; second instruction
faddp st(1), st
```

<sup>1</sup>[MSDN](#)

```
fcomp [esp+50h+var_3C]
fnstsw ax
test ah, 41h
jz short loc_100040B7
```

The first `FXCH` instruction swaps `ST(0)` and `ST(1)`, the second do the same, so both do nothing. This is a program uses MFC42.dll, so it could be MSVC 6.0, 5.0 or maybe even MSVC 4.2 from 1990s.

This pair do nothing, so it probably wasn't caught by MSVC compiler tests. Or maybe I wrong?

### 10.4.3 Summary

Other compiler anomalies here in this book: [1.22.2 on page 315](#), [3.7.3 on page 492](#), [3.15.7 on page 532](#), [1.20.7 on page 302](#), [1.14.4 on page 148](#), [1.22.5 on page 333](#).

Such cases are demonstrated here in this book, to show that such compilers errors are possible and sometimes one should not to rack one's brain while thinking why did the compiler generate such strange code.

## 10.5 Itanium

Although almost failed, Intel Itanium ([IA64](#)) is a very interesting architecture.

While [OOE](#) CPUs decides how to rearrange their instructions and execute them in parallel, [EPIC<sup>2</sup>](#) was an attempt to shift these decisions to the compiler: to let it group the instructions at the compile stage.

This resulted in notoriously complex compilers.

Here is one sample of [IA64](#) code: simple cryptographic algorithm from the Linux kernel:

Listing 10.3: Linux kernel 3.2.0.4

```
#define TEA_ROUNDS          32
#define TEA_DELTA            0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
    u32 y, z, n, sum = 0;
    u32 k0, k1, k2, k3;
    struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
    const __le32 *in = (const __le32 *)src;
    __le32 *out = (__le32 *)dst;

    y = le32_to_cpu(in[0]);
    z = le32_to_cpu(in[1]);

    k0 = ctx->KEY[0];
    k1 = ctx->KEY[1];
    k2 = ctx->KEY[2];
    k3 = ctx->KEY[3];

    n = TEA_ROUNDS;

    while (n-- > 0) {
        sum += TEA_DELTA;
        y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
        z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
    }

    out[0] = cpu_to_le32(y);
    out[1] = cpu_to_le32(z);
}
```

Here is how it was compiled:

<sup>2</sup>Explicitly parallel instruction computing

Listing 10.4: Linux Kernel 3.2.0.4 for Itanium 2 (McKinley)

```

0090|          tea_encrypt:
0090|08 80 80 41 00 21    adds r16 = 96, r32           // ptr to ctx->KEY[2]
0096|80 C0 82 00 42 00    adds r8 = 88, r32            // ptr to ctx->KEY[0]
009C|00 00 04 00    nop.i 0
00A0|09 18 70 41 00 21    adds r3 = 92, r32           // ptr to ctx->KEY[1]
00A6|F0 20 88 20 28 00    ld4 r15 = [r34], 4          // load z
00AC|44 06 01 84    adds r32 = 100, r32;;        // ptr to ctx->KEY[3]
00B0|08 98 00 20 10 10    ld4 r19 = [r16]           // r19=k2
00B6|00 01 00 00 42 40    mov r16 = r0             // r0 always contain zero
00BC|00 08 CA 00    mov.i r2 = ar.lc          // save lc register
00C0|05 70 00 44 10 10
         9E FF FF FF 7F 20    ld4 r14 = [r34]           // load y
00CC|92 F3 CE 6B    movl r17 = 0xFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00    nop.m 0
00D6|50 01 20 20 20 00    ld4 r21 = [r8]           // r21=k0
00DC|F0 09 2A 00    mov.i ar.lc = 31          // TEA_ROUNDS is 32
00E0|0A A0 00 06 10 10    ld4 r20 = [r3];;        // r20=k1
00E6|20 01 80 20 20 00    ld4 r18 = [r32]          // r18=k3
00EC|00 00 04 00    nop.i 0
00F0|
00F0|          loc_F0:
00F0|09 80 40 22 00 20    add r16 = r16, r17          // r16=sum, r17=TEA_DELTA
00F6|D0 71 54 26 40 80    shladd r29 = r14, 4, r21        // r14=y, r21=k0
00FC|A3 70 68 52    extr.u r28 = r14, 5, 27;;
0100|03 F0 40 1C 00 20    add r30 = r16, r14
0106|B0 E1 50 00 40 40    add r27 = r28, r20;;        // r20=k1
010C|D3 F1 3C 80    xor r26 = r29, r30;;
0110|0B C8 6C 34 0F 20    xor r25 = r27, r26;;
0116|F0 78 64 00 40 00    add r15 = r15, r25          // r15=z
011C|00 00 04 00    nop.i 0;;
0120|00 00 00 00 01 00    nop.m 0
0126|80 51 3C 34 29 60    extr.u r24 = r15, 5, 27
012C|F1 98 4C 80    shladd r11 = r15, 4, r19          // r19=k2
0130|0B B8 3C 20 00 20    add r23 = r15, r16;;
0136|A0 C0 48 00 40 00    add r10 = r24, r18          // r18=k3
013C|00 00 04 00    nop.i 0;;
0140|0B 48 28 16 0F 20    xor r9 = r10, r11;;
0146|60 B9 24 1E 40 00    xor r22 = r23, r9
014C|00 00 04 00    nop.i 0;;
0150|11 00 00 00 01 00    nop.m 0
0156|E0 70 58 00 40 A0    add r14 = r14, r22
015C|A0 FF FF 48    br.cloop.sptk.few loc_F0;;
0160|09 20 3C 42 90 15    st4 [r33] = r15, 4          // store z
0166|00 00 00 02 00 00    nop.m 0
016C|20 08 AA 00    mov.i ar.lc = r2;;        // restore lc register
0170|11 00 38 42 90 11    st4 [r33] = r14          // store y
0176|00 00 00 02 00 80    nop.i 0
017C|08 00 84 00    br.ret.sptk.many b0;;

```

First of all, all [IA64](#) instructions are grouped into 3-instruction bundles.

Each bundle has a size of 16 bytes (128 bits) and consists of template code (5 bits) + 3 instructions (41 bits for each).

[IDA](#) shows the bundles as 6+6+4 bytes —you can easily spot the pattern.

All 3 instructions from each bundle usually executes simultaneously, unless one of instructions has a “stop bit”.

Supposedly, Intel and HP engineers gathered statistics on most frequent instruction patterns and decided to bring bundle types ([AKA](#) “templates”): a bundle code defines the instruction types in the bundle. There are 12 of them.

For example, the zeroth bundle type is [MII](#), which implies the first instruction is Memory (load or store), the second and third ones are I (integer instructions).

Another example is the bundle of type 0x1d: [MFB](#): the first instruction is Memory (load or store), the second one is Float ([FPU](#) instruction), and the third is Branch (branch instruction).

If the compiler cannot pick a suitable instruction for the relevant bundle slot, it may insert a [NOP](#): you can

## 10.6. 8086 MEMORY MODEL

see here the `nop.i` instructions (**NOP** at the place where the integer instruction might be) or `nop.m` (a memory instruction might be at this slot).

**NOPs** are inserted automatically when one uses assembly language manually.

And that is not all. Bundles are also grouped.

Each bundle may have a “stop bit”, so all the consecutive bundles with a terminating bundle which has the “stop bit” can be executed simultaneously.

In practice, Itanium 2 can execute 2 bundles at once, resulting in the execution of 6 instructions at once.

So all instructions inside a bundle and a bundle group cannot interfere with each other (i.e., must not have data hazards).

If they do, the results are to be undefined.

Each stop bit is marked in assembly language as two semicolons (`;;`) after the instruction.

So, the instructions at [90-ac] may be executed simultaneously: they do not interfere. The next group is [b0-cc].

We also see a stop bit at 10c. The next instruction at 110 has a stop bit too.

This implies that these instructions must be executed isolated from all others (as in **CISC**).

Indeed: the next instruction at 110 uses the result from the previous one (the value in register r26), so they cannot be executed at the same time.

Apparently, the compiler was not able to find a better way to parallelize the instructions, in other words, to load **CPU** as much as possible, hence too much stop bits and **NOPs**.

Manual assembly programming is a tedious job as well: the programmer has to group the instructions manually.

The programmer is still able to add stop bits to each instructions, but this will degrade the performance that Itanium was made for.

An interesting examples of manual **IA64** assembly code can be found in the Linux kernel’s sources:

<http://go.yurichev.com/17322>.

Another introductory paper on Itanium assembly: [Mike Burrell, *Writing Efficient Itanium 2 Assembly Code* (2010)]<sup>3</sup>, [papasutra of haquebright, *WRITING SHELLCODE FOR IA-64* (2001)]<sup>4</sup>.

Another very interesting Itanium feature is the *speculative execution* and the **NaT** (“not a thing”) bit, somewhat resembling **NaN** numbers:

[MSDN](#).

## 10.6 8086 memory model

When dealing with 16-bit programs for MS-DOS or Win16 ( [8.5.3 on page 831](#) or [3.30.5 on page 659](#)), we can see that the pointers consist of two 16-bit values. What do they mean? Oh yes, that is another weird MS-DOS and 8086 artifact.

8086/8088 was a 16-bit CPU, but was able to address 20-bit address in RAM (thus being able to access 1MB of external memory).

The external memory address space was divided between **RAM** (640KB max), **ROM**, windows for video memory, EMS cards, etc.

Let’s also recall that 8086/8088 was in fact an inheritor of the 8-bit 8080 CPU.

The 8080 has a 16-bit memory space, i.e., it was able to address only 64KB.

And probably because of reason of old software porting<sup>5</sup>, 8086 can support many 64KB windows simultaneously, placed within the 1MB address space.

This is some kind of a toy-level virtualization.

<sup>3</sup>Also available as <http://yurichev.com/mirrors/RE/itanium.pdf>

<sup>4</sup>Also available as <http://phrack.org/issues/57/5.html>

<sup>5</sup>The author is not 100% sure here

## 10.7. BASIC BLOCKS REORDERING

All 8086 registers are 16-bit, so to address more, special segment registers (CS, DS, ES, SS) were introduced.

Each 20-bit pointer is calculated using the values from a segment register and an address register pair (e.g. DS:BX) as follows:

$$\text{real\_address} = (\text{segment\_register} \ll 4) + \text{address\_register}$$

For example, the graphics ([EGA<sup>6</sup>](#), [VGA<sup>7</sup>](#)) video [RAM](#) window on old IBM PC-compatibles has a size of 64KB.

To access it, a value of 0xA000 has to be stored in one of the segment registers, e.g. into DS.

Then DS:0 will address the first byte of video [RAM](#) and DS:0xFFFF — the last byte of RAM.

The real address on the 20-bit address bus, however, will range from 0xA0000 to 0xFFFF.

The program may contain hard-coded addresses like 0x1234, but the [OS](#) may need to load the program at arbitrary addresses, so it recalculates the segment register values in a way that the program does not have to care where it's placed in the RAM.

So, any pointer in the old MS-DOS environment in fact consisted of the segment address and the address inside segment, i.e., two 16-bit values. 20-bit was enough for that, though, but we needed to recalculate the addresses very often: passing more information on the stack seemed a better space/convenience balance.

By the way, because of all this it was not possible to allocate a memory block larger than 64KB.

The segment registers were reused at 80286 as selectors, serving a different function.

When the 80386 CPU and computers with bigger [RAM](#) were introduced, MS-DOS was still popular, so the DOS extenders emerged: these were in fact a step toward a “serious” [OS](#), switching the CPU in protected mode and providing much better memory [APIs](#) for the programs which still needed to run under MS-DOS.

Widely popular examples include DOS/4GW (the DOOM video game was compiled for it), Phar Lap, PMODE.

By the way, the same way of addressing memory was used in the 16-bit line of Windows 3.x, before Win32.

## 10.7 Basic blocks reordering

### 10.7.1 Profile-guided optimization

This optimization method can move some [basic blocks](#) to another section of the executable binary file.

Obviously, there are parts of a function which are executed more frequently (e.g., loop bodies) and less often (e.g., error reporting code, exception handlers).

The compiler adds instrumentation code into the executable, then the developer runs it with a lot of tests to collect statistics.

Then the compiler, with the help of the statistics gathered, prepares final the executable file with all infrequently executed code moved into another section.

As a result, all frequently executed function code is compacted, and that is very important for execution speed and cache usage.

An example from Oracle RDBMS code, which was compiled with Intel C++:

Listing 10.5: orageneric11.dll (win32)

```
_skgfsync    public _skgfsync
              proc near
; address 0x6030D86A
              db      66h
              nop
              push    ebp
              mov     ebp, esp
              mov     edx, [ebp+0Ch]
              test   edx, edx
```

<sup>6</sup>Enhanced Graphics Adapter

<sup>7</sup>Video Graphics Array

## 10.7. BASIC BLOCKS REORDERING

```
jz      short loc_6030D884
mov    eax, [edx+30h]
test   eax, 400h
jnz    __VInfreq_skgfsync ; write to log
continue:
        mov    eax, [ebp+8]
        mov    edx, [ebp+10h]
        mov    dword ptr [eax], 0
        lea    eax, [edx+0Fh]
        and    eax, 0FFFFFFFCh
        mov    ecx, [eax]
        cmp    ecx, 45726963h
        jnz    error           ; exit with error
        mov    esp, ebp
        pop    ebp
        retn
_skgfsync endp

...
; address 0x60B953F0

__VInfreq_skgfsync:
        mov    eax, [edx]
        test   eax, eax
        jz    continue
        mov    ecx, [ebp+10h]
        push   ecx
        mov    ecx, [ebp+8]
        push   edx
        push   ecx
        push   offset ...
        push   dword ptr [edx+4]
        call   dword ptr [eax] ; write to log
        add    esp, 14h
        jmp    continue

error:
        mov    edx, [ebp+8]
        mov    dword ptr [edx], 69AAh ; 27050 "function called with invalid FIB/IOV ↵
        ↓ structure"
        mov    eax, [eax]
        mov    [edx+4], eax
        mov    dword ptr [edx+8], 0FA4h ; 4004
        mov    esp, ebp
        pop    ebp
        retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

The distance of addresses between these two code fragments is almost 9 MB.

All infrequently executed code was placed at the end of the code section of the DLL file, among all function parts.

This part of the function was marked by the Intel C++ compiler with the `VInfreq` prefix.

Here we see that a part of the function that writes to a log file (presumably in case of error or warning or something like that) which was probably not executed very often when Oracle's developers gathered statistics (if it was executed at all).

The writing to log basic block eventually returns the control flow to the “hot” part of the function.

Another “infrequent” part is the `basic block` returning error code 27050.

In Linux ELF files, all infrequently executed code is moved by Intel C++ into the separate `text.unlikely` section, leaving all “hot” code in the `text.hot` section.

From a reverse engineer's perspective, this information may help to split the function into its core and error handling parts.

# Chapter 11

## Books/blogs worth reading

### 11.1 Books and other materials

#### 11.1.1 Reverse Engineering

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)

Also, Kris Kaspersky's books.

#### 11.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*

Blogs:

- Microsoft: Raymond Chen
- nynaeve.net

#### 11.1.3 C/C++

- [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, (1988)]
- ISO/IEC 9899:TC3 (C C99 standard), (2007)<sup>1</sup>
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard<sup>2</sup>
- Agner Fog, *Optimizing software in C++* (2015)<sup>3</sup>
- Marshall Cline, *C++ FAQ*<sup>4</sup>
- Dennis Yurichev, *C/C++ programming language notes*<sup>5</sup>
- JPL Institutional Coding Standard for the C Programming Language<sup>6</sup>

<sup>1</sup>Also available as <http://go.yurichev.com/17274>

<sup>2</sup>Also available as <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>.

<sup>3</sup>Also available as [http://agner.org/optimize/optimizing\\_cpp.pdf](http://agner.org/optimize/optimizing_cpp.pdf).

<sup>4</sup>Also available as <http://go.yurichev.com/17291>

<sup>5</sup>Also available as <http://yurichev.com/C-book.html>

<sup>6</sup>Also available as [https://yurichev.com/mirrors/C/JPL\\_Coding\\_Standard\\_C.pdf](https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf)

**11.1.4 x86 / x86-64**

- Intel manuals<sup>7</sup>
- AMD manuals<sup>8</sup>
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)<sup>9</sup>
- Agner Fog, *Calling conventions* (2015)<sup>10</sup>
- [*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)]
- [*Software Optimization Guide for AMD Family 16h Processors*, (2013)]

Somewhat outdated, but still interesting to read:

Michael Abrash, *Graphics Programming Black Book*, 1997<sup>11</sup> (he is known for his work on low-level optimization for such projects as Windows NT 3.1 and id Quake).

**11.1.5 ARM**

- ARM manuals<sup>12</sup>
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)
- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)]<sup>13</sup>
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)<sup>14</sup>

**11.1.6 Assembly language**

Richard Blum — Professional Assembly Language.

**11.1.7 Java**

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] <sup>15</sup>.

**11.1.8 UNIX**

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

**11.1.9 Programming in general**

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002).
- (For hard-core geeks with computer science and mathematical background) Donald E. Knuth, *The Art of Computer Programming*.

<sup>7</sup>Also available as <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

<sup>8</sup>Also available as <http://developer.amd.com/resources/developer-guides-manuals/>

<sup>9</sup>Also available as <http://agner.org/optimize/microarchitecture.pdf>

<sup>10</sup>Also available as [http://www.agner.org/optimize/calling\\_conventions.pdf](http://www.agner.org/optimize/calling_conventions.pdf)

<sup>11</sup>Also available as <https://github.com/jagregory/abrush-black-book>

<sup>12</sup>Also available as [http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc\\_subset.architecture.reference/index.html](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc_subset.architecture.reference/index.html)

<sup>13</sup>Also available as [http://yurichev.com/mirrors/ARMv8-A\\_Architecture\\_Reference\\_Manual\\_\(Issue\\_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

<sup>14</sup>Also available as <http://go.yurichev.com/17273>

<sup>15</sup>Also available as <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>; <http://docs.oracle.com/javase/specs/jvms/se7/html/>

## 11.1.10 Cryptography

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) Ivh, *Crypto 101*<sup>16</sup>
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*<sup>17</sup>.

---

<sup>16</sup>Also available as <https://www.cryptol101.io/>  
<sup>17</sup>Also available as <https://crypto.stanford.edu/~dabo/cryptobook/>

# Chapter 12

## Communities

There are two excellent RE<sup>1</sup>-related subreddits on reddit.com: [reddit.com/r/ReverseEngineering/](https://www.reddit.com/r/ReverseEngineering/) and [reddit.com/r/remath](https://www.reddit.com/r/remath) (on the topics for the intersection of RE and mathematics).

There is also a RE part of the Stack Exchange website: [reverseengineering.stackexchange.com](https://reverseengineering.stackexchange.com).

On IRC there's a ##re channel on FreeNode<sup>2</sup>.

---

<sup>1</sup>Reverse Engineering

<sup>2</sup>[freenode.net](https://freenode.net)

# **Afterword**

## 12.1 Questions?

Do not hesitate to mail any questions to the author:

`<dennis(a)yurichev.com>`. Do you have any suggestion on new content for the book? Please do not hesitate to send any corrections (including grammar (you see how horrible my English is?)), etc.

The author is working on the book a lot, so the page and listing numbers, etc., are changing very rapidly. Please do not refer to page and listing numbers in your emails to me. There is a much simpler method: make a screenshot of the page, in a graphics editor underline the place where you see the error, and send it to me. He'll fix it much faster. And if you familiar with git and  $\text{\LaTeX}$  you can fix the error right in the source code:

[GitHub](#).

Do not worry to bother me while writing me about any petty mistakes you found, even if you are not very confident. I'm writing for beginners, after all, so beginners' opinions and comments are crucial for my job.

# **Appendix**

## **.1.1 Terminology**

Common for 16-bit (8086/80286), 32-bit (80386, etc.), 64-bit.

**byte** 8-bit. The DB assembly directive is used for defining variables and arrays of bytes. Bytes are passed in the 8-bit part of registers: AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL/R\*L.

**word** 16-bit. DW assembly directive —“—. Words are passed in the 16-bit part of the registers: AX/BX/CX/DX/SI/DI/R\*W.

**double word** (“dword”) 32-bit. DD assembly directive —“—. Double words are passed in registers (x86) or in the 32-bit part of registers (x64). In 16-bit code, double words are passed in 16-bit register pairs.

**quad word** (“qword”) 64-bit. DQ assembly directive —“—. In 32-bit environment, quad words are passed in 32-bit register pairs.

**tbyte** (10 bytes) 80-bit or 10 bytes (used for IEEE 754 FPU registers).

**paragraph** (16 bytes)— term was popular in MS-DOS environment.

Data types of the same width (BYTE, WORD, DWORD) are also the same in Windows API.

## **.1.2 General purpose registers**

It is possible to access many registers by byte or 16-bit word parts. It is all inheritance from older Intel CPUs (up to the 8-bit 8080) still supported for backward compatibility. Older 8-bit CPUs (8080) had 16-bit registers divided by two. Programs written for 8080 could access the low byte part of 16-bit registers, high byte part or the whole 16-bit register. Perhaps, this feature was left in 8086 as a helper for easier porting. This feature is usually not present in RISC CPUs.

Registers prefixed with R- appeared in x86-64, and those prefixed with E- —in 80386. Thus, R-registers are 64-bit, and E-registers—32-bit.

8 more GPR’s were added in x86-86: R8-R15.

N.B.: In the Intel manuals the byte parts of these registers are prefixed by L, e.g.: R8L, but IDA names these registers by adding the B suffix, e.g.: R8B.

### **RAX/EAX/AX/AL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RAX <sup>x64</sup>							
EAX							
AX							
AH AL							

AKA accumulator. The result of a function is usually returned via this register.

### **RBX/EBX/BX/BL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RBX <sup>x64</sup>							
EBX							
BX							
BH BL							

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RCX <sup>x64</sup>							
ECX							
CX							
CH CL							

**AKA** counter: in this role it is used in REP prefixed instructions and also in shift instructions (SHL/SHR/RxL/RxR).

### **RDX/EDX/DX/DL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RDX <sup>x64</sup>							
EDX							
DX							
DH DL							

### **RSI/ESI/SI/SIL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RSI <sup>x64</sup>							
ESI							
SI							
SIL <sup>x64</sup>							

**AKA** “source index”. Used as source in the instructions REP MOVSx, REP CMPSx.

### **RDI/EDI/DI/DIL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RDI <sup>x64</sup>							
EDI							
DI							
DIL <sup>x64</sup>							

**AKA** “destination index”. Used as a pointer to the destination in the instructions REP MOVSx, REP STOSx.

### **R8/R8D/R8W/R8L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
R8							
R8D							
R8W							
R8L							

### **R9/R9D/R9W/R9L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
R9							
R9D							
R9W							
R9L							

**R10/R10D/R10W/R10L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R10			
					R10D		
						R10W	
							R10L

**R11/R11D/R11W/R11L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R11			
					R11D		
						R11W	
							R11L

**R12/R12D/R12W/R12L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R12			
					R12D		
						R12W	
							R12L

**R13/R13D/R13W/R13L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R13			
					R13D		
						R13W	
							R13L

**R14/R14D/R14W/R14L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R14			
					R14D		
						R14W	
							R14L

**R15/R15D/R15W/R15L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R15			
					R15D		
						R15W	
							R15L

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				RSP			
					ESP		
						SP	
							SPL

**AKA stack pointer.** Usually points to the current stack except in those cases when it is not yet initialized.

### **RBP/EBP/BP/BPL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
			RBP				
				EBP			
					BP		
						BPL	

**AKA frame pointer.** Usually used for local variables and accessing the arguments of the function. More about it: ([1.9.1 on page 68](#)).

### **RIP/EIP/IP**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
			RIP <sup>x64</sup>				
				EIP			
					IP		

**AKA “instruction pointer”**<sup>3</sup>. Usually always points to the instruction to be executed right now. Cannot be modified, however, it is possible to do this (which is equivalent):

```
MOV EAX, ...
JMP EAX
```

Or:

```
PUSH value
RET
```

### **CS/DS/ES/SS/FS/GS**

16-bit registers containing code selector (CS), data selector (DS), stack selector (SS).

FS in win32 points to [TLS](#), GS took this role in Linux. It is made so for faster access to the [TLS](#) and other structures like the [TIB](#).

In the past, these registers were used as segment registers ([10.6 on page 992](#)).

### **Flags register**

**AKA EFLAGS.**

<sup>3</sup>Sometimes also called “program counter”

## .1. X86

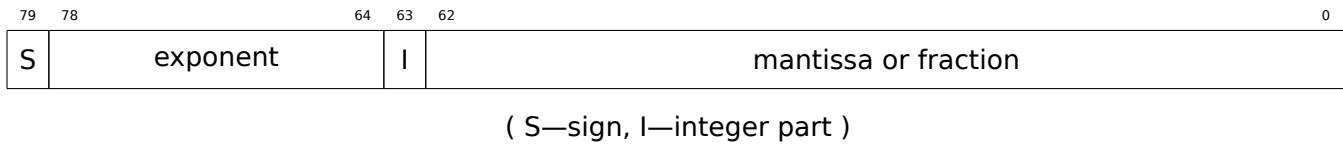
Bit (mask)	Abbreviation (meaning)	Description
0 (1)	CF (Carry)	The CLC/STC/CMC instructions are used for setting/resetting/toggling this flag ( <a href="#">1.19.7 on page 235</a> ).
2 (4)	PF (Parity)	Exist solely for work with <a href="#">BCD</a> -numbers
4 (0x10)	AF (Adjust)	
6 (0x40)	ZF (Zero)	Setting to 0 if the last operation's result is equal to 0.
7 (0x80)	SF (Sign)	
8 (0x100)	TF (Trap)	Used for debugging. If turned on, an exception is to be generated after each instruction's execution.
9 (0x200)	IF (Interrupt enable)	Are interrupts enabled. The CLI/STI instructions are used for setting/resetting the flag
10 (0x400)	DF (Direction)	A direction is set for the REP MOVSx/CMPSx/LODSx/SCASx instructions. The CLD/STD instructions are used for setting/resetting the flag See also: <a href="#">3.25 on page 641</a> .
11 (0x800)	OF (Overflow)	
12, 13 (0x3000)	IOPL (I/O privilege level) <sup>i286</sup>	
14 (0x4000)	NT (Nested task) <sup>i286</sup>	
16 (0x10000)	RF (Resume) <sup>i386</sup>	Used for debugging. The CPU ignores the hardware breakpoint in DRx if the flag is set.
17 (0x20000)	VM (Virtual 8086 mode) <sup>i386</sup>	
18 (0x40000)	AC (Alignment check) <sup>i486</sup>	
19 (0x80000)	VIF (Virtual interrupt) <sup>i586</sup>	
20 (0x100000)	VIP (Virtual interrupt pending) <sup>i586</sup>	
21 (0x200000)	ID (Identification) <sup>i586</sup>	

All the rest flags are reserved.

### .1.3 FPU registers

8 80-bit registers working as a stack: ST(0)-ST(7). N.B.: [IDA](#) calls ST(0) as just ST. Numbers are stored in the IEEE 754 format.

*long double* value format:



### Control Word

Register controlling the behavior of the [FPU](#).

## 1. X86

Bit	Abbreviation (meaning)	Description
0	IM (Invalid operation Mask)	
1	DM (Denormalized operand Mask)	
2	ZM (Zero divide Mask)	
3	OM (Overflow Mask)	
4	UM (Underflow Mask)	
5	PM (Precision Mask)	
7	IEM (Interrupt Enable Mask)	Exceptions enabling, 1 by default (disabled)
8, 9	PC (Precision Control)	00 — 24 bits (REAL4) 10 — 53 bits (REAL8) 11 — 64 bits (REAL10)
10, 11	RC (Rounding Control)	00 — (by default) round to nearest 01 — round toward $-\infty$ 10 — round toward $+\infty$ 11 — round toward 0
12	IC (Infinity Control)	0 — (by default) treat $+\infty$ and $-\infty$ as unsigned 1 — respect both $+\infty$ and $-\infty$

The PM, UM, OM, ZM, DM, IM flags define if to generate exception in the case of a corresponding error.

## Status Word

Read-only register.

Bit	Abbreviation (meaning)	Description
15	B (Busy)	Is FPU do something (1) or results are ready (0)
14	C3	
13, 12, 11	TOP	points to the currently zeroth register
10	C2	
9	C1	
8	C0	
7	IR (Interrupt Request)	
6	SF (Stack Fault)	
5	P (Precision)	
4	U (Underflow)	
3	O (Overflow)	
2	Z (Zero)	
1	D (Denormalized)	
0	I (Invalid operation)	

The SF, P, U, O, Z, D, I bits signal about exceptions.

About the C3, C2, C1, C0 you can read more here: ([1.19.7 on page 234](#)).

N.B.: When ST(x) is used, the FPU adds  $x$  to TOP (by modulo 8) and that is how it gets the internal register's number.

## Tag Word

The register has current information about the usage of numbers registers.

Bit	Abbreviation (meaning)
15, 14	Tag(7)
13, 12	Tag(6)
11, 10	Tag(5)
9, 8	Tag(4)
7, 6	Tag(3)
5, 4	Tag(2)
3, 2	Tag(1)
1, 0	Tag(0)

Each tag contains information about a physical FPU register (R(x)), not logical (ST(x)).

For each tag:

- 00 — The register contains a non-zero value

## .1. X86

---

- 01 — The register contains 0
- 10 — The register contains a special value ([NAN<sup>4</sup>](#),  $\infty$ , or denormal)
- 11 — The register is empty

## .1.4 SIMD registers

### MMX registers

8 64-bit registers: MM0..MM7.

### SSE and AVX registers

SSE: 8 128-bit registers: XMM0..XMM7. In the x86-64 8 more registers were added: XMM8..XMM15. AVX is the extension of all these registers to 256 bits.

## .1.5 Debugging registers

Used for hardware breakpoints control.

- DR0 — address of breakpoint #1
- DR1 — address of breakpoint #2
- DR2 — address of breakpoint #3
- DR3 — address of breakpoint #4
- DR6 — a cause of break is reflected here
- DR7 — breakpoint types are set here

### DR6

Bit (mask)	Description
0 (1)	B0 — breakpoint #1 has been triggered
1 (2)	B1 — breakpoint #2 has been triggered
2 (4)	B2 — breakpoint #3 has been triggered
3 (8)	B3 — breakpoint #4 has been triggered
13 (0x2000)	BD — modification attempt of one of the DRx registers. may be raised if GD is enabled
14 (0x4000)	BS — single step breakpoint (TF flag has been set in EFLAGS). Highest priority. Other bits may also be set.
15 (0x8000)	BT (task switch flag)

N.B. A single step breakpoint is a breakpoint which occurs after each instruction. It can be enabled by setting TF in EFLAGS ([.1.2 on page 1005](#)).

### DR7

Breakpoint types are set here.

---

<sup>4</sup>Not a Number

Bit (mask)	Description
0 (1)	L0 — enable breakpoint #1 for the current task
1 (2)	G0 — enable breakpoint #1 for all tasks
2 (4)	L1 — enable breakpoint #2 for the current task
3 (8)	G1 — enable breakpoint #2 for all tasks
4 (0x10)	L2 — enable breakpoint #3 for the current task
5 (0x20)	G2 — enable breakpoint #3 for all tasks
6 (0x40)	L3 — enable breakpoint #4 for the current task
7 (0x80)	G3 — enable breakpoint #4 for all tasks
8 (0x100)	LE — not supported since P6
9 (0x200)	GE — not supported since P6
13 (0x2000)	GD — exception is to be raised if any MOV instruction tries to modify one of the DRx registers
16,17 (0x30000)	breakpoint #1: R/W — type
18,19 (0xC0000)	breakpoint #1: LEN — length
20,21 (0x300000)	breakpoint #2: R/W — type
22,23 (0xC00000)	breakpoint #2: LEN — length
24,25 (0x3000000)	breakpoint #3: R/W — type
26,27 (0xC000000)	breakpoint #3: LEN — length
28,29 (0x30000000)	breakpoint #4: R/W — type
30,31 (0xC0000000)	breakpoint #4: LEN — length

The breakpoint type is to be set as follows (R/W):

- 00 — instruction execution
- 01 — data writes
- 10 — I/O reads or writes (not available in user-mode)
- 11 — on data reads or writes

N.B.: breakpoint type for data reads is absent, indeed.

Breakpoint length is to be set as follows (LEN):

- 00 — one-byte
- 01 — two-byte
- 10 — undefined for 32-bit mode, eight-byte in 64-bit mode
- 11 — four-byte

## .1.6 Instructions

Instructions marked as (M) are not usually generated by the compiler: if you see one of them, it is probably a hand-written piece of assembly code, or a compiler intrinsic ([10.3 on page 988](#)).

Only the most frequently used instructions are listed here. You can read [11.1.4 on page 996](#) for a full documentation.

Do you have to know all instruction's opcodes by heart? No, only those which are used for code patching ([10.1.2 on page 987](#)). All the rest of the opcodes don't need to be memorized.

### Prefixes

**LOCK** forces CPU to make exclusive access to the RAM in multiprocessor environment. For the sake of simplification, it can be said that when an instruction with this prefix is executed, all other CPUs in a multiprocessor system are stopped. Most often it is used for critical sections, semaphores, mutexes. Commonly used with ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. You can read more about critical sections here ([6.5.4 on page 786](#)).

**REP** is used with the MOVSx and STOSx instructions: execute the instruction in a loop, the counter is located in the CX/ECX/RCX register. For a detailed description, read more about the MOVSx ([.1.6 on page 1012](#)) and STOSx ([.1.6 on page 1014](#)) instructions.

The instructions prefixed by REP are sensitive to the DF flag, which is used to set the direction.

**REPE/REPNE** (AKA REPZ/REPNZ) used with CMPSx and SCASx instructions: execute the last instruction in a loop, the count is set in the `CX / ECX / RCX` register. It terminates prematurely if ZF is 0 (REPE) or if ZF is 1 (REPNE).

For a detailed description, you can read more about the CMPSx ([.1.6 on page 1015](#)) and SCASx ([.1.6 on page 1013](#)) instructions.

Instructions prefixed by REPE/REPNE are sensitive to the DF flag, which is used to set the direction.

## Most frequently used instructions

These can be memorized in the first place.

**ADC** (add with carry) add values, [increment](#) the result if the CF flag is set. ADC is often used for the addition of large values, for example, to add two 64-bit values in a 32-bit environment using two ADD and ADC instructions. For example:

```
; work with 64-bit values: add val1 to val2.
; .lo means lowest 32 bits, .hi means highest.
ADD val1.lo, val2.lo
ADC val1.hi, val2.hi ; use CF set or cleared at the previous instruction
```

One more example: [1.28 on page 397](#).

**ADD** add two values

**AND** logical “and”

**CALL** call another function:

`PUSH address_after_CALL_instruction; JMP label`

**CMP** compare values and set flags, the same as `SUB` but without writing the result

**DEC** [decrement](#).Unlike other arithmetic instructions, `DEC` doesn't modify CF flag.

**IMUL** signed multiply `IMUL` often used instead of `MUL`, read more about it: [2.2.1](#).

**INC** [increment](#).Unlike other arithmetic instructions, `INC` doesn't modify CF flag.

**JCXZ, JCXZ, JRCXZ** (M) jump if CX/ECX/RCX=0

**JMP** jump to another address. The opcode has a [jump offset](#).

**Jcc** (where cc—condition code)

A lot of these instructions have synonyms (denoted with AKA), this was done for convenience. Synonymous instructions are translated into the same opcode. The opcode has a [jump offset](#).

**JAE** AKA JNC: jump if above or equal (unsigned): CF=0

**JA** AKA JNBE: jump if greater (unsigned): CF=0 and ZF=0

**JBE** jump if lesser or equal (unsigned): CF=1 or ZF=1

**JB** AKA JC: jump if below (unsigned): CF=1

**JC** AKA JB: jump if CF=1

**JE** AKA JZ: jump if equal or zero: ZF=1

**JGE** jump if greater or equal (signed): SF=OF

**JG** jump if greater (signed): ZF=0 and SF=OF

**JLE** jump if lesser or equal (signed): ZF=1 or SF≠OF

**JL** jump if lesser (signed): SF≠OF

**JNAE** AKA JC: jump if not above or equal (unsigned) CF=1

**JNA** jump if not above (unsigned) CF=1 and ZF=1

**JNBE** jump if not below or equal (unsigned): CF=0 and ZF=0

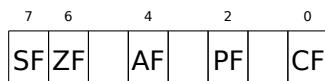
**JNB** AKA JNC: jump if not below (unsigned): CF=0

**JNC** AKA JAE: jump CF=0 synonymous to JNB.

## 1. X86

**JNE AKA JNZ:** jump if not equal or not zero: ZF=0  
**JNGE** jump if not greater or equal (signed): SF≠OF  
**JNG** jump if not greater (signed): ZF=1 or SF≠OF  
**JNLE** jump if not lesser (signed): ZF=0 and SF=OF  
**JNL** jump if not lesser (signed): SF=OF  
**JNO** jump if not overflow: OF=0  
**JNS** jump if SF flag is cleared  
**JNZ AKA JNE:** jump if not equal or not zero: ZF=0  
**JO** jump if overflow: OF=1  
**JPO** jump if PF flag is cleared (Jump Parity Odd)  
**JP AKA JPE:** jump if PF flag is set  
**JS** jump if SF flag is set  
**JZ AKA JE:** jump if equal or zero: ZF=1

**LAHF** copy some flag bits to AH:



This instruction is often used in [FPU](#)-related code.

**LEAVE** equivalent of the **MOV ESP, EBP** and **POP EBP** instruction pair—in other words, this instruction sets the [stack pointer](#) (**ESP**) back and restores the **EBP** register to its initial state.

**LEA** (*Load Effective Address*) form an address

This instruction was intended not for summing values and multiplication but for forming an address, e.g., for calculating the address of an array element by adding the array address, element index, with multiplication of element size<sup>5</sup>.

So, the difference between **MOV** and **LEA** is that **MOV** forms a memory address and loads a value from memory or stores it there, but **LEA** just forms an address.

But nevertheless, it is can be used for any other calculations.

**LEA** is convenient because the computations performed by it does not alter [CPU](#) flags. This may be very important for [OOE](#) processors (to create less data dependencies).

Aside from this, starting at least at Pentium, LEA instruction is executed in 1 cycle.

```
int f(int a, int b)
{
    return a*8+b;
};
```

Listing 1: Optimizing MSVC 2010

```
_a$ = 8          ; size = 4
_b$ = 12         ; size = 4
_f      PROC
    mov     eax, DWORD PTR _b$[esp-4]
    mov     ecx, DWORD PTR _a$[esp-4]
    lea     eax, DWORD PTR [eax+ecx*8]
    ret     0
_f      ENDP
```

Intel C++ uses LEA even more:

```
int f1(int a)
{
    return a*13;
};
```

<sup>5</sup>See also: [wikipedia](#)

```
_f1    PROC NEAR
      mov     ecx, DWORD PTR [4+esp]      ; ecx = a
      lea     edx, DWORD PTR [ecx+ecx*8]   ; edx = a*9
      lea     eax, DWORD PTR [edx+ecx*4]   ; eax = a*9 + a*4 = a*13
      ret
```

These two instructions performs faster than one IMUL.

**MOVSB/MOVSW/MOVSD/MOVSQ** copy byte/ 16-bit word/ 32-bit word/ 64-bit word from the address which is in SI/ESI/RSI into the address which is in DI/EDI/RDI.

Together with the REP prefix, it is to be repeated in a loop, the count is to be stored in the CX/ECX/RCX register: it works like memcpy() in C. If the block size is known to the compiler in the compile stage, memcpy() is often inlined into a short code fragment using REP MOVSw, sometimes even as several instructions.

The memcpy(EDI, ESI, 15) equivalent is:

```
; copy 15 bytes from ESI to EDI
CLD          ; set direction to forward
MOV ECX, 3
REP MOVSD    ; copy 12 bytes
MOVSW        ; copy 2 more bytes
MOVSB        ; copy remaining byte
```

( Supposedly, it works faster than copying 15 bytes using just one REP MOVSb).

**MOVSX** load with sign extension see also: ([1.17.1 on page 201](#))

**MOVZX** load and clear all other bits see also: ([1.17.1 on page 202](#))

**MOV** load value. this instruction name is misnomer, resulting in some confusion (data is not moved but copied), in other architectures the same instructions is usually named “LOAD” and/or “STORE” or something like that.

One important thing: if you set the low 16-bit part of a 32-bit register in 32-bit mode, the high 16 bits remains as they were. But if you modify the low 32-bit part of the register in 64-bit mode, the high 32 bits of the register will be cleared.

Supposedly, it was done to simplify porting code to x86-64.

**MUL** unsigned multiply. **IMUL** often used instead of **MUL**, read more about it: [2.2.1](#).

**NEG** negation:  $op = -op$  Same as **NOT op / ADD op, 1**.

**NOP** **NOP**. Its opcode is 0x90, it is in fact the **XCHG EAX,EAX** idle instruction. This implies that x86 does not have a dedicated **NOP** instruction (as in many RISC). This book has at least one listing where GDB shows NOP as 16-bit XCHG instruction: [1.8.1 on page 48](#).

More examples of such operations: ([1.7 on page 1021](#)).

**NOP** may be generated by the compiler for aligning labels on a 16-byte boundary. Another very popular usage of **NOP** is to replace manually (patch) some instruction like a conditional jump to **NOP** in order to disable its execution.

**NOT**  $op1: op1 = \neg op1$ . logical inversion Important feature—the instruction doesn’t change flags.

**OR** logical “or”

**POP** get a value from the stack:  $value=SS:[ESP]; ESP=ESP+4$  (or 8)

**PUSH** push a value into the stack:  $ESP=ESP-4$  (or 8);  $SS:[ESP]=value$

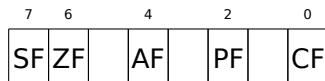
**RET** return from subroutine: **POP tmp; JMP tmp**.

In fact, RET is an assembly language macro, in Windows and \*NIX environment it is translated into RETN (“return near”) or, in MS-DOS times, where the memory was addressed differently ([10.6 on page 992](#)), into RETF (“return far”).

**RET** can have an operand. Then it works like this:

**POP tmp; ADD ESP op1; JMP tmp**. **RET** with an operand usually ends functions in the *stdcall* calling convention, see also: [6.1.2 on page 733](#).

**SAHF** copy bits from AH to CPU flags:



This instruction is often used in FPU-related code.

**SBB** (*subtraction with borrow*) subtract values, [decrement](#) the result if the CF flag is set. SBB is often used for subtraction of large values, for example, to subtract two 64-bit values in 32-bit environment using two SUB and SBB instructions. For example:

```
; work with 64-bit values: subtract val2 from val1.
; .lo means lowest 32 bits, .hi means highest.
SUB val1.lo, val2.lo
SBB val1.hi, val2.hi ; use CF set or cleared at the previous instruction
```

One more example: [1.28 on page 397](#).

**SCASB/SCASW/SCASD/SCASQ** (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word that's stored in AX/EAX/RAX with a variable whose address is in DI/EDI/RDI. Set flags as [CMP](#) does.

This instruction is often used with the REPNE prefix: continue to scan the buffer until a special value stored in AX/EAX/RAX is found. Hence “NE” in REPNE: continue to scan while the compared values are not equal and stop when equal.

It is often used like the `strlen()` C standard function, to determine an [ASCIIIZ](#) string's length:

Example:

```
lea    edi, string
mov   ecx, 0xFFFFFFFFh ; scan  $2^{32}-1$  bytes, i.e., almost infinitely
xor   eax, eax        ; 0 is the terminator
repne scasb
add   edi, 0xFFFFFFFFh ; correct it

; now EDI points to the last character of the ASCIIIZ string.

; lets determine string length
; current ECX = -1-strlen

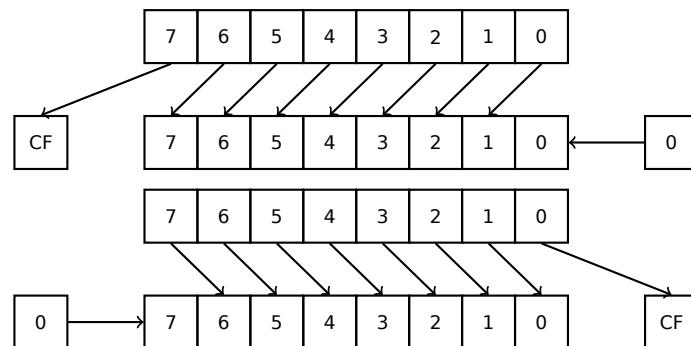
not   ecx
dec   ecx

; now ECX contain string length
```

If we use a different AX/EAX/RAX value, the function acts like the `memchr()` standard C function, i.e., it finds a specific byte.

**SHL** shift value left

**SHR** shift value right:



These instructions are frequently used for multiplication and division by  $2^n$ . Another very frequent application is processing bit fields: [1.22 on page 304](#).

**SHRD** op1, op2, op3: shift value in op2 right by op3 bits, taking bits from op1.

Example: [1.28 on page 397](#).

**STOSB/STOSW/STOSD/STOSQ** store byte/ 16-bit word/ 32-bit word/ 64-bit word from AX/EAX/RAX into the address which is in DI/EDI/RDI.

Together with the REP prefix, it is to be repeated in a loop, the counter is in the CX/ECX/RCX register: it works like `memset()` in C. If the block size is known to the compiler on compile stage, `memset()` is often inlined into a short code fragment using REP MOVSx, sometimes even as several instructions.

`memset(EDI, 0xAA, 15)` equivalent is:

```
; store 15 0xAA bytes to EDI
CLD           ; set direction to forward
MOV EAX, 0AAAAAAAAh
MOV ECX, 3
REP STOSD      ; write 12 bytes
STOSW         ; write 2 more bytes
STOSB         ; write remaining byte
```

( Supposedly, it works faster than storing 15 bytes using just one REP STOSB).

**SUB** subtract values. A frequently occurring pattern is `SUB reg, reg`, which implies zeroing of `reg`.

**TEST** same as AND but without saving the result, see also: [1.22 on page 304](#)

**XOR** op1, op2: [XOR<sup>6</sup>](#) values.  $op1 = op1 \oplus op2$ . A frequently occurring pattern is `XOR reg, reg`, which implies zeroing of `reg`. See also: [2.5 on page 461](#).

## Less frequently used instructions

**BSF** bit scan forward, see also: [1.29.2 on page 420](#)

**BSR** bit scan reverse

**BSWAP** (byte swap), change value [endianness](#).

**BTC** bit test and complement

**BTR** bit test and reset

**BTS** bit test and set

**BT** bit test

**CBW/CWD/CWDE/CDQ/CDQE** Sign-extend value:

**CBW** convert byte in AL to word in AX

**CWD** convert word in AX to doubleword in DX:AX

**CWDE** convert word in AX to doubleword in EAX

**CDQ** convert doubleword in EAX to quadword in EDX:EAX

**CDQE** (x64) convert doubleword in EAX to quadword in RAX

These instructions consider the value's sign, extending it to high part of the newly constructed value. See also: [1.28.5 on page 406](#).

Interestingly to know these instructions was initially named as `SEX` (*Sign EXtend*), as Stephen P. Morse (one of Intel 8086 CPU designers) wrote in [Stephen P. Morse, *The 8086 Primer*, (1980)]<sup>7</sup>:

The process of stretching numbers by extending the sign bit is called sign extension. The 8086 provides instructions (Fig. 3.29) to facilitate the task of sign extension. These instructions were initially named SEX (sign extend) but were later renamed to the more conservative CBW (convert byte to word) and CWD (convert word to double word).

**CLD** clear DF flag.

**CLI** (M) clear IF flag

**CMC** (M) toggle CF flag

<sup>6</sup>eXclusive OR

<sup>7</sup>Also available as <https://archive.org/details/The8086Primer>

## .1. X86

**CMOVcc** conditional MOV: load if the condition is true. The condition codes are the same as in the Jcc instructions ( [.1.6 on page 1010](#)).

**CMPSB/CMPSW/CMPSD/CMPSQ** (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word from the address which is in SI/ESI/RSI with the variable at the address stored in DI/EDI/RDI. Set flags as **CMP** does.

Together with the REP prefix, it is to be repeated in a loop, the counter is stored in the CX/ECX/RCX register, the process will run until the ZF flag is zero (e.g., until the compared values are equal to each other, hence "E" in REPE).

It works like memcmp() in C.

Example from the Windows NT kernel ([WRK v1.2](#)):

Listing 3: base\ntos\rtl\i386\movemem.asm

```
; ULONG
; RtlCompareMemory (
;     IN PVOID Source1,
;     IN PVOID Source2,
;     IN ULONG Length
; )
;
; Routine Description:
;
; This function compares two blocks of memory and returns the number
; of bytes that compared equal.
;
; Arguments:
;
; Source1 (esp+4) - Supplies a pointer to the first block of memory to
; compare.
;
; Source2 (esp+8) - Supplies a pointer to the second block of memory to
; compare.
;
; Length (esp+12) - Supplies the Length, in bytes, of the memory to be
; compared.
;
; Return Value:
;
; The number of bytes that compared equal is returned as the function
; value. If all bytes compared equal, then the length of the original
; block of memory is returned.
;
;--
;
RcmSource1      equ      [esp+12]
RcmSource2      equ      [esp+16]
RcmLength       equ      [esp+20]

CODE_ALIGNMENT
cPublicProc _RtlCompareMemory,3
cPublicFpo 3,0

    push    esi          ; save registers
    push    edi          ;
    cld                ; clear direction
    mov     esi,RcmSource1 ; (esi) -> first block to compare
    mov     edi,RcmSource2 ; (edi) -> second block to compare

;
; Compare dwords, if any.
;

rcm10:  mov     ecx,RcmLength      ; (ecx) = length in bytes
        shr     ecx,2          ; (ecx) = length in dwords
        jz      rcm20          ; no dwds, try bytes
        repe   cmpsd          ; compare dwds
        jnz    rcm40          ; mismatch, go find byte
```

```

;
; Compare residual bytes, if any.
;

rcm20:  mov      ecx,RcmLength          ; (ecx) = length in bytes
        and      ecx,3                ; (ecx) = length mod 4
        jz       rcm30              ; 0 odd bytes, go do dwords
        repe    cmpsb               ; compare odd bytes
        jnz     rcm50               ; mismatch, go report how far we got

;
; All bytes in the block match.
;

rcm30:  mov      eax,RcmLength          ; set number of matching bytes
        pop      edi                ; restore registers
        pop      esi                ;
        stdRET _RtlCompareMemory

;
; When we come to rcm40, esi (and edi) points to the dword after the
; one which caused the mismatch. Back up 1 dword and find the byte.
; Since we know the dword didn't match, we can assume one byte won't.
;

rcm40:  sub      esi,4                ; back up
        sub      edi,4                ; back up
        mov      ecx,5                ; ensure that ecx doesn't count out
        repe   cmpsb               ; find mismatch byte

;
; When we come to rcm50, esi points to the byte after the one that
; did not match, which is TWO after the last byte that did match.
;

rcm50:  dec      esi                ; back up
        sub      esi,RcmSource1      ; compute bytes that matched
        mov      eax,esi               ;
        pop      edi                ; restore registers
        pop      esi                ;
        stdRET _RtlCompareMemory

stdENDP _RtlCompareMemory

```

N.B.: this function uses a 32-bit word comparison (CMPSD) if the block size is a multiple of 4, or per-byte comparison (CMPSB) otherwise.

**CPUID** get information about the CPU's features. see also: ([1.24.6 on page 370](#)).

**DIV** unsigned division

**IDIV** signed division

**INT** (M): `INT x` is analogous to `PUSHF; CALL dword ptr [x*4]` in 16-bit environment. It was widely used in MS-DOS, functioning as a syscall vector. The registers AX/BX/CX/DX/SI/DI were filled with the arguments and then the flow jumped to the address in the Interrupt Vector Table (located at the beginning of the address space). It was popular because INT has a short opcode (2 bytes) and the program which needs some MS-DOS services is not bother to determine the address of the service's entry point. The interrupt handler returns the control flow to caller using the IRET instruction.

The most busy MS-DOS interrupt number was 0x21, serving a huge part of its API. See also: [Ralf Brown *Ralf Brown's Interrupt List*], for the most comprehensive interrupt lists and other MS-DOS information.

In the post-MS-DOS era, this instruction was still used as syscall both in Linux and Windows ([6.3 on page 746](#)), but was later replaced by the SYSENTER or SYSCALL instructions.

**INT 3** (M): this instruction is somewhat close to `INT`, it has its own 1-byte opcode (`0xCC`), and is actively used while debugging. Often, the debuggers just write the `0xCC` byte at the address of the breakpoint to be set, and when an exception is raised, the original byte is restored and the original

## 1. X86

instruction at this address is re-executed.

As of Windows NT, an `EXCEPTION_BREAKPOINT` exception is to be raised when the CPU executes this instruction. This debugging event may be intercepted and handled by a host debugger, if one is loaded. If it is not loaded, Windows offers to run one of the registered system debuggers. If MSVS<sup>8</sup> is installed, its debugger may be loaded and connected to the process. In order to protect from reverse engineering, a lot of anti-debugging methods check integrity of the loaded code.

MSVC has compiler intrinsic for the instruction: `__debugbreak()`<sup>9</sup>.

There is also a win32 function in kernel32.dll named `DebugBreak()`<sup>10</sup>, which also executes `INT 3`.

**IN** (M) input data from port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example ( [8.5.3 on page 831](#)).

**IRET** : was used in the MS-DOS environment for returning from an interrupt handler after it was called by the INT instruction. Equivalent to `POP tmp; POPF; JMP tmp`.

**LOOP** (M) decrement CX/ECX/RCX, jump if it is still not zero.

LOOP instruction was often used in DOS-code which works with external devices. To add small delay, this was done:

```
MOV    CX, nnnn
LABEL: LOOP   LABEL
```

Drawback is obvious: length of delay depends on CPU speed.

**OUT** (M) output data to port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example ( [8.5.3 on page 831](#)).

**POPA** (M) restores values of (R|E)DI, (R|E)SI, (R|E)BP, (R|E)BX, (R|E)DX, (R|E)CX, (R|E)AX registers from the stack.

**POPCNT** population count. Counts the number of 1 bits in the value.

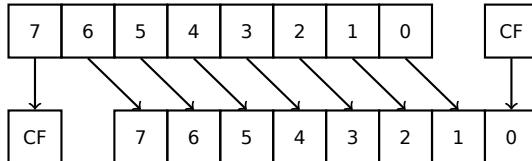
See: [2.6 on page 463](#).

**POPF** restore flags from the stack (AKA EFLAGS register)

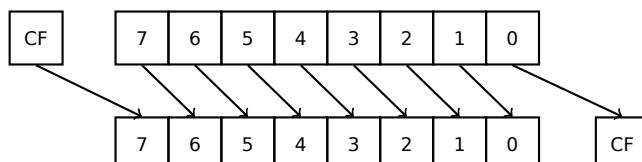
**PUSHA** (M) pushes the values of the (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI registers to the stack.

**PUSHF** push flags (AKA EFLAGS register)

**RCL** (M) rotate left via CF flag:

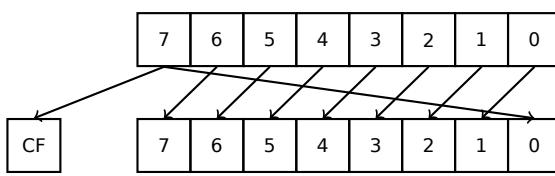


**RCR** (M) rotate right via CF flag:



**ROL/ROR** (M) cyclic shift

ROL: rotate left:

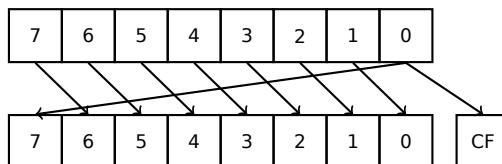


<sup>8</sup>Microsoft Visual Studio

<sup>9</sup>MSDN

<sup>10</sup>MSDN

ROR: rotate right:

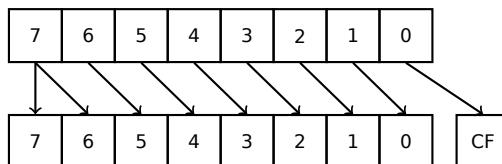


Despite the fact that almost all CPUs have these instructions, there are no corresponding operations in C/C++, so the compilers of these PLs usually do not generate these instructions.

For the programmer's convenience, at least MSVC has the pseudofunctions (compiler intrinsics) `_rotl()` and `_rotr()`<sup>11</sup>, which are translated by the compiler directly to these instructions.

**SAL** Arithmetic shift left, synonymous to `SHL`

**SAR** Arithmetic shift right



Hence, the sign bit always stays at the place of the **MSB**.

**SETcc** op: load 1 to operand (byte only) if the condition is true or zero otherwise. The condition codes are the same as in the Jcc instructions ( [.1.6 on page 1010](#)).

**STC** (M) set CF flag

**STD** (M) set DF flag. This instruction is not generated by compilers and generally rare. For example, it can be found in the `ntoskrnl.exe` Windows kernel file, in the hand-written memory copy routines.

**STI** (M) set IF flag

**SYSCALL** (AMD) call syscall ( [6.3 on page 746](#))

**SYSENTER** (Intel) call syscall ( [6.3 on page 746](#))

**UD2** (M) undefined instruction, raises exception. Used for testing.

**XCHG** (M) exchange the values in the operands

This instruction is rare: compilers don't generate it, because starting at Pentium, XCHG with address in memory in operand executes as if it has LOCK prefix ([Michael Abrash, *Graphics Programming Black Book*, 1997 chapter 19]). Perhaps, Intel engineers did so for compatibility with synchronizing primitives. Hence, XCHG starting at Pentium can be slow. On the other hand, XCHG was very popular in assembly language programmers. So if you see XCHG in code, it can be a sign that this piece of code is written manually. However, at least Borland Delphi compiler generates this instruction.

## FPU instructions

-R suffix in the mnemonic usually implies that the operands are reversed, -P suffix implies that one element is popped from the stack after the instruction's execution, -PP suffix implies that two elements are popped.

-P instructions are often useful when we do not need the value in the FPU stack to be present anymore after the operation.

**FABS** replace value in ST(0) by absolute value in ST(0)

**FADD** op: ST(0)=op+ST(0)

**FADD** ST(0), ST(i): ST(0)=ST(0)+ST(i)

**FADDP** ST(1)=ST(0)+ST(1); pop one element from the stack, i.e., the values in the stack are replaced by their sum

**FCHS** ST(0)=-ST(0)

<sup>11</sup>MSDN

**FCOM** compare ST(0) with ST(1)

**FCOM** op: compare ST(0) with op

**FCOMP** compare ST(0) with ST(1); pop one element from the stack

**FCOMPP** compare ST(0) with ST(1); pop two elements from the stack

**FDIVR** op: ST(0)=op/ST(0)

**FDIVR** ST(i), ST(j): ST(i)=ST(j)/ST(i)

**FDIVRP** op: ST(0)=op/ST(0); pop one element from the stack

**FDIVRP** ST(i), ST(j): ST(i)=ST(j)/ST(i); pop one element from the stack

**FDIV** op: ST(0)=ST(0)/op

**FDIV** ST(i), ST(j): ST(i)=ST(i)/ST(j)

**FDIVP** ST(1)=ST(0)/ST(1); pop one element from the stack, i.e., the dividend and divisor values in the stack are replaced by quotient

**FILD** op: convert integer and push it to the stack.

**FIST** op: convert ST(0) to integer op

**FISTP** op: convert ST(0) to integer op; pop one element from the stack

**FLD1** push 1 to stack

**FLDCW** op: load FPU control word ([.1.3 on page 1006](#)) from 16-bit op.

**FLDZ** push zero to stack

**FLD** op: push op to the stack.

**FMUL** op: ST(0)=ST(0)\*op

**FMUL** ST(i), ST(j): ST(i)=ST(i)\*ST(j)

**FMULP** op: ST(0)=ST(0)\*op; pop one element from the stack

**FMULP** ST(i), ST(j): ST(i)=ST(i)\*ST(j); pop one element from the stack

**FSINCOS** : tmp=ST(0); ST(1)=sin(tmp); ST(0)=cos(tmp)

**FSQRT** : ST(0) =  $\sqrt{ST(0)}$

**FSTCW** op: store FPU control word ([.1.3 on page 1006](#)) into 16-bit op after checking for pending exceptions.

**FNSTCW** op: store FPU control word ([.1.3 on page 1006](#)) into 16-bit op.

**FSTSW** op: store FPU status word ([.1.3 on page 1007](#)) into 16-bit op after checking for pending exceptions.

**FNSTSW** op: store FPU status word ([.1.3 on page 1007](#)) into 16-bit op.

**FST** op: copy ST(0) to op

**FSTP** op: copy ST(0) to op; pop one element from the stack

**FSUBR** op: ST(0)=op-ST(0)

**FSUBR** ST(0), ST(i): ST(0)=ST(i)-ST(0)

**FSUBRP** ST(1)=ST(0)-ST(1); pop one element from the stack, i.e., the value in the stack is replaced by the difference

**FSUB** op: ST(0)=ST(0)-op

**FSUB** ST(0), ST(i): ST(0)=ST(0)-ST(i)

**FSUBP** ST(1)=ST(1)-ST(0); pop one element from the stack, i.e., the value in the stack is replaced by the difference

**FUCOM** ST(i): compare ST(0) and ST(i)

**FUCOM** compare ST(0) and ST(1)

**FUCOMP** compare ST(0) and ST(1); pop one element from stack.

---

**FUCOMPP** compare ST(0) and ST(1); pop two elements from stack.

The instructions perform just like FCOM, but an exception is raised only if one of the operands is SNaN, while QNaN numbers are processed smoothly.

**FXCH** ST(i) exchange values in ST(0) and ST(i)

**FXCH** exchange values in ST(0) and ST(1)

### Instructions having printable ASCII opcode

(In 32-bit mode).

These can be suitable for shellcode construction. See also: [8.12.1 on page 912](#).

ASCII character	hexadecimal code	x86 instruction
0	30	XOR
1	31	XOR
2	32	XOR
3	33	XOR
4	34	XOR
5	35	XOR
7	37	AAA
8	38	CMP
9	39	CMP
:	3a	CMP
;	3b	CMP
<	3c	CMP
=	3d	CMP
?	3f	AAS
@	40	INC
A	41	INC
B	42	INC
C	43	INC
D	44	INC
E	45	INC
F	46	INC
G	47	INC
H	48	DEC
I	49	DEC
J	4a	DEC
K	4b	DEC
L	4c	DEC
M	4d	DEC
N	4e	DEC
O	4f	DEC
P	50	PUSH
Q	51	PUSH
R	52	PUSH
S	53	PUSH
T	54	PUSH
U	55	PUSH
V	56	PUSH
W	57	PUSH
X	58	POP
Y	59	POP
Z	5a	POP
[	5b	POP
\	5c	POP
]	5d	POP
^	5e	POP
~	5f	POP
a	60	PUSHA
f	61	POPA
	66	(in 32-bit mode) switch to

g	67	16-bit operand size in 32-bit mode) switch to 16-bit address size	
h	68	PUSH	
i	69	IMUL	
j	6a	PUSH	
k	6b	IMUL	
p	70	JO	
q	71	JNO	
r	72	JB	
s	73	JAE	
t	74	JE	
u	75	JNE	
v	76	JBE	
w	77	JA	
x	78	JS	
y	79	JNS	
z	7a	JP	

In summary: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSH, XOR.

## .1.7 npad

It is an assembly language macro for aligning labels on a specific boundary.

That's often needed for the busy labels to where the control flow is often passed, e.g., loop body starts. So the CPU can load the data or code from the memory effectively, through the memory bus, cache lines, etc.

Taken from `listing.inc` (MSVC):

By the way, it is a curious example of the different `NOP` variations. All these instructions have no effects whatsoever, but have a different size.

Having a single idle instruction instead of couple of NOP-s, is accepted to be better for CPU performance.

```
; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
    nop
else
    if size eq 2
        mov edi, edi
    else
        if size eq 3
            ; lea ecx, [ecx+00]
            DB 8DH, 49H, 00H
        else
            if size eq 4
                ; lea esp, [esp+00]
                DB 8DH, 64H, 24H, 00H
            else
                if size eq 5
                    add eax, DWORD PTR 0
                else
                    if size eq 6
```

## 2. ARM

## .2 ARM

## **.2.1 Terminology**

ARM was initially developed as 32-bit CPU, so that's why a *word* here, unlike x86, is 32-bit.

**byte** 8-bit. The DB assembly directive is used for defining variables and arrays of bytes.

**halfword** 16-bit. DCW assembly directive —"—.

## .2. ARM

---

**word** 32-bit. DCD assembly directive —"——.

**doubleword** 64-bit.

**quadword** 128-bit.

### .2.2 Versions

- ARMv4: Thumb mode introduced.
- ARMv6: used in iPhone 1st gen., iPhone 3G (Samsung 32-bit RISC ARM 1176JZ(F)-S that supports Thumb-2)
- ARMv7: Thumb-2 was added (2003). was used in iPhone 3GS, iPhone 4, iPad 1st gen. (ARM Cortex-A8), iPad 2 (Cortex-A9), iPad 3rd gen.
- ARMv7s: New instructions added. Was used in iPhone 5, iPhone 5c, iPad 4th gen. (Apple A6).
- ARMv8: 64-bit CPU, [AKA](#) ARM64 [AKA](#) AArch64. Was used in iPhone 5S, iPad Air (Apple A7). There is no Thumb mode in 64-bit mode, only ARM (4-byte instructions).

### .2.3 32-bit ARM (AArch32)

#### General purpose registers

- R0— function result is usually returned using R0
- R1...R12—[GPRs](#)
- R13—[AKA](#) SP ([stack pointer](#))
- R14—[AKA](#) LR ([link register](#))
- R15—[AKA](#) PC (program counter)

R0 - R3 are also called “scratch registers”: the function’s arguments are usually passed in them, and the values in them are not required to be restored upon the function’s exit.

#### Current Program Status Register (CPSR)

Bit	Description
0..4	M—processor mode
5	T—Thumb state
6	F—FIQ disable
7	I—IRQ disable
8	A—imprecise data abort disable
9	E—data endianness
10..15, 25, 26	IT—if-then state
16..19	GE—greater-than-or-equal-to
20..23	DNM—do not modify
24	J—Java state
27	Q—sticky overflow
28	V—overflow
29	C—carry/borrow/extend
30	Z—zero bit
31	N—negative/less than

#### VFP (floating point) and NEON registers

0..31 <sup>bits</sup>	32..64	65..96	97..127
Q0 <sup>128 bits</sup>			
D0 <sup>64 bits</sup>			D1
S0 <sup>32 bits</sup>	S1	S2	S3

## .2. ARM

---

S-registers are 32-bit, used for the storage of single precision numbers.

D-registers are 64-bit ones, used for the storage of double precision numbers.

D- and S-registers share the same physical space in the CPU—it is possible to access a D-register via the S-registers (it is senseless though).

Likewise, the **NEON** Q-registers are 128-bit ones and share the same physical space in the CPU with the other floating point registers.

In VFP 32 S-registers are present: S0..S31.

In VFPv2 there 16 D-registers are added, which in fact occupy the same space as S0..S31.

In VFPv3 (**NEON** or “Advanced SIMD”) there are 16 more D-registers, D0..D31, but the D16..D31 registers are not sharing space with any other S-registers.

In **NEON** or “Advanced SIMD” another 16 128-bit Q-registers were added, which share the same space as D0..D31.

## .2.4 64-bit ARM (AArch64)

### General purpose registers

The number of registers was doubled since AArch32.

- X0— function result is usually returned using X0
- X0...X7—Function arguments are passed here.
- X8
- X9...X15—are temporary registers, the callee function can use and not restore them.
- X16
- X17
- X18
- X19...X29—callee function can use them, but must restore them upon exit.
- X29—used as **FP** (at least GCC)
- X30—“Procedure Link Register” **AKA LR (link register)**.
- X31—register always contains zero **AKA XZR or “Zero Register”**. It’s 32-bit part is called **WZR**.
- **SP**, not a general purpose register anymore.

See also: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]<sup>12</sup>.

The 32-bit part of each X-register is also accessible via W-registers (W0, W1, etc.).

High 32-bit part	low 32-bit part
X0	
	W0

## .2.5 Instructions

There is a **-S** suffix for some instructions in ARM, indicating that the instruction sets the flags according to the result. Instructions which lacks this suffix are not modify flags. For example **ADD** unlike **ADDS** will add two numbers, but the flags will not be touched. Such instructions are convenient to use between **CMP** where the flags are set and, e.g. conditional jumps, where the flags are used. They are also better in terms of data dependency analysis (because less number of registers are modified during execution).

<sup>12</sup>Also available as <http://go.yurichev.com/17287>

**Conditional codes table**

Code	Description	Flags
EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS AKA HS (Higher or Same)	Carry set / Unsigned, Greater than, equal	C == 1
CC AKA LO (LOwer)	Carry clear / Unsigned, Less than	C == 0
MI	Minus, negative / Less than	N == 1
PL	Plus, positive or zero / Greater than, equal	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
HI	Unsigned higher / Greater than	C == 1 and Z == 0
LS	Unsigned lower or same / Less than or equal	C == 0 or Z == 1
GE	Signed greater than or equal / Greater than or equal	N == V
LT	Signed less than / Less than	N != V
GT	Signed greater than / Greater than	Z == 0 and N == V
LE	Signed less than or equal / Less than, equal	Z == 1 or N != V
None / AL	Always	Any

## .3 MIPS

### .3.1 Registers

( O32 calling convention )

#### General purpose registers GPR

Number	Pseudoname	Description
\$0	\$ZERO	Always zero. Writing to this register is like <a href="#">NOP</a> .
\$1	\$AT	Used as a temporary register for assembly macros and pseudo instructions.
\$2 ...\$3	\$V0 ...\$V1	Function result is returned here.
\$4 ...\$7	\$A0 ...\$A3	Function arguments.
\$8 ...\$15	\$T0 ...\$T7	Used for temporary data.
\$16 ...\$23	\$S0 ...\$S7	Used for temporary data*.
\$24 ...\$25	\$T8 ...\$T9	Used for temporary data.
\$26 ...\$27	\$K0 ...\$K1	Reserved for <a href="#">OS</a> kernel.
\$28	\$GP	Global Pointer**.
\$29	\$SP	<a href="#">SP</a> *.
\$30	\$FP	<a href="#">FP</a> *
\$31	\$RA	<a href="#">RA</a> .
n/a	PC	<a href="#">PC</a> .
n/a	HI	high 32 bit of multiplication or division remainder***.
n/a	LO	low 32 bit of multiplication and division remainder***.

#### Floating-point registers

Name	Description
\$F0..\$F1	Function result returned here.
\$F2..\$F3	Not used.
\$F4..\$F11	Used for temporary data.
\$F12..\$F15	First two function arguments.
\$F16..\$F19	Used for temporary data.
\$F20..\$F31	Used for temporary data*.

\*—Callee must preserve the value.

\*\*—Callee must preserve the value ( except in [PIC](#) code).

#### 4. SOME GCC LIBRARY FUNCTIONS

\*\*\*—accessible using the MFHI and MFL0 instructions.

### .3.2 Instructions

There are 3 kinds of instructions:

- R-type: those which have 3 registers. R-instruction usually have the following form:

instruction destination, source1, source2
---

One important thing to keep in mind is that when the first and second register are the same, IDA may show the instruction in its shorter form:

instruction destination/source1, source2
--

That somewhat reminds us of the Intel syntax for x86 assembly language.

- I-type: those which have 2 registers and a 16-bit immediate value.
- J-type: jump/branch instructions, have 26 bits for encoding the offset.

### Jump instructions

What is the difference between B- instructions (BEQ, B, etc.) and J- ones (JAL, JALR, etc.)?

The B-instructions have an I-type, hence, the B-instructions' offset is encoded as a 16-bit immediate. JR and JALR are R-type and jump to an absolute address specified in a register. J and JAL are J-type, hence the offset is encoded as a 26-bit immediate.

In short, B-instructions can encode a condition (B is in fact pseudo instruction for BEQ \$ZERO, \$ZERO, LABEL), while J-instructions can't.

## 4 Some GCC library functions

name	meaning
__divdi3	signed division
__moddi3	getting remainder (modulo) of signed division
__udivdi3	unsigned division
__umoddi3	getting remainder (modulo) of unsigned division

## 5 Some MSVC library functions

ll in function name stands for “long long”, e.g., a 64-bit data type.

name	meaning
__alldiv	signed division
__allmul	multiplication
__allrem	remainder of signed division
__allshl	shift left
__allshr	signed shift right
__aulldiv	unsigned division
__aullrem	remainder of unsigned division
__aullshr	unsigned shift right

## .6. CHEATSHEETS

Multiplication and shift left procedures are the same for both signed and unsigned numbers, hence there is only one function for each operation here. .

The source code of these function can be found in the installed [MSVS](#), in `VC/crt/src/intel/*.asm` `VC/crt/src/intel/*.asm`.

## .6 Cheatsheets

### .6.1 IDA

Hot-keys cheatsheet:

key	meaning
Space	switch listing and graph view
C	convert to code
D	convert to data
A	convert to string
*	convert to array
U	undefine
O	make offset of operand
H	make decimal number
R	make char
B	make binary number
Q	make hexadecimal number
N	rename identifier
?	calculator
G	jump to address
:	add comment
Ctrl-X	show references to the current function, label, variable (incl. in local stack)
X	show references to the function, label, variable, etc.
Alt-I	search for constant
Ctrl-I	search for the next occurrence of constant
Alt-B	search for byte sequence
Ctrl-B	search for the next occurrence of byte sequence
Alt-T	search for text (including instructions, etc.)Text suchen (inkl. Anweisungen, usw.)
Ctrl-T	search for the next occurrence of text
Alt-P	edit current function
Enter	jump to function, variable, etc.
Esc	get back
Num -	fold function or selected area
Num +	unhide function or area

Function/area folding may be useful for hiding function parts when you realize what they do. . this is used in my [script<sup>13</sup>](#) for hiding some often used patterns of inline code. .

### .6.2 OllyDbg

Hot-keys cheatsheet:

hot-key	meaning
F7	trace into
F8	step over
F9	run
Ctrl-F2	restart

### .6.3 MSVC

Some useful options which were used through this book. .

<sup>13</sup>[GitHub](#)

## .6. CHEATSHEETS

option	meaning
/O1	minimize space
/Ob0	no inline expansion
/Ox	maximum optimizations
/GS-	disable security checks (buffer overflows)
/Fa(file)	generate assembly listing
/Zi	enable debugging information
/Zp(n)	pack structs on <i>n</i> -byte boundary
/MD	produced executable will use MSVCR*.DLL MSVCR*.DLL

Some information about MSVC versions: [5.1.1 on page 705](#).

## .6.4 GCC

Some useful options which were used through this book.

option	meaning
-Os	code size optimization
-O3	maximum optimization
-regparm=	how many arguments are to be passed in registers
-o file	set name of output file
-g	produce debugging information in resulting executable
-S	generate assembly listing file
-masm=intel	produce listing in Intel syntax
-fno-inline	do not inline functions

## .6.5 GDB

Some of commands we used in this book:

## .6. CHEATSHEETS

option	meaning
break filename.c:number	set a breakpoint on line number in source code
break function	set a breakpoint on function
break *address	set a breakpoint on address
b	—
p variable	print value of variable
run	run
r	—
cont	continue execution
c	—
bt	print stack
set disassembly-flavor intel	set Intel syntax
disas	disassemble current function
disas function	disassemble function
disas function,+50	disassemble portion
disas \$eip,+0x10	—
disas/r	disassemble with opcodes
info registers	print all registers
info float	print FPU-registers
info locals	dump local variables (if known)
x/w ...	dump memory as 32-bit word
x/w \$rdi	dump memory as 32-bit word
x/10w ...	at address in RDI
x/s ...	dump 10 memory words
x/i ...	dump memory as string
x/10c ...	dump memory as code
x/b ...	dump 10 characters
x/h ...	dump bytes
x/g ...	dump 16-bit halfwords
finish	dump giant (64-bit) words
next	execute till the end of function
step	next instruction (don't dive into functions)
set step-mode on	next instruction (dive into functions)
frame n	do not use line number information while stepping
info break	switch stack frame
del n	list of breakpoints
set args ...	delete breakpoint
	set command-line arguments

## **Acronyms used**

.6. CHEATSHEETS	
<b>OS</b> Operating System .....	xvi
<b>OOP</b> Object-Oriented Programming.....	546
<b>PL</b> Programming language.....	xiv
<b>PRNG</b> Pseudorandom number generator.....	ix
<b>ROM</b> Read-only memory .....	81
<b>ALU</b> Arithmetic logic unit.....	27
<b>PID</b> Program/process ID.....	806
<b>LF</b> Line feed (10 or '\n' in C/C++).....	525
<b>CR</b> Carriage return (13 or '\r' in C/C++).....	525
<b>LIFO</b> Last In First Out .....	30
<b>MSB</b> Most significant bit .....	317
<b>LSB</b> Least significant bit	
<b>NSA</b> National Security Agency .....	463
<b>RA</b> Return Address .....	22
<b>PE</b> Portable Executable .....	5
<b>SP</b> stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.....	19
<b>DLL</b> Dynamic-link library.....	756
<b>PC</b> Program Counter. IP/EIP/RIP in x86/64. PC in ARM.....	20
<b>LR</b> Link Register .....	6
<b>IDA</b> Interactive Disassembler and debugger developed by <a href="#">Hex-Rays</a> .....	6
<b>IAT</b> Import Address Table.....	756
<b>INT</b> Import Name Table.....	756
<b>RVA</b> Relative Virtual Address .....	756
<b>VA</b> Virtual Address.....	756
<b>OEP</b> Original Entry Point .....	746
<b>MSVC</b> Microsoft Visual C++	

## .6. CHEATSHEETS

<b>MSVS</b> Microsoft Visual Studio .....	1017
<b>ASLR</b> Address Space Layout Randomization .....	626
<b>MFC</b> Microsoft Foundation Classes .....	759
<b>TLS</b> Thread Local Storage .....	xiv
<b>AKA</b> Also Known As .....	30
<b>CRT</b> C runtime library .....	10
<b>CPU</b> Central processing unit .....	xvi
<b>FPU</b> Floating-point unit .....	v
<b>CISC</b> Complex instruction set computing .....	20
<b>RISC</b> Reduced instruction set computing .....	2
<b>GUI</b> Graphical user interface .....	752
<b>RTTI</b> Run-time type information .....	561
<b>BSS</b> Block Started by Symbol .....	25
<b>SIMD</b> Single instruction, multiple data .....	196
<b>BSOD</b> Blue Screen of Death .....	746
<b>DBMS</b> Database management systems .....	xiv
<b>ISA</b> Instruction Set Architecture .....	2
<b>HPC</b> High-Performance Computing .....	517
<b>SEH</b> Structured Exception Handling .....	37
<b>ELF</b> Executable file format widely used in *NIX systems including Linux .....	xiv
<b>TIB</b> Thread Information Block .....	284
<b>PIC</b> Position Independent Code: <a href="#">6.4.1 on page 747</a> .....	xiv
<b>NAN</b> Not a Number .....	1008
<b>NOP</b> No OPeration .....	6
<b>BEQ</b> (PowerPC, ARM) Branch if Equal .....	96
<b>BNE</b> (PowerPC, ARM) Branch if Not Equal .....	209

.6. CHEATSHEETS	
<b>BLR</b> (PowerPC) Branch to Link Register.....	815
<b>XOR</b> eXclusive OR.....	1014
<b>MCU</b> Microcontroller unit.....	494
<b>RAM</b> Random-access memory .....	3
<b>GCC</b> GNU Compiler Collection .....	4
<b>EGA</b> Enhanced Graphics Adapter .....	993
<b>VGA</b> Video Graphics Array .....	993
<b>API</b> Application programming interface .....	635
<b>ASCII</b> American Standard Code for Information Interchange .....	450
<b>ASCIIZ</b> ASCII Zero (null-terminated ASCII string ).....	93
<b>IA64</b> Intel Architecture 64 (Itanium): <a href="#">10.5 on page 990</a> .....	464
<b>EPIC</b> Explicitly parallel instruction computing.....	990
<b>OOE</b> Out-of-order execution .....	465
<b>MSDN</b> Microsoft Developer Network.....	628
<b>STL</b> (C++) Standard Template Library: <a href="#">3.19.4 on page 563</a> .....	568
<b>PODT</b> (C++) Plain Old Data Type .....	579
<b>HDD</b> Hard disk drive.....	591
<b>VM</b> Virtual Memory	
<b>WRK</b> Windows Research Kernel .....	722
<b>GPR</b> General Purpose Registers .....	2
<b>SSDT</b> System Service Dispatch Table.....	746
<b>RE</b> Reverse Engineering.....	998
<b>RAID</b> Redundant Array of Independent Disks .....	vii
<b>BCD</b> Binary-coded decimal .....	449
<b>BOM</b> Byte order mark.....	712
<b>GDB</b> GNU debugger.....	48

.6. CHEATSHEETS	
<b>FP</b> Frame Pointer	24
<b>MBR</b> Master Boot Record	718
<b>JPE</b> Jump Parity Even (x86 instruction)	239
<b>CIDR</b> Classless Inter-Domain Routing	484
<b>STMFD</b> Store Multiple Full Descending (ARM instruction)	
<b>LDMFD</b> Load Multiple Full Descending (ARM instruction)	
<b>STMED</b> Store Multiple Empty Descending (ARM instruction)	31
<b>LDMED</b> Load Multiple Empty Descending (ARM instruction)	31
<b>STMFA</b> Store Multiple Full Ascending (ARM instruction)	31
<b>LDMFA</b> Load Multiple Full Ascending (ARM instruction)	31
<b>STMEA</b> Store Multiple Empty Ascending (ARM instruction)	31
<b>LDMEA</b> Load Multiple Empty Ascending (ARM instruction)	31
<b>APSR</b> (ARM) Application Program Status Register	262
<b>FPSCR</b> (ARM) Floating-Point Status and Control Register	262
<b>RFC</b> Request for Comments	716
<b>TOS</b> Top Of Stack	667
<b>LVA</b> (Java) Local Variable Array	673
<b>JVM</b> Java virtual machine	.ix
<b>JIT</b> Just-in-time compilation	666
<b>CDFS</b> Compact Disc File System	729
<b>CD</b> Compact Disc	
<b>ADC</b> Analog-to-digital converter	725
<b>EOF</b> End of file	85
<b>DIY</b> Do It Yourself	631
<b>MMU</b> Memory management unit	625
<b>CPRNG</b> Cryptographically secure Pseudorandom Number Generator	955

## *.6. CHEATSHEETS*

---

<b>DES</b> Data Encryption Standard .....	450
<b>MIME</b> Multipurpose Internet Mail Extensions .....	450
<b>DBI</b> Dynamic Binary Instrumentation .....	523
<b>XML</b> Extensible Markup Language .....	640
<b>JSON</b> JavaScript Object Notation .....	640
<b>URL</b> Uniform Resource Locator .....	4

# Glossary

**heap** usually, a big chunk of memory provided by the OS so that applications can divide it by themselves as they wish. malloc()/free() work with the heap. [31](#), [349](#), [564](#), [566](#), [567](#), [579](#), [581](#), [596](#), [597](#), [641](#), [755](#), [756](#)

**real number** numbers which may contain a dot. this is *float* and *double* in C/C++ . [218](#)

**decrement** Decrease by 1 . [19](#), [185](#), [203](#), [442](#), [732](#), [857](#), [1010](#), [1013](#), [1017](#)

**increment** Increase by 1 . [16](#), [20](#), [185](#), [189](#), [203](#), [209](#), [327](#), [330](#), [442](#), [854](#), [1010](#)

**integral data type** usual numbers, but not a real ones. may be used for passing variables of boolean data type and enumerations . [232](#)

**product** Multiplication result . [99](#), [224](#), [227](#), [408](#), [433](#), [456](#)

**arithmetic mean** a sum of all values divided by their count . [518](#)

**stack pointer** A register pointing to a place in the stack. [9](#), [11](#), [20](#), [30](#), [31](#), [35](#), [42](#), [54](#), [56](#), [73](#), [100](#), [548](#), [655](#), [733–736](#), [1005](#), [1011](#), [1023](#), [1031](#)

**tail call** It is when the compiler (or interpreter) transforms the recursion (with which it is possible: *tail recursion*) into an iteration for efficiency : [wikipedia](#). [480](#)

**quotient** Division result . [218](#), [220](#), [222](#), [223](#), [227](#), [432](#), [496](#), [520](#)

**anti-pattern** Generally considered as bad practice . [33](#), [76](#), [465](#)

**atomic operation** “*ατομός*” stands for “indivisible” in Greek, so an atomic operation is guaranteed not to be interrupted by other threads . [651](#), [787](#)

**basic block** a group of instructions that do not have jump/branch instructions, and also don't have jumps inside the block from the outside. In IDA it looks just like as a list of instructions without empty lines . [696](#), [993](#), [994](#)

**callee** A function being called by another . [33](#), [46](#), [67](#), [86](#), [98](#), [100](#), [102](#), [421](#), [465](#), [548](#), [655](#), [733–736](#), [738](#), [739](#), [1025](#)

**caller** A function calling another . [5–8](#), [10](#), [30](#), [46](#), [86](#), [98](#), [99](#), [101](#), [109](#), [156](#), [421](#), [468](#), [548](#), [733](#), [735](#), [736](#), [739](#)

**compiler intrinsic** A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a call to it. Often, it's a pseudofunction for a specific CPU instruction. Read more: ([10.3 on page 988](#)) . [1017](#)

**CP/M** Control Program for Microcomputers: a very basic disk OS used before MS-DOS. [912](#)

**dongle** Dongle is a small piece of hardware connected to LPT printer port (in past) or to USB . Its function was similar to a security token, it has some memory and, sometimes, a secret (crypto-)hashing algorithmi . [814](#)

**endianness** Byte order: [2.7 on page 463](#). [22](#), [78](#), [347](#), [1014](#)

**GiB** Gibibyte:  $2^{30}$  or 1024 mebibytes or 1073741824 bytes . [15](#)

**jump offset** a part of the JMP or Jcc instruction's opcode, to be added to the address of the next instruction, and this is how the new PC is calculated. May be negative as well . [94](#), [134](#), [1010](#)

**kernel mode** A restrictions-free CPU mode in which the OS kernel and drivers execute. cf. [user mode](#). [1037](#)

**leaf function** A function which does not call any other function . [29](#), [32](#)

**link register** (RISC) A register where the return address is usually stored. This makes it possible to call leaf functions without using the stack, i.e., faster . [32](#), [815](#), [1023](#), [1024](#)

**loop unwinding** It is when a compiler, instead of generating loop code for  $n$  iterations, generates just  $n$  copies of the loop body, in order to get rid of the instructions for loop maintenance . [187](#)

**name mangling** used at least in C++, where the compiler needs to encode the name of class, method and argument types in one string, which will become the internal name of the function. You can read more about it here : [3.19.1 on page 546](#). [546](#), [706](#), [707](#)

**NaN** not a number: a special cases for floating point numbers, usually signaling about errors . [235](#), [257](#), [992](#)

**NEON** AKA “Advanced SIMD”—SIMD from ARM. [1024](#)

**NOP** “no operation”, idle instruction . [732](#)

**NTAPI** API available only in the Windows NT line. Largely not documented by Microsoft . [793](#)

**padding** *Padding* in English language means to stuff a pillow with something to give it a desired (bigger) form. In computer science, padding means to add more bytes to a block so it will have desired size, like  $2^n$  bytes. . [714](#)

**PDB** (Win32) Debugging information file, usually just function names, but sometimes also function arguments and local variables names . [705](#), [758](#), [793](#), [794](#), [801](#), [802](#), [895](#)

**POKE** BASIC language instruction for writing a byte at a specific address . [732](#)

**register allocator** The part of the compiler that assigns CPU registers to local variables . [202](#), [307](#), [421](#)

**reverse engineering** act of understanding how the thing works, sometimes in order to clone it . [v](#), [1017](#)

**security cookie** A random value, different at each execution. You can read more about it here : [1.20.3 on page 283](#). [777](#)

**stack frame** A part of the stack that contains information specific to the current function: local variables, function arguments, RA, etc. . [68](#), [98](#), [99](#), [476](#), [777](#)

**stdout** standard output. [22](#), [36](#), [156](#)

**thunk function** Tiny function with a single role: call another function . [23](#), [394](#), [815](#), [824](#)

**tracer** My own simple debugging tool. You can read more about it here : [7.2.1 on page 789](#). [190–192](#), [709](#), [720](#), [723](#), [773](#), [782](#), [897](#), [903](#), [907](#), [908](#), [910](#), [987](#)

**user mode** A restricted CPU mode in which it all application software code is executed. cf. [kernel mode](#). [831](#), [1037](#)

**Windows NT** Windows NT, 2000, XP, Vista, 7, 8, 10. [293](#), [419](#), [654](#), [713](#), [746](#), [757](#), [786](#), [915](#), [1017](#)

**word** data type fitting in [GPR](#). In the computers older than PCs, the memory size was often measured in words rather than bytes . [449–452](#), [456](#), [457](#), [570](#), [642](#)

**xoring** often used in the English language, which implying applying the [XOR](#) operation . [777](#), [826](#), [829](#)

# Index

.NET, 762  
0x0BADF00D, 76  
0xCCCCCCCC, 76

Ada, 107  
AES, 867  
Alpha AXP, 3  
AMD, 738  
Angry Birds, 263, 264  
Apollo Guidance Computer, 211  
ARM, 209, 540, 815, 1022  
    Addressing modes, 441  
    ARM mode, 2  
    ARM1, 452  
    armel, 228  
    armhf, 228  
    Condition codes, 137  
    D-registers, 227, 1023  
    Data processing instructions, 498  
    DCB, 20  
    hard float, 228  
    if-then block, 263  
Instructions  
    ADC, 400  
    ADD, 21, 106, 137, 193, 321, 334, 498, 1024  
    ADDAL, 137  
    ADDCC, 175  
    ADDS, 104, 400, 1024  
    ADR, 19, 137  
    ADRcc, 137, 164, 165, 465  
    ADRP/ADD pair, 24, 55, 82, 290, 303, 444  
    ANDcc, 536  
    ASR, 337  
    ASRS, 315, 499  
    B, 54, 137, 138  
    Bcc, 96, 97, 149  
    BCS, 138, 265  
    BEQ, 95, 164  
    BGE, 138  
    BIC, 315, 316, 320, 339  
    BL, 20–24, 137, 445  
    BLcc, 137  
    BLE, 138  
    BLS, 138  
    BLT, 193  
    BLX, 22  
    BNE, 138  
    BX, 104, 177  
    CMP, 95, 96, 137, 165, 175, 193, 334, 1024  
    CSEL, 146, 151, 153, 335  
    EOR, 320  
    FCMPE, 265  
    FCSEL, 265  
    FMOV, 443  
    FMRS, 321  
    IT, 153, 263, 286  
    LDMccFD, 137  
    LDMEA, 31  
    LDMED, 31  
    LDMFA, 31  
    LDMFD, 20, 31, 137  
    LDP, 25  
    LDR, 56, 73, 81, 272, 289, 441  
    LDRB, 365  
    LDRB.W, 209  
    LDRSB, 209  
    LEA, 465  
    LSL, 334, 337  
    LSL.W, 334  
    LSLR, 536  
    LSLS, 273, 321, 536  
    LSR, 337  
    LSRS, 321  
    MADD, 104  
    MLA, 104  
    MOV, 8, 20, 21, 334, 498  
    MOVcc, 149, 153  
    MOVK, 443  
    MOVT, 21, 498  
    MOVT.W, 22  
    MOVW, 22  
    MUL, 106  
    MULS, 104  
    MVNS, 210  
    NEG, 506  
    ORR, 315  
    POP, 19–21, 30, 32  
    PUSH, 21, 30, 32  
    RET, 25  
    RSB, 143, 299, 334, 505  
    SBC, 400  
    SMMUL, 498  
    STMEA, 31  
    STMED, 31  
    STMFA, 31, 57  
    STMFD, 19, 31  
    STMIA, 56  
    STMIB, 57  
    STP, 24, 55  
    STR, 55, 272  
    SUB, 56, 299, 334  
    SUBcc, 536  
    SUBEQ, 210  
    SUBS, 400  
    SXTB, 366  
    SXTW, 303  
    TEST, 202

- 
- TST, 308, 334
  - VADD, 227
  - VDIV, 227
  - VLDR, 227
  - VMOV, 227, 262
  - VMOVGT, 262
  - VMRS, 262
  - VMUL, 227
  - XOR, 143, 321
  - Leaf function, 32
  - Mode switching, 104, 177
  - mode switching, 22
  - Optional operators
    - ASR, 334, 498
    - LSL, 272, 299, 334, 443
    - LSR, 334, 498
    - ROR, 334
    - RRX, 334
  - Pipeline, 175
  - Registers
    - APSR, 262
    - FPSCR, 262
    - Link Register, 20, 32, 54, 177, 1023
    - R0, 107, 1023
    - scratch registers, 209, 1023
    - X0, 1024
    - Z, 96, 1023
  - S-registers, 227, 1023
  - soft float, 228
  - Thumb mode, 2, 138, 176
  - Thumb-2 mode, 2, 176, 263, 264
  - ARM64
    - lo12, 55
  - ASLR, 756
  - AT&T syntax, 12, 37
  - AWK, 722
    - Base address, 756
    - base32, 715
    - Base64, 714
    - base64, 716, 864, 955
    - base64scanner, 463, 715
    - bash, 108
    - BASIC
      - POKE, 732
    - binary grep, 720, 788
    - Binary Ninja, 788
    - Binary tree, 586
    - BIND.EXE, 761
    - BinNavi, 788
    - binutils, 381
    - Binwalk, 949
    - Bitcoin, 648
    - Booth's multiplication algorithm, 217
    - Borland C++, 623
    - Borland C++Builder, 707
    - Borland Delphi, 707, 711, 987, 1018
    - BSoD, 746
    - BSS, 757
    - Buffer Overflow, 275, 282, 777
    - C language elements
      - C99, 110
      - bool, 304
      - restrict, 515
    - variable length arrays, 287
    - const, 9, 81
    - for, 185, 482
    - if, 125, 155
    - Pointers, 67, 73, 110, 385, 421, 615
    - Post-decrement, 441
    - Post-increment, 441
    - Pre-decrement, 441
    - Pre-increment, 441
    - return, 10, 86, 109
    - switch, 154, 155, 164
    - while, 201

C standard library

    - alloca(), 35, 287, 465, 768
    - assert(), 292, 717
    - atexit(), 569
    - atoi(), 499, 886
    - calloc(), 847
    - close(), 750
    - exit(), 468
    - fread(), 638
    - free(), 465, 597
    - fseek(), 846
    - ftell(), 846
    - fwrite(), 638
    - getenv(), 887
    - localtime(), 664
    - localtime\_r(), 356
    - longjmp, 643
    - longjmp(), 156
    - malloc(), 349, 465, 597
    - memchr(), 1013
    - memcmp(), 454, 513, 718, 1015
    - memcpy(), 12, 67, 511, 641, 1012
    - memmove(), 641
    - memset(), 267, 510, 907, 1013, 1014
    - open(), 750
    - pow(), 230
    - puts(), 21
    - qsort(), 386
    - rand(), 339, 708, 799, 801, 835
    - read(), 638, 750
    - realloc(), 465
    - scanf(), 66
    - setjmp, 643
    - strcat(), 514
    - strcmp(), 454, 507, 750
    - strcpy(), 12, 509, 836
    - strlen(), 201, 417, 509, 526, 1013
    - strstr(), 468
    - strtok, 212
    - time(), 664
    - tolower(), 852
    - toupper(), 533
    - va\_arg, 519
    - va\_list, 522
    - vprintf, 522
    - write(), 638

C++, 898

    - C++11, 579, 741
    - exceptions, 768
    - ostream, 561
    - References, 562
    - RTTI, 561

---

STL, 705  
     std::forward\_list, 579  
     std::list, 570  
     std::map, 586  
     std::set, 586  
     std::string, 563  
     std::vector, 579  
 C11, 741  
 Callbacks, 385  
 Canary, 283  
 cdecl, 42, 733  
 Cipher Feedback mode, 869  
 clusterization, 953  
 COFF, 822  
 column-major order, 294  
 Compiler intrinsic, 36, 456, 988  
 Compiler's anomalies, 148, 230, 302, 315, 333, 492, 532, 989  
 Cray-1, 452  
 CRC32, 465, 481  
 CRT, 752, 774  
 CryptoMiniSat, 428  
 CryptoPP, 867  
 Cygwin, 706, 709, 762, 790  
 Data general Nova, 217  
 DES, 408, 421  
 dlopen(), 750  
 dlsym(), 750  
 Donald E. Knuth, 452  
 DOSBox, 915  
 DosBox, 723  
 double, 219, 739  
 Doubly linked list, 462, 570  
 dtruss, 790  
 Duff's device, 493  
 Dynamically loaded libraries, 23  
 Edsger W. Dijkstra, 598  
 EICAR, 912  
 ELF, 79  
 Entropy, 929, 947  
 Error messages, 716  
 fastcall, 14, 34, 66, 306, 734  
 fetchmail, 450  
 FidoNet, 715  
 float, 219, 739  
 Forth, 688  
 FORTRAN, 23  
 Fortran, 294, 515, 598, 706  
 FreeBSD, 718  
 Function epilogue, 30, 54, 56, 137, 365, 722  
 Function prologue, 10, 30, 32, 55, 283, 722  
 Fused multiply-add, 104  
 Fuzzing, 506  
 Garbage collector, 689  
 GCC, 706, 1026, 1028  
 GDB, 29, 48, 51, 282, 394, 395, 789, 1028  
 GeoIP, 948  
 GHex, 788  
 Glibc, 394, 642, 746  
 Global variables, 76  
 GnuPG, 955  
 grep usage, 192, 264, 705, 720, 723, 896  
 Hash functions, 465  
 HASP, 718  
 Heartbleed, 641, 874  
 Heisenbug, 648, 654  
 Hex-Rays, 108, 304, 630  
 Hiew, 93, 134, 710, 716, 758, 759, 762, 788, 987  
 IDA, 86, 381, 514, 699, 713, 788, 789, 977, 1027  
     var\_?, 56, 73  
 IEEE 754, 219, 317, 377, 428, 1002  
 Inline code, 194, 315, 506, 552, 583  
 Integer overflow, 107  
 Intel  
     8080, 209  
     8086, 209, 314, 831  
         Memory model, 662, 992  
     8253, 914  
     80286, 831, 993  
     80386, 314, 993  
     80486, 218  
         FPU, 218  
     Intel 4004, 449  
     Intel C++, 9, 408, 989, 993, 1011  
     Intel syntax, 12, 19  
     iPod/iPhone/iPad, 19  
     Itanium, 990  
 Java, 451, 666  
 John Carmack, 524  
 JPEG, 952  
 jumpable, 169, 176  
 Keil, 19  
 kernel panic, 746  
 kernel space, 746  
 LAPACK, 23  
 LD\_PRELOAD, 750  
 Linker, 81, 546  
 Linux, 307, 747, 899  
     libc.so.6, 306, 394  
 LISP, 619  
 LLDB, 789  
 LLVM, 19  
 long double, 219  
 Loop unwinding, 187  
 LZMA, 950  
 Mac OS Classic, 814  
 Mac OS X, 790  
 Mathematica, 598, 811  
 MD5, 466, 718  
 memfrob(), 866  
 MFC, 759, 887  
 Microsoft Word, 641  
 MIDI, 718  
 MinGW, 706  
 minifloat, 443  
 MIPS, 3, 541, 726, 757, 815, 952  
     Branch delay slot, 8  
     Global Pointer, 25, 299  
     Instructions  
         ADD, 107

ADD.D, 230  
ADDIU, 26, 84, 85  
ADDU, 107  
AND, 317  
BC1F, 267  
BC1T, 267  
BEQ, 97, 139  
BLTZ, 144  
BNE, 139  
BNEZ, 178  
BREAK, 499  
C.LT.D, 267  
DIV.D, 230  
J, 6, 8, 26  
JAL, 107  
JALR, 26, 107  
JR, 167  
LB, 199  
LBU, 198  
LI, 445  
LUI, 26, 84, 85, 230, 320, 445  
LW, 26, 74, 85, 167, 446  
LWC1, 230  
MFC1, 233  
MFHI, 107, 499, 1026  
MFLO, 107, 499, 1026  
MTC1, 383  
MUL.D, 230  
MULT, 107  
NOR, 211  
OR, 29  
ORI, 317, 445  
SB, 198  
SLL, 178, 213, 336  
SLLV, 336  
SLT, 139  
SLTIU, 178  
SLTU, 139, 141, 178  
SRL, 218  
SUBU, 144  
SW, 61  
Load delay slot, 167  
O32, 61, 66, 1025  
Pseudoinstructions  
    B, 196  
    BEQZ, 141  
    L.D, 230  
    LA, 29  
    LI, 8  
    MOVE, 26, 83  
    NEGU, 144  
    NOP, 29, 83  
    NOT, 211  
Registers  
    FCCR, 266  
    HI, 499  
    LO, 499  
MS-DOS, 34, 284, 623, 659, 718, 723, 732, 756, 831, 912, 913, 957, 987, 992, 1002, 1012, 1016, 1017  
    DOS extenders, 993  
MSVC, 1026, 1027  
Name mangling, 546  
Native API, 757  
NEC V20, 915  
Non-a-numbers (NaNs), 257  
Notepad, 950  
objdump, 381, 749, 762, 788  
octet, 450  
OEP, 756, 762  
OllyDbg, 44, 69, 78, 99, 112, 128, 170, 189, 204, 221, 236, 247, 270, 277, 280, 294, 295, 325, 347, 364, 365, 370, 373, 389, 759, 789, 1027  
OOP  
    Polymorphism, 546  
opaque predicate, 543  
OpenMP, 648, 708  
OpenSSL, 641, 874  
OpenWatcom, 706, 735  
Oracle RDBMS, 9, 408, 716, 765, 899, 907, 908, 969, 979, 989, 993  
Page (memory), 419  
Pascal, 711  
PDP-11, 441  
PGP, 714  
Phrack, 715  
Pin, 523  
PNG, 951  
position-independent code, 19, 747  
PowerPC, 3, 25, 814  
Propagating Cipher Block Chaining, 880  
puts() instead of printf(), 21, 71, 108, 135  
Python, 524, 597  
Quake, 524  
Quake III Arena, 385  
rada.re, 13  
Radare, 789  
radare2, 954  
rafind2, 788  
RAID4, 461  
RAM, 81  
Raspberry Pi, 19  
ReactOS, 771  
Recursion, 30, 32, 480  
    Tail recursion, 480  
Register allocation, 421  
Relocation, 23  
Reverse Polish notation, 267  
RISC pipeline, 138  
ROM, 81  
ROT13, 866  
row-major order, 294  
RSA, 5  
RVA, 756  
SAP, 705, 895  
SCO OpenServer, 821  
Scratch space, 737  
Security cookie, 283, 777  
Security through obscurity, 717  
SHA1, 466  
SHA512, 648  
Shadow space, 101, 102, 429  
Shellcode, 542, 746, 756, 913, 1020

## INDEX

Signed numbers, 126, 454  
SIMD, 428, 513  
SQLite, 629  
SSE, 428  
SSE2, 428  
Stack, 30, 98, 156  
    Stack frame, 68  
    Stack overflow, 32  
stdcall, 733, 987  
strace, 750, 790  
strtoll(), 878  
Stuxnet, 718  
Syntactic Sugar, 155  
syscall, 306, 746, 790  
Sysinternals, 716, 790  
  
Tagged pointers, 619  
TCP/IP, 464  
thiscall, 546, 548, 735  
Thumb-2 mode, 22  
thunk-functions, 23, 761, 815, 824  
TLS, 284, 741, 757, 762, 1005  
    Callbacks, 744, 762  
Tor, 715  
tracer, 190, 391, 393, 709, 720, 723, 773, 782, 789, 867, 897, 903, 907, 908, 910, 987  
Turbo C++, 623  
  
uClibc, 642  
UCS-2, 450  
UFS2, 718  
Unicode, 711  
UNIX  
    chmod, 4  
    fork, 643  
     getopt, 878  
    grep, 716, 988  
    mmap(), 623  
    od, 788  
    strings, 715, 788  
    xxd, 788, 935  
Unrolled loop, 194, 286, 493, 496, 510  
uptime, 750  
UPX, 954  
USB, 816  
UseNet, 715  
user space, 746  
UTF-16, 450  
UTF-16LE, 711, 712  
UTF-8, 711, 956  
Uuencode, 955  
Uuencoding, 715  
  
VA, 756  
Valgrind, 654  
Variance, 864  
  
Watcom, 706  
win32  
    FindResource(), 619  
    GetOpenFileName, 212  
    GetProcAddress(), 628  
    HINSTANCE, 629  
    HMODULE, 629  
    LoadLibrary(), 628  
    MAKEINTRESOURCE(), 619  
    WinDbg, 789  
Windows, 786  
    API, 1002  
    IAT, 756  
    INT, 756  
    KERNEL32.DLL, 305  
    MSVCR80.DLL, 387  
    NTAPI, 793  
    ntoskrnl.exe, 899  
    PDB, 705, 758, 793, 801, 895  
    Structured Exception Handling, 37, 763  
    TIB, 284, 763, 1005  
    Win32, 304, 712, 750, 756, 993  
        GetProcAddress, 762  
        LoadLibrary, 762  
        MulDiv(), 456, 810  
        Ordinal, 759  
        RaiseException(), 763  
        SetUnhandledExceptionFilter(), 765  
    Windows 2000, 757  
    Windows 3.x, 654, 993  
    Windows NT4, 757  
    Windows Vista, 756, 793  
    Windows XP, 757, 762, 801  
Wine, 771  
Wolfram Mathematica, 929  
  
x86  
    AVX, 408  
    Flags  
        CF, 34, 1010, 1013, 1014, 1017, 1018  
        DF, 1014, 1018  
        IF, 1014, 1018  
    FPU, 1006  
    Instructions  
        AAA, 1021  
        AAS, 1021  
        ADC, 399, 659, 1010  
        ADD, 9, 42, 98, 501, 659, 1010  
        ADDSD, 429  
        ADDSS, 441  
        ADRcc, 145  
        AESDEC, 867  
        AESENC, 867  
        AESKEYGENASSIST, 870  
        AND, 10, 305, 309, 323, 338, 372, 1010, 1014  
        BSF, 420, 1014  
        BSR, 1014  
        BSWAP, 464, 1014  
        BT, 1014  
        BTC, 319, 1014  
        BTR, 319, 787, 1014  
        BTS, 319, 1014  
        CALL, 9, 31, 538, 761, 880, 947, 1010  
        CBW, 455, 1014  
        CDQ, 407, 455, 1014  
        CDQE, 455, 1014  
        CLD, 1014  
        CLI, 1014  
        CMC, 1014  
        CMOVcc, 138, 145, 147, 149, 153, 465, 1014  
        CMP, 86, 1010, 1021  
        CMPSB, 718, 1015

---

CMPSD, 1015  
 CMPSQ, 1015  
 CMPSW, 1015  
 COMISD, 437  
 COMISS, 441  
 CPUID, 370, 1016  
 CWD, 455, 659, 924, 1014  
 CWDE, 455, 1014  
 DEC, 203, 1010, 1021  
 DIV, 455, 1016  
 DIVSD, 429, 721  
 FABS, 1018  
 FADD, 1018  
 FADDP, 220, 226, 1018  
 FATRET, 332, 333  
 FCHS, 1018  
 FCMOVcc, 259  
 FCOM, 246, 257, 1018  
 FCOMP, 234, 1018  
 FCOMPP, 1018  
 FDIV, 220, 720, 1019  
 FDIVP, 220, 1019  
 FDIVR, 226, 1019  
 FDIVRP, 1019  
 FDUP, 688  
 FILD, 1019  
 FIST, 1019  
 FISTP, 1019  
 FLD, 231, 234, 1019  
 FLD1, 1019  
 FLDCW, 1019  
 FLDZ, 1019  
 FMUL, 220, 1019  
 FMULP, 1019  
 FNSTCW, 1019  
 FNSTSW, 234, 257, 1019  
 FSCALE, 383  
 FSINCOS, 1019  
 FSQRT, 1019  
 FST, 1019  
 FSTCW, 1019  
 FSTP, 231, 1019  
 FSTSW, 1019  
 FSUB, 1019  
 FSUBP, 1019  
 FSUBR, 1019  
 FSUBRP, 1019  
 FUCOM, 257, 1019  
 FUCOMI, 259  
 FUCOMP, 1019  
 FUCOMPP, 257, 1019  
 FWAIT, 218  
 FXCH, 990, 1020  
 IDIV, 455, 496, 1016  
 IMUL, 98, 302, 455, 456, 619, 1010, 1021  
 IN, 538, 831, 914, 1017  
 INC, 203, 987, 1010, 1021  
 INT, 34, 912, 1016  
 INT3, 709  
 IRET, 1016, 1017  
 JA, 126, 258, 455, 1010, 1021  
 JAE, 126, 1010, 1021  
 JB, 126, 455, 1010, 1021  
 JBE, 126, 1010, 1021  
 JC, 1010  
 Jcc, 97, 148  
 JCXZ, 1010  
 JE, 155, 1010, 1021  
 JECXZ, 1010  
 JG, 126, 455, 1010  
 JGE, 126, 1010  
 JL, 126, 455, 1010  
 JLE, 126, 1010  
 JMP, 31, 54, 761, 987, 1010  
 JNA, 1010  
 JNAE, 1010  
 JNB, 1010  
 JNBE, 258, 1010  
 JNC, 1010  
 JNE, 86, 126, 1010, 1021  
 JNG, 1010  
 JNGE, 1010  
 JNL, 1010  
 JNLE, 1010  
 JNO, 1010, 1021  
 JNS, 1010, 1021  
 JNZ, 1010  
 JO, 1010, 1021  
 JP, 235, 915, 1010, 1021  
 JPO, 1010  
 JRCXZ, 1010  
 JS, 1010, 1021  
 JZ, 96, 155, 989, 1010  
 LAHF, 1011  
 LEA, 68, 101, 352, 470, 483, 501, 738, 797, 880, 1011  
 LEAVE, 11, 1011  
 LES, 836, 923  
 LOCK, 786  
 LODSB, 914  
 LOOP, 185, 200, 722, 923, 1017  
 MAXSD, 437  
 MOV, 8, 10, 12, 510, 511, 538, 759, 880, 947, 987, 1012  
 MOVDQA, 411  
 MOVDQU, 411  
 MOVS, 1012  
 MOVSD, 436, 512, 851, 1012  
 MOVSDX, 436  
 MOVSQ, 1012  
 MOVSS, 441  
 MOVSW, 1012  
 MOVSX, 201, 209, 364–366, 455, 1012  
 MOVSXD, 288  
 MOVZX, 202, 349, 815, 1012  
 MUL, 455, 456, 619, 1012  
 MULSD, 429  
 NEG, 504, 1012  
 NOP, 483, 987, 1012, 1021  
 NOT, 208, 210, 855, 1012  
 OR, 309, 526, 1012  
 OUT, 538, 831, 1017  
 PADD, 411  
 PCMPEQB, 420  
 PLMULHW, 408  
 PLMULLD, 408  
 PMOVMSKB, 420  
 POP, 9, 30, 31, 1012, 1021

POPA, 1017, 1021  
POPCNT, 1017  
POPF, 914, 1017  
PUSH, 9, 11, 30, 31, 68, 538, 880, 947, 1012, 1021  
PUSHA, 1017, 1021  
PUSHF, 1017  
PXOR, 419  
RCL, 722, 1017  
RCR, 1017  
RET, 5, 7, 10, 31, 283, 548, 655, 987, 1012  
ROL, 333, 988, 1017  
ROR, 988, 1017  
SAHF, 257, 1012  
SAL, 1018  
SALC, 915  
SAR, 337, 455, 517, 923, 1018  
SBB, 399, 1013  
SCASB, 914, 915, 1013  
SCASD, 1013  
SCASQ, 1013  
SCASW, 1013  
SET, 467  
SETALC, 915  
SETcc, 139, 202, 258, 1018  
SHL, 213, 269, 337, 1013  
SHR, 217, 337, 372, 1013  
SHRD, 406, 1013  
STC, 1018  
STD, 1018  
STI, 1018  
STOSB, 495, 1013  
STOSD, 1013  
STOSQ, 510, 1013  
STOSW, 1013  
SUB, 10, 86, 155, 501, 1010, 1014  
SYSCALL, 1016, 1018  
SYSENTER, 747, 1016, 1018  
TEST, 201, 305, 308, 338, 1014  
UD2, 1018  
XADD, 787  
XCHG, 1012, 1018  
XOR, 10, 86, 208, 518, 721, 826, 987, 1014, 1021  
MMX, 407  
Prefixes  
LOCK, 787, 1009  
REP, 1009, 1012, 1014  
REPE/REPNE, 1009  
REPNE, 1013  
Registers  
AF, 449  
AH, 1011, 1012  
CS, 992  
DF, 642  
DR6, 1008  
DR7, 1008  
DS, 992  
EAX, 86, 107  
EBP, 68, 98  
ECX, 546  
ES, 923, 992  
ESP, 42, 68  
Flags, 86, 128, 1005  
FS, 743  
GS, 284, 743, 746  
JMP, 174  
RIP, 749  
SS, 992  
ZF, 86, 305  
SSE, 407  
SSE2, 407  
x86-64, 14, 50, 67, 72, 94, 100, 421, 428, 539, 736, 749, 1002, 1008  
Xcode, 19  
XML, 714, 863  
XOR, 869  
Z80, 450  
zlib, 643, 866