

## Praise for *Reverse Engineering for Beginners*

- “It’s very well done .. and for free .. amazing.”<sup>1</sup> Daniel Bilar, Siege Technologies, LLC.
- “...excellent and free”<sup>2</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.
- “... 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>3</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>4</sup> (Sergey Bratus, Research Assistant Professor at the Computer Science Department at Dartmouth College)

---

<sup>1</sup>[https://twitter.com/daniel\\_bilar/status/436578617221742593](https://twitter.com/daniel_bilar/status/436578617221742593)

<sup>2</sup><https://twitter.com/petefinnigan/status/400551705797869568>

<sup>3</sup>[http://www.reddit.com/r/IAMa/comments/24nb6f/i\\_was\\_a\\_professional\\_password\\_cracker\\_who\\_taught/](http://www.reddit.com/r/IAMa/comments/24nb6f/i_was_a_professional_password_cracker_who_taught/)

<sup>4</sup><https://twitter.com/sergeybratus/status/505590326560833536>

---

# Reverse Engineering for Beginners

Dennis Yurichev  
<dennis(a)yurichev.com>



©2013-2014, Dennis Yurichev.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Text version (September 3, 2014).

There is probably a newer version of this text, and Russian language version also accessible at <http://beginners.re>. A4-format version is also available on the page.

You may also subscribe to my twitter, to get information about updates of this text, etc: @yurichev<sup>5</sup>, or to subscribe to mailing list<sup>6</sup>.

---

<sup>5</sup><https://twitter.com/yurichev>

<sup>6</sup>[http://yurichev.com/mailling\\_lists.html](http://yurichev.com/mailling_lists.html)

---

# Please donate!

I worked more than one year on this book, here are more than 750 pages, and it's free. Same level books has price tag from \$20 to \$50.

More about it: [0.0.1](#).

I also look for a publisher who may want to translate and publish my “Reverse Engineering for Beginners” book to a language other than English/Russian, under condition that English/Russian version will remain freely available in open-source form. Interesting?  
dennis(a)yurichev.com



# Short contents

<b>I</b>	<b>Code patterns</b>	<b>1</b>
<b>II</b>	<b>Important fundamentals</b>	<b>629</b>
<b>III</b>	<b>Finding important/interesting stuff in the code</b>	<b>636</b>
<b>IV</b>	<b>OS-specific</b>	<b>668</b>
<b>V</b>	<b>Tools</b>	<b>741</b>
<b>VI</b>	<b>More examples</b>	<b>747</b>
<b>VII</b>	<b>Examples of reversing proprietary file formats</b>	<b>918</b>
<b>VIII</b>	<b>Other things</b>	<b>940</b>
<b>IX</b>	<b>Books/blogs worth reading</b>	<b>967</b>
<b>X</b>	<b>Exercises</b>	<b>971</b>
	<b>Afterword</b>	<b>1004</b>
	<b>Appendix</b>	<b>1006</b>

# Contents

0.0.1	Donate	vi
<b>I</b>	<b>Code patterns</b>	<b>1</b>
<b>1</b>	<b>Short introduction to the CPU</b>	<b>3</b>
1.1	Couple words about x86 and ARM	4
<b>2</b>	<b>Hello, world!</b>	<b>5</b>
2.1	x86	5
2.1.1	MSVC	5
2.1.2	GCC	7
2.1.3	GCC: AT&T syntax	9
2.2	x86-64	11
2.2.1	MSVC-x86-64	11
2.2.2	GCC-x86-64	11
2.3	GCC—one more thing	13
2.4	ARM	14
2.4.1	Non-optimizing Keil 6/2013 (ARM mode)	14
2.4.2	Non-optimizing Keil 6/2013 (thumb mode)	16
2.4.3	Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	17
2.4.4	Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)	18
2.4.5	ARM64	20
2.5	Conclusion	22
2.6	Exercises	22
2.6.1	Exercise #1	22
<b>3</b>	<b>Function prologue and epilogue</b>	<b>23</b>
3.1	Recursion	23
<b>4</b>	<b>Stack</b>	<b>25</b>
4.1	Why does the stack grow backward?	26
4.2	What is the stack used for?	26

4.2.1	Save the return address where a function must return control after execution	26
4.2.2	Passing function arguments	28
4.2.3	Local variable storage	29
4.2.4	x86: <code>alloca()</code> function	30
4.2.5	(Windows) SEH	33
4.2.6	Buffer overflow protection	33
4.3	Typical stack layout	33
4.4	Noise in stack	33
4.5	Exercises	37
4.5.1	Exercise #1	37
4.5.2	Exercise #2	37
<b>5</b>	<b><code>printf()</code> with several arguments</b>	<b>39</b>
5.1	x86: 3 arguments	39
5.1.1	MSVC	39
5.1.2	MSVC and OllyDbg	40
5.1.3	GCC	44
5.1.4	GCC and GDB	45
5.2	x64: 8 arguments	48
5.2.1	MSVC	48
5.2.2	GCC	49
5.2.3	GCC + GDB	50
5.3	ARM: 3 arguments	53
5.3.1	32-bit ARM	53
5.3.2	ARM64	55
5.4	ARM: 8 arguments	55
5.4.1	Optimizing Keil 6/2013: ARM mode	56
5.4.2	Optimizing Keil 6/2013: thumb mode	57
5.4.3	Optimizing Xcode 4.6.3 (LLVM): ARM mode	58
5.4.4	Optimizing Xcode 4.6.3 (LLVM): thumb-2 mode	59
5.4.5	ARM64	60
5.5	Conclusion	61
5.6	By the way	62
<b>6</b>	<b><code>scanf()</code></b>	<b>63</b>
6.1	About pointers	63
6.2	x86	64
6.2.1	MSVC	64
6.2.2	MSVC + OllyDbg	66
6.2.3	GCC	68
6.3	x64	68
6.3.1	MSVC	68
6.3.2	GCC	69

---

6.4	ARM	70
6.4.1	Optimizing Keil 6/2013 (thumb mode)	70
6.5	Global variables	71
6.5.1	MSVC: x86	71
6.5.2	MSVC: x86 + OllyDbg	73
6.5.3	GCC: x86	74
6.5.4	MSVC: x64	75
6.5.5	ARM: Optimizing Keil 6/2013 (thumb mode)	75
6.6	scanf() result checking	77
6.6.1	MSVC: x86	78
6.6.2	MSVC: x86: IDA	79
6.6.3	MSVC: x86 + OllyDbg	82
6.6.4	MSVC: x86 + Hiew	83
6.6.5	GCC: x86	85
6.6.6	MSVC: x64	85
6.6.7	ARM: Optimizing Keil 6/2013 (thumb mode)	86
<b>7</b>	<b>Accessing passed arguments</b>	<b>88</b>
7.1	x86	88
7.1.1	MSVC	88
7.1.2	MSVC + OllyDbg	89
7.1.3	GCC	90
7.2	x64	91
7.2.1	MSVC	91
7.2.2	GCC	93
7.2.3	GCC: uint64_t instead of int	94
7.3	ARM	95
7.3.1	Non-optimizing Keil 6/2013 (ARM mode)	95
7.3.2	Optimizing Keil 6/2013 (ARM mode)	97
7.3.3	Optimizing Keil 6/2013 (thumb mode)	97
7.3.4	ARM64	97
<b>8</b>	<b>More about results returning</b>	<b>100</b>
8.1	Attempt to use result of function returning <i>void</i>	100
8.2	What if not to use function result?	101
8.3	Returning a structure	102
<b>9</b>	<b>Pointers</b>	<b>105</b>
9.1	Global variables example	105
9.2	Local variables example	109
9.3	Conclusion	111



<b>10 GOTO</b>	<b>112</b>
10.1 Dead code	114
10.2 Exercise	115
<b>11 Conditional jumps</b>	<b>116</b>
11.1 Simple example	116
11.1.1 x86	117
11.1.2 ARM	125
11.2 Conditional operator	129
11.2.1 x86	129
11.2.2 ARM	131
11.2.3 ARM64	132
11.2.4 Let's rewrite it in if/else way	132
11.2.5 Conclusion	133
11.2.6 Exercise	133
11.3 Conclusion	133
11.3.1 x86	133
11.3.2 ARM	134
<b>12 switch()/case/default</b>	<b>135</b>
12.1 Few number of cases	135
12.1.1 x86	135
12.1.2 ARM: Optimizing Keil 6/2013 (ARM mode)	142
12.1.3 ARM: Optimizing Keil 6/2013 (thumb mode)	143
12.1.4 ARM64: Non-optimizing GCC (Linaro) 4.9	143
12.1.5 ARM64: Optimizing GCC (Linaro) 4.9	144
12.1.6 Conclusion	145
12.2 A lot of cases	145
12.2.1 x86	146
12.2.2 ARM: Optimizing Keil 6/2013 (ARM mode)	151
12.2.3 ARM: Optimizing Keil 6/2013 (thumb mode)	153
12.2.4 Conclusion	156
12.3 When there are several <i>case</i> in one block	157
12.3.1 MSVC	158
12.3.2 GCC	159
12.4 Fallthrough	160
12.4.1 MSVC x86	160
12.4.2 ARM64	161
12.5 Exercises	162
12.5.1 Exercise #1	162

<b>13 Loops</b>	<b>163</b>
13.0.2 Simple example	163
13.0.3 Several iterators	173
13.1 Conclusion	179
13.2 Exercises	182
13.2.1 Exercise #1	182
13.2.2 Exercise #2	182
13.2.3 Exercise #3	182
13.2.4 Exercise #4	183
<b>14 Simple C-strings processings</b>	<b>186</b>
14.1 strlen()	186
14.1.1 x86	186
14.1.2 ARM	192
14.2 Strings trimming	197
14.2.1 x64: Optimizing MSVC 2013	198
14.2.2 x64: Non-optimizing GCC 4.9.1	200
14.2.3 x64: Optimizing GCC 4.9.1	202
14.2.4 ARM64: Non-optimizing GCC (Linaro) 4.9	203
14.2.5 ARM64: Optimizing GCC (Linaro) 4.9	205
14.2.6 ARM: Optimizing Keil 6/2013 (ARM mode)	206
14.2.7 ARM: Optimizing Keil 6/2013 (thumb mode)	207
14.3 Exercises	208
14.3.1 Exercise #1	208
<b>15 Replacing arithmetic instructions to other ones</b>	<b>210</b>
15.1 Multiplication	210
15.1.1 Multiplication using addition	210
15.1.2 Multiplication using shifting	211
15.1.3 Multiplication using shifting/subtracting/adding	212
15.2 Division	216
15.2.1 Division using shifts	216
15.3 Division by 9	217
15.3.1 x86	217
15.3.2 ARM	219
15.3.3 How it works	220
15.3.4 Getting divisor	222
15.4 Exercises	224
15.4.1 Exercise #1	224
15.4.2 Exercise #2	224

<b>16 Floating-point unit</b>	<b>226</b>
16.1 Simple example	227
16.1.1 x86	227
16.1.2 ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	233
16.1.3 ARM: Optimizing Keil 6/2013 (thumb mode)	234
16.1.4 ARM64: Optimizing GCC (Linaro) 4.9	235
16.1.5 ARM64: Non-optimizing GCC (Linaro) 4.9	236
16.2 Passing floating point number via arguments	237
16.2.1 x86	237
16.2.2 ARM + Non-optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)	238
16.2.3 ARM + Non-optimizing Keil 6/2013 (ARM mode)	239
16.2.4 ARM64 + Optimizing GCC (Linaro) 4.9	240
16.3 Comparison example	240
16.3.1 x86	241
16.3.2 ARM	264
16.3.3 ARM64	268
16.4 x64	270
16.5 Exercises	270
16.5.1 Exercise #1	270
16.5.2 Exercise #2	271
<b>17 Arrays</b>	<b>272</b>
17.1 Simple example	272
17.1.1 x86	272
17.1.2 ARM + Non-optimizing Keil 6/2013 (ARM mode)	276
17.1.3 ARM + Optimizing Keil 6/2013 (thumb mode)	277
17.2 Buffer overflow	278
17.2.1 Reading behind array bounds	278
17.2.2 Writing behind array bounds	281
17.3 Buffer overflow protection methods	286
17.3.1 Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)	289
17.4 One more word about arrays	291
17.5 Array of pointers to strings	292
17.5.1 x64	292
17.5.2 32-bit ARM	294
17.5.3 ARM64	296
17.6 Multidimensional arrays	300
17.6.1 Two-dimensional array example	301
17.6.2 Access two-dimensional array as one-dimensional	303
17.6.3 Three-dimensional array example	306
17.6.4 More examples	309
17.7 Pack of strings as two-dimensional array	309
17.7.1 32-bit ARM	312
17.7.2 ARM64	313

17.7.3	Conclusion	313
17.8	Negative array indices	313
17.9	Conclusion	317
17.10	Exercises	317
17.10.1	Exercise #1	317
17.10.2	Exercise #2	319
17.10.3	Exercise #3	322
17.10.4	Exercise #4	323
17.10.5	Exercise #5	324
<b>18</b>	<b>Working with specific bits</b>	<b>328</b>
18.1	Specific bit checking	328
18.1.1	x86	328
18.1.2	ARM	332
18.2	Specific bit setting/clearing	334
18.2.1	x86	334
18.2.2	ARM + Optimizing Keil 6/2013 (ARM mode)	339
18.2.3	ARM + Optimizing Keil 6/2013 (thumb mode)	339
18.2.4	ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	339
18.2.5	ARM: more about BIC instruction	340
18.2.6	ARM64: Optimizing GCC (Linaro) 4.9	340
18.2.7	ARM64: Non-optimizing GCC (Linaro) 4.9	341
18.3	Shifts	341
18.4	Counting bits set to 1	341
18.4.1	x86	344
18.4.2	x64	348
18.4.3	ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	352
18.4.4	ARM + Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)	353
18.4.5	ARM64 + Optimizing GCC 4.9	353
18.4.6	ARM64 + Non-optimizing GCC 4.9	354
18.5	CRC32 calculation example	355
18.6	Network address calculation example	360
18.6.1	calc_network_address()	362
18.6.2	form_IP()	363
18.6.3	print_as_IP()	365
18.6.4	form_netmask() and set_bit()	367
18.6.5	Summary	368
18.7	Conclusion	368
18.7.1	Check for specific bit (known at compiling stage)	369
18.7.2	Check for specific bit (specified at runtime)	369
18.7.3	Set specific bit (known at compiling stage)	370
18.7.4	Set specific bit (specified at runtime)	370
18.7.5	Clear specific bit (known at compiling stage)	371
18.7.6	Clear specific bit (specified at runtime)	371

---

18.8 Exercises	371
18.8.1 Exercise #1	371
18.8.2 Exercise #2	373
18.8.3 Exercise #3	374
18.8.4 Exercise #4	375
<b>19 Structures</b>	<b>377</b>
19.1 MSVC: SYSTEMTIME example	377
19.1.1 OllyDbg	379
19.1.2 Replacing the structure by array	380
19.2 Let's allocate space for structure using malloc()	382
19.3 UNIX: struct tm	384
19.3.1 Linux	384
19.3.2 ARM + Optimizing Keil 6/2013 (thumb mode)	395
19.3.3 ARM + Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)	396
19.4 Fields packing in structure	397
19.4.1 x86	398
19.4.2 x86 + OllyDbg + fields are packed by default	401
19.4.3 x86 + OllyDbg + fields aligning by 1 byte boundary	402
19.4.4 ARM + Optimizing Keil 6/2013 (thumb mode)	403
19.4.5 ARM + Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)	403
19.5 Nested structures	404
19.5.1 OllyDbg	406
19.6 Bit fields in structure	407
19.6.1 CPUID example	407
19.6.2 Working with the float type as with a structure	413
19.7 Exercises	417
19.7.1 Exercise #1	417
19.7.2 Exercise #2	417
<b>20 Unions</b>	<b>421</b>
20.1 Pseudo-random number generator example	421
<b>21 Pointers to functions</b>	<b>424</b>
21.1 MSVC	425
21.1.1 MSVC + OllyDbg	428
21.1.2 MSVC + tracer	429
21.1.3 MSVC + tracer (code coverage)	430
21.2 GCC	431
21.2.1 GCC + GDB (with source code)	433
21.2.2 GCC + GDB (no source code)	434

<b>22 64-bit values in 32-bit environment</b>	<b>438</b>
22.1 Arguments passing, addition, subtraction	438
22.2 Multiplication, division	440
22.3 Shifting right	443
22.4 Converting 32-bit value into 64-bit one	443
<b>23 SIMD</b>	<b>446</b>
23.1 Vectorization	447
23.1.1 Intel C++	448
23.1.2 GCC	452
23.2 SIMD <code>strlen()</code> implementation	455
<b>24 64 bits</b>	<b>461</b>
24.1 x86-64	461
24.2 ARM	471
24.3 Float point numbers	471
<b>25 Working with float point numbers using SIMD</b>	<b>472</b>
25.1 Simple example	472
25.1.1 x64	473
25.1.2 x86	474
25.2 Passing floating point number via arguments	479
25.3 Comparison example	481
25.3.1 x64	481
25.3.2 x86	482
25.4 Summary	483
<b>26 Temperature converting</b>	<b>485</b>
26.1 Integer values	485
26.1.1 Optimizing MSVC 2012 x86	486
26.1.2 Optimizing MSVC 2012 x64	488
26.2 Float point values	488
<b>27 Fibonacci numbers</b>	<b>493</b>
27.1 Example #1	493
27.2 Example #2	497
27.3 Summary	500
<b>28 C99 restrict</b>	<b>501</b>
<b>29 Inline functions</b>	<b>505</b>
29.1 Strings and memory functions	506
29.1.1 <code>strcmp()</code>	506
29.1.2 <code>strlen()</code>	509
29.1.3 <code>strcpy()</code>	510

---

29.1.4	memcpy()	510
29.1.5	memcmp()	514
29.1.6	IDA script	515
<b>30</b>	<b>Incorrectly disassembled code</b>	<b>516</b>
30.1	Disassembling started incorrectly (x86)	516
30.2	How random noise looks disassembled?	517
<b>31</b>	<b>C++</b>	<b>524</b>
31.1	Classes	524
31.1.1	Simple example	524
31.1.2	Class inheritance	533
31.1.3	Encapsulation	537
31.1.4	Multiple inheritance	539
31.1.5	Virtual methods	544
31.2	ostream	548
31.3	References	549
31.4	STL	550
31.4.1	std::string	550
31.4.2	std::list	560
31.4.3	std::vector	575
31.4.4	std::map and std::set	585
<b>32</b>	<b>Obfuscation</b>	<b>602</b>
32.1	Text strings	602
32.2	Executable code	603
32.2.1	Inserting garbage	603
32.2.2	Replacing instructions to bloated equivalents	604
32.2.3	Always executed/never executed code	604
32.2.4	Making a lot of mess	604
32.2.5	Using indirect pointers	605
32.3	Virtual machine / pseudo-code	605
32.4	Other thing to mention	606
32.5	Exercises	606
32.5.1	Exercise #1	606
<b>33</b>	<b>More about ARM</b>	<b>607</b>
33.1	Addressing modes	607
33.2	Loading constants into register	608
33.2.1	32-bit ARM	608
33.2.2	ARM64	609
33.3	Relocs in ARM64	610

<b>34 Windows 16-bit</b>	<b>612</b>
34.1 Example#1	612
34.2 Example #2	613
34.3 Example #3	614
34.4 Example #4	616
34.5 Example #5	619
34.6 Example #6	624
34.6.1 Global variables	626
<b>II Important fundamentals</b>	<b>629</b>
<b>35 Signed number representations</b>	<b>630</b>
35.1 Integer overflow	631
<b>36 Endianness</b>	<b>632</b>
36.1 Big-endian	632
36.2 Little-endian	632
36.3 Bi-endian	633
36.4 Converting data	633
<b>37 Memory</b>	<b>634</b>
<b>38 CPU</b>	<b>635</b>
38.1 Branch predictors	635
38.2 Data dependencies	635
<b>III Finding important/interesting stuff in the code</b>	<b>636</b>
<b>39 Identification of executable files</b>	<b>638</b>
39.1 Microsoft Visual C++	638
39.1.1 Name mangling	638
39.2 GCC	638
39.2.1 Name mangling	639
39.2.2 Cygwin	639
39.2.3 MinGW	639
39.3 Intel FORTRAN	639
39.4 Watcom, OpenWatcom	639
39.4.1 Name mangling	639
39.5 Borland	639
39.5.1 Delphi	640
39.6 Other known DLLs	642



---

<b>40</b>	<b>Communication with the outer world (win32)</b>	<b>643</b>
40.1	Often used functions in Windows API . . . . .	644
40.2	tracer: Intercepting all functions in specific module . . . . .	645
<b>41</b>	<b>Strings</b>	<b>647</b>
41.1	Text strings . . . . .	647
41.1.1	C/C++ . . . . .	647
41.1.2	Borland Delphi . . . . .	648
41.1.3	Unicode . . . . .	648
41.2	Error/debug messages . . . . .	652
<b>42</b>	<b>Calls to assert()</b>	<b>653</b>
<b>43</b>	<b>Constants</b>	<b>655</b>
43.1	Magic numbers . . . . .	656
43.1.1	DHCP . . . . .	656
43.2	Constant searching . . . . .	657
<b>44</b>	<b>Finding the right instructions</b>	<b>658</b>
<b>45</b>	<b>Suspicious code patterns</b>	<b>661</b>
45.1	XOR instructions . . . . .	661
45.2	Hand-written assembly code . . . . .	661
<b>46</b>	<b>Using magic numbers while tracing</b>	<b>663</b>
<b>47</b>	<b>Other things</b>	<b>665</b>
47.1	General idea . . . . .	665
47.2	C++ . . . . .	665
<b>48</b>	<b>Old-school techniques, nevertheless, interesting to know</b>	<b>666</b>
48.1	Memory “snapshots” comparing . . . . .	666
48.1.1	Windows registry . . . . .	667
<b>IV</b>	<b>OS-specific</b>	<b>668</b>
<b>49</b>	<b>Arguments passing methods (calling conventions)</b>	<b>669</b>
49.1	cdecl . . . . .	669
49.2	stdcall . . . . .	669
49.2.1	Variable arguments number functions . . . . .	671
49.3	fastcall . . . . .	671
49.3.1	GCC regparm . . . . .	672
49.3.2	Watcom/OpenWatcom . . . . .	673
49.4	thiscall . . . . .	673

---

49.5	x86-64	673
49.5.1	Windows x64	673
49.5.2	Linux x64	676
49.6	Returning values of <i>float</i> and <i>double</i> type	677
49.7	Modifying arguments	677
<b>50</b>	<b>Thread Local Storage</b>	<b>679</b>
<b>51</b>	<b>System calls (syscall-s)</b>	<b>680</b>
51.1	Linux	680
51.2	Windows	681
<b>52</b>	<b>Linux</b>	<b>682</b>
52.1	Position-independent code	682
52.1.1	Windows	685
52.2	<i>LD_PRELOAD</i> hack in Linux	686
<b>53</b>	<b>Windows NT</b>	<b>690</b>
53.1	CRT (win32)	690
53.2	Win32 PE	695
53.2.1	Terminology	695
53.2.2	Base address	696
53.2.3	Subsystem	697
53.2.4	OS version	697
53.2.5	Sections	697
53.2.6	Relocations (relocs)	699
53.2.7	Exports and imports	699
53.2.8	Resources	703
53.2.9	.NET	703
53.2.10	TLS	703
53.2.11	Tools	704
53.2.12	Further reading	704
53.3	Windows SEH	704
53.3.1	Let's forget about MSVC	704
53.3.2	Now let's get back to MSVC	712
53.3.3	Windows x64	732
53.3.4	Read more about SEH	737
53.4	Windows NT: Critical section	738
<b>V</b>	<b>Tools</b>	<b>741</b>
<b>54</b>	<b>Disassembler</b>	<b>742</b>
54.1	IDA	742

<b>55 Debugger</b>	<b>743</b>
55.1 tracer	743
55.2 OllyDbg	743
55.3 GDB	743
<b>56 System calls tracing</b>	<b>744</b>
56.0.1 strace / dtruss	744
<b>57 Decompilers</b>	<b>745</b>
<b>58 Other tools</b>	<b>746</b>
<b>VI More examples</b>	<b>747</b>
<b>59 Task manager practical joke (Windows Vista)</b>	<b>748</b>
59.1 Using LEA to load values	751
<b>60 Color Lines game practical joke</b>	<b>753</b>
<b>61 Minesweeper (Windows XP)</b>	<b>757</b>
61.1 Exercises	763
<b>62 Hand decompiling + Z3 SMT solver</b>	<b>764</b>
62.1 Hand decompiling	765
62.2 Now let's use Z3 SMT solver	770
<b>63 Dongles</b>	<b>776</b>
63.1 Example #1: MacOS Classic and PowerPC	776
63.2 Example #2: SCO OpenServer	787
63.2.1 Decrypting error messages	798
63.3 Example #3: MS-DOS	802
<b>64 "QR9": Rubik's cube inspired amateur crypto-algorithm</b>	<b>811</b>
<b>65 SAP</b>	<b>855</b>
65.1 About SAP client network traffic compression	855
65.2 SAP 6.0 password checking functions	871
<b>66 Oracle RDBMS</b>	<b>878</b>
66.1 V\$VERSION table in the Oracle RDBMS	878
66.2 X\$KSMLRU table in Oracle RDBMS	890
66.3 V\$TIMER table in Oracle RDBMS	893
<b>67 Handwritten assembly code</b>	<b>899</b>
67.1 EICAR test file	899

<b>68 Demos</b>	<b>901</b>
68.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10	901
68.1.1 Trixter's 42 byte version	902
68.1.2 My attempt to reduce Trixter's version: 27 bytes	903
68.1.3 Take a random memory garbage as a source of randomness	904
68.1.4 Conclusion	905
68.2 Mandelbrot set	905
68.2.1 Theory	906
68.2.2 Let's back to the demo	911
68.2.3 My "fixed" version	915
<b>VII Examples of reversing proprietary file formats</b>	<b>918</b>
<b>69 Millenium game save file</b>	<b>919</b>
<b>70 Oracle RDBMS: .SYM-files</b>	<b>924</b>
<b>71 Oracle RDBMS: .MSB-files</b>	<b>934</b>
71.1 Summary	939
<b>VIII Other things</b>	<b>940</b>
<b>72 npad</b>	<b>941</b>
<b>73 Executable files patching</b>	<b>944</b>
73.1 Text strings	944
73.2 x86 code	944
<b>74 Compiler intrinsic</b>	<b>946</b>
<b>75 Compiler's anomalies</b>	<b>947</b>
<b>76 OpenMP</b>	<b>949</b>
76.1 MSVC	952
76.2 GCC	955
<b>77 Itanium</b>	<b>958</b>
<b>78 8086 memory model</b>	<b>962</b>
<b>79 Basic blocks reordering</b>	<b>964</b>
79.1 Profile-guided optimization	964

<b>IX</b>	<b>Books/blogs worth reading</b>	<b>967</b>
<b>80</b>	<b>Books</b>	<b>968</b>
80.1	Windows	968
80.2	C/C++	968
80.3	x86 / x86-64	968
80.4	ARM	968
<b>81</b>	<b>Blogs</b>	<b>969</b>
81.1	Windows	969
<b>82</b>	<b>Other</b>	<b>970</b>
<b>X</b>	<b>Exercises</b>	<b>971</b>
<b>83</b>	<b>Level 1</b>	<b>973</b>
83.1	Exercise 1.4	973
<b>84</b>	<b>Level 2</b>	<b>974</b>
84.1	Exercise 2.4	974
84.1.1	Optimizing MSVC 2010	974
84.1.2	GCC 4.4.1	975
84.1.3	Optimizing Keil (ARM mode)	977
84.1.4	Optimizing Keil (thumb mode)	978
84.2	Exercise 2.5	978
84.2.1	Optimizing MSVC 2010	979
84.3	Exercise 2.6	980
84.3.1	Optimizing MSVC 2010	980
84.3.2	Optimizing Keil (ARM mode)	981
84.3.3	Optimizing Keil (thumb mode)	982
84.4	Exercise 2.13	983
84.4.1	Optimizing MSVC 2012	983
84.4.2	Keil (ARM mode)	983
84.4.3	Keil (thumb mode)	984
84.5	Exercise 2.14	984
84.5.1	MSVC 2012	984
84.5.2	Keil (ARM mode)	986
84.5.3	GCC 4.6.3 for Raspberry Pi (ARM mode)	986
84.6	Exercise 2.15	987
84.6.1	Optimizing MSVC 2012 x64	988
84.6.2	Optimizing GCC 4.4.6 x64	992
84.6.3	Optimizing GCC 4.8.1 x86	993
84.6.4	Keil (ARM mode): Cortex-R4F CPU as target	994

84.7 Exercise 2.16	996
84.7.1 Optimizing MSVC 2012 x64	996
84.7.2 Optimizing Keil (ARM mode)	997
84.7.3 Optimizing Keil (thumb mode)	997
84.8 Exercise 2.17	998
84.9 Exercise 2.18	998
84.10 Exercise 2.19	998
<b>85 Level 3</b>	<b>999</b>
85.1 Exercise 3.2	999
85.2 Exercise 3.3	999
85.3 Exercise 3.4	1000
85.4 Exercise 3.5	1000
85.5 Exercise 3.6	1000
85.6 Exercise 3.8	1001
<b>86 crackme / keygenme</b>	<b>1002</b>
<b>Afterword</b>	<b>1004</b>
<b>87 Questions?</b>	<b>1004</b>
<b>Appendix</b>	<b>1006</b>
<b>A x86</b>	<b>1006</b>
A.1 Terminology	1006
A.2 General purpose registers	1006
A.2.1 RAX/EAX/AX/AL	1007
A.2.2 RBX/EBX/BX/BL	1007
A.2.3 RCX/ECX/CX/CL	1007
A.2.4 RDX/EDX/DX/DL	1007
A.2.5 RSI/ESI/SI/SIL	1008
A.2.6 RDI/EDI/DI/DIL	1008
A.2.7 R8/R8D/R8W/R8L	1008
A.2.8 R9/R9D/R9W/R9L	1008
A.2.9 R10/R10D/R10W/R10L	1008
A.2.10 R11/R11D/R11W/R11L	1009
A.2.11 R12/R12D/R12W/R12L	1009
A.2.12 R13/R13D/R13W/R13L	1009
A.2.13 R14/R14D/R14W/R14L	1009
A.2.14 R15/R15D/R15W/R15L	1009
A.2.15 RSP/ESP/SP/SPL	1010

---

A.2.16	RBP/EBP/BP/BPL	1010
A.2.17	RIP/EIP/IP	1010
A.2.18	CS/DS/ES/SS/FS/GS	1011
A.2.19	Flags register	1011
A.3	FPU-registers	1012
A.3.1	Control Word	1012
A.3.2	Status Word	1013
A.3.3	Tag Word	1013
A.4	SIMD-registers	1014
A.4.1	MMX-registers	1014
A.4.2	SSE and AVX-registers	1014
A.5	Debugging registers	1014
A.5.1	DR6	1014
A.5.2	DR7	1015
A.6	Instructions	1016
A.6.1	Prefixes	1016
A.6.2	Most frequently used instructions	1016
A.6.3	Less frequently used instructions	1023
A.6.4	FPU instructions	1030
A.6.5	SIMD instructions	1032
A.6.6	Instructions having printable ASCII opcode	1032
<b>B</b>	<b>ARM</b>	<b>1035</b>
B.1	Terminology	1035
B.2	Versions	1035
B.3	32-bit ARM (AArch32)	1036
B.3.1	General purpose registers	1036
B.3.2	Current Program Status Register (CPSR)	1036
B.3.3	VFP (floating point) and NEON registers	1036
B.4	64-bit ARM (AArch64)	1037
B.4.1	General purpose registers	1037
B.5	Instructions	1038
<b>C</b>	<b>Some GCC library functions</b>	<b>1039</b>
<b>D</b>	<b>Some MSVC library functions</b>	<b>1040</b>
<b>E</b>	<b>Cheatsheets</b>	<b>1041</b>
E.1	IDA	1041
E.2	OllyDbg	1042
E.3	MSVC	1043
E.4	GCC	1043
E.5	GDB	1043

<b>F</b>	<b>Exercise solutions</b>	<b>1045</b>
F.1	Per chapter	1045
F.1.1	“Stack” chapter	1045
F.1.2	“switch()/case/default” chapter	1046
F.1.3	Exercise #1	1046
F.1.4	“Loops” chapter	1046
F.1.5	Exercise #3	1046
F.1.6	Exercise #4	1046
F.1.7	“Simple C-strings processings” chapter	1046
F.1.8	“Replacing arithmetic instructions to other ones” chapter	1047
F.1.9	“Floating-point unit” chapter	1047
F.1.10	“Arrays” chapter	1047
F.1.11	“Working with specific bits” chapter	1049
F.1.12	“Structures” chapter	1051
F.1.13	“Obfuscation” chapter	1053
F.2	Level 1	1053
F.2.1	Exercise 1.1	1053
F.2.2	Exercise 1.4	1053
F.3	Level 2	1053
F.3.1	Exercise 2.4	1053
F.3.2	Exercise 2.5	1054
F.3.3	Exercise 2.6	1054
F.3.4	Exercise 2.13	1055
F.3.5	Exercise 2.14	1055
F.3.6	Exercise 2.15	1055
F.3.7	Exercise 2.16	1055
F.3.8	Exercise 2.17	1056
F.3.9	Exercise 2.18	1056
F.3.10	Exercise 2.19	1056
F.4	Level 3	1056
F.4.1	Exercise 3.2	1056
F.4.2	Exercise 3.3	1056
F.4.3	Exercise 3.4	1056
F.4.4	Exercise 3.5	1057
F.4.5	Exercise 3.6	1057
F.4.6	Exercise 3.8	1057
F.5	Other	1057
F.5.1	“Minesweeper (Windows XP)” example	1057
	<b>Acronyms used</b>	<b>1059</b>
	<b>Glossary</b>	<b>1065</b>





---

# Preface

Here are some of my notes in English for beginners about [reverse engineering](#) who would like to learn to understand x86 (which accounts for almost all executable software in the world) and ARM code created by C/C++ compilers.

There are several popular meanings of the term “[reverse engineering](#)”: 1) reverse engineering of software: researching compiled programs; 2) 3D model scanning and reworking in order to make a copy of it; 3) recreating [DBMS](#)<sup>7</sup> structure. This book is related to the first meaning.

## Topics discussed

x86/x64, ARM/ARM64.

## Topics touched

Oracle RDBMS (66), Itanium (77), copy-protection dongles (63), LD\_PRELOAD (52.2), stack overflow, [ELF](#)<sup>8</sup>, win32 PE file format (53.2), x86-64 (24.1), critical sections (53.4), syscalls (51), [TLS](#)<sup>9</sup>, position-independent code ([PIC](#)<sup>10</sup>) (52.1), profile-guided optimization (79.1), C++ STL (31.4), OpenMP (76), SEH (53.3).

## Notes

Why one should learn assembly language these days? Unless you are OS developer, you probably don't need write in assembly: modern compilers perform optimizations much better than humans do. <sup>11</sup> Also, modern [CPU](#)<sup>12</sup>s are very complex devices and assembly knowledge would not help you understand its internals. That said, there are at least two areas where a good understanding of assembly may help: first, security/malware research. Second, gaining a better understanding of your compiled code while debugging.

Therefore this book is intended for those who want to understand assembly language rather than to write in it, which is why there are many examples of compiler output.

How would one find a reverse engineering job?

There are hiring threads that appear from time to time on reddit devoted to RE<sup>13</sup> (2013 Q3, 2014). Try looking there. A somewhat related hiring thread can be found

---

<sup>7</sup>Database management systems

<sup>8</sup>Executable file format widely used in \*NIX system including Linux

<sup>9</sup>Thread Local Storage

<sup>10</sup>Position Independent Code: [52.1](#)

<sup>11</sup>A very good text about this topic: [[Fog13b](#)]

<sup>12</sup>Central processing unit

<sup>13</sup><http://www.reddit.com/r/ReverseEngineering/>

in the “netsec” subreddit: [2014 Q2](#).

## About the author



Dennis Yurichev is an experienced reverse engineer and programmer. His CV is available on his website<sup>14</sup>.

## Thanks

For patiently answering all my questions: Andrey “herm1t” Baranovich, Slava “Avid” Kazakov.

For sending me notes about mistakes and inaccuracies: Stanislav “Beaver” Bobrytsky, Alexander Lysenko, Shell Rocket, Zhu Ruijin.

For helping me in other ways: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC).

For translating to Chinese simplified: Xian Chi.

For proofreading: Alexander “Lstar” Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark “Logxen” Cooper, Yuan Jochen Kang.

Thanks also to all the folks on github.com who have contributed notes and corrections.

A lot of  $\LaTeX$  packages were used: I would like to thank the authors as well.

### 0.0.1 Donate

As it turns out, (technical) writing takes a lot of effort and work.

This book is free, available freely and available in source code form <sup>15</sup> (LaTeX), and it will be so forever.

<sup>14</sup>[http://yurichev.com/Dennis\\_Yurichev.pdf](http://yurichev.com/Dennis_Yurichev.pdf)

<sup>15</sup><https://github.com/dennis714/RE-for-beginners>

It's also ad-free.

My current plan for this book is to add lots of information about: PLANS<sup>16</sup>.

If you want me to continue writing on all these topics you may consider donating.

I worked more than year on this book <sup>17</sup>, there are more than 750 pages. There are at least  $\approx$  400 T<sub>E</sub>X-files,  $\approx$  150 C/C++ source codes,  $\approx$  470 various listings,  $\approx$  160 screenshots.

Price of other books on the same subject varies between \$20 and \$50 on amazon.com.

Ways to donate are available on the page: <http://beginners.re/donate.html>

Every donor's name will be included in the book! Donors also have a right to ask me to rearrange items in my writing plan.

## Donors

14 \* anonymous, 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), Pawel Szczur (40 CHF), Justin Simms (\$20), Shawn the R0ck (\$27), Ki Chan Ahn (\$50), Triop AB (100 SEK), Ange Albertini (10 EUR), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (40 EUR), Sergey Volchkov (10 AUD), Vankayala Vigneswararao (\$50), Philippe Teuwen (\$4), Martin Haeberli (\$10), Victor Cazacov (5 EUR), Tobias Sturzenegger (10 CHF), Sonny Thai (\$15), Bayna AlZaabi (\$75), Redfive B.V. (25 EUR), Joonas Oskari Heikkilä (5 EUR), Marshall Bishop (\$50), Nicolas Werner (12 EUR), Jeremy Brown (\$100), Alexandre Borges (\$25), Vladimir Dikovski (50 EUR).

## About illustrations

Readers who are used to reading a lot on the Internet expect to see illustrations exactly where they should be. This is because there are no pages at all, it's just one single page. It's not possible to place illustrations in the book at the suitable context. So in this book illustrations are at the end of each section, and there are references in the text, such as "fig.1.1".

This is the A5-format version for e-book readers. Although, the content is mostly the same, the illustrations are resized and probably not readable. Sorry about that! You can always view them in A4-format version here: <http://beginners.re>.

---

<sup>16</sup><https://github.com/dennis714/RE-for-beginners/blob/master/PLANS>

<sup>17</sup>Initial git commit from March 2013:  
<https://github.com/dennis714/RE-for-beginners/tree/1e57ef540d827c7f7a92fcb3a4626af3e13c7ee4>

## **About Korean translation**

You can free to download and read my book online. However DO NOT distribute any translation WITHOUT MY PERMISSION. Please contact me at [dennis\(a\)yurichev.com](mailto:dennis@yurichev.com) or the Korean translation copyright holder at [acornpub\(a\)acornpub.co.kr](mailto:acornpub@acornpub.co.kr) if you are interested in the Korean translation.

# **Part I**

# **Code patterns**

---

When I first learned C and then C++, I wrote small pieces of code, compiled them, and saw what was produced in the assembly language. This was easy for me. I did it many times and the relation between the C/C++ code and what the compiler produced was imprinted in my mind so deep that I can quickly understand what was in the original C code when I look at produced x86 code. Perhaps this technique may be helpful for someone else so I will try to describe some examples here.

# Chapter 1

## Short introduction to the CPU

The **CPU** is the unit which executes all of the programs.

Short glossary:

**Instruction** : a primitive command to the **CPU**. Simplest examples: moving data between registers, working with memory, arithmetic primitives. As a rule, each **CPU** has its own instruction set architecture (**ISA**<sup>1</sup>).

**Machine code** : code for the **CPU**. Each instruction is usually encoded by several bytes.

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

**CPU register** : Each **CPU** has a fixed set of general purpose registers (**GPR**<sup>2</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 you are working with a high-level **PL**<sup>3</sup> and you have only 8 32-bit (or 64-bit) variables. A lot of things can be done using only these!

What is the difference between machine code and a **PL**? 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** which can execute high-level **PL** code, but it would be much more complex. On the contrary, it is very inconvenient for humans to use assembly language due to its low-levelness. Besides, it is very hard to do it without making a huge amount of annoying mistakes. The program which converts high-level **PL** code into assembly is called a *compiler*.

---

<sup>1</sup>Instruction Set Architecture

<sup>2</sup>General Purpose Registers

<sup>3</sup>Programming language



## 1.1 Couple words about x86 and ARM

x86 was always an **ISA** with variable-length opcodes, so when 64-bit era came, x64 extensions was not affect **ISA** very much. x86 have a lot of instructions appeared in 16-bit 8086 CPU and still present in latest CPUs.

ARM is a **RISC<sup>4</sup> CPU** designed with constant opcode length in mind, which had some advantages in past. So at very start, ARM has all instructions encoded in 4 bytes<sup>5</sup>. This is now called “ARM mode”.

Then they thought it’s not very frugal. In fact, most used **CPU** instructions<sup>6</sup> in real world applications can be encoded using lesser amount of information. So they added another **ISA** called Thumb, where each instruction encoded in just 2 bytes. Now this is called “Thumb mode”. But not all ARM instructions could be encoded in just 2 bytes, so Thumb instruction set is somewhat limited. Code compiled for ARM mode and Thumb mode may coexist in one program, of course.

Then ARM creators thought Thumb can be extended: Thumb-2 was appeared (in ARMv7). Thumb-2 is still 2-byte instructions, but some new instructions has size of 4 bytes. There is a common misconception that thumb-2 is a mix of ARM and thumb. This is not correct. Rather, thumb-2 was extended to fully support processor features so it could compete with ARM mode. On instruction richness, Thumb-2 can compete with original ARM mode. The majority of iPod/iPhone/iPad applications are compiled for the thumb-2 instruction set because Xcode does this by default.

Then 64-bit ARM came, this **ISA** has 4-byte opcodes again, without any additional Thumb mode. But 64-bit requirements affected **ISA**, so, summarizing, we now have 3 ARM instruction sets: ARM mode, Thumb mode (including Thumb-2) and ARM64. These **ISA**’s intersects partially, but I would say, they are much more different **ISA**’s than variations of one. Therefore, I try to add fragments of code in all 3 ARM **ISA**’s in this book.

---

<sup>4</sup>Reduced instruction set computing

<sup>5</sup>By the way, fixed-length instructions are handy in a way that one can calculate the next (or previous) instruction’s address without effort. This feature will be discussed in switch() (12.2.2) section.

<sup>6</sup>These are MOV/PUSH/CALL/Jcc

# Chapter 2

## Hello, world!

Let's start with the famous example from the "The C programming Language"[[Ker88](#)] book:

```
#include <stdio.h>

int main()
{
    printf("hello, world");
    return 0;
};
```

### 2.1 x86

#### 2.1.1 MSVC

Let's compile it in MSVC 2010:

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

(/Fa option means generate assembly listing file)

Listing 2.1: MSVC 2010

```
CONST SEGMENT
$SG3830 DB      'hello, world', 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 hereafter: [2.1.3](#).

The compiler generated `1.obj` file will be linked into `1.exe`.

In our case, the file contain two segments: `CONST` (for data constants) and `_TEXT` (for code).

The string ``hello, world'` in C/C++ has type `const char[]` [[Str13](#), p176, 7.3.2], however 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.

So the example may be rewritten as:

```

#include <stdio.h>

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

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

```

Let's 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 strings: [41.1.1](#).

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>1</sup>.

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

Before the call the 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 flow control to the `main()` function, string address (or pointer to it) is still in stack.

Since we do not need it anymore the [stack pointer](#) (the `ESP` register) needs to be corrected.

<sup>1</sup>Read more about it in section about function prolog and epilog ([3](#)).

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

Why 4? Since it is 32-bit code we need exactly 4 bytes for address passing through the stack. It is 8 bytes in x64-code.

`ADD ESP, 4` is effectively equivalent to `POP register` but without using any register<sup>2</sup>.

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

Here is an example from it:

Listing 2.2: Oracle RDBMS 10.2 Linux (app.o file)

<code>.text:0800029A</code>	<code>push</code>	<code>ebx</code>
<code>.text:0800029B</code>	<code>call</code>	<code>qksfroChild</code>
<code>.text:080002A0</code>	<code>pop</code>	<code>ecx</code>

Read more about the stack in section (4).

After the call to `printf()`, in the original C/C++ code was `return 0` –return 0 as the result of the `main()` function.

In the generated code this is implemented by instruction `XOR EAX, EAX`. `XOR` is in fact, just “eXclusive OR”<sup>3</sup> but compilers often use it instead of `MOV EAX, 0` –again because it is a slightly shorter opcode (2 bytes against 5).

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

The last instruction `RET` returns control flow to the *caller*. Usually, it is C/C++ `CRT`<sup>4</sup> code which in turn returns control to the `OS`<sup>5</sup>.

## 2.1.2 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`<sup>6</sup> disassembler, let's see how the `main()` function was created.

(`IDA`, like `MSVC`, shows code in Intel-syntax).

N.B. We could also have `GCC` produce assembly listings in Intel-syntax by applying the options `-S -masm=intel`

<sup>2</sup>CPU flags, however, are modified

<sup>3</sup>[http://en.wikipedia.org/wiki/Exclusive\\_or](http://en.wikipedia.org/wiki/Exclusive_or)

<sup>4</sup>C runtime library: `sec:CRT`

<sup>5</sup>Operating System

<sup>6</sup>Interactive Disassembler

Listing 2.3: GCC

```

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"
                    ↵
                    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 saved in the EAX register first and then it is stored on the stack. Also in the function prologue we see `AND ESP, 0FFFFFFF0h` –this instruction aligns the value in the ESP register on a 16-byte boundary. This results in all values in the stack being aligned. (The CPU performs better if the values it is dealing with are located in memory at addresses aligned on a 4- or 16-byte boundary)<sup>7</sup>.

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

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 written directly onto the stack space 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 (executing `MOV EBP, ESP / AND ESP, ...`).

<sup>7</sup>[Wikipedia: Data structure alignment](#)

### 2.1.3 GCC: AT&T syntax

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

Listing 2.4: let's compile in GCC 4.7.3

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

We get this:

Listing 2.5: GCC 4.7.3

```
.file "1_1.c"
.section .rodata
.LC0:
.string "hello, world"
.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
```

There are a lot of macros (beginning with dot). These are not very interesting to us so far. 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>8</sup>:

<sup>8</sup>This GCC option can be used to eliminate "unnecessary" macros: *-fno-asynchronous-unwind-tables*

## Listing 2.6: GCC 4.7.3

```

.LC0:
    .string "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:

- Operands are written backwards.

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

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

Here is a way to think about them: when you deal with Intel-syntax, you can put in equality sign (=) in your mind between operands and when you deal with AT&T-syntax put in a right arrow (→)<sup>9</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 special symbol is to be added to each instruction defining the type of data:
  - 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: 0FFFFFFF0h is written as \$-16. It is the same: 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 usual MOV, not XOR. MOV just loads value to a register. Its name is not intuitive (data is not moved). In other architectures, this instruction has the name "LOAD" or something like that.

<sup>9</sup> By the way, in some C standard functions (e.g., memcpy(), strcpy()) arguments are listed in the same way as in Intel-syntax: pointer to destination memory block at the beginning and then pointer to source memory block.

## 2.2 x86-64

### 2.2.1 MSVC—x86-64

Let's also try 64-bit MSVC:

Listing 2.7: MSVC 2012 x64

```

$SG2989 DB      'hello, world', 00H

main PROC
    sub     rsp, 40
    lea    rcx, OFFSET FLAT:$SG2923
    call   printf
    xor    eax, eax
    add    rsp, 40
    ret    0
main ENDP

```

As of x86-64, all registers were extended to 64-bit and now have a 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: 49.3). I.e., one part of function arguments are passed in registers, other part—via stack. In Win64, 4 function arguments are passed in RCX, RDX, R8, R9 registers. That is what we see here: a pointer to the string for `printf()` is now passed not in stack, but in the RCX register.

Pointers are 64-bit now, so they are passed in the 64-bit part of registers (which have the R- prefix). But for backward compatibility, it is still possible to access 32-bit parts, using the E- prefix.

This is how RAX/EAX/AX/AL looks like in 64-bit x86-compatible CPUs:

7th (byte number)	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 the 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., 32-bit part of register) instead of RAX.

### 2.2.2 GCC—x86-64

Let's also try GCC in 64-bit Linux:

Listing 2.8: GCC 4.4.6 x64



```
.string "hello, world"
main:
    sub     rsp, 8
    mov     edi, OFFSET FLAT:.LC0 ; "hello, world"
    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 [Mit13]. The first 6 arguments are passed in the RDI, RSI, RDX, RCX, R8, R9 registers, and others—via stack.

So the pointer to the string is passed in EDI (32-bit part of 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 writing something into the lower 32-bit register part, also clear the higher 32-bits[Int13]. I.e., the MOV EAX, 011223344h will write a value correctly into RAX, since the higher bits will be cleared.

If we open the compiled object file (.o), we will also see all instruction's opcodes <sup>10</sup>:

Listing 2.9: GCC 4.4.6 x64

```
.text:0000000004004D0          main  proc near
.text:0000000004004D0 48 83 EC 08      sub    rsp, 8
.text:0000000004004D4 BF E8 05 40 00  mov    edi, offset ↵
    ↵ format ; "hello, world"
.text:0000000004004D9 31 C0          xor    eax, eax
.text:0000000004004DB E8 D8 FE FF FF  call   _printf
.text:0000000004004E0 31 C0          xor    eax, eax
.text:0000000004004E2 48 83 C4 08     add    rsp, 8
.text:0000000004004E6 C3            retn
.text:0000000004004E6          main  endp
```

As we can see, the instruction writing into EDI at 0x4004D4 occupies 5 bytes. The same instruction writing a 64-bit value into RDI will occupy 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 EAX register clearance before printf() function call. This is done because a number of used vector registers is passed in EAX by standard: “with variable arguments passes information about the number of vector registers used”[Mit13].

<sup>10</sup>This must be enabled in Options → Disassembly → Number of opcode bytes

## 2.3 GCC—one more thing

The fact *anonymous* C-string has *const* type (2.1.1), and the fact C-strings allocated in constants segment are guaranteed to be immutable, has interesting consequence: compiler may use specific part of 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++-compiler (including MSVC) will allocate two strings, but let's see what GCC 4.8.1 is doing:

Listing 2.10: GCC 4.8.1 + IDA listing

```
f1          proc near
s           = dword ptr -1Ch

           sub     esp, 1Ch
           mov     [esp+1Ch+s], offset s ; "world"
           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',0

```

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

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

This clever trick is often used by at least GCC and can save some memory bytes.

## 2.4 ARM

For my experiments with ARM processors I used several compilers:

- Popular in the embedded area Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE (with LLVM-GCC 4.2 compiler <sup>11</sup>).
- GCC 4.9 (Linaro) (for ARM64), available as win32-executables at <http://www.linaro.org/projects/armv8/>.

32-bit ARM code is used in all cases in this book, if not mentioned otherwise. If we talk about 64-bit ARM here, it will be called ARM64.

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

Let’s start by compiling our example in Keil:

```
armcc.exe --arm --c90 -00 1.c
```

The `armcc` compiler produces assembly listings in Intel-syntax but it has high-level ARM-processor related macros<sup>12</sup>, but it is more important for us to see the instructions “as is” so let’s see the compiled result in `IDA`.

Listing 2.11: Non-optimizing Keil 6/2013 (ARM mode) `IDA`

```

.text:00000000          main
.text:00000000 10 40 2D E9      STMFD  SP!, {R4,LR}

```

<sup>11</sup>It is indeed so: Apple Xcode 4.6.3 uses open-source GCC as front-end compiler and LLVM code generator

<sup>12</sup>e.g. ARM mode lacks PUSH/POP instructions

```
.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>13</sup>, works as an x86 PUSH instruction, writing the values of two registers (R4 and LR<sup>15</sup>) 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 it is not quite precise. The PUSH instruction is only available in thumb mode. So, to make things less messy, that is why I suggested working in IDA.

This instruction first [decrements SP](#)<sup>16</sup> so it will point to the place in the stack that is free for new entries, then it writes the values of the R4 and LR registers at the address in changed SP.

This instruction (like the PUSH instruction in thumb mode) is able to save several register values at once and this may be useful. By the way, there is no such thing 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 pack of registers at the specified memory place.

The `ADR R0, aHelloWorld` instruction adds the value in the PC<sup>17</sup> register to the offset where the *“hello, world”* string is located. How is the PC register used here, one might ask? This is so-called “position-independent code”.<sup>18</sup> It is intended to be executed at a non-fixed address in memory. In the opcode of the ADR instruction, the difference between the address of this instruction and the place where the string is located is encoded. The difference will always be the same, independent of the address where the 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 address of our C-string in memory.

The `BL __2printf`<sup>19</sup> instruction calls the `printf()` function. Here’s how this instruction works:

- write the address following the BL instruction (0xC) into the LR;

<sup>13</sup>STMFD<sup>14</sup>

<sup>15</sup>Link Register

<sup>16</sup>stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.

<sup>17</sup>Program Counter. IP/EIP/RIP in x86/64. PC in ARM.

<sup>18</sup>Read more about it in relevant section (52.1)

<sup>19</sup>Branch with Link

- then pass control flow into `printf()` by writing its address into the **PC** register.

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

That is the difference between “pure” **RISC**-processors like ARM and **CISC**<sup>20</sup>-processors like x86, where the return address is usually stored on the stack<sup>21</sup>.

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 remember, all ARM-mode instructions have a size of 4 bytes (32 bits). Hence they can only be located on 4-byte boundary addresses. This means the 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>22</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>23</sup> is an inverse instruction of **STMFd**. 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** saves the **R4** and **LR** registers pair on the stack, but **R4** and **PC** are *restored* during execution of **LDMFD**.

As I wrote before, the address of the place to where each function must return control is usually saved in the **LR** register. The very first function saves its value in the stack because our `main()` function will use the register in order to call `printf()`. In the function end this value can be written directly to the **PC** register, thus passing control to where our function was called. Since our `main()` function is usually the primary function in C/C++, control will be returned to the **OS** loader or to a point in **CRT**, or something like that.

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

## 2.4.2 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 will get (in **IDA**):

<sup>20</sup>Complex instruction set computing

<sup>21</sup>Read more about this in next section (4)

<sup>22</sup>MOVE

<sup>23</sup>LDMFD<sup>24</sup>

Listing 2.12: Non-optimizing Keil 6/2013 (thumb mode) + IDA

```
.text:00000000          main
.text:00000000 10 B5          PUSH    {R4,LR}
.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 I mentioned, 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 into PC while using the small space in one 16-bit opcode. So 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 I mentioned, all instructions in thumb mode have a size of 2 bytes (or 16 bits). This means 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, BL thumb-instruction can encode the address  $current\_PC \pm \approx 2M$ .

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

### 2.4.3 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 compiler switch `-O3`.

Listing 2.13: 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 `STMTD` 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 [App10]) is a frame pointer. More on it 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. Remember, 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). However, its usage here is redundant because the `MOV R0, #0x1686` instruction above cleared the higher part of the register. This is probably 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 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()`.

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

*Almost* because we need to be sure the string will not contain `printf`-control statements starting with `%`: then the effect of these two functions would be different <sup>25</sup>.

Why did the compiler replace the `printf()` with `puts()`? Probably because `puts()` is faster <sup>26</sup>.

`puts()` works faster because it just passes characters to `stdout` without comparing each to the `%` symbol.

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

## 2.4.4 Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)

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

Listing 2.14: Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)

<code>__text:00002B6C</code>	<code>__hello_world</code>
<code>__text:00002B6C 80 B5</code>	<code>PUSH {R7,LR}</code>
<code>__text:00002B6E 41 F2 D8 30</code>	<code>MOVW R0, #0x13D8</code>
<code>__text:00002B72 6F 46</code>	<code>MOV R7, SP</code>
<code>__text:00002B74 C0 F2 00 00</code>	<code>MOVT.W R0, #0</code>

<sup>25</sup>It should also 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>26</sup>[http://www.cisellant.de/projects/gcc\\_printf/gcc\\_printf.html](http://www.cisellant.de/projects/gcc_printf/gcc_printf.html)

```

__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's easily observable – opcodes of thumb-2 instructions also begin with 0xFx or 0xE<sub>x</sub>. But in the IDA listings two opcode bytes are swapped (for thumb and thumb-2 modes). For instructions in ARM mode, the order is the fourth byte, then the third, then the second and finally the first (due to different [endianness](#)). 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 writes a 16-bit value into the lower part of the R0 register, clearing higher bits.

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

Among other differences, here the BLX instruction is used instead of BL. The difference is that, besides saving the [RA](#)<sup>27</sup> in the LR register and passing control to the puts() function, the processor is also switching from thumb mode to ARM (or back). This instruction is placed here since the instruction to which control is passed looks like (it is encoded in ARM mode):

```

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

```

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). Often-used library functions are stored in dynamic libraries, 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) being imported from external modules along with the names of these modules.

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

<sup>27</sup>Return Address



In our case, `__imp_puts` is a 32-bit variable where the OS loader will write the correct address of the function in an external library. Then the LDR instruction just takes 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 that an OS loader needs for doing this procedure, it is good idea for it to write the address of each symbol only once to a specially-allocated place just for it.

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. So, it is optimal to allocate a separate function working in ARM mode with only one goal – to pass control to the dynamic library and then to jump to this short one-instruction function (the so-called **thunk function**) from thumb-code.

By the way, in the previous example (compiled for ARM mode) control passed by the BL instruction goes to the same **thunk function**. However the processor mode is not switched (hence the absence of an “X” in the instruction mnemonic).

## 2.4.5 ARM64

### GCC

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

Listing 2.15: Non-optimizing GCC 4.8.1 + objdump

```

1 000000000400590 <main>:
2   400590:   a9bf7bfd      stp     x29, x30, [sp,#-16]!
3   400594:   910003fd      mov     x29, sp
4   400598:   90000000      adrp   x0, 400000 <_init-0x3b8>
      ↙ >
5   40059c:   91192000      add     x0, x0, #0x648
6   4005a0:   97fffffa0     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 6f210000 .....Hello!..

```

There are no thumb and thumb-2 modes in ARM64, only ARM, so there are 32-bit instructions only. Registers count is doubled: B.4.1. 64-bit registers has X- prefixes, while its 32-bit parts—W-.

STP instruction (*Store Pair*) saves two registers in stack simultaneously: X29 in X30. Of course, this instruction is able to save this pair at random place of

memory, but `SP` register is specified here, so the pair is saved in stack. ARM64 registers are 64-bit ones, each has size of 8 bytes, so one need 16 bytes for saving two registers.

Exclamation mark after operand mean that 16 will be subtracted from `SP` first, and only then values from registers pair will be written into the stack. This is also called *pre-index*. About difference between *post-index* and *pre-index*, read here: [33.1](#).

Hence, in terms of more familiar x86, the first instruction is just analogous to pair of `PUSH X29` and `PUSH X30`. `X29` is used as `FP`<sup>28</sup> in ARM64, and `X30` as `LR`, so that's why they are saved in function prologue and restored in function epilogue.

The second instruction copies `SP` in `X29` (or `FP`). This is needed for function stack frame setup.

`ADRP` and `ADD` instructions are needed for forming address of the string "Hello!" in the `X0` register, because first function argument is passed in this register. But there are no instructions in ARM allowing to write large number into register (because instruction length is limited by 4 bytes, read more about it here: [33.2.1](#)). So several instructions must be used. The first instruction (`ADRP`) writes address of 4Kb page where string is located into `X0`, and the second one (`ADD`) just adds remainder to the address. Read more about: [33.3](#).

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

`puts()` is called then using `BL` instruction, this was already discussed before: [2.4.3](#).

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

High 32-bit part	low 32-bit part
X0	
	W0

Function result is returning via `X0` and `main()` returning 0, so that's how returning result is prepared. But why 32-bit part? Because `int` data type in ARM64, just like in x86-64, is still 32-bit, for better compatibility. So if function returning 32-bit `int`, only 32 lowest bits of `X0` register should be filled.

In order to get sure about it, I changed by example slightly and recompiled it. Now `main()` returns 64-bit value:

Listing 2.16: `main()` returning a value of `uint64_t` type

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

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

<sup>28</sup>Frame Pointer

```
    return 0;
};
```

Result is very same, but that's how MOV at that line is now looks like:

Listing 2.17: Non-optimizing GCC 4.8.1 + objdump

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

LDP (*Load Pair*) then restores X29 and X30 registers. There are no exclamation mark after instruction: this mean, the value is first loaded from the stack, only then SP value is increased by 16. This is called *post-index*.

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

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

## 2.5 Conclusion

The main difference between x86/ARM and x64/ARM64 code is that pointer to the string is now 64-bit. Indeed, modern CPUs are 64-bit now because memory is cheaper nowadays, we can add much more of it to computers, so 32-bit pointers are not enough to address it. So all pointers are 64-bit now.

## 2.6 Exercises

### 2.6.1 Exercise #1

```
main:
    push 0xFFFFFFFF
    call MessageBeep
    xor  eax,eax
    retn
```

What this win32-function does?

## Chapter 3

# 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 instructions do: saves the value in the EBP register, sets the value of the EBP register to the value of the ESP and then allocates space on the stack for local variables.

The value in the EBP is fixed over a period of function execution and it is to be used for local variables and arguments access. One can use ESP, but it is changing over time and it is not convenient.

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

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

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

### 3.1 Recursion

Epilogues and prologues can make recursion performance worse.

For example, once upon a time I wrote a function to seek the correct node in a binary tree. As a recursive function it would look stylish but since additional time is to be 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 compilers use [tail call](#).

# Chapter 4

## Stack

A stack is one of the most fundamental data structures in computer science <sup>1</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 the block.

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

POP is the reverse operation: get the data from memory pointed to by SP, put it in the operand (often a register) and then add 4 (or 8) to the [stack pointer](#).

After stack allocation the [stack pointer](#) points to the end of stack. PUSH decreases the [stack pointer](#) and POP increases it. The end 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.

Besides support of descending stacks, ARM has also support of ascending stacks.

For example the [STMFD/LDMFD](#), [STMED<sup>2</sup>/LDMED<sup>3</sup>](#) instructions are intended to deal with a descending stack. The [STMFA<sup>4</sup>/LDMFA<sup>5</sup>](#), [STMEA<sup>6</sup>/LDMEA<sup>7</sup>](#) instructions are intended to deal with an ascending stack.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Call\\_stack](http://en.wikipedia.org/wiki/Call_stack)

<sup>2</sup>Store Multiple Empty Descending (ARM instruction)

<sup>3</sup>Load Multiple Empty Descending (ARM instruction)

<sup>4</sup>Store Multiple Full Ascending (ARM instruction)

<sup>5</sup>Load Multiple Full Ascending (ARM instruction)

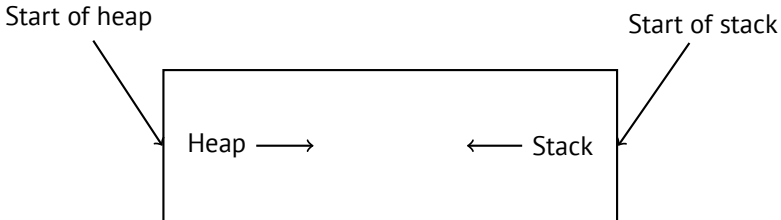
<sup>6</sup>Store Multiple Empty Ascending (ARM instruction)

<sup>7</sup>Load Multiple Empty Ascending (ARM instruction)

## 4.1 Why does the stack grow backward?

Intuitively, we might think that, like any other data structure, the stack may grow upward, i.e., towards higher addresses.

The reason the stack grows backward is probably historical. When 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 [\[RT74\]](#) 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.

## 4.2 What is the stack used for?

### 4.2.1 Save the return address where a function must return control after execution

#### x86

While 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 –it 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 optimization (`/Ox` option) the optimized code will not overflow the stack but instead will work *correctly*<sup>8</sup>:

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

<sup>8</sup>irony here



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

## ARM

ARM programs also use the stack for saving return addresses, but differently. As mentioned in “Hello, world!” (2.4), the **RA** is saved to the **LR** (link register). However, if one needs to call another function and use the **LR** register one more time its value should 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**.

Nevertheless, if a function never calls any other function, in ARM terminology it is called a *leaf function*<sup>9</sup>. As a consequence, leaf functions do not save the **LR** register (because doesn’t modify it). If this 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. This can be faster than on older x86 because external RAM is not used for the stack<sup>10</sup>. It can be useful for such situations when memory for the stack is not yet allocated or not available.

Some examples of leaf functions here are: listing. [7.3.2](#), [7.3.3](#), [18.17](#), [18.26](#), [18.4.4](#), [14.4](#), [14.2](#), [16.1.2](#).

## 4.2.2 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, 4*3
```

**Callee** functions get their arguments via the stack pointer.

Consequently, this is how values will be located in the stack before execution of the very first instruction of the `f()` function:

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

<sup>9</sup><http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faq/ka13785.html>

<sup>10</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 common sense that big number of small function is [anti-pattern](#)[[Ray03](#), Chapter 4, Part II].

See also the section about other calling conventions (49). 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.

For example, it is possible to allocate a space for arguments in the [heap](#), fill it and pass it to a function via a pointer to this block in the EAX register. This will work.<sup>11</sup> However, it is a convenient custom in x86 and ARM to use the stack for this.

By the way, the [callee](#) function does not have any information about how many arguments were passed. Functions with a variable number of arguments (like `printf()`) determine the number by specifiers (which begin with a % sign) in the format string. If we write something like

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

`printf()` will print 1234, and then also two random numbers, which were laying near it in the stack, by chance.

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 not used. If you declare `main()` as `main(int argc, char *argv[])`, you will use two arguments, and third will remain “invisible” for your function. Even more than that, it is possible to declare `main(int argc)`, and it will work.

### 4.2.3 Local variable storage

A function could allocate space in the stack for its local variables just by shifting the [stack pointer](#) towards the stack bottom. Hence, it's very fast, no matter how many local variables defined.

---

<sup>11</sup>For example, in the “The Art of Computer Programming” book by Donald Knuth, in section 1.4.1 dedicated to subroutines[Knu98, section 1.4.1], we can read about one way to supply arguments to a subroutine is simply to list them after the `JMP` instruction passing control to subroutine. Knuth writes this method was particularly convenient on System/360.

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.

## 4.2.4 x86: `alloca()` function

It is worth noting the `alloca()` function.<sup>12</sup>

This function works like `malloc()` but allocates memory just on the stack.

The allocated memory chunk does not need to be freed via a `free()` function call, since the function epilogue (3) will return ESP back to its initial state and the allocated memory will be just annulled.

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:

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

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (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 to the `buf` buffer. `puts()` copies `buf` contents to `stdout`. Of course, these two function calls might be replaced by one `printf()` call, but I would like to illustrate small buffer usage.)

### MSVC

Let's compile (MSVC 2010):

<sup>12</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`

## Listing 4.1: 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            ; 0000001cH

...

```

The sole `alloca()` argument is passed via EAX (instead of pushing into stack) <sup>13</sup>. After the `alloca()` call, ESP points to the block of 600 bytes and we can use it as memory for the `buf` array.

**GCC + Intel syntax**

GCC 4.4.1 can do the same without calling external functions:

## Listing 4.2: GCC 4.7.3

```

.LC0:
    .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

```

<sup>13</sup>It is because `alloca()` is rather compiler intrinsic (74) than usual function.

One of the reason there is a separate function instead of couple instructions just in the code, because [MSVC<sup>14</sup>](#) implementation of the `alloca()` function also has a code which reads from the memory just allocated, in order to let [OS](#) to map physical memory to this [VM<sup>15</sup>](#) region.

```

    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"
↳ %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 4.3: 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)` is analogous to `mov DWORD PTR [esp+20], 3` in Intel-syntax – when addressing memory in form *register+offset*, it is written as *offset(register)* in AT&T syntax.

## 4.2.5 (Windows) SEH

SEH<sup>16</sup> records are also stored on the stack (if they present)..

Read more about it: [\(53.3\)](#).

## 4.2.6 Buffer overflow protection

More about it here [\(17.2\)](#).

## 4.3 Typical stack layout

A very typical stack layout in a 32-bit environment at the start of a function, before first instruction executed:

...	...
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
...	...

## 4.4 Noise in stack

Often in this book, I write about “noise” or “garbage” values in stack or memory. Where are they came from? These are what was left in there after other function’s 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);
};
```

<sup>16</sup>Structured Exception Handling: [53.3](#)

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

Compiling...

Listing 4.4: 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
    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 for 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 I run...

```

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

```

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

Let’s load the example into OllyDbg: [fig.4.1](#).

When `f1()` writes to `a`, `b` and `c` variables, they are stored at the address `0x14F85C` and so on.

And when `f2()` executed: [fig.4.2](#).

... `a`, `b` and `c` of `f2()` are located at the same address! No one overwritten values yet, so they are still untouched here.

So, for this weird situation, several functions should be called one after another and `SP` should be the same at each function entry (i.e., they should has same number of arguments). Then, local variables will be located at the same point of stack.



Summarizing, all values in stack (and memory cells at all) has values left there from previous function executions. They are not random in strict sense, but rather has unpredictable values.

How else? Probably, it would be possible to clear stack portions before each function execution, but that's too much extra (and needless) work.

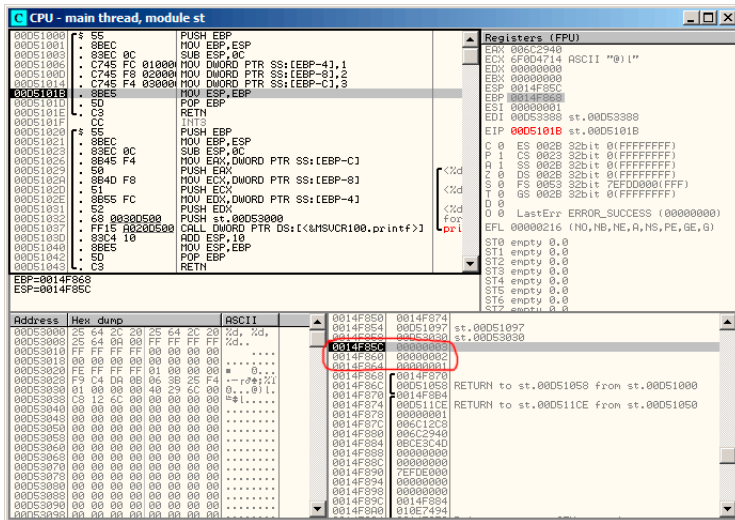


Figure 4.1: OllyDbg: f1 ( )

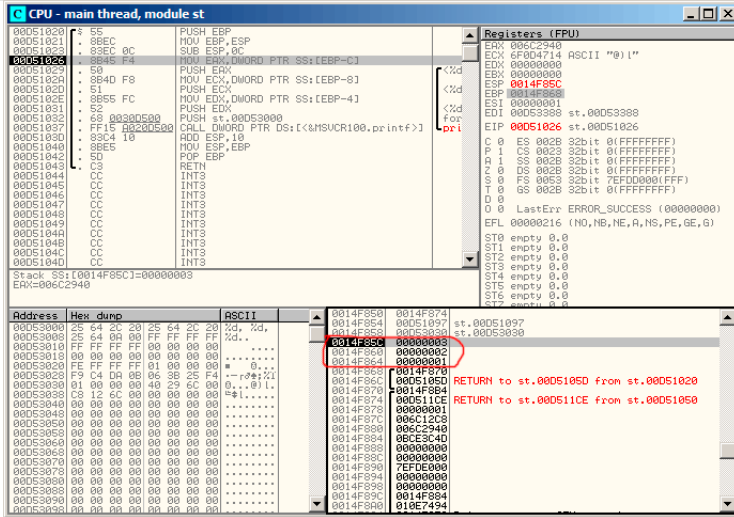


Figure 4.2: OllyDbg: f2()

## 4.5 Exercises

### 4.5.1 Exercise #1

If to compile this piece of code in MSVC and run, a three number will be printed. Where they are came from? Where they are came from if to compile it in MSVC with optimization (/Ox)? Why the situation is completely different in GCC?

```
#include <stdio.h>

int main()
{
    printf ("%d, %d, %d\n");

    return 0;
};
```

Answer: [F.1.1](#).

### 4.5.2 Exercise #2

What this code does?

Listing 4.5: Optimizing MSVC 2010

```
$SG3103 DB      '%d', 0aH, 00H
```

```

_main PROC
    push    0
    call   DWORD PTR __imp__time64
    push   edx
    push   eax
    push   OFFSET $SG3103 ; '%d'
    call  DWORD PTR __imp__printf
    add    esp, 16
    xor    eax, eax
    ret    0
_main ENDP

```

Listing 4.6: Optimizing Keil 6/2013 (ARM mode)

```

main PROC
    PUSH    {r4,lr}
    MOV     r0,#0
    BL     time
    MOV     r1,r0
    ADR     r0,|L0.32|
    BL     __2printf
    MOV     r0,#0
    POP     {r4,pc}
    ENDP

|L0.32|
    DCB     "%d\n",0

```

Listing 4.7: Optimizing Keil 6/2013 (thumb mode)

```

main PROC
    PUSH    {r4,lr}
    MOVS    r0,#0
    BL     time
    MOVS    r1,r0
    ADR     r0,|L0.20|
    BL     __2printf
    MOVS    r0,#0
    POP     {r4,pc}
    ENDP

|L0.20|
    DCB     "%d\n",0

```

Answer: [F.1.1.](#)

## Chapter 5

# printf() with several arguments

Now let's extend the *Hello, world!* (2) example, replacing `printf()` in the `main()` function body by this:

```
#include <stdio.h>

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

## 5.1 x86: 3 arguments

### 5.1.1 MSVC

Let's compile it by MSVC 2010 Express and we got:

```
$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 here 4 arguments.  $4 * 4 = 16$  – 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) is changed back by the ``ADD ESP, X'` instruction after a function call, often, the number of function arguments can be deduced here: just divide X by 4.

Of course, this is specific to the *cdecl* calling convention.

See also the section about calling conventions ([49](#)).

It is also possible for the compiler to merge several ``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

```

## 5.1.2 MSVC and OllyDbg

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

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

Find this code by scrolling the code to the very top (MSVC allocates `main()` function at the very beginning of the code section): [fig.5.3](#).

Click on the `PUSH EBP` instruction, press F2 (set breakpoint) and press F9 (run). We need to do these manipulations in order to skip [CRT-code](#), because, we aren't

really interested in it, yet.

Press F8 (step over) 6 times, i.e., skip 6 instructions: fig.5.4.

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

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

```

0031FE40  6D848584  MSUCR110.6D848584
0031FE44  0031FE68
0031FE48  01281127  1.01281127
0031FE4C  01285140  OFFSET 1.argc
0031FE50  01285144  OFFSET 1.argv
0031FE54  01285000  format = "a=%d; b=%d; c=%d"
0031FE58  00000001  <%d> = 1
0031FE5C  00000002  <%d> = 2
0031FE60  00000003  <%d> = 3
0031FE64  0031FEH4
0031FE68  01281240  RETURN to 1.01281240 from 1.01281005

```

Figure 5.1: OllyDbg: stack after values pushed (I made the round red mark here in a graphics editor)

So 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 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 window at the left-bottom part, where some memory part is always seen. These memory values can be edited. It is possible to change the format string, and then the result of our example will be different. It is probably not very useful now, but it’s a very good idea for doing it as an exercise, to get a feeling of how everything works here.

Press F8 (step over).

In the console we’ll see the output:

```

C:\Polygon\ollydbg\1.exe
a=1; b=2; c=3

```

Figure 5.2: printf() function executed

Let’s see how registers and stack state are changed: fig.5.5.

EAX register now contains 0xD (13). That's correct, since `printf()` returns the number of characters printed. The EIP value is changed: indeed, now there is the address of the instruction after `CALL printf`. ECX and EDX values are changed as well. Apparently, `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 is changed! We clearly see that the format string and corresponding 3 values are still there. Indeed, that's the *cdecl* calling convention: *callee* doesn't return ESP back to its previous value. It's the *caller's* duty to do so.

Press F8 again to execute `ADD ESP, 10` instruction: fig.5.6.

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

The screenshot shows the OllyDbg interface with the following details:

- File View Debug Plugins Options Window Help**
- CPU - main thread, module 1**
- Registers (FPU)**:
  - EAX: 0D0E5634 OFFSET MSUCR110...
  - ECX: 00000013
  - EDX: 00000000
  - ESI: 00000000
  - EDI: 00000000
  - EIP: 01281010 1.main
- Disassembly**:
  - 01281000 CC INT3
  - 01281001 CC INT3
  - 01281002 CC INT3
  - 01281003 > 8E PUSH EBP
  - 01281004 \* 6A 03 MOV EBP, ESP
  - 01281005 \* 5A 03 PUSH 3
  - 01281006 \* 6A 02 PUSH 2
  - 01281007 \* 6A 01 PUSH 1
  - 01281008 \* 68 00002801 PUSH 1, 01280000
  - 01281009 \* FF15 12802801 CALL DWORD PTR DS:[MSUCR110.printf]
  - 0128100A \* 83C4 10 ADD ESP, 10
  - 0128100B \* 5BC XOR EAX, EAX
  - 0128100C \* 50 POP EBP
  - 0128100D \* C3 RETN
  - 0128100E CC INT3
  - 0128100F CC INT3
  - 01281010 CC INT3
  - 01281011 CC INT3
  - 01281012 CC INT3
  - 01281013 CC INT3
  - 01281014 CC INT3
  - 01281015 CC INT3
  - 01281016 CC INT3
  - 01281017 CC INT3
  - 01281018 CC INT3
  - 01281019 CC INT3
  - 0128101A CC INT3
  - 0128101B CC INT3
  - 0128101C CC INT3
  - 0128101D CC INT3
  - 0128101E CC INT3
  - 0128101F CC INT3
  - 01281020 CC INT3
  - 01281021 CC INT3
  - 01281022 CC INT3
  - 01281023 CC INT3
  - 01281024 CC INT3
  - 01281025 CC INT3
  - 01281026 CC INT3
  - 01281027 CC INT3
  - 01281028 CC INT3
  - 01281029 CC INT3
  - 0128102A CC INT3
  - 0128102B CC INT3
  - 0128102C CC INT3
  - 0128102D CC INT3
  - 0128102E CC INT3
  - 0128102F CC INT3
  - 01281030 CC INT3
  - 01281031 CC INT3
  - 01281032 \* FF25 F812801 JMP DWORD PTR DS:[MSUCR110.printf]
  - 01281033 \* 8B 405A0000 MOV EAX, 5A40
  - 01281034 \* 56 3905 00002 CMP WORD PTR DS:[1280000], AX
  - 01281035 \* 74 04 JZ SHORT 1, 01281048
  - 01281036 > 33C XOR EAX, EAX
  - 01281037 \* EB 34 JMP SHORT 1, 0128107E
  - 01281038 > 8B0 3C002801 MOV ECX, DWORD PTR DS:[128003C]
  - 01281039 \* 81B9 00002801 CMP DWORD PTR DS:[ECX+1280000], 4550
  - 0128103A \* 75 E8 JNC SHORT 1, 01281046
  - 0128103B \* 8B 0B010000 MOV EAX, 10B
  - 0128103C \* 52 2801 12802801 CMOB WORD PTR DS:[ECX+1280018], AX
- Stack**:
  - 0031FE63 01281240 RETURN to 1, 01281240 from 00000001
  - 0031FE70 006A6F0 0
  - 0031FE74 006A41E3 0
  - 0031FE78 0D0E5634 0
  - 0031FE7C 00000000 0
  - 0031FE80 00000000 0
  - 0031FE84 7EFD0E00 0
  - 0031FE88 00000000 0
  - 0031FE8C 0031FE78 0
  - 0031FE90 00000204 0
- EBP=0031FE84**
- Jump from 01281005**
- 1.c:\d. (**
- Address Hex dump ASCII**
- 01280000 51 50 25 44 38 28 65 30 87d1 b5
- 01280005 25 64 38 20 63 3D 5E 64 7d; c=0d
- 01280010 00 00 00 00 01 00 00 00 ...0..0
- 01280015 00 00 00 00 00 00 00 00 .....
- 01280020 FE FF FF FF FF FF #
- 01280025 00 61 37 0E 7F 5E C8 24 Ra700000
- 01280030 00 00 00 00 00 00 00 00 .....
- 01280035 00 00 00 00 00 00 00 00 .....
- 01280040 00 00 00 00 00 00 00 00 .....
- 01280045 00 00 00 00 00 00 00 00 .....

Figure 5.3: OllyDbg: the very start of the `main()` function

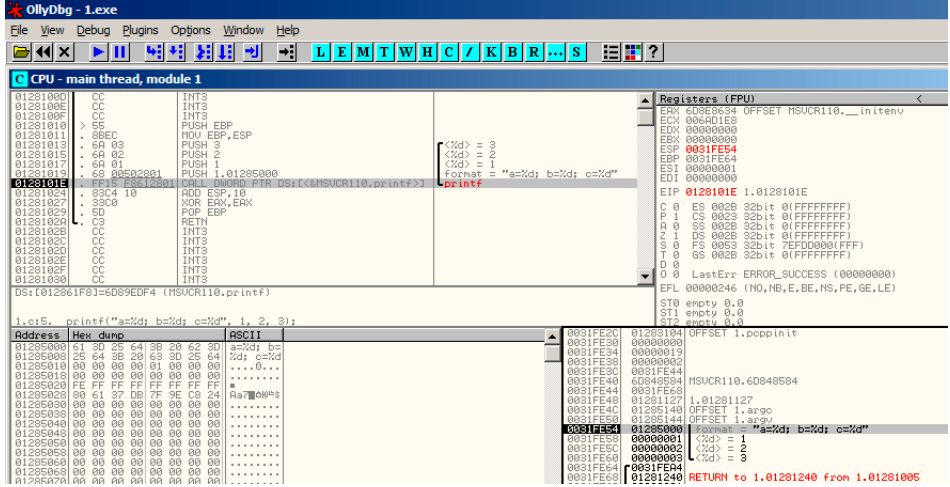


Figure 5.4: OllyDbg: before printf() execution

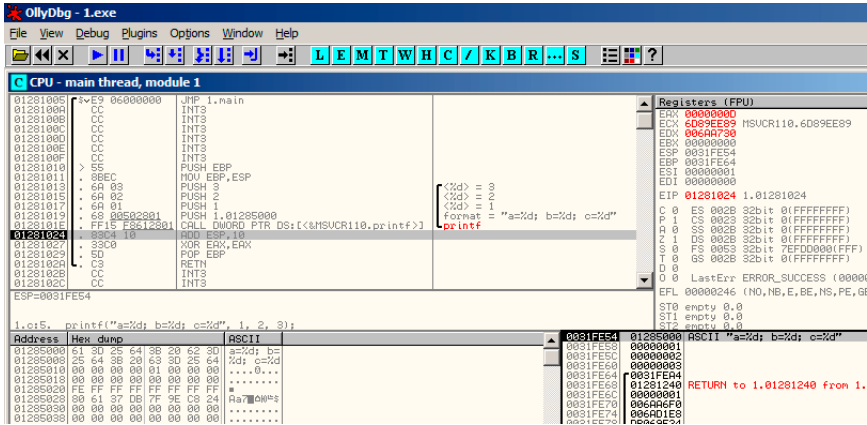


Figure 5.5: OllyDbg: after printf() execution



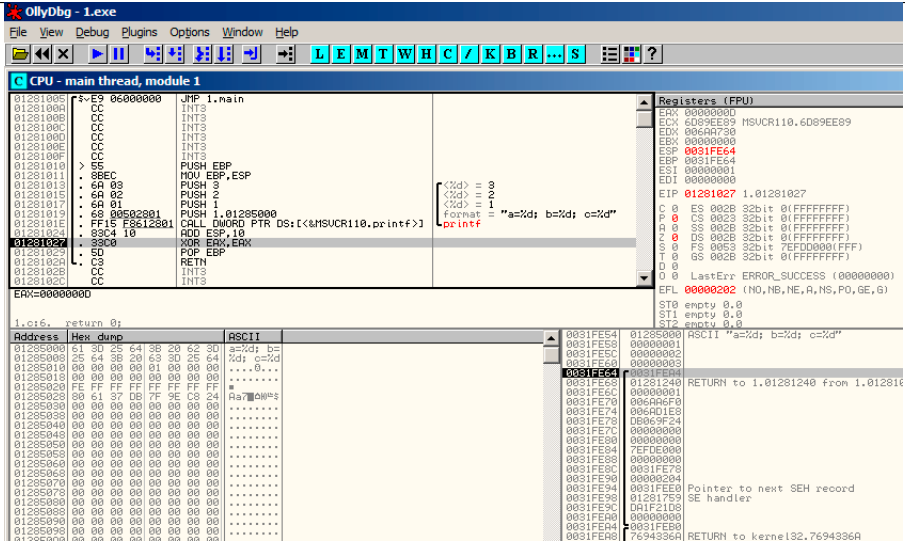


Figure 5.6: OllyDbg: after ADD ESP, 10 instruction execution

### 5.1.3 GCC

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

```

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\n"
                                        ↪
                                        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 can be said that the difference between code from MSVC and code from GCC is only in the method of placing arguments on the stack. Here GCC is working directly with the stack without PUSH/POP.

### 5.1.4 GCC and GDB

Let's try this example also in [GDB](#)<sup>1</sup> in Linux.

-g mean produce debug information into executable file.

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

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/↵
↳ licenses/gpl.html>
This is free software: you are free to change and redistribute ↵
↳ it.
There is NO WARRANTY, to the extent permitted by law. Type "↵
↳ show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 5.1: let's set breakpoint on printf()

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

Run. There are no printf() function source code here, so [GDB](#) can't show its source, 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 left column is an address in stack.

<sup>1</sup>GNU debugger

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

The very first element is the **RA** (0x0804844a). We can make sure by disassembling the memory at this address:

```
(gdb) x/5i 0x0804844a
0x0804844a <main+45>: mov    $0x0,%eax
0x0804844f <main+50>: leave
0x08048450 <main+51>: ret
0x08048451:  xchg  %ax,%ax
0x08048453:  xchg  %ax,%ax
```

Two XCHG instructions, apparently, is some random garbage, which we can ignore so far.

The second element (0x080484f0) is an address of format string:

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

Other 3 elements (1, 2, 3) are printf() arguments. Other elements may be just “garbage” present in stack, but also may be values from other functions, their local variables, etc. We can ignore it for now.

Execute “finish”. This mean, execute all instructions till the function end. Here it means: execute till the finish of printf().

```
(gdb) finish
Run till exit from #0  __printf (format=0x080484f0 "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 number of characters printed, just like in the example with OllyDbg.

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 which C-code line is being executed now? 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 registers. 13 in EAX:

```
(gdb) info registers
eax          0xd          13
ecx          0x0          0
edx          0x0          0
ebx          0xb7fc0000   -1208221696
esp          0xbffff120   0xbffff120
ebp          0xbffff138   0xbffff138
esi          0x0          0
edi          0x0          0
eip          0x804844a     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** shows disassembly in AT&T syntax by default. It's 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, this ends the block of the function.

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

Let's see the registers after the `MOV EAX, 0` instruction execution. EAX here is zero indeed.

```
(gdb) info registers
eax                0x0          0
ecx                0x0          0
edx                0x0          0
ebx                0xb7fc0000    -1208221696
esp                0xbffff120    0xbffff120
ebp                0xbffff138    0xbffff138
esi                0x0          0
edi                0x0          0
eip                0x804844f     0x804844f <main+50>
...
```

## 5.2 x64: 8 arguments

To see how other arguments will be passed via the stack, let's change our example again by increasing the number of arguments to be passed 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",
    ↵ n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
};
```

### 5.2.1 MSVC

As we saw before, the first 4 arguments are passed in the RCX, RDX, R8, R9 registers in Win64, while all the rest—via the stack. That is what we see here. However, the `MOV` instruction, instead of `PUSH`, is used for preparing the stack, so the values are written to the stack in a straightforward manner.

## Listing 5.2: 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

```

Observant reader may ask, what 8 bytes are allocated for *int* values, while 4 is enough? Yes, this should be memorized: 8 bytes are allocated for any data type shorter than 64 bits. It's done for convenience: it's always easy to calculate address of this or that argument. Besides, all they are located at aligned memory addresses. The same story in 32-bit environments: 4 bytes are reserved for all data types.

## 5.2.2 GCC

In \*NIX OS-es, it's the same story for x86-64, except that the first 6 arguments are passed in the RDI, RSI, RDX, RCX, R8, R9 registers. All the rest—via the stack. GCC generates the code writing string pointer into EDI instead if RDI—we saw this thing before: [2.2.2](#).

We also saw before the EAX register being cleared before a `printf()` call: [2.2.2](#).

## Listing 5.3: 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
```

### 5.2.3 GCC + GDB

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

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

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
↳ licenses/gpl.html>
This is free software: you are free to change and redistribute it
↳ it.
There is NO WARRANTY, to the extent permitted by law. Type "show
↳ show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 5.4: 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 has the values which are should be there. RIP has an 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             0x7fffffffce0 140737488346336
r11             0x7ffff7a65f60 140737348263776
r12             0x400440 4195392
r13             0x7fffffff040 140737488347200
r14             0x0             0
r15             0x0             0
rip             0x7ffff7a65f60 0x7ffff7a65f60 <__printf>
...
```

## Listing 5.5: 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` means *giant words*, i.e., 64-bit words.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x000000000400576      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 in stack: 6, 7, 8. We also see that 8 is passed with high 32-bits not cleared: 0x00007fff00000008. That's OK, because the values have *int* type, which is 32-bit type. So, the high register or stack element part may contain "random garbage".

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

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x000000000400576
Dump of assembler code for function main:
0x00000000040052d <+0>:    push    rbp
0x00000000040052e <+1>:    mov     rbp, rsp
0x000000000400531 <+4>:    sub     rsp, 0x20
0x000000000400535 <+8>:    mov     DWORD PTR [rsp+0x10], 0x8
0x00000000040053d <+16>:   mov     DWORD PTR [rsp+0x8], 0x7
0x000000000400545 <+24>:   mov     DWORD PTR [rsp], 0x6
0x00000000040054c <+31>:   mov     r9d, 0x5
0x000000000400552 <+37>:   mov     r8d, 0x4
0x000000000400558 <+43>:   mov     ecx, 0x3
0x00000000040055d <+48>:   mov     edx, 0x2
0x000000000400562 <+53>:   mov     esi, 0x1
0x000000000400567 <+58>:   mov     edi, 0x400628
0x00000000040056c <+63>:   mov     eax, 0x0
0x000000000400571 <+68>:   call   0x400410 <printf@plt>
0x000000000400576 <+73>:   mov     eax, 0x0
0x00000000040057b <+78>:   leave
0x00000000040057c <+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
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
```

rsi	0x7fffffff	2147483609
rdi	0x0	0
rbp	0x7fffffffdf60	0x7fffffffdf60
rsp	0x7fffffffdf40	0x7fffffffdf40
r8	0x7ffff7d26a0	140737351853728
r9	0x7ffff7a60134	140737348239668
r10	0x7fffffff5b0	140737488344496
r11	0x7ffff7a95900	140737348458752
r12	0x400440	4195392
r13	0x7fffffff040	140737488347200
r14	0x0	0
r15	0x0	0
rip	0x40057b	0x40057b <main+78>
...		

## 5.3 ARM: 3 arguments

Traditionally, ARM's scheme for passing arguments (calling convention) is as follows: the first 4 arguments are passed in the R0-R3 registers; the remaining arguments via the stack. This resembles the arguments passing scheme in fastcall (49.3) or win64 (49.5.1).

### 5.3.1 32-bit ARM

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

Listing 5.6: Non-optimizing Keil 6/2013 (ARM mode)

```
.text:00000000 main
.text:00000000 10 40 2D E9   STMFID  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, aADBDCD ; "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 generate same code.

**Optimizing Keil 6/2013 (thumb mode)**

Listing 5.7: 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, aADBDCD ; "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 5.8: 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, aADBDCD ; "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 here 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 also in the fact that there is no function prologue and epilogue (instructions that save R0 and LR registers values). The B instruction just jumps to another address, without any manipulation of the LR register, that is, it is analogous 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 after it. After finishing, the `printf()` function will just return control to the address stored in `LR`. But the address of the point from where our function was called is now in `LR`! Consequently, control from `printf()` will be returned to that point. As a consequence, we do not need to save `LR` since we do not need to modify `LR`. We do not need to modify `LR` since there are no other function calls except `printf()`. Furthermore, after this call we do not to do anything! That's why this optimization is possible.

This optimization is often used in functions where the last statement is a call to another function.

Another similar example will be described in “`switch()/case/default`” section, in (12.1.1).

## 5.3.2 ARM64

### Non-optimizing GCC (Linaro) 4.9

```
.LC1:
    .string "a=%d; b=%d; c=%d"
f2:
    stp    x29, x30, [sp, -16]!
    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
    ldp   x29, x30, [sp], 16
    ret
```

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

`%d` in `printf()` string format is 32-bit *int*, so the 1, 2 and 3 are loaded into 32-bit register parts.

Optimizing GCC (Linaro) 4.9 makes the same code.

## 5.4 ARM: 8 arguments

Let's use again the example with 9 arguments from the previous section: 5.2.

```
void printf_main2()
```

```
{
    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);
};
```

## 5.4.1 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, aADBCDDDEDFDGD ; "a=%d; ↵
    ↵ b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
.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 will use this register for the `printf()` call. Exclamation mark at the end mean *pre-index*. This mean, **SP** will be decreased by 4 at the beginning, then **LR** will be written by the address stored in **SP**. This is analogous to **PUSH** in x86. Read more about it: [33.1](#).

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

- Passing 5, 6, 7 and 8 via stack: they are written to the R0, R1, R2 and R3 registers respectively. Then, the ```ADD R12, SP, #0x18+var_14''` instruction writes an address of the point in the stack, where these 4 variables will be written, into the R12 register. `var_14` is an assembly macro, equal to `-0x14`, which is created by IDA to succinctly denote code accessing the stack. `var_?` macros created by IDA reflect local variables in the stack. So, `SP + 4` will be written into the R12 register. The next ```STMIA R12, R0-R3''` instruction writes the contents of registers R0-R3 to the point in memory to which R12 is pointing. `STMIA` means *Store Multiple Increment After*. *Increment After* means that R12 will be increased by 4 after each register value is written.
- Passing 4 via stack: 4 is stored in R0 and then, this value, with the help of ```STR R0, [SP,#0x18+var_18]''` instruction, is saved on the stack. `var_18` is `-0x18`, so the offset will be 0, so, the value from the R0 register (4) will be written to the point where `SP` is pointing to.
- Passing 1, 2 and 3 via registers:
 

The values of the first 3 numbers (a, b, c) (1, 2, 3 respectively) are passed in 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 returns the `SP` pointer back to its former point, thus cleaning the stack. Of course, what was written on the stack will stay there, but it all will 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 are no exclamation mark—indeed, first `PC` is loaded from the point `SP` pointing to  $(4 + var_4 = 4 + (-4) = 0)$ , so this instruction is analogous to `LDR PC, [SP], #4`, then `SP` increased by 4. This is called *post-index*<sup>2</sup>. Why IDA shows the instruction like that? `var_4` is allocated for saved value of `LR` in the local stack. This instruction is somewhat analogous to `POP PC` in x86<sup>3</sup>.

## 5.4.2 Optimizing Keil 6/2013: thumb mode

```
.text:0000001C          printf_main2
.text:0000001C
```

<sup>2</sup>Read more about it: [33.1](#).

<sup>3</sup>It's impossible to set value of IP/EIP/RIP using `POP` in x86, but anyway, you got the idea, I hope.

```

.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, aADBDCDDDEDFDGD ; "a=%d\
    ↵ ; b=%d; c=%d; d=%d; e=%d; f=%d; g%"...
.text:0000003A 06 F0 D9 F8 BL      __2printf
.text:0000003E
.text:0000003E          loc_3E ; CODE XREF: example13_f+16
.text:0000003E 05 B0      ADD    SP, SP, #0x14
.text:00000040 00 BD      POP    {PC}

```

Almost like the previous example. However, this is thumb code and values are packed into stack differently: 8 for the first time, then 5, 6, 7 for the second, and 4 for the third.

### 5.4.3 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 memory, rather than the opposite order.

Another thing we easily spot is that the instructions are ostensibly located randomly. For instance, the value in the R0 register is prepared in three places, at addresses 0x2918, 0x2920 and 0x2928, when it would be possible to do it in one single point. However, the optimizing compiler has its own reasons for how to place instructions better. 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 modifying the R0 register. On the other hand, ``MOVT R0, #0'' and ``MOV R2, #4'' instructions can be executed simultaneously since effects of their execution are not conflicting with each other. Presumably, compiler tries to generate code in such a way, where it is possible, of course.

#### 5.4.4 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
__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}

```

Almost the same as in the previous example, with the exception that thumb-instructions are used here instead.

## 5.4.5 ARM64

### 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:
    sub    sp, sp, #32
    stp    x29, x30, [sp,16]
    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
    ldp    x29, x30, [sp,16]
    add    sp, sp, 32
    ret

```

First 8 arguments are passed in X- or W-registers: [ARM13c]. Pointer to the string require 64-bit register, so it's passed in X0. All other values has *int* 32-bit type, so these are written in 32-bit part of registers (W-). 9th argument (8) is passed via the stack. Indeed: it's not possible to pass large number of arguments in registers, its count is limited.

Optimizing GCC (Linaro) 4.9 makes the same code.

## 5.5 Conclusion

So here is rough skeleton of function call:

Listing 5.9: x86

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

Listing 5.10: x64 (MSVC)

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

Listing 5.11: x64 (GCC)

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

Listing 5.12: ARM

```
MOV R0, argument 1
MOV R1, argument 2
MOV R2, argument 3
MOV R3, argument 4
; add arguments 5, 6, etc into stack (if needed)
BL function
; modify stack pointer (if needed)
```

Listing 5.13: ARM64

```
MOV X0, argument 1
MOV X1, argument 2
MOV X2, argument 3
MOV X3, argument 4
MOV X4, argument 5
MOV X5, argument 6
MOV X6, argument 7
MOV X7, argument 8
; add arguments 9, 10, etc into stack (if needed)
BL function
; modify stack pointer (if needed)
```

## 5.6 By the way

By the way, this difference between passing arguments in x86, x64, fastcall and ARM is a good illustration of the fact that the CPU is not aware of how arguments are passed to functions. It is also possible to create a hypothetical compiler that is able to pass arguments via a special structure not using stack at all.

# Chapter 6

## scanf()

Now let's use `scanf()`.

```
#include <stdio.h>

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

    scanf ("%d", &x);

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

    return 0;
};
```

OK, I agree, it is not clever to use `scanf()` for user interactions today. But I wanted to illustrate passing pointer to *int* type variable.

### 6.1 About pointers

It is one of the most fundamental things in computer science. Often, large array, structure or object, it is too costly to pass to other function, while passing its address is much easier. More than that: if calling function must modify something in the large array or structure, to return it as a whole is absurd as well. So the simplest thing to do is to pass an address of array or structure to **callee** function, and let **callee** change what must be changed.

In C/C++ it is just an address of some point in memory.

In x86, address is represented as 32-bit number (i.e., occupying 4 bytes), while in x86-64 it is 64-bit number (occupying 8 bytes). By the way, that is the reason

of some people's indignation related to switching to x86-64 –all pointers on x64-architecture require twice as more space.

With some effort, it is possible to work with untyped pointers only; e.g. standard C function `memcpy()`, copying a block from one place in memory to another, takes 2 pointers of `void*` type on input, since it is impossible to predict block data type you would like to copy. Data types is not even important to know, only block size matters.

Also pointers are widely used when function needs to return more than one value (we will back to this further (9)). `scanf()` is just that case. In addition to the function's need to show how many values were read successfully, it also should return all these values.

In C/C++ pointer type is needed only for type checking on compiling stage. Internally, in compiled code, there is no information about pointer types at all.

## 6.2 x86

### 6.2.1 MSVC

What we got after compiling in 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

```

Variable `x` is local.

C/C++ standard tell us it must be visible only in this function and not from any other point. Traditionally, local variables are placed in the local stack. Probably, there could be other ways, but in x86 it is so.

Next instruction after function prologue, `PUSH ECX`, has not a goal to save `ECX` state (notice absence of corresponding `POP ECX` at the function end).

In fact, this instruction just allocates 4 bytes on the stack for `x` variable storage.

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

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

It is also possible to use `ESP`, but since it's changing too often, it's not very convenient. So it can be said, the value of the `EBP` is *frozen state* of the value of the `ESP` at the moment of function execution start.

A very typical [stack frame](#) layout in 32-bit environment is:

...	...
EBP-8	local variable #2, marked in <a href="#">IDA</a> as <code>var_8</code>
EBP-4	local variable #1, marked in <a href="#">IDA</a> as <code>var_4</code>
EBP	saved value of <code>EBP</code>
EBP+4	return address
EBP+8	argument#1, marked in <a href="#">IDA</a> as <code>arg_0</code>
EBP+0xC	argument#2, marked in <a href="#">IDA</a> as <code>arg_4</code>
EBP+0x10	argument#3, marked in <a href="#">IDA</a> as <code>arg_8</code>
...	...

Function `scanf()` in our example has two arguments.

First is pointer to the string containing `'%d'` and second – address of `x` variable.

First of all, address of the `x` variable is placed into the `EAX` register by `lea eax, DWORD PTR _x$[ebp]` instruction

`LEA` meaning *load effective address*, and is often used for forming address of something ([A.6.2](#)).

It can be said, LEA here just stores sum of the value in the EBP register and `_x$` macro to the EAX register.

It is the same as `lea eax, [ebp-4]`.

So, 4 subtracting from value in the EBP register and result is placed to the EAX register. And then value in the EAX register is pushing into stack and `scanf()` is called.

`printf()` is called after that. First argument is pointer to string: `` `You entered %d... \n'`.

Second argument is prepared as: `mov ecx, [ebp-4]`, this instruction places to the ECX not address of the `x` variable, but its contents.

After, value in the ECX is placed on the stack and the last `printf()` called.

## 6.2.2 MSVC + OllyDbg

Let's try this example in OllyDbg. Let's load, press F8 (step over) until we get into our executable file instead of `ntdll.dll`. Scroll up until `main()` appears. Let's click on the first instruction (`PUSH EBP`), press F2, then F9 (Run) and breakpoint triggers on the `main()` begin.

Let's trace to the place where the address of `x` variable is prepared: [fig.6.2](#).

It is possible to right-click on EAX in registers window and then "Follow in stack". This address will appear in stack window. Look, this is a variable in the local stack. I drew a red arrow there. And there are some garbage (`0x77D478`). Now address of the stack element, with the help of `PUSH`, will be written to the same stack, nearly. Let's trace by F8 until `scanf()` execution finished. During the moment of `scanf()` execution, we enter, for example, 123, in the console window:

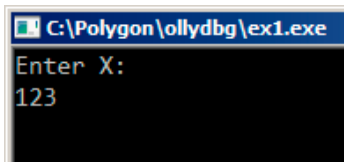


Figure 6.1: Console output

`scanf()` executed here: [fig.6.3](#). `scanf()` returns 1 in EAX, which means, it have read one value successfully. The element of stack of our attention now contain `0x7B` (123).

Further, this value is copied from the stack to the ECX register and passed into `printf()`: [fig.6.4](#).

**CPU - main thread, module ex1**

Address	Hex dump	ASCII
00361270	70 12 36 00 01 00 00 00	...F3...
00361274	FF 3B 89 18 00 00 00 00	...R.....
00361278	00 00 00 00 00 00 00 00	.....

**Registers (FPU)**

EIP	00361025	ex1.00361025
EAX	00000001	
ECX	6C42F1E7	HSUCR110.6C42F1E7
EDX	0077C960	
EBX	00000000	
ESP	00361274	
EBP	00361270	
ESI	00000001	
EDI	00000000	

Figure 6.2: OllyDbg: address of the local variable is computed

**CPU - main thread, module ex1**

Address	Hex dump	ASCII
00361270	70 12 36 00 01 00 00 00	...F3...
00361274	FF 3B 89 18 00 00 00 00	...R.....
00361278	00 00 00 00 00 00 00 00	.....

**Registers (FPU)**

EIP	00361031	ex1.00361031
EAX	00000000	
ECX	6C42F1E7	HSUCR110.6C42F1E7
EDX	0077C960	
EBX	00000000	
ESP	00361274	
EBP	00361270	
ESI	00000001	
EDI	00000000	

Figure 6.3: OllyDbg: scanf() executed

**CPU - main thread, module ex1**

Address	Hex dump	ASCII
00361270	70 12 36 00 01 00 00 00	...F3...
00361274	FF 3B 89 18 00 00 00 00	...R.....
00361278	00 00 00 00 00 00 00 00	.....

**Registers (FPU)**

EIP	00361037	ex1.00361037
EAX	0000007B	
ECX	00000000	
EDX	0077C960	
EAX	00000000	
ESP	00361274	
EBP	00361270	
ESI	00000001	
EDI	00000000	

Figure 6.4: OllyDbg: preparing the value for passing into printf()



## 6.2.3 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
main                endp

```

GCC replaced first the `printf()` call to the `puts()`, it was already described (2.4.3) why it was done.

As before – arguments are placed on the stack by `MOV` instruction.

## 6.3 x64

All the same, but registers are used instead of stack for arguments passing.

### 6.3.1 MSVC

Listing 6.1: MSVC 2012 x64

<code>_DATA</code>	<code>SEGMENT</code>
--------------------	----------------------

```

$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

```

## 6.3.2 GCC

Listing 6.2: 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]

```

```

mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
xor     eax, eax
call    printf

; return 0
xor     eax, eax
add     rsp, 24
ret

```

## 6.4 ARM

### 6.4.1 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}

```

A pointer to a *int*-typed variable must be passed to a `scanf()` so it can return value via it. *int* is 32-bit value, so we need 4 bytes for storing it somewhere in memory, and it fits exactly in 32-bit register. A place for the local variable `x` is allocated in the stack and **IDA** named it `var_8`, however, it is not necessary to allocate it since **SP stack pointer** is already pointing to the space may be used instantly. So, **SP stack pointer** value is copied to the R1 register and, together with format-string, passed into `scanf()`. Later, with the help of the LDR instruction, this value is moved from stack into the R1 register in order to be passed into `printf()`.

Examples compiled for ARM-mode and also examples compiled with Xcode 4.6.3 LLVM are not differ significantly from what we saw here, so they are omitted.

## 6.5 Global variables

What if `x` variable from previous example will not be local but global variable? Then it will be accessible from any point, not only from function body. Global variables are considered as [anti-pattern](#), but for the sake of experiment we could do this.

```
#include <stdio.h>

// make x as global variable
int x;

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

    scanf ("%d", &x);

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

    return 0;
};
```

### 6.5.1 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
call   _printf
add    esp, 8
xor    eax, eax
pop    ebp
ret    0
_main  ENDP
_TEXT  ENDS

```

Now `x` variable is defined in the `._DATA` segment. Memory in local stack is not allocated anymore. All accesses to it are not via stack but directly to process memory. Not initialized global variables takes no place in the executable file (indeed, why we should allocate a place in the executable file for initially zeroed variables?), but when someone will access this place in memory, **OS** will allocate a block of zeroes there<sup>1</sup>.

Now let's assign value to variable explicitly:

```
int x=10; // default value
```

We got:

```

._DATA  SEGMENT
_x      DD      0aH
...

```

Here we see value `0xA` of `DWORD` type (`DD` meaning `DWORD` = 32 bit).

If you will open compiled `.exe` in **IDA**, you will see the `x` variable placed at the beginning of the `._DATA` segment, and after you'll see text strings.

If you will open compiled `.exe` in **IDA** from previous example where `x` value is not defined, you'll see something like this:

```

.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

```

<sup>1</sup>That is how **VM** behaves

```

.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` marked as `?` among other variables not required to be initialized. This means that after loading `.exe` to memory, a space for all these variables will be allocated and filled by zeroes [ISO07, 6.7.8p10]. But in an `.exe` file these not initialized variables are not occupy anything. E.g. it is suitable for large arrays.

## 6.5.2 MSVC: x86 + OllyDbg

Things are even simpler here: fig.6.5. Variable is located in the data segment. By the way, after `PUSH` instruction (pushing `x` address) is executed, the address will appear in stack, and it is possible to right-click on that element and select “Follow in dump”. And the variable will appear in the memory window at left.

After we enter 123 in the console, 0x7B will appear here.

But why the very first byte is 7B? Thinking logically, a 00 00 00 7B should be there. This is what called [endianness](#), and *little-endian* is used in x86. This mean that lowest byte is written first, and highest written last. More about it: [36](#).

Some time after, 32-bit value from this place of memory is loaded into `EAX` and passed into `printf()`.

`x` variable address in the memory is 0xDC3390. In OllyDbg we can see process memory map (Alt-M) and we will see that this address is inside of `.data` PE-segment of our program: fig.6.6.

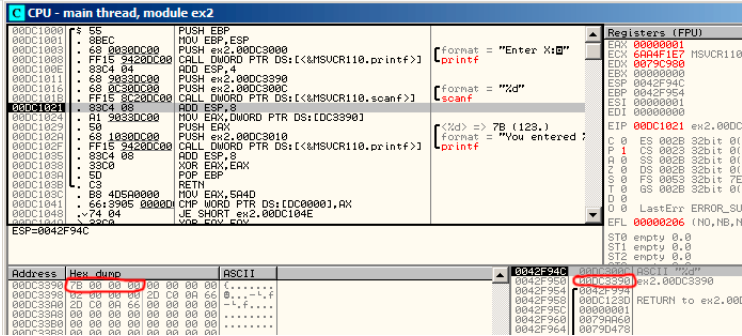


Figure 6.5: OllyDbg: after scanf() execution

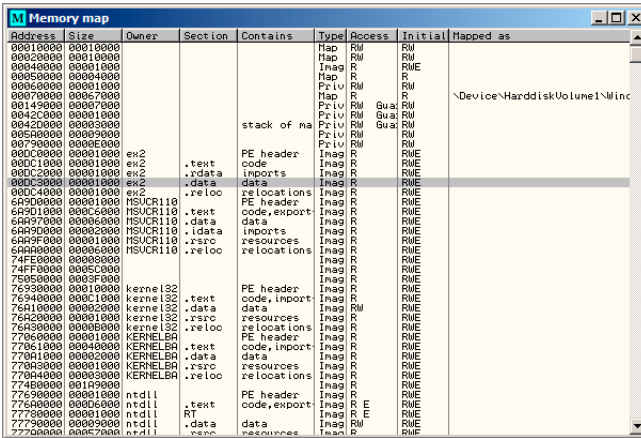


Figure 6.6: OllyDbg: process memory map

### 6.5.3 GCC: x86

It is almost the same in Linux, except segment names and properties: not initialized variables are located in the `_bss` segment. In `ELF` file format this segment has such attributes:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

If to statically assign a value to variable, e.g. 10, it will be placed in the `_data` segment, this is segment with the following attributes:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

## 6.5.4 MSVC: x64

Listing 6.3: 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

_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

```

Almost the same code as in x86. Take a notice that *x* variable address is passed to `scanf()` using LEA instruction, while the value of variable is passed to the second `printf()` using MOV instruction. ``DWORD PTR' '—is a part of assembly language (no related to machine codes), showing that the variable data type is 32-bit and the MOV instruction should be encoded accordingly.

## 6.5.5 ARM: Optimizing Keil 6/2013 (thumb mode)

```

.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

```

So, `x` variable is now global and somehow, it is now located in another segment, namely data segment (*.data*). One could ask, why text strings are located in code segment (*.text*) and `x` can be located right here? Since this is variable, and by its definition, it can be changed. And probably, can be changed very often.

Segment of code can sometimes be located in microcontroller ROM (remember, we now deal with embedded microelectronics, and memory scarcity is common

here), and changeable variables – in RAM<sup>2</sup>. It is not very economically to store constant variables in RAM when one have ROM. Furthermore, constant variables in RAM must be initialized before, since after RAM turning on, it, obviously, contain random information.

Onwards, we see a pointer to the x (off\_2C) variable in code segment, and all operations with variable occurred via this pointer. This is because x variable can be located somewhere far from this code fragment, so its address must be saved somewhere in close proximity to the code. LDR instruction in thumb mode can address only variable in range of 1020 bytes from the point it is located. Same instruction in ARM-mode – variables in range of  $\pm 4095$  bytes. This, address of the x variable must be located somewhere in close proximity, because, there is no guarantee the linker will able to place this variable near the code, it could be even in external memory chip!

One more thing: if variable will be declared as *const*, Keil compiler shall allocate it in the .constdata segment. Perhaps, thereafter, linker will able to place this segment in ROM too, along with code segment.

## 6.6 scanf() result checking

As I noticed before, it is slightly old-fashioned to use scanf() today. But if we have to, we need at least check if scanf() finished correctly without 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, scanf()<sup>3</sup> function returns number of fields it successfully read. In our case, if everything went fine and user entered a number, scanf() will return 1 or 0 (or EOF) in case of error.

I added C code for scanf() result checking and printing error message in case of error.

<sup>2</sup>Random-access memory

<sup>3</sup>MSDN: scanf, wscanf: [http://msdn.microsoft.com/en-us/library/9y6s16x1\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/9y6s16x1(VS.71).aspx)

This works predictably:

```
C:\...\>ex3.exe
Enter X:
123
You entered 123...

C:\...\>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

### 6.6.1 MSVC: x86

What we got in assembly language (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
```

**Caller** function (`main()`) must have access to the result of **callee** function (`scanf()`), so **callee** leaves this value in the EAX register.

After, we check it with the help of instruction `CMP EAX, 1` (*CoMPare*), in other words, we compare value in the EAX register with 1.

JNE conditional jump follows `CMP` instruction. JNE means *Jump if Not Equal*.

So, if value in the EAX register not equals to 1, then the processor will pass execution to the address mentioned in operand of JNE, in our case it is `$LN2@main`. Passing control to this address, **CPU** will execute function `printf()` with argument `'What you entered? Huh?'`. But if everything is fine, conditional

jump will not be taken, and another `printf()` call will be executed, with two arguments: 'You entered %d...' and value of variable `x`.

Since second subsequent `printf()` not needed to be executed, there is `JMP` after (unconditional jump), it will pass control to the point after second `printf()` and before `XOR EAX, EAX` instruction, which implement `return 0`.

So, it can be said that comparing a value with another is *usually* implemented by `CMP/Jcc` instructions pair, where *cc* is *condition code*. `CMP` comparing two values and set processor flags<sup>4</sup>. `Jcc` check flags needed to be checked and pass control to mentioned address (or not pass).

But in fact, this could be perceived paradoxical, but `CMP` instruction is in fact `SUB` (subtract). All arithmetic instructions set processor flags too, not just `CMP`. If we compare 1 and 1,  $1 - 1$  will be 0 in result, `ZF` flag will be set (meaning the last result was 0). There is no any other circumstances when it is possible except when operands are equal. `JNE` checks only `ZF` flag and jumping only if it is not set. `JNE` is in fact a synonym of `JNZ` (*Jump if Not Zero*) instruction. Assembler translating both `JNE` and `JNZ` instructions into same opcode. So, `CMP` instruction can be replaced to `SUB` instruction and almost everything will be fine, but the difference is in the `SUB` alter the value of the first operand. `CMP` is "*SUB without saving result*".

## 6.6.2 MSVC: x86: IDA

It's time to run `IDA` and try to do something in it. By the way, it is good idea to use `/MD` option in `MSVC` for beginners: this mean that all these standard functions will not be linked with executable file, but will be imported from the `MSVCR*.DLL` file instead. Thus it will be easier to see which standard function used and where.

While analysing code in `IDA`, it is very advisable to do notes for oneself (and others). For example, analysing this example, we see that `JNZ` will be triggered in case of error. So it's possible to move cursor to the label, press "n" and rename it to "error". Another label—into "exit". What I've got:

```
.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
```

<sup>4</sup>About x86 flags, see also: [http://en.wikipedia.org/wiki/FLAGS\\_register\\_\(computing\)](http://en.wikipedia.org/wiki/FLAGS_register_(computing)).

```

.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\r
    ↵  "
.text:00401032    call   ds:printf
.text:00401038    add    esp, 8
.text:0040103B    jmp    short exit
.text:0040103D    error: ; CODE XREF: _main+27
.text:0040103D    push   offset aWhat ; "What you entered? \r
    ↵  Huh?\n"
.text:00401042    call   ds:printf
.text:00401048    add    esp, 4
.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's slightly easier to understand the code. However, it's not good idea to comment every instruction excessively.

A part of function can also be hidden in [IDA](#): a block should be marked, then “-” on numerical pad is pressed and text to be entered.

I've hide two parts and gave names to them:

```

.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    error: ; CODE XREF: _main+27
.text:0040103D    push   offset aWhat ; "What you entered? Huh\r
    ↵  ?\n"
.text:00401042    call   ds:printf

```

```
.text:00401048      add  esp, 4
.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
```

To unhide these parts, “+” on numerical pad can be used.

By pressing “space”, we can see how **IDA** can represent a function as a graph: fig.6.7. There are two arrows after each conditional jump: green and red. Green arrow pointing to the block which will be executed if jump is triggered, and red if otherwise.

It is possible to fold nodes in this mode and give them names as well (“group nodes”). I did it for 3 blocks: fig.6.8.

It’s very useful. It can be said, a very important part of reverse engineer’s job is to reduce information he/she have.

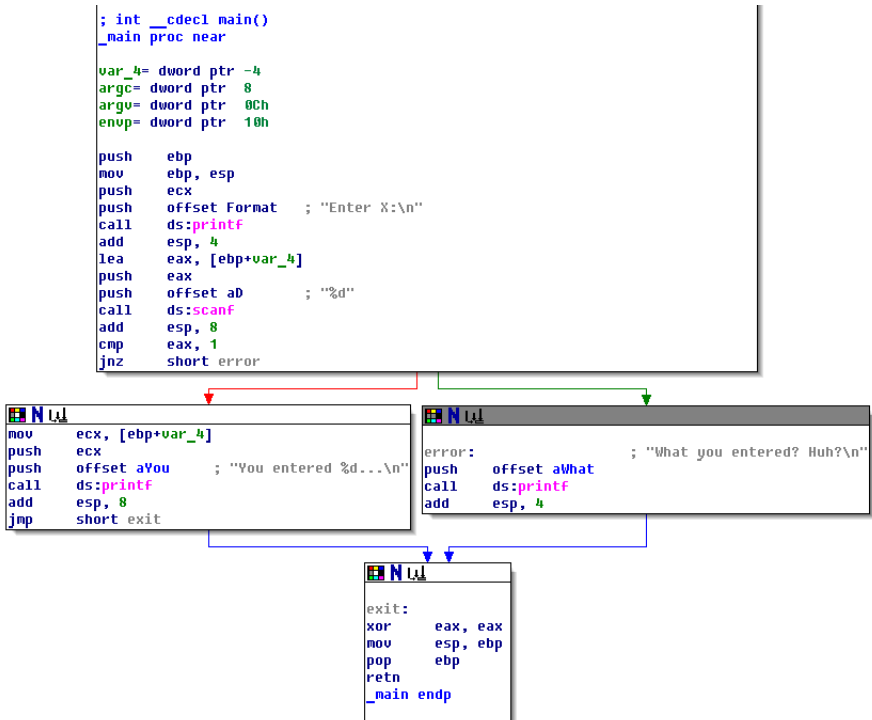


Figure 6.7: Graph mode in IDA

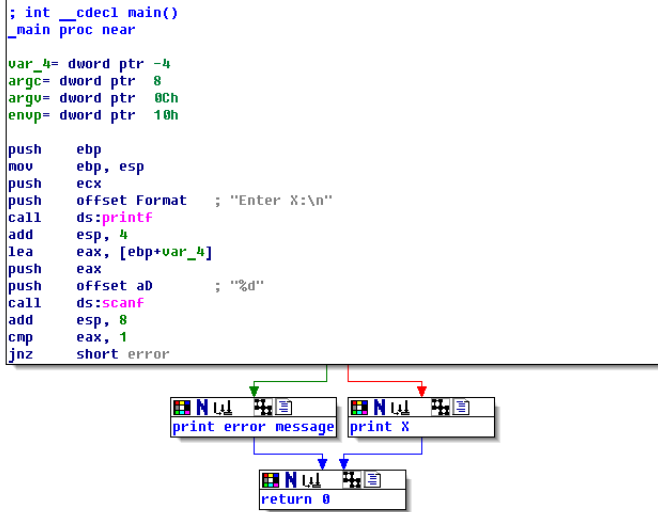


Figure 6.8: Graph mode in IDA with 3 nodes folded

### 6.6.3 MSVC: x86 + OllyDbg

Let's try to hack our program in OllyDbg, forcing it to think `scanf()` working always without error.

When address of local variable is passed into `scanf()`, initially this variable contain some random garbage, that is `0x4CD478` in case: fig.6.10.

When `scanf()` is executing, I entered something definitely not a number in the console, like "asdasd". `scanf()` finishing with 0 in EAX, which mean, an error occurred: fig.6.11.

We can also see to the local variable in the stack and notice that it's not changed. Indeed, what `scanf()` would write there? It just did nothing except returning zero.

Now let's try to "hack" our program. Let's right-click on EAX, there will also be "Set to 1" among other options. This is what we need.

1 now in EAX, so the following check will executed as we need, and `printf()` will print value of variable in the stack.

Let's run (F9) and we will see this in console window:

Figure 6.9: console window

Indeed, 5035128 is a decimal representation of the number in stack (0x4CD478)!

Figure 6.10: OllyDbg: passing variable address into scanf ( )

Figure 6.11: OllyDbg: scanf ( ) returning error

## 6.6.4 MSVC: x86 + Hiew

This can be also a simple example of executable file patching. We may try to patch executable, so the program will always print numbers, no matter who we entered.

Assuming the executable compiled against external MSVC\*.DLL (i.e., with /MD option)<sup>5</sup>, we may find main ( ) function at the very beginning of .text section. Let's open executable in Hiew, find the very beginning of .text section (Enter, F8, F6, Enter, Enter).

We will see this: fig.6.12.

Hiew finds ASCIIZ<sup>6</sup> strings and displays them, as well as imported function names.

<sup>5</sup>that's what also called "dynamic linking"

<sup>6</sup>ASCII Zero (null-terminated ASCII string)



Move cursor to the address `.00401027` (with the `JNZ` instruction we should bypass), press `F3`, and then type “9090” (meaning two `NOP`<sup>7</sup>-s): fig.6.13.

Then `F9` (update). Now the executable saved to disk. It will behave as we wanted.

Two `NOP`-s are probably not quite aesthetically as it could be. Other 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 can do the opposite: replace first byte to `EB` while not touching the second byte (`jump offset`). We’ll got here always triggered unconditional jump. The error message will be printed always, no matter what number was entered.

```

C:\Polygon\ollydbg\ex3.exe  FRO ----- a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 5F          push     ebp
.00401001: 8BEC       mov     ebp,esp
.00401003: 51        push     ecx
.00401004: 6800304000 push     000403000 ; 'Enter X:' --E1
.00401009: FF1594204000 call    printf
.0040100F: 83C404    add     esp,4
.00401012: 8D45FC    lea    eax,[ebp][-4]
.00401015: 50        push     eax
.00401016: 680C304000 push     00040300C --E2
.0040101B: FF159C204000 call    scanf
.00401021: 83C408    add     esp,8
.00401024: 83F801    cmp     eax,1
.00401027: 7514     jnz     .00040103D --E3
.00401029: 8B4DFC    mov     ecx,[ebp][-4]
.0040102C: 51        push     ecx
.0040102D: 6810304000 push     000403010 ; 'You entered %d...' --E4
.00401032: FF1594204000 call    printf
.00401038: 83C408    add     esp,8
.0040103B: EB0E     jmps    .00040104B --E5
.0040103D: 6824304000 push     000403024 ; 'What you entered? Huh?' --E6
.00401042: FF1594204000 call    printf
.00401048: 83C404    add     esp,4
.0040104B: 33C0     xor     eax,eax
.0040104D: 8BE5     mov     esp,ebp
.0040104F: 5D        pop     ebp
.00401050: C3        retn    ; ~~~~~
.00401051: B84D5A0000 mov     eax,00005A4D ; 'ZM'
1Global 2Fi1Bk 3CryBk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9Ibyte 10Leave 11Naked 12AddName

```

Figure 6.12: Hiew: `main()` function

<sup>7</sup>No Operation



```

    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?'
    call   printf
$LN1@main:
    ; return 0
    xor    eax, eax
    add    rsp, 56
    ret    0
main     ENDP
_TEXT   ENDS
END

```

## 6.6.7 ARM: Optimizing Keil 6/2013 (thumb mode)

Listing 6.5: 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, aYouEnteredD___ ; "You entered %d↵
        ↵ ... \n"
        BL    __2printf
        B     loc_1A

```

New instructions here are `CMP` and `BEQ`<sup>8</sup>.

`CMP` is akin to the x86 instruction bearing the same name, it subtracts one argument from another and saves flags.

`BEQ` is jumping to another address if operands while comparing were equal to each other, or, if result of last computation was 0, or if Z flag is 1. Same thing as `JZ` in x86.

Everything else is simple: execution flow is forking into two branches, then the branches are converging at the point where 0 is written into the `R0`, as a value returned from the function, and then function finishing.

---

<sup>8</sup>(PowerPC, ARM) Branch if Equal

# Chapter 7

## Accessing passed arguments

Now we figured out the [caller](#) function passing arguments to the [callee](#) via stack. But how [callee](#) access them?

Listing 7.1: 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;
};
```

### 7.1 x86

#### 7.1.1 MSVC

What we have after compilation (MSVC 2010 Express):

Listing 7.2: 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 the 3 numbers are pushing to stack in function `main()` and `f(int, int, int)` is called then. Argument access inside `f()` is organized with the help of macros like: `_a$ = 8`, in the same way as local variables accessed, but the difference in that these offsets are positive (addressed with *plus* sign). So, adding `_a$` macro to the value in the EBP register, *outer* side of [stack frame](#) is addressed.

Then `a` value is stored into EAX. After `IMUL` instruction execution, value in the EAX is a [product](#) of value in EAX and what is stored in `_b`. After `IMUL` execution, `ADD` is summing value in EAX and what is stored in `_c`. Value in the EAX is not needed to be moved: it is already in place it must be. Now return to [caller](#) – it will take value from the EAX and used it as `printf()` argument.

### 7.1.2 MSVC + OllyDbg

Let's illustrate this in OllyDbg. When we trace until the very first instruction in `f()` that uses one of the arguments (first one), we see that EBP is pointing to the [stack](#)

frame, I marked its begin with red arrow. The first element of **stack frame** is saved EBP value, second is **RA**, third is first function argument, then second argument and third one. To access the first function argument, one need to add exactly 8 (2 32-bit words) to EBP.

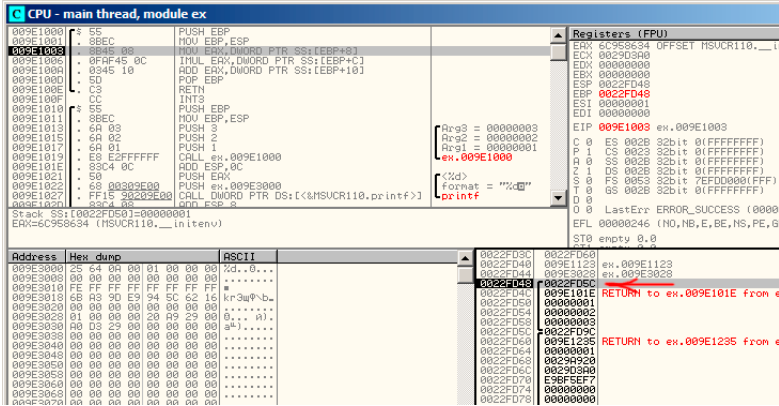


Figure 7.1: OllyDbg: inside of f() function

### 7.1.3 GCC

Let's compile the same in GCC 4.4.1 and let's see results in [IDA](#):

Listing 7.3: GCC 4.4.1

```

f
public f
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] ; 1st argument
    imul   eax, [ebp+arg_4] ; 2nd argument
    add    eax, [ebp+arg_8] ; 3rd argument
    pop    ebp
    retn

f
endp

public 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

```

Almost the same result.

The [stack pointer](#) is not returning back after both function execution, because penultimate LEAVE ([A.6.2](#)) instruction will do this, at the end.

## 7.2 x64

The story is a bit different in x86-64, function arguments (4 or 6) are passed in registers, and a [callee](#) reading them from there instead of stack.

### 7.2.1 MSVC

Optimizing MSVC:

Listing 7.4: 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
        ; R8D - 3rd argument
        imul   ecx, edx
        lea    eax, DWORD PTR [r8+rcx]
        ret    0
f      ENDP

```

As we can see, very compact `f()` function takes arguments right from the registers. LEA instruction is used here for addition, apparently, compiler considered this instruction here faster than ADD. LEA is also used in `main()` for the first and third arguments preparing, apparently, compiler thinks that it will work faster than usual value loading to the register using MOV instruction.

Let's try to take a look on output of non-optimizing MSVC:

Listing 7.5: 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

```

Somewhat puzzling: all 3 arguments from registers are saved to the stack for some reason. This is called “shadow space”<sup>1</sup>: every Win64 may (but not required to) save all 4 register values there. This is done by two reasons: 1) it is too lavish to allocate the whole register (or even 4 registers) for the input argument, so it will be accessed via stack; 2) debugger is always aware where to find function arguments at a break<sup>2</sup>.

So, some large functions can save input arguments in “shadows space” if they need to use it while execution, but some small functions, like ours, may not do this.

It is duty of **caller** to allocate “shadow space” in stack.

## 7.2.2 GCC

Optimizing GCC does more or less understandable code:

Listing 7.6: 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

```

<sup>1</sup>[http://msdn.microsoft.com/en-us/library/zthk2dkh\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/zthk2dkh(v=vs.80).aspx)

<sup>2</sup>[http://msdn.microsoft.com/en-us/library/ew5tede7\(v=VS.90\).aspx](http://msdn.microsoft.com/en-us/library/ew5tede7(v=VS.90).aspx)

```

xor    eax, eax
add    rsp, 8
ret

```

Non-optimizing GCC:

Listing 7.7: 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” requirement in System V \*NIX[Mit13], but *callee* may need to save arguments somewhere, in case of registers shortage.

### 7.2.3 GCC: `uint64_t` instead of `int`

Our example worked with 32-bit *int*, that is why 32-bit register parts were 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,
                        0x1111111122222222,
                        0x3333333344444444));
    return 0;
};

```

Listing 7.8: 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, 3333333344444444h ; 3rd argument
                mov   rsi, 1111111122222222h ; 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 very same, but registers (prefixed by R-) are *used as a whole*.

## 7.3 ARM

### 7.3.1 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}

```

In `main()` function, two other functions are simply called, and three values are passed to the first one (`f`).

As I mentioned before, first 4 values are usually passed in first 4 registers (R0-R3) in ARM.

`f` function, as it seems, use first 3 registers (R0-R2) as arguments.

`MLA` (*Multiply Accumulate*) instruction multiplies two first operands (R3 and R1), adds third operand (R2) to product and places result into zeroth register (R0), via which, by standard, values are returned from functions.

Multiplication and addition at once<sup>3</sup> (*Fused multiply-add*) is very useful operation, by the way, there was no such instruction in x86 before FMA-instructions appearance in SIMD<sup>4</sup>.

The very first `MOV R3, R0` instruction, apparently, redundant (single `MLA` instruction could be used here instead), compiler was not optimized it, since this is non-optimizing compilation.

`BX` instruction returns control to the address stored in the `LR` register and, if it is necessary, switches processor mode from thumb to ARM or vice versa. This can be necessary since, as we can see, `f` function is not aware, from which code it may be called, from ARM or thumb. This, if it will be called from thumb code, `BX` will not only return control to the calling function, but also will switch processor mode to thumb mode. Or not switch, if the function was called from ARM code [ARM12, A2.3.2].

<sup>3</sup>wikipedia: [Multiply-accumulate operation](#)

<sup>4</sup>[https://en.wikipedia.org/wiki/FMA\\_instruction\\_set](https://en.wikipedia.org/wiki/FMA_instruction_set)

### 7.3.2 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 `f` function compiled by Keil compiler in full optimization mode (-O3). `MOV` instruction was optimized (or reduced) and now `MLA` uses all input registers and also places result right into `R0`, exactly where calling function will read it and use.

### 7.3.3 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
```

`MLA` instruction is not available in thumb mode, so, compiler generates the code doing these two operations separately. First `MULS` instruction multiply `R0` by `R1` leaving result in the `R1` register. Second (`ADDS`) instruction adds result and `R2` leaving result in the `R0` register.

### 7.3.4 ARM64

#### Optimizing GCC (Linaro) 4.9

That's easy. `MADD` is just an instruction doing fused multiply/add (similar to `MLA` we already saw). All 3 arguments are passed in 32-bit part of X-registers. Indeed, argument types are 32-bit *int*'s. The result is returned in `W0`.

```
f:
    madd    w0, w0, w1, w2
    ret

main:
    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
```

```

        mov     w0, 0
        ldp    x29, x30, [sp], 16
        ret

.LC7:
        .string "%d\n"

```

I also extended all data types to 64-bit `uint64_t` and tried it:

```

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
{
    printf ("%lld\n", f9(0x1122334455667788,
                        0x1111111122222222,
                        0x3333333344444444));
    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"

```

Function `f()` is just the same, but whole 64-bit X-registers are now used. Long 64-bit values are loaded into registers by parts, I described it also here: [33.2.1](#).

## Non-optimizing GCC (Linaro) 4.9

Non-optimizing compiler 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 input arguments in the local stack, for a case if someone (or something) in this function will need to use W0 . . . W2 registers by overwriting original function arguments, but they may be needed again in future. So this is *Register Save Area* [ARM13c] however, callee is not obliged to save them. This is somewhat similar to “Shadow Space”: 7.2.1.

Why optimizing GCC 4.9 dropped this arguments saving code? Because it did some additional optimizing work and concluded that function arguments will not be needed in future and W0 . . . W2 registers will also not be used.

We also see MUL/ADD instruction pair instead of single MADD.



## Chapter 8

# More about results returning

As of x86, function execution result is usually returned <sup>1</sup> in the EAX register. If it is byte type or character (*char*) –then in the lowest register EAX part –AL. If a function returns *float* number, the FPU register ST(0) is to be used instead. In ARM, result is usually returned in the R0 register.

### 8.1 Attempt to use result of function returning *void*

By the way, what if returning value of the `main()` function will be declared not as *int* but as *void*?

so-called startup-code is calling `main()` roughly as:

```
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* and nothing will be returned explicitly (by *return* statement), then something random, that was stored in the EAX register at the moment of the `main()` finish, will come into the sole `exit()` function argument. Most likely, there will be a random value, left from your function execution. So, exit code of program will be pseudorandom.

---

<sup>1</sup>See also: MSDN: Return Values (C++): <http://msdn.microsoft.com/en-us/library/7572ztz4.aspx>

I can illustrate this fact. Please notice, the `main()` function has `void` type:

```
#include <stdio.h>

void main()
{
    printf ("Hello, world!\n");
};
```

Let's compile it in Linux.

GCC 4.8.1 replaced `printf()` to `puts()` (we saw this before: [2.4.3](#)), but that's OK, since `puts()` returns number of characters printed, just like `printf()`. Please notice that `EAX` is not zeroed before `main()` finish. This means, `EAX` value at the `main()` finish will contain what `puts()` left there.

Listing 8.1: 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 bash script, showing exit status:

Listing 8.2: `tst.sh`

```
#!/bin/sh
./hello_world
echo $?
```

And run it:

```
$ tst.sh
Hello, world!
14
```

14 is a number of characters printed.

## 8.2 What if not to use function result?

`printf()` returns count of characters successfully sent to output, but result of this function is rarely used in practice. It's possible to call functions which essence in returning values, but not to use them explicitly:

```
int f()
{
    // skip first 3 random values
    rand();
    rand();
    rand();
    // and use 4th
    return rand();
};
```

Result of `rand()` function will always be leaved in EAX, in all four cases. But in first 3 cases, a value in EAX will be just thrown away.

## 8.3 Returning a structure

Let's back to the fact the returning value is left in the EAX register. That is why old C compilers cannot create functions capable of returning something not fitting in one register (usually type `int`) but if one needs it, one should return information via pointers passed in function arguments. So, usually, if a function should return several values, it returns only one, and all the rest—via pointers. Now it is possible, to return, let's say, whole structure, but still it is not very popular. If function must return a large structure, `caller` must allocate it and pass pointer to it via first argument, transparently for programmer. That is almost the same as to pass pointer in first argument manually, but compiler hide this.

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 /Ox):

```

$T3853 = 8                ; size = 4
_a$ = 12                 ; size = 4
?get_some_values@@YA?AUs@@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
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP                ; get_some_values

```

Macro name for internal variable passing pointer to structure is \$T3853 here. This example can be rewritten using 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 8.3: 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 may see, the function is just filling fields in the structure, allocated by caller function. As if a pointer to the structure was passed. So there are no performance drawbacks.

# Chapter 9

## Pointers

Pointers are often used to return values from function (recall `scanf()` case (6)). For example, when function should return two values.

### 9.1 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 compiling into:

Listing 9.1: 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]
    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 ; ↗
↳ 000001c8H
    push    123 ; ↗
↳ 0000007bH
    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 ; ↗
↳ 0000001cH
    xor     eax, eax
    ret     0
_main ENDP

```

Let's see this in OllyDbg: fig.9.1. At first, global variables addresses are passed into `f1()`. We can click "Follow in dump" on the stack element, and we will see a place in data segment allocated for two variables. These variables are cleared, because non-initialized data (BSS<sup>1</sup>) are cleared before execution begin: [ISO07, 6.7.8p10]. They are residing in data segment, we can be sure it is so, by pressing Alt-M and seeing memory map: fig.9.5.

<sup>1</sup>Block Started by Symbol

Let's trace (F7) until execution of `f1()`: [fig.9.2](#). Two values are seen in the stack 456 (0x1C8) and 123 (0x7B), and two global variables addresses as well.

Let's trace until the end of `f1()`. At the window at left we see how calculation results are appeared in the global variables: [fig.9.3](#).

Now values of global variables are loaded into registers for passing into `printf()`: [fig.9.4](#).

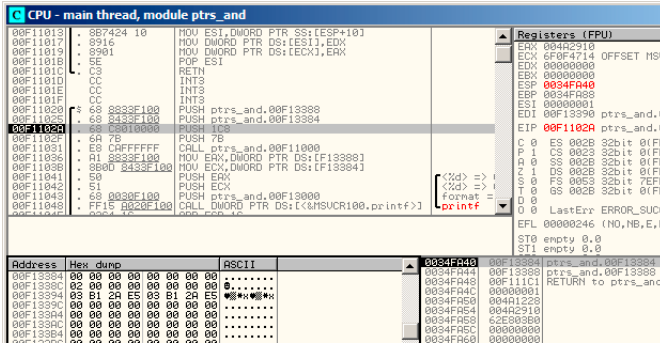


Figure 9.1: OllyDbg: global variables addresses are passing into `f1()`

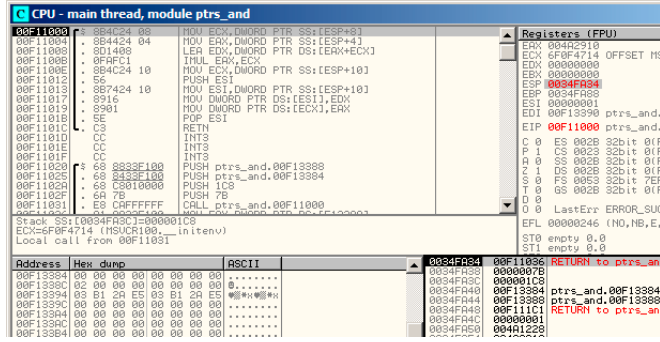


Figure 9.2: OllyDbg: `f1()` is started



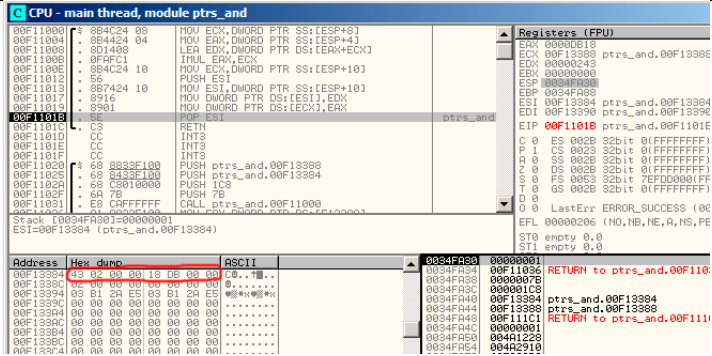


Figure 9.3: OllyDbg: f1 ( ) finishes

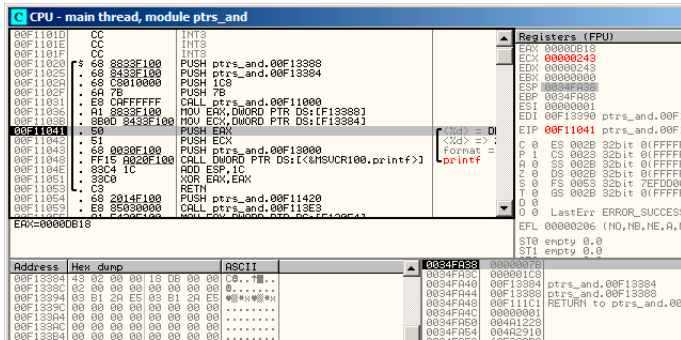


Figure 9.4: OllyDbg: global variables addresses are passed into printf ( )

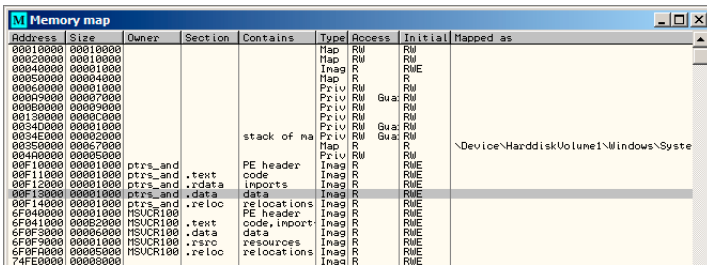


Figure 9.5: OllyDbg: memory map

## 9.2 Local variables example

Let's rework our example slightly:

Listing 9.2: now 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() function code will not be changed. Only main() code will:

Listing 9.3: 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
    add    esp, 36 ; ↗
    ↪ 00000024H
    ret    0
```

Let's again take a look into OllyDbg. Local variable addresses in the stack are 0x35FCF4 and 0x35FCF8. We see how these are pushed into the stack: fig.9.6.

f1() is started. Random garbage are at 0x35FCF4 and 0x35FCF8 so far: fig.9.7.

f1() finished. There are 0xDB18 and 0x243 now at 0x35FCF4 and 0x35FCF8 addresses, these values are f1() function result.

The screenshot shows the CPU window with the following registers:

- ECX: 0035FCF8
- EDI: 00F83388 ptrs\_and.00F83388
- EIP: 00F8102B ptrs\_and.00F8102B

The memory dump window shows the following data:

Address	Hex dump	ASCII
0035FCF4	01 00 00 00 C0 11 F8 00 0...	4
0035FCF8	01 00 00 00 28 12 15 00 0...	...
0035FD00	10 29 15 00 13 DE E5 19  13 11 4	
0035FD10	00 00 00 00 00 00 00 00 .....	
0035FD18	00 00 00 00 78 FD 35 00 ...HMS	
0035FD20	00 00 00 00 0C FD 35 00 ...HMS	
0035FD28	48 C2 DE F3 78 FD 35 00 ...HMS	
0035FD30	09 16 F8 00 17 02 28 19  .. 04	
0035FD38	00 00 00 00 48 FD 35 00 ...HMS	
0035FD40	6A 58 94 76 00 59 FD 7E  33 0*	
0035FD48	58 FD 35 00 72 9F 6C 77  HMS.r1lw	

Figure 9.6: OllyDbg: addresses of local variables are pushed into the stack

The screenshot shows the CPU window with the following registers:

- ECX: 0035FCF4
- EDI: 00000000
- EIP: 00F81000 ptrs\_and.00

The memory dump window shows the following data:

Address	Hex dump	ASCII
0035FCF4	01 00 00 00 C0 11 F8 00 0...	4
0035FD00	01 00 00 00 28 12 15 00 0...	...
0035FD08	10 29 15 00 13 DE E5 19  13 11 4	
0035FD10	00 00 00 00 00 00 00 00 .....	
0035FD18	00 00 00 00 78 FD 35 00 ...HMS	
0035FD20	00 00 00 00 0C FD 35 00 ...HMS	
0035FD28	48 C2 DE F3 78 FD 35 00 ...HMS	
0035FD30	09 16 F8 00 17 02 28 19  .. 04	
0035FD38	00 00 00 00 48 FD 35 00 ...HMS	
0035FD40	6A 58 94 76 00 59 FD 7E  33 0*	
0035FD48	58 FD 35 00 72 9F 6C 77  HMS.r1lw	

Figure 9.7: OllyDbg: f1() starting

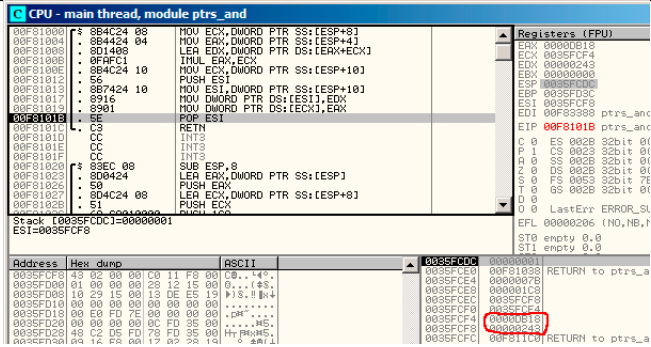


Figure 9.8: OllyDbg: `f1()` finished

### 9.3 Conclusion

`f1()` can return results to any place in memory, located anywhere. This is essence and usefulness of pointers.

By the way, C++ *references* works just in the same way. Read more about them: (31.3).

# Chapter 10

## GOTO

GOTO operator considered harmful [[Dij68](#)], but nevertheless, can be used reasonably [[Knu74](#)], [[Yur13](#), p. 1.3.2].

Here is a simplest possible example:

```
#include <stdio.h>

int main()
{
    printf ("begin\n");
    goto exit;
    printf ("skip me!\n");
exit:
    printf ("end\n");
};
```

Here is what we've got is MSVC 2012:

Listing 10.1: MSVC 2012

```
$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

```

So the *goto* statement is just replaced by JMP instruction, which has the very same effect: unconditional jump to another place.

The second `printf()` call can be executed only with the help of human intervention, using debugger or patching.

This also could be a simple patching exercise. Let's open resulting executable in Hiew: fig.10.1.

Place cursor to the address of JMP (0x410), press F3 (edit), press two zeroes, so the opcode will be EB 00: fig.10.2.

The second byte of JMP opcode mean relative offset of jump, 0 means the point right after current instruction. So now JMP will not skip second `printf()` call.

Now press F9 (save) and exit. Now we run executable and we see this: fig.10.3.

The same effect can be achieved if to replace JMP instruction by 2 NOP instructions. NOP has 0x90 opcode and length of 1 byte, so we need 2 instructions as replacement.

```

Hiew: goto.exe
C:\Polygon\goto.exe
FRO ----- a32 PE
.00401000: 55      push    ebp
.00401001: 8BEC    mov     ebp,esp
.00401003: 6800304000  push   000403000 ;'begin
.00401008: FF1590204000 call   printf
.0040100E: 83C404  add    esp,4
.00401011: EB0E    jmps   .000401021 --2
.00401013: 6808304000  push   000403008 ;'skip
.00401018: FF1590204000 call   printf
.0040101E: 83C404  add    esp,4
.00401021: 6814304000 2push  000403014 --4
.00401026: FF1590204000 call   printf
.0040102C: 83C404  add    esp,4
.0040102F: 33C0    xor    eax,eax
.00401031: 5D     pop    ebp
.00401032: C3     retn ; _^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_

```

Figure 10.1: Hiew

```

Hiew: goto.exe
C:\Polygon\goto.exe  FFWO EDITMODE  a32 PE
00000400: 55          push        ebp
00000401: 8BEC       mov        ebp,esp
00000403: 6800304000 push       000403000 ; '@0'
00000408: FF1590204000 call      d,[000402090]
0000040E: 83C404     add        esp,4
00000411: EB00       jmps      000000413
00000413: 6808304000 push       000403008 ; '@0'
00000418: FF1590204000 call      d,[000402090]
0000041E: 83C404     add        esp,4
00000421: 6814304000 push       000403014 ; '@0'
00000426: FF1590204000 call      d,[000402090]
0000042C: 83C404     add        esp,4
0000042F: 33C0      xor        eax,eax
00000431: 5D        pop        ebp
00000432: C3        retm ; _^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_

```

Figure 10.2: Hiew

```

C:\Polygon>goto.exe
begin
skip me!
end

```

Figure 10.3: Result

## 10.1 Dead code

The second `printf()` call is also called “dead code” in compiler’s term. This mean, the code will never be executed. So when you compile this example with optimization, compiler removing “dead code” leaving no trace of it:

Listing 10.2: 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:
    call    _printf
    add    esp, 8
    xor    eax, eax
    ret    0
_main    ENDP
```

However, compiler forgot to remove the “skip me!” string.

## 10.2 Exercise

Try to achieve the same result using your favorite compiler and debugger.



# Chapter 11

## Conditional jumps

### 11.1 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;
};
```

**11.1.1 x86****x86 + MSVC**

What we have in the `f_signed()` function:

Listing 11.1: 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'
    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

```

First instruction `JLE` means *Jump if Less or Equal*. In other words, if second operand is larger than first or equal, control flow will be passed to address or label mentioned in instruction. But if this condition will not trigger (second operand less than first), control flow will not be altered and first `printf()` will be called. The second check is `JNE`: *Jump if Not Equal*. Control flow will not be altered if operands are equals to each other. The third check is `JGE`: *Jump if Greater or Equal*—jump if the first operand is larger than the second or if they are equals to each other. By the way, if all three conditional jumps are triggered, no `printf()` will be called whatsoever. But, without special intervention, it is impossible.

`f_unsigned()` function is likewise, with the exception the `JBE` and `JAE` instructions are used here instead of `JLE` and `JGE`, see below about it:

Now let's take a look to the `f_unsigned()` function

Listing 11.2: 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

```

Almost the same, with exception of instructions: `JBE`—*Jump if Below or Equal* and `JAE`—*Jump if Above or Equal*. These instructions (`JA`/`JAE`/`JBE`/`JBE`) are distinct from `JG`/`JGE`/`JL`/`JLE` in that way, they works with unsigned numbers.

See also section about signed number representations (35). So, where we see usage of `JG`/`JL` instead of `JA`/`JBE` or otherwise, we can almost be sure about signed or unsigned type of variable.

Here is also `main()` function, where nothing much new to us:

Listing 11.3: `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

```

### x86 + MSVC + OllyDbg

We can see how flags are set by running this example in OllyDbg. Let's begin with `f_unsigned()` function, which works with unsigned number. `CMP` executed thrice here, but for the same arguments, so flags will be the same each time.

First comparison results: fig.11.1. So, the flags are: `C=1`, `P=1`, `A=1`, `Z=0`, `S=1`, `T=0`, `D=0`, `O=0`. Flags are named by one character for brevity in OllyDbg.

OllyDbg gives a hint that (`JBE`) jump will be triggered. Indeed, if to take a look into `[Int13]`, we will read there that `JBE` will trigger if `CF=1` or `ZF=1`. Condition is true here, so jump is triggered.

The next conditional jump: fig.11.2. OllyDbg gives a hint that `JNZ` will trigger. Indeed, `JNZ` will trigger if `ZF=0` (zero flag).

The third conditional jump `JNB`: fig.11.3. In `[Int13]` we may find that `JNB` will trigger if `CF=0` (carry flag). It's not true in our case, so the third `printf()` will execute.

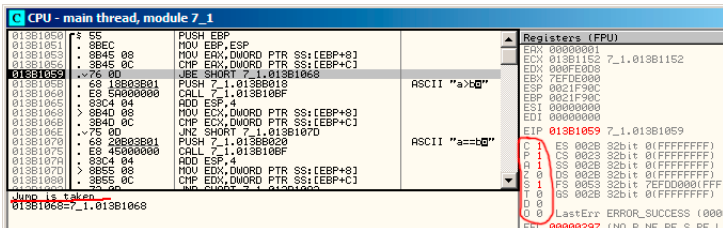


Figure 11.1: OllyDbg: `f_unsigned()`: first conditional jump

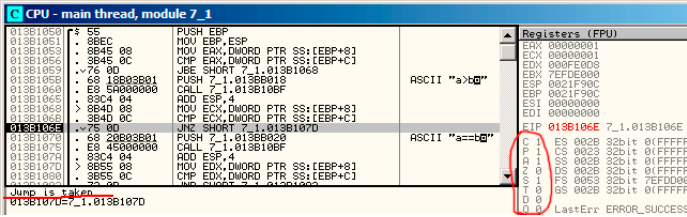


Figure 11.2: OllyDbg: `f_unsigned()`: second conditional jump

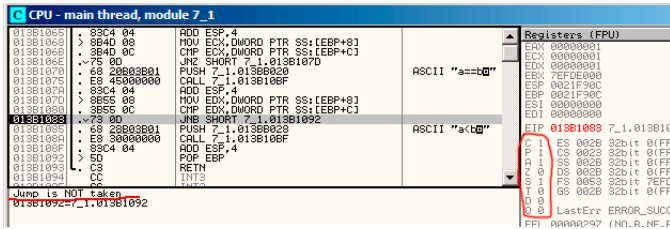


Figure 11.3: OllyDbg: `f_unsigned()`: third conditional jump

Now we can try in OllyDbg the `f_signed()` function which works with signed values.

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` will trigger: fig.11.4. In [Int13] we may find that this instruction is triggering if `ZF=1` or `SF≠OF`. `SF≠OF` in our case, so jump is triggering.

The next `JNZ` conditional jump will trigger: it does if `ZF=0` (zero flag): fig.11.5.

The third conditional jump `JGE` will not trigger because it will only if `SF=OF`, and that is not true in our case: fig.11.6.

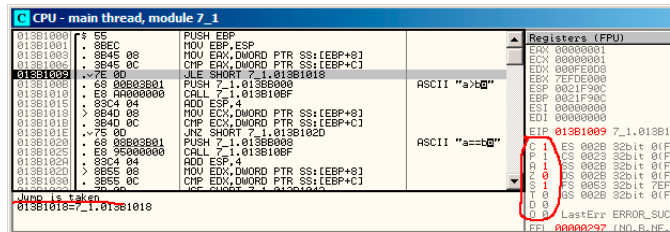


Figure 11.4: OllyDbg: `f_unsigned()`: first conditional jump

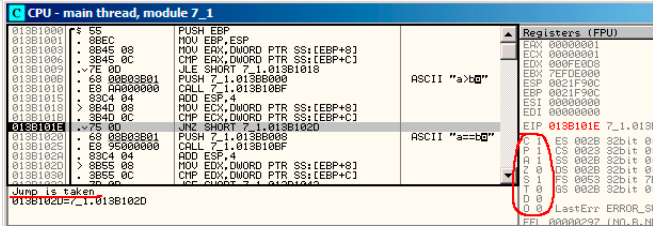


Figure 11.5: OllyDbg: f\_unsigned(): second conditional jump

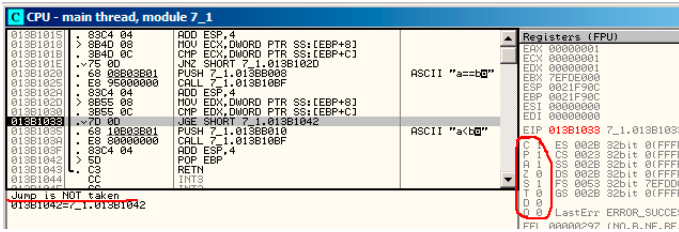


Figure 11.6: OllyDbg: f\_unsigned(): third conditional jump

x86 + MSVC + Hiew

We can try patch executable file in that way, that f\_unsigned() function will always print “a==b”, for any input values.

Here is how it looks in Hiew: fig.11.7.

Essentially, we've got three tasks:

- force first jump to be always triggered;
- force second jump to be never triggered;
- force third jump to be always triggered.

Thus we can point code flow into the second printf(), and it always print “a==b”.

Three instructions (or bytes) should be patched:

- The first jump will now be JMP, but jump offset will be same.
- The second jump may be triggered sometimes, but in any case it will jump to the next instruction, because, we set jump offset to 0. Jump offset is just to be added to the address of the next instruction in these instructions. So if offset is 0, jump will be done to the next instruction.

- The third jump we convert into `JMP` just as the first one, so it will be triggered always.

That's what we do: [fig.11.8](#).

If we could forget about any of these jumps, then several `printf()` calls may execute, but this is not behaviour we're need.

```

C:\Polygon\ollydbg\7_1.exe  FRO -----  a32 PE .00401000  Hiew 8.02 (c)SEN
.00401000: 55      push   ebp
.00401001: 8BEC   mov    ebp, esp
.00401003: 8B4508 mov    eax, [ebp][8]
.00401006: 3B450C cmp    eax, [ebp][00C]
.00401009: 7E0D   jle    .00401018 --E1
.0040100B: 6800B04000 push 0040B000 --E2
.00401010: E8AA000000 call   .004010BF --E3
.00401015: 83C404 add    esp, 4
.00401018: 8B4D08 mov    ecx, [ebp][8]
.0040101B: 3B4D0C cmp    ecx, [ebp][00C]
.0040101E: 750D   jnz    .0040102D --E4
.00401020: 6800B04000 push 0040B008; 'a=b' --E5
.00401025: E895000000 call   .004010BF --E3
.0040102A: 83C404 add    esp, 4
.0040102D: 8B5508 mov    edx, [ebp][8]
.00401030: 3B550C cmp    edx, [ebp][00C]
.00401033: 7D0D   jge    .00401042 --E6
.00401035: 6810B04000 push 0040B010 --E7
.0040103A: E880000000 call   .004010BF --E3
.0040103F: 83C404 add    esp, 4
.00401042: 5D     pop    ebp
.00401043: C3     retn  ; -----
.00401044: CC     int    3
.00401045: CC     int    3
.00401046: CC     int    3
.00401047: CC     int    3
.00401048: CC     int    3
1Global 2Fib1k 3Cry61k 4ReLoac 5ondLdr 6String 7Direct 8Table 9Ibyte 10Leave 11Naked 12AddNam

```

Figure 11.7: Hiew: `f_unsigned()` function

```

C:\Polygon\ollydbg\7_1.exe  FWO 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: 68000400  push   00040B00 ; '*'
00000410: E8AA0000  call   000004BF
00000415: 83C404     add     esp, 4
00000418: 8B4D08     mov     ecx, [ebp+8]
0000041B: 3B4D0C     cmp     ecx, [ebp+00C]
0000041E: 7509      jnz    00000420
00000420: 68000400  push   00040B00 ; '*'
00000425: E8950000  call   000004BF
0000042A: 83C404     add     esp, 4
0000042D: 8B5508     mov     edx, [ebp+8]
00000430: 3B550C     cmp     edx, [ebp+00C]
00000433: EB0D      jmps    00000442
00000435: 68100400  push   00040B10 ; '*'
0000043A: E8800000  call   000004BF
0000043F: 83C404     add     esp, 4
00000442: 5D        pop     ebp
00000443: C3        ret    ; ~~~~~
00000444: CC        int    3
00000445: CC        int    3
00000446: CC        int    3
00000447: CC        int    3
00000448: CC        int    3
1 2 NOps 3 4 5 6 7 8 Table 9 10 11 12

```

Figure 11.8: Hiew: let's modify `f_unsigned()` function

## Non-optimizing GCC

Non-optimizing GCC 4.4.1 produce almost the same code, but with `puts()` (2.4.3) instead of `printf()`.

## Optimizing GCC

Observant reader may ask, why to execute `CMP` several times, if flags are same after each execution? Perhaps, optimizing MSVC can't do this, but optimizing GCC 4.8.1 can optimize more deeply:

Listing 11.4: 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:
    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 be described later: [12.1.1](#).

Needless to say, that 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 it somewhere, it may be a good probability that the code is hand-written.

`f_unsigned()` function is not that aesthetically short:

Listing 11.5: 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      ; instruction may 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
---------------

But nevertheless, there are two CMP instructions instead of three. So, optimization algorithms of GCC 4.8.1 are probably not always perfect yet.

## 11.1.2 ARM

### 32-bit ARM

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

Listing 11.6: 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                        STMFD    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
```

A lot of instructions in ARM mode can be executed only when specific flags are set. E.g. this is often used while numbers comparing.

For instance, ADD instruction is ADDAL internally in fact, where AL meaning *Always*, i.e., execute always. Predicates are encoded in 4 high bits of 32-bit ARM instructions (*condition field*). B instruction of unconditional jump is in fact conditional and encoded just like any other conditional jumps, but has AL in the *condition field*, and what it means, executing always, ignoring flags.

ADRGT instructions works just like ADR but will execute only in the case when previous CMP instruction, while comparing two numbers, found one number greater than another (*Greater Than*).

The next BLGT instruction behaves exactly as BL and will be triggered only if result of comparison was the same (*Greater Than*). ADRGT writes a pointer to the string ``a>b\n'', into R0 and BLGT calls printf(). Consequently, these instructions with -GT suffix, will be executed only in the case when value in the R0 (*a* is there) was bigger than value in the R4 (*b* is there).

Then we see ADREQ and BLEQ instructions. They behave just like ADR and BL but is to be executed only in the case when operands were equal to each other during last comparison. Another CMP is located before them (because printf() call may tamper state of flags).

Then we see LDMGEFD, this instruction works just like LDMFD<sup>1</sup>, but will be triggered only in the case when one value was greater or equal to another while comparison (*Greater or Equal*).

The sense of ``LDMGEFD SP!, {R4-R6,PC}'' instruction is that is like function epilogue, but it will be triggered only if  $a \geq b$ , only then function execution will be finished. But if it is not true, i.e.,  $a < b$ , then control flow come to next ``LDMFD SP!, {R4-R6,LR}'' instruction, this is one more function epilogue, this instruction restores R4-R6 registers state, but also LR instead of PC, thus, it does not returns from function. Last two instructions calls printf() with the string «a<b\n» as sole argument. Unconditional jump to the printf() function instead of function return, is what we already examined in «printf() with several arguments» section, here (5.3.1).

f\_unsigned is likewise, but ADRHI, BLHI, and LDMCSFD instructions are used there, these predicates (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) are analogical to those examined before, but serving for unsigned values.

There is not much new in the main() function for us:

Listing 11.7: main()

```
.text:00000128          EXPORT main
.text:00000128          main
.text:00000128  10 40 2D E9          STMFD  SP!, {R4,LR}
.text:0000012C  02 10 A0 E3          MOV    R1, #2
.text:00000130  01 00 A0 E3          MOV    R0, #1
.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's how to get rid of conditional jumps in ARM mode.

Why it is so good? Read here: [38.1](#).

<sup>1</sup>LDMFD

There is no such feature in x86, if not to consider CMOVcc instruction, it is the same as MOV, but triggered only when specific flags are set, usually set while value comparison by CMP.

## Optimizing Keil 6/2013 (thumb mode)

Listing 11.8: 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
.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
.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
.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 usual conditional jump *Less than or Equal*, BNE—*Not Equal*, BGE—*Greater than or Equal*.

f\_unsigned function is just likewise, but other instructions are used while dealing with unsigned values: BLS (*Unsigned lower or same*) and BCS (*Carry Set (Greater than or equal)*).

## ARM64: Optimizing GCC (Linaro) 4.9

Listing 11.9: 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
        adrp  x0, .LC11      ; "a<b"
        add   x0, x0, :lo12:.LC11
        b     puts
.L19:
    adrp  x0, .LC9      ; "a>b"
    add   x0, x0, :lo12:.LC9
    b     puts
.L15:
    ; impossible here
    ret
.L20:
    adrp  x0, .LC10     ; "a==b"
    add   x0, x0, :lo12:.LC10
    b     puts

```

Listing 11.10: 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

```

I added comments. What is also striking is that compiler is not aware that some conditions are not possible at all, so there are dead code at some places, which never will be executed.

## Exercise

Try to optimize these functions manually, by size, removing redundant instructions, while not adding new ones.

## 11.2 Conditional operator

Conditional operator in C/C++ is:

```
expression ? expression : expression
```

Now here is an example:

```

const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
};

```

### 11.2.1 x86

Listing 11.11: 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
      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 11.12: 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

```

Latest compilers may be more concise:

Listing 11.13: 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 RDX value ('it is ten')
; if not, do nothing. pointer to the string 'it is not ten' is ↘
      ↙ still in RDX as for now.
      cmov  rax, rdx
      ret   0
f      ENDP

```

Optimizing GCC 4.8 for x86 also use `CMOVcc` instruction, while non-optimizing GCC 4.8 use conditional jumps.

## 11.2.2 ARM

Optimizing Keil for ARM mode also use conditional instructions `ADRcc`:

Listing 11.14: 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

```

Without manual intervention, both `ADREQ` and `ADRNE` instructions cannot be executed.

Optimizing Keil for Thumb mode ought to use conditional jump instructions, since there are no load instruction supporting conditional flags:



Listing 11.15: 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

```

### 11.2.3 ARM64

Optimizing GCC (Linaro) 4.9 for ARM64 also use conditional jumps:

Listing 11.16: 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's because ARM64 hasn't simple load instruction with conditional flags, like `ADRCC` in 32-bit ARM mode or `CMOVcc` in x86 [ARM13a, p390, C5.5]. It has, however, "Conditional SElect" instruction (CSEL), but GCC 4.9 is probably not that good to generate it in such piece of code.

### 11.2.4 Let's rewrite it in 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 also was able to generate CMOVcc in this case:

Listing 11.17: 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 a code identical to listing.11.14. But optimizing MSVC 2012 is not that good (yet).

## 11.2.5 Conclusion

Why optimizing compilers try to get rid of conditional jumps? Read here about it: [38.1](#).

## 11.2.6 Exercise

Try to rewrite the code in listing.11.16 by removing all conditional jump instructions, and use CSEL instruction.

# 11.3 Conclusion

## 11.3.1 x86

Rough skeleton of conditional jump:

## Listing 11.18: 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:

```

## Listing 11.19: 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:

```

## 11.3.2 ARM

It's possible to use conditional suffixes in ARM mode for some instructions:

## Listing 11.20: 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 are no limit of number of instructions with conditional code suffixes, as long as CPU flags are not modified by any of them.

Thumb mode has IT instruction, allowing to add conditional suffixes to the next four instructions. Read more about it: [16.3.2](#).

## Listing 11.21: 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

```

# Chapter 12

## switch()/case/default

### 12.1 Few 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
};
```

#### 12.1.1 x86

##### Non-optimizing MSVC

Result (MSVC 2010):

Listing 12.1: 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
    pop     ebp
    ret     0
_f   ENDP

```

Our function with a few cases in `switch()`, in fact, is 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, was it real `switch()` in source code, or just pack of `if()` statements. This means, `switch()` is like syntactic sugar for large number of nested checks constructed using `if()`.

Nothing especially new to us in generated code, with the exception the compiler moving input variable `a` to temporary local variable `tv64`<sup>1</sup>.

If to compile the same in GCC 4.4.1, we'll get almost the same, 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 12.2: 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, 00
    ↪ H
    jmp     _printf
$LN3@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00
    ↪ H
    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.

<sup>1</sup>Local variables in stack prefixed with `tv` – that's how MSVC names internal variables for its needs

First: the value of the  $\alpha$  variable is placed into EAX and 0 subtracted from it. Sounds absurd, but it may needs to check if 0 was in the EAX register before? If yes, flag ZF will be set (this also means that subtracting from 0 is 0) and first conditional jump JE (*Jump if Equal* or synonym JZ –*Jump if Zero*) will be triggered and control flow passed to the \$LN4@f label, where 'zero' message is being printed. If first jump was not triggered, 1 subtracted from the input value and if at some stage 0 will be resulted, corresponding jump will be triggered.

And if no jump triggered at all, control flow passed to the printf() with argument 'something unknown' string.

Second: we see unusual thing for us: string pointer is placed into the  $\alpha$  variable, and then printf() is called not via CALL, but via JMP. This could be explained simply. **Caller** pushing to stack a value and calling our function via CALL. CALL itself pushing returning address (RA) to stack and do unconditional jump to our function address. Our function at any point of execution (since it do not contain any instruction moving stack pointer) has the following stack layout:

- ESP—pointing to RA
- ESP+4—pointing to the  $\alpha$  variable

On the other side, when we need to call printf() here, we need exactly the same stack layout, except of first printf() argument pointing to string. And that is what our code does.

It replaces function's first argument to address of the string and jumping to the printf(), as if not our function f() was called firstly, but immediately printf(). printf() printing a string to **stdout** and then execute RET instruction, which POPping RA from stack and control flow is returned not to f() but rather to the f()'s **callee**, bypassing end of f() function.

All this is possible since printf() is called right at the end of the f() function in any case. In some way, it is all similar to the longjmp()<sup>2</sup> function. And of course, it is all done for the sake of speed.

Similar case with ARM compiler described in “printf() with several arguments”, section, here (5.3.1).

## OllyDbg

Since this example is tricky, let's trace it in OllyDbg.

OllyDbg can detect such switch() constructs, so its add some useful comments. EAX is 2 at start, that's function's input value: fig.12.1

0 is subtracted from 2 in EAX. Of course, EAX is still contain 2. But ZF flag is now 0, indicating that resulting value is non-zero: fig.12.2.

<sup>2</sup><http://en.wikipedia.org/wiki/Setjmp.h>

DEC is executed and EAX now contain 1. But 1 is non-zero, so the ZF flag is still 0: fig.12.3.

Next DEC is executed. EAX is finally 0 and ZF flag is set, because the result is zero: fig.12.4. OllyDbg shows that this jump will be taken now.

A pointer to the string “two” will now be written into the stack: fig.12.5. Please note: current argument of the function is 2 and 2 is now in the stack at the address 0x0020FA44.

MOV wrote pointer to the string at the address 0x0020FA44 (see stack window). Jump is happen. This is the first instruction of `printf()` function in `MSVCR100.DLL` (I compiled the example with `/MD` switch): fig.12.6. Now the `printf()` will treat the string at 0x0020FA44 as its sole argument and will print the string.

This is the very last instruction of `printf()`: fig.12.7. “two” string was just printed to the console window.

Let’s press F7 or F8 (step over) and we will return...not to `f()` function, but rather to the `main()`: fig.12.8. Yes, the jump was direct, from the guts of `printf()` to `main()`. Because `RA` in the stack pointed not to some place in `f()` function, but rather to `main()`. And `CALL 0x01201000` was the actual instruction which called `f()` function.

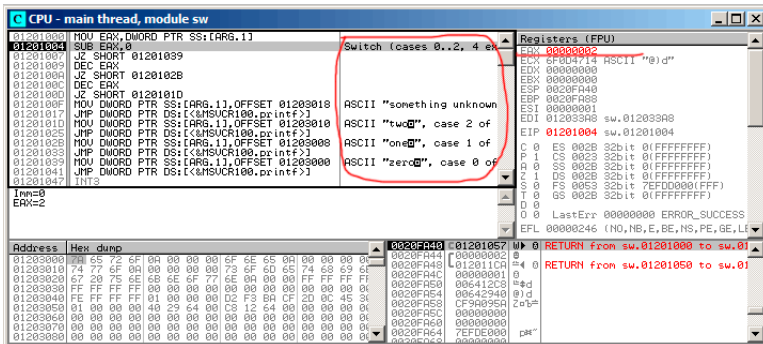


Figure 12.1: OllyDbg: EAX now contain first (and sole) function argument



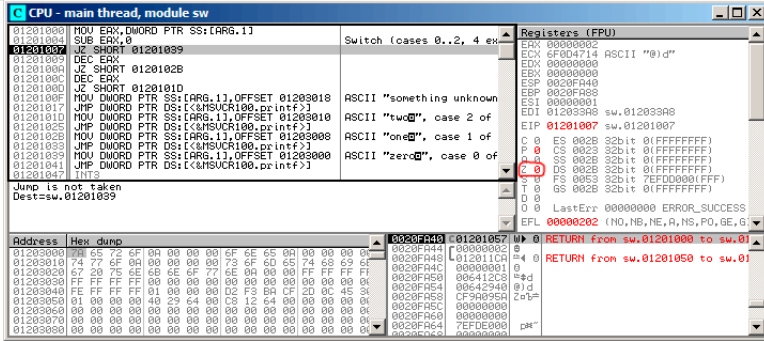


Figure 12.2: OllyDbg: SUB executed

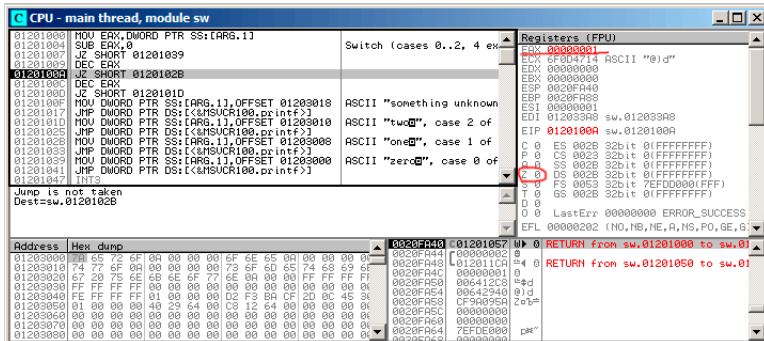


Figure 12.3: OllyDbg: first DEC executed

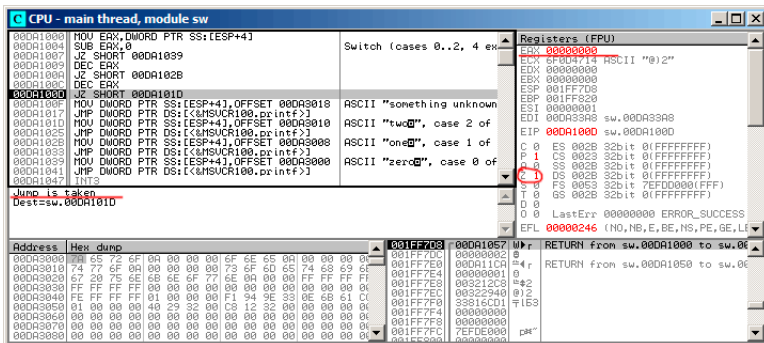


Figure 12.4: OllyDbg: second DEC executed

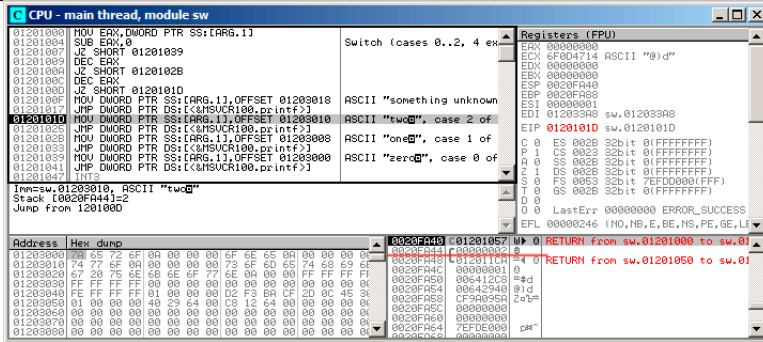


Figure 12.5: OllyDbg: pointer to the string is to be written at the place of first argument

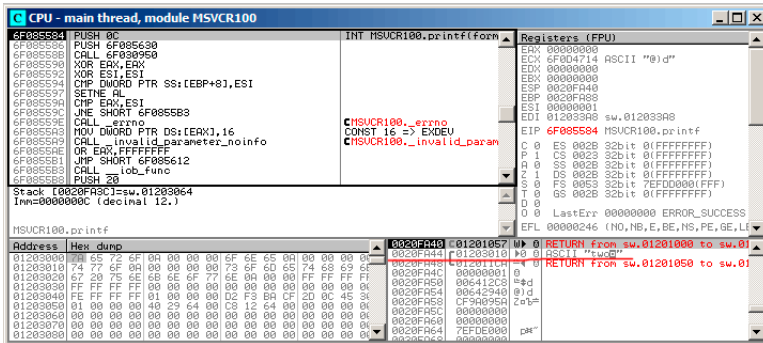


Figure 12.6: OllyDbg: first instruction of printf () in MSVCR100.DLL

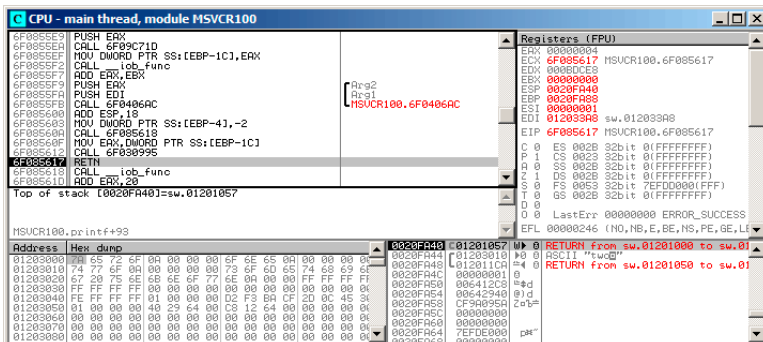


Figure 12.7: OllyDbg: last instruction of printf () in MSVCR100.DLL

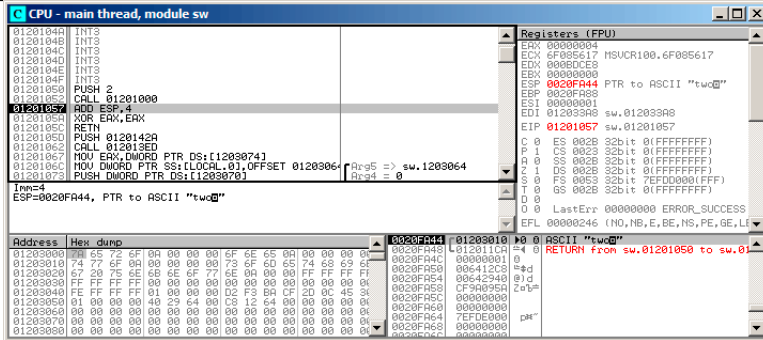


Figure 12.8: OllyDbg: return to main()

## 12.1.2 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, aSomethingUnkno ; "\
    ↪ something unknown\n"
.text:0000016C 4E 0F 8F 02      ADREQ  R0, aTwo ; "two\n"
.text:00000170
.text:00000170          loc_170: ; CODE XREF: f1+8
.text:00000170          ; f1+14
.text:00000170 78 18 00 EA      B      __2printf
```

Again, by investigating this code, we cannot say, was it switch() in the original source code, or pack of if() statements.

Anyway, we see here predicated instructions again (like ADREQ (*Equal*)) which will be triggered only in  $R0 = 0$  case, and the address of the «zero\n» string will be loaded into the R0. The next instruction BEQ will redirect control flow to loc\_170, if  $R0 = 0$ . By the way, astute reader may ask, will BEQ triggered right since ADREQ before it is already filled the R0 register with another value. Yes, it will since BEQ checking flags set by CMP instruction, and ADREQ not modifying flags at all.

Other instructions are already familiar to us. There is only one call to printf(), at the end, and we already examined this trick here (5.3.1). There are three paths to printf() at the end.

The last 'CMP R0, #2' instruction is needed here to figure out, if  $a = 2$  or not. If it is not true, then ADRNE will load pointer to the string «something

unknown \n» into R0 since  $a$  was already checked to be equal to 0 or 1 before, so we can be assured the  $a$  variable is not equal to these numbers at this point. And if  $R0 = 2$ , a pointer to string «two\n» will be loaded by ADREQ into R0.

### 12.1.3 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 ; "\n
    ↘ 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}
.text:000000F4          ; End of function f1
```

As I already mentioned, there is no feature of adding conditional predicates to majority of instructions in thumb mode, so the thumb-code here is somewhat similar to the easily understandable x86 CISC-style code.

### 12.1.4 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
    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

```

Input value has *int* type, hence W0 register is used as input value instead of the whole X0 register. String pointers are passed to puts() using ADRP/ADD instructions pair just like I showed in “Hello, world!” example: [2.4.5](#).

### 12.1.5 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 do jump if W0 is zero. There is also direct jump to puts ( ) instead of calling it, like I described before: [12.1.1](#).

### 12.1.6 Conclusion

*switch()* with few number of cases is indistinguishable from *if/else* construction, for example: listing.[12.1.1](#).

## 12.2 A lot of cases

If *switch()* statement contain a lot of case's, it is not very convenient for 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;
    }
}

```

```

};
};

int main()
{
    f (2); // test
};

```

## 12.2.1 x86

### Non-optimizing MSVC

We got (MSVC 2010):

Listing 12.3: 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
_f       ENDP

```

OK, what we see here is: there is a set of the `printf()` calls with various arguments. All they has not only addresses in process memory, but also internal symbolic labels assigned by compiler. All these labels are also mentioned in `$LN11@f` internal table.

At the function beginning, if  $a$  is greater than 4, control flow is passed to label `$LN1@f`, where `printf()` with argument 'something unknown' is called.

But if  $a$  value is less or equals to 4, let's multiply it by 4 and add `$LN11@f` table address. That is how address inside of 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 within 32-bit process that is why all elements contain 4 bytes). Address of the `$LN11@f` table + 8 – it will be table element where `$LN4@f` label is stored. `JMP` fetches `$LN4@f` address from the table and jump to it.

This table sometimes called *jumptable* or *branch table*<sup>3</sup>.

Then corresponding `printf()` is called with argument 'two'. Literally, `jmp DWORD PTR $LN11@f[ecx*4]` instruction means *jump to DWORD, which is stored at address `$LN11@f + ecx * 4`*.

`npad (72)` is assembly language macro, aligning next label so that it will be stored at address aligned on a 4 byte (or 16 byte) border. This is very suitable for processor since it is able to fetch 32-bit values from memory through memory bus, cache memory, etc, in much effective way if it is aligned.

<sup>3</sup>The whole method once called *computed GOTO* in early FORTRAN versions: [http://en.wikipedia.org/wiki/Branch\\_table](http://en.wikipedia.org/wiki/Branch_table). Not quite relevant these days, but what a term!



## OllyDbg

Let's try this example in OllyDbg. Function's input value (2) is loaded into EAX: [fig.12.9](#).

Input value is checked, if it's bigger than 4? No, "default" jump is not taken: [fig.12.10](#).

Here we see a jumtable: [fig.12.11](#). By the way, I clicked "Follow in Dump" → "Address constant", so now we see a jumtable in data window. These are 4 32-bit values<sup>4</sup>. ECX is 2 now, so the second element (counting from zeroth) of table will be used. By the way, it's also possible to click "Follow in Dump" → "Memory address" and OllyDbg will show the element JMP instruction being address now. That's 0x0116103A.

Jump occurred and we now at 0x0116103A: the code printing "two" string will now be executed: [fig.12.12](#).

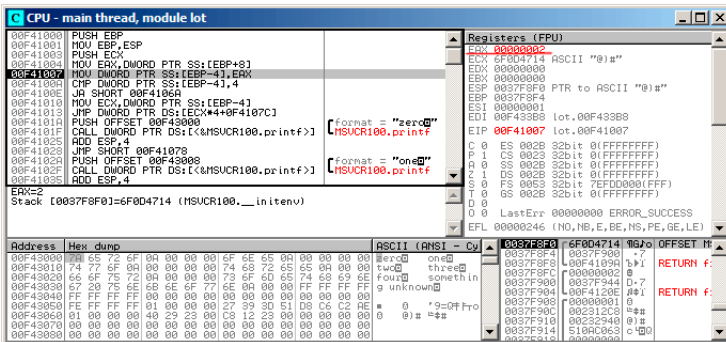


Figure 12.9: OllyDbg: function's input value is loaded in EAX

<sup>4</sup>They are underlined by OllyDbg because these are also FIXUps: [53.2.6](#), we will back to them later

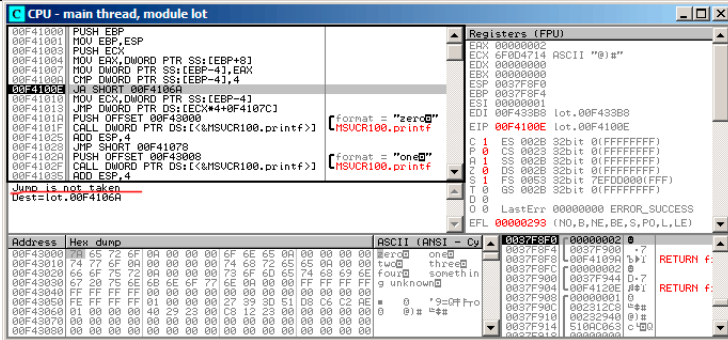


Figure 12.10: OllyDbg: 2 is no bigger than 4: no jump is taken

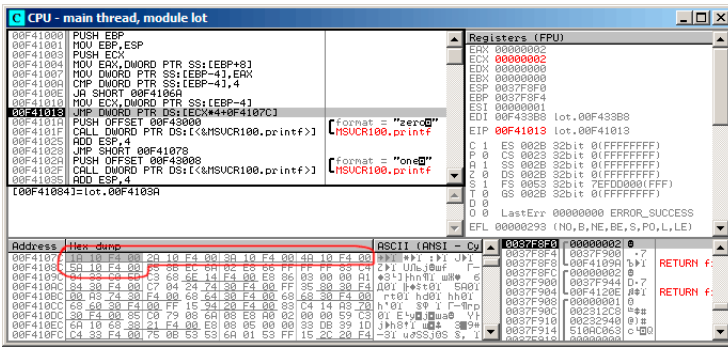


Figure 12.11: OllyDbg: calculating destination address using jumtable

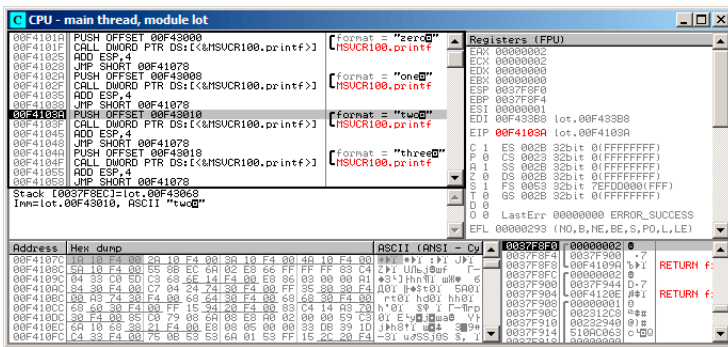


Figure 12.12: OllyDbg: now we at corresponding case: label

**Non-optimizing GCC**

Let's see what GCC 4.4.1 generates:

Listing 12.4: 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
    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, except 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) (15.2.1). Then label address is taken from `off_804855C` array, address is calculated and stored into `EAX`, then ```JMP EAX``` do actual jump.

## 12.2.2 ARM: 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        ; jumtable ↪
    ↪ 00000178 default case

00000180
00000180                loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA        B      zero_case          ; jumtable ↪
    ↪ 00000178 case 0

00000184
00000184                loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA        B      one_case           ; jumtable ↪
    ↪ 00000178 case 1

00000188
00000188                loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA        B      two_case          ; jumtable ↪
    ↪ 00000178 case 2

0000018C

```

```

0000018C          loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA      B      three_case      ; jumtable ↗
    ↳ 00000178 case 3

00000190
00000190          loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA      B      four_case      ; jumtable ↗
    ↳ 00000178 case 4

00000194
00000194          zero_case ; CODE XREF: f2+4
00000194                          ; f2:loc_180
00000194 EC 00 8F E2      ADR    R0, aZero      ; jumtable ↗
    ↳ 00000178 case 0
00000198 06 00 00 EA      B      loc_1B8

0000019C
0000019C          one_case ; CODE XREF: f2+4
0000019C                          ; f2:loc_184
0000019C EC 00 8F E2      ADR    R0, aOne      ; jumtable ↗
    ↳ 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      ; jumtable ↗
    ↳ 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    ; jumtable ↗
    ↳ 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     ; jumtable ↗
    ↳ 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
000001C0          ; End of function f2

```

This code makes use of the ARM mode feature in which all instructions has fixed size of 4 bytes.

Let's keep in mind the maximum value for  $a$  is 4 and any greater value must cause «something unknown» string printing.

The very first ``CMP R0, #5`` instruction compares  $a$  input value with 5.

The next ``ADDCC PC, PC, R0, LSL#2``<sup>5</sup> instruction will execute only if  $R0 < 5$  ( $CC=Carry\ clear / Less\ than$ ). Consequently, if ADDCC will not trigger (it is a  $R0 \geq 5$  case), a jump to `default_case` label will be occurred.

But if  $R0 < 5$  and ADDCC will trigger, following events will happen:

Value in the R0 is multiplied by 4. In fact, LSL#2 at the instruction's suffix means "shift left by 2 bits". But as we will see later (15.2.1) in "Shifts" section, shift left by 2 bits is just equivalently to multiplying by 4.

Then,  $R0 * 4$  value we got, is added to current value in the PC, thus jumping to one of B (Branch) instructions located below.

At the moment of ADDCC execution, value in the PC is 8 bytes ahead (0x180) than address at which ADDCC instruction is located (0x178), or, in other words, 2 instructions ahead.

This is how ARM processor pipeline works: when ADDCC instruction is executed, the processor at the moment is beginning to process instruction after the next one, so that is why PC pointing there. This fact should be memorized.

If  $a = 0$ , then nothing will be added to the value in the PC, and actual value in the PC is to be written into the PC (which is 8 bytes ahead) and jump to the label `loc_180` will happen, this is 8 bytes ahead of the point where ADDCC instruction is.

In case of  $a = 1$ , then  $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$  will be written to the PC, this is the address of the `loc_184` label.

With every 1 added to  $a$ , resulting PC increasing by 4. 4 is also instruction length in ARM mode and also, length of each B instruction length, there are 5 of them in row.

Each of these five B instructions passing control further, where something is going on, what was programmed in `switch0`. Pointer loading to corresponding string occurring there, etc.

### 12.2.3 ARM: Optimizing Keil 6/2013 (thumb mode)

<sup>5</sup>ADD—addition

```

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 ; jumtable ↗
    ↘ 000000FA case 0
00000108 06 E0                            B      loc_118

0000010A
0000010A                                one_case ; CODE XREF: f2+4
0000010A 8E A0                            ADR    R0, aOne ; jumtable ↗
    ↘ 000000FA case 1
0000010C 04 E0                            B      loc_118

0000010E
0000010E                                two_case ; CODE XREF: f2+4
0000010E 8F A0                            ADR    R0, aTwo ; jumtable ↗
    ↘ 000000FA case 2
00000110 02 E0                            B      loc_118

00000112
00000112                                three_case ; CODE XREF: f2+4
00000112 90 A0                            ADR    R0, aThree ; jumtable ↗
    ↘ 000000FA case 3
00000114 00 E0                            B      loc_118

00000116
00000116                                four_case ; CODE XREF: f2+4
00000116 91 A0                            ADR    R0, aFour ; jumtable ↗
    ↘ 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: __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 all instructions in thumb and thumb-2 modes will have same size. It is even can be said that in these modes instructions has variable length, just like in x86.

So there is a special table added, containing information about how much cases are there, not including default-case, and offset, for each, each encoding a label, to which control must be passed in corresponding case.

A special function here present in order to deal with the table and pass control, named

`__ARM_common_switch8_thumb`. It is beginning with ``BX PC'' instruction, which function is to switch processor to ARM-mode. Then you may see the function for table processing. It is too complex for describing it here now, so I will omit all elaborations.

But it is interesting to note the function uses `LR` register as a pointer to the table. Indeed, after this function calling, `LR` will contain address after ``BL `__ARM_common_switch8_thumb`'' instruction, and the table is beginning right there.

It is also worth noting the code is generated as a separate function in order to reuse it, so compiler will not generate the very same code for each `switch()` statement.



IDA successfully perceived it as a service function and table automatically, and added commentaries to labels like  
jumptable 000000FA case 0.

## 12.2.4 Conclusion

Rough skeleton of *switch()*:

Listing 12.5: 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
    JMP exit

default:
    ...

exit:
    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

Jump to the address in jump table may also be implemented using this instruction: `JMP jump_table[REG*4]`. Or `JMP jump_table[REG*8]` in x64.

A jump table is just array of pointers, like it was described here: [17.5](#).

## 12.3 When there are several *case* in one block

Here is also a very often used construction: several *case* statements may be used in 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 each block for each possible case, so what is usually done, is each block generated plus some kind of dispatcher.

### 12.3.1 MSVC

Listing 12.6: 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, ↵
17         ↵ 7, 10'
18         jmp     DWORD PTR __imp__printf
19 $LN4@f:
20         mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, ↵
21         ↵ 5'
22         jmp     DWORD PTR __imp__printf
23 $LN3@f:
24         mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, ↵
25         ↵ 21'
26         jmp     DWORD PTR __imp__printf
27 $LN2@f:
28         mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
29         jmp     DWORD PTR __imp__printf
30 $LN1@f:
31         mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'↵
32         ↵ '
33         jmp     DWORD PTR __imp__printf
34         npad   2 ; align $LN11@f table on 16-byte boundary
35 $LN11@f:
36         DD     $LN5@f ; print '1, 2, 7, 10'
37         DD     $LN4@f ; print '3, 4, 5'
38         DD     $LN3@f ; print '8, 9, 21'
39         DD     $LN2@f ; print '22'
40         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 index table, and the second table (\$LN11@f) is an array of pointers to blocks.

First, input value is used as index in index table (line 13).

Here is short legend for values in the table: 0 is first *case* block (for values 1, 2, 7, 10), 1 is second (for values 3, 4, 5), 2 is third (for values 8, 9, 21), 3 is fourth (for value 22), 4 is for default block.

We get there index for the second table of block pointers and we we jump there (line 14).

What is also worth to note that there are no case for input value 0. Hence, we see DEC instruction at line 10, and the table is beginning at  $a = 1$ . Because there are no need to allocate table element for  $a = 0$ .

This is very often used pattern.

So where economy is? Why it's not possible to make it as it was already discussed (12.2.1), just with one table, consisting of block pointers? The reason is because elements in index table has 8-bit byte type, hence it's all more compact.

## 12.3.2 GCC

GCC do the job like it was already discussed (12.2.1), using just one table of pointers.

## 12.4 Fallthrough

Another very popular usage of `switch()` is fallthrough. Here is a small example:

```

1 #define R 1
2 #define W 2
3 #define RW 3
4
5 void f(int type)
6 {
7     int r=0, w=0;
8
9     switch (type)
10    {
11    case RW:
12        r=1;
13    case W:
14        w=1;
15        break;
16    case R:
17        r=1;
18        break;
19    default:
20        break;
21    };
22    printf ("r=%d, w=%d\n", r, w);
23 };

```

If `type = 1 (R)`, `r` is to be set to 1, if `type = 2 (W)`, `w` is to be set to 2. In case of `type = 3 (RW)`, both `r` and `w` are to be set to 1.

The piece of code at line 14 is executed in two cases: if `type = RW` or if `type = W`. There are no “break” for “case RW”, that’s OK.

### 12.4.1 MSVC x86

Listing 12.7: MSVC 2012

```

$SG1305 DB      'r=%d, w=%d', 0aH, 00H

_w$ = -12      ; size = 4
_r$ = -8       ; size = 4
tv64 = -4      ; size = 4
_type$ = 8     ; size = 4
_f          PROC
    push     ebp
    mov     ebp, esp
    sub     esp, 12

```

```

mov     DWORD PTR _r$[ebp], 0
mov     DWORD PTR _w$[ebp], 0
mov     eax, DWORD PTR _type$[ebp]
mov     DWORD PTR tv64[ebp], eax
cmp     DWORD PTR tv64[ebp], 1 ; R
je     SHORT $LN2@f
cmp     DWORD PTR tv64[ebp], 2 ; W
je     SHORT $LN3@f
cmp     DWORD PTR tv64[ebp], 3 ; RW
je     SHORT $LN4@f
jmp     SHORT $LN5@f
$LN4@f: ; case RW:
mov     DWORD PTR _r$[ebp], 1
$LN3@f: ; case W:
mov     DWORD PTR _w$[ebp], 1
jmp     SHORT $LN5@f
$LN2@f: ; case R:
mov     DWORD PTR _r$[ebp], 1
$LN5@f: ; default
mov     ecx, DWORD PTR _w$[ebp]
push   ecx
mov     edx, DWORD PTR _r$[ebp]
push   edx
push   OFFSET $SG1305 ; 'r=%d, w=%d'
call   _printf
add     esp, 12
mov     esp, ebp
pop     ebp
ret     0
_f     ENDP

```

The code is mostly resembles what is in source. There are no jumps between labels `$LN4@f` and `$LN3@f`: so when code flow are at `$LN4@f`, `r` value are first set to 1, then `w`. This is probably why it's called fallthrough: code flow is falling through one piece of code (setting `r`) to another (setting `w`). If `type = W`, we are landing at `$LN3@f`, so no code setting `r` to 1 is executing.

## 12.4.2 ARM64

Listing 12.8: GCC (Linaro) 4.9

```

.LC0:
.string "r=%d, w=%d\n"
f:
stp     x29, x30, [sp, -48]!
add     x29, sp, 0
str     w0, [x29, 28]

```

```

    str    wzr, [x29,44] ; set r and w local variables to ↵
↳ zero
    str    wzr, [x29,40]
    ldr    w0, [x29,28] ; load 'type' argument
    cmp    w0, 2        ; type=W?
    beq    .L3
    cmp    w0, 3        ; type=RW?
    beq    .L4
    cmp    w0, 1        ; type=R?
    beq    .L5
    b      .L6          ; otherwise...
.L4: ; case RW
    mov    w0, 1
    str    w0, [x29,44] ; r=1
.L3: ; case W
    mov    w0, 1
    str    w0, [x29,40] ; w=1
    b      .L6
.L5: ; case R
    mov    w0, 1
    str    w0, [x29,44] ; r=1
    nop
.L6: ; default
    adrp   x0, .LC0 ; "r=%d, w=%d\n"
    add    x0, x0, :lo12:LC0
    ldr    w1, [x29,44] ; load r
    ldr    w2, [x29,40] ; load w
    bl     printf
    ldp    x29, x30, [sp], 48
    ret

```

Merely the same thing. There are no jumps between labels .L4 and .L3.

## 12.5 Exercises

### 12.5.1 Exercise #1

It's possible to rework C example in [12.2](#) in such way, so the compiler will produce even smaller code, but it will work just the same. Try to achieve it.

Hint: [F.1.3](#).

# Chapter 13

## Loops

### 13.0.2 Simple example

#### x86

There is a special LOOP instruction in x86 instruction set, it is checking value in the ECX register and if it is not 0, do ECX [decrement](#) and pass control flow to the label mentioned in the LOOP operand. Probably, this instruction is not very convenient, so, I did not ever see any modern compiler emit it automatically. So, if you see the instruction somewhere in code, it is most likely this is manually written piece of assembly code.

In C/C++ loops are usually constructed using `for()`, `while()`, `do/while()` statements.

Let's start with `for()`.

This statement defines loop initialization (set loop counter to initial value), loop condition (is counter is bigger than a limit?), what is done at each iteration ([increment/decrement](#)) and of course loop body.

```
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

So, generated code will be consisted of four parts as well.

Let's start with simple example:

```
#include <stdio.h>

void f(int i)
{
    printf ("f(%d)\n", i);
}
```



```

};

int main()
{
    int i;

    for (i=2; i<10; i++)
        f(i);

    return 0;
};

```

Result (MSVC 2010):

Listing 13.1: 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 $LN1@main         ; if i is biggest or equals ↵
    ↵ to 10, let's finish loop
    mov     ecx, DWORD PTR _i$[ebp] ; loop body: call f(i)
    push    ecx
    call    _f
    add     esp, 4
    jmp     SHORT $LN2@main         ; jump to loop begin
$LN1@main:
    ; loop end
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP

```

Nothing very special, as we see.

GCC 4.4.1 emits almost the same code, with one subtle difference:

Listing 13.2: 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    f
                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 will get if optimization is turned on (/Ox):

Listing 13.3: Optimizing MSVC

```

_main              PROC
                push    esi
                mov     esi, 2
$LL3@main:
                push    esi
                call    _f
                inc     esi
                add     esp, 4
                cmp     esi, 10 ; 0000000aH
                jl      SHORT $LL3@main
                xor     eax, eax
                pop     esi
                ret     0
_main              ENDP

```

What is going on here is: space for the *i* variable is not allocated in local stack

anymore, but individual register is allocated for it: the ESI. This is possible in such small functions where not so many local variables are present.

One very important property is the `f()` function must not change the value in the ESI. Our compiler is sure here. And if compiler decided to use the ESI register in `f()` too, its value would be saved then at the `f()` function's prologue and restored at the `f()` function's epilogue. Almost like in our listing: please note `PUSH ESI/POP ESI` at the function begin and end.

Let's try GCC 4.4.1 with maximal optimization turned on (`-O3` option):

Listing 13.4: 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    f
              mov     [esp+10h+var_10], 3
              call    f
              mov     [esp+10h+var_10], 4
              call    f
              mov     [esp+10h+var_10], 5
              call    f
              mov     [esp+10h+var_10], 6
              call    f
              mov     [esp+10h+var_10], 7
              call    f
              mov     [esp+10h+var_10], 8
              call    f
              mov     [esp+10h+var_10], 9
              call    f
              xor     eax, eax
              leave
              retn
main          endp
```

Huh, GCC just unwind our loop.

[Loop unwinding](#) has advantage in these cases when there is not so much iterations and we could cut some execution speed by removing all loop supporting instructions. On the other side, resulting code is obviously larger.

OK, let's increase maximal value of the `i` variable to 100 and try again. GCC will do:

## Listing 13.5: GCC

```

main                public 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
                   nop    ; aligning label loc_80484D0 (loop body ↙
                   ↘ begin) by 16-byte border

loc_80484D0:
                   mov     [esp+20h+var_20], ebx ; pass i as first ↙
                   ↘ argument to f()
                   add     ebx, 1    ; i++
                   call    f
                   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
                   retn
main                endp

```

It is quite similar to what MSVC 2010 with optimization (`/Ox`) produce. With the exception the `EBX` register will be fixed to 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.

### x86: OllyDbg

Let's compile our example in MSVC 2010 with `/Ox` and `/Ob0` options and load it into OllyDbg.

It seems, OllyDbg is able to detect simple loops and show them in square brackets, for convenience: [fig.13.1](#).

By tracing (F8 (step over)) we see how `ESI` [incrementing](#). Here, for instance,  $ESI = i = 6$ : [fig.13.2](#).

9 is a last loop value. That's why `JL` will not trigger after [increment](#), and function is finishing: [fig.13.3](#).

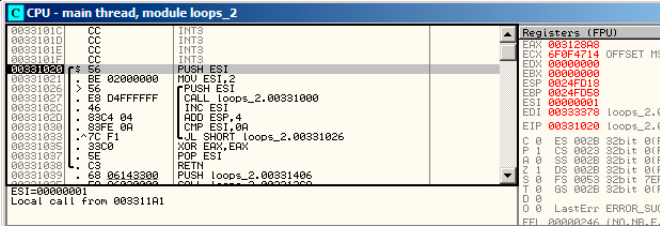


Figure 13.1: OllyDbg: main() begin

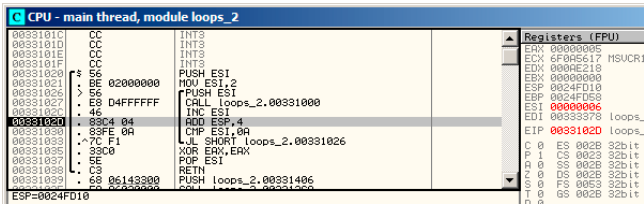


Figure 13.2: OllyDbg: loop body just executed with i = 6

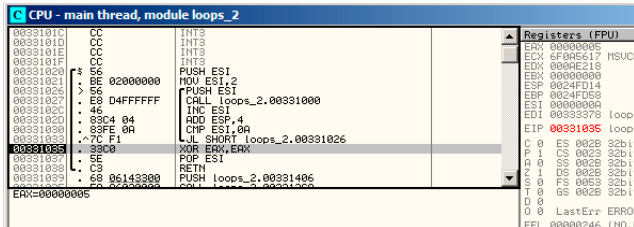


Figure 13.3: OllyDbg: ESI = 10, loop end

### x86: tracer

As we might see, it is not very convenient to trace in debugger manually. That's one of the reasons I wrote [tracer](#) for myself.

I open compiled example in [IDA](#), I find the address of the instruction PUSH ESI (passing sole argument into f()) and this is 0x401026 for me and I run [tracer](#):

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX just sets breakpoint at address and then will print registers state.

In the `tracer.log` I see after running:

```
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
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 value of ESI register is changed from 2 to 9.

Even more than that, [tracer](#) can collect register values on all addresses within function. This is called *trace* there. Each instruction is being traced, all interesting register values are noticed and collected. [IDA](#).idc-script is generated then. So, in the [IDA](#) I've learned that `main()` function address is `0x00401020` and I run:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF mean set breakpoint on function.

As a result, I have got loops\_2.exe.idc and loops\_2.exe\_clear.idc scripts. I'm loading loops\_2.exe.idc into IDA and I see: fig.13.4

We see that ESI can be from 2 to 9 at the begin of loop body, but from 3 to 0xA (10) after increment. We can also see that main() is finishing with 0 in EAX.

tracer also generates loops\_2.exe.txt, containing information about how many times each instruction was executed and register values:

Listing 13.6: 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=0
  ↳ x8d1000
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 [RET] EAX=0
```

grep can be used here.

```
.text:00401020
.text:00401020 ; ===== SUBROUTINE =====
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main      proc near          ; CODE XREF: __tmainCRTStartup+11D↓j
.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
.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:00401027      inc     esi          ; ESI=2..9
.text:0040102C      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 13.4: IDA with .idc-script loaded

**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     f
    ADD     R4, R4, #1

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.

``MOV R4, #2'' instruction just initializing  $i$ .

``MOV R0, R4'' and ``BL f'' instructions are compose loop body, the first instruction preparing argument for  $f()$  function and the second is calling it.

``ADD R4, R4, #1'' instruction is just adding 1 to the  $i$  variable during each iteration.

``CMP R4, #0xA'' comparing  $i$  with 0xA (10). Next instruction BLT (*Branch Less Than*) will jump if  $i$  is less than 10.

Otherwise, 0 will be written into R0 (since our function returns 0) and function execution ended.

**ARM: Optimizing Keil 6/2013 (thumb mode)**

```

_main
    PUSH    {R4,LR}
    MOVS   R4, #2

loc_132    ; CODE XREF: _main+E
    MOVS   R0, R4
    BL     example7_f
    ADDS   R4, R4, #1
    CMP    R4, #0xA
    BLT    loc_132
    MOVS   R0, #0
    POP    {R4,PC}

```

Practically, the same.



## ARM: 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
    POP         {R4,R7,PC}

```

In fact, this was in my `f()` function:

```

void f(int i)
{
    // do something here
    printf ("%d\n", i);
};

```

So, LLVM not just *unrolled* the loop, but also represented my very simple function `f()` as *inlined*, and inserted its body 8 times instead of loop. This is possible when function is so primitive (like mine) and when it is called not many times (like here).

## One more thing

On the code generated we can see: after  $i$  initialization, loop body will not be executed, but  $i$  condition checked first, and only after loop body is to be executed. And that is correct. Because, if loop condition is not met at the beginning, loop body must not be executed. For example, this is possible in the following case:

```
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

If *total\_entries\_to\_process* equals to 0, loop body must not be executed whatsoever. So that is why condition checked before loop body execution.

However, optimizing compiler may swap condition check and loop body, if it sure that the situation described here is not possible (like in case of our very simple example and Keil, Xcode (LLVM), MSVC in optimization mode).

### 13.0.3 Several iterators

Common loop has only one iterator, but there could be several in resulting code.

Here is 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 this is costly operation. Can we optimize it somehow? Yes, if we will notice that both array indices are leaping on places we can calculate easily without multiplication.

### Three iterators

Listing 13.7: Optimizing MSVC 2013 x64

```
f      PROC
; RDX=a1
; RCX=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: *cnt* variable and two indices, they are increased by 12 and 28 at each iteration, pointing to new array elements. 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 on each iteration instead of one, two multiplication operations are dropped.

## Two iterators

GCC 4.9 did even more, leaving only 2 iterators:

Listing 13.8: 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=RDY*4=cnt*4
    sal    rdx, 5
; RDX=RDY<<5=cnt*32
    sub    rdx, rax
; RDX=RDY-RAX=cnt*32-cnt*4=cnt*28
    add    rdx, rsi
; RDX=RDY+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
    jne    .L3
.L1:
    rep   ret

```

There are no *counter* variable: GCC concluded it not needed. Last element of *a2* array is calculated before loop begin (which is easy:  $cnt * 7$ ) and that's how loop will be stopped: just iterate until second index not reached this precalculated value.

About multiplication using shifts/additions/subtractions read here: [15.1.3](#).

This code can be rewritten into C/C++ like that:

```

#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    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 do the same, but precalculates last index of *a1* array instead of *a2*, and this has the same effect, of course:

Listing 13.9: 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

```

## Intel C++ 2011 case

Compiler optimizations can also be weird, but nevertheless, still correct. Here is what Intel C++ compiler 2011 do:

Listing 13.10: Optimizing Intel C++ 2011 x64

```

f      PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8 = cnt
.B1.1::          ; Preds .B1.0
    test    r8, r8                                ↙
    ↪ ;8.14
    jbe    exit          ; Prob 50%                ↙
    ↪ ;8.14
                                           ; LOE rdx rcx rbx rbp rsi rdi ↙
    ↪ r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 ↙
    ↪ xmm13 xmm14 xmm15
.B1.2::          ; Preds .B1.1
    cmp    r8, 6                                    ↙
    ↪ ;8.2
    jbe    just_copy          ; Prob 50%           ↙
    ↪ ;8.2
                                           ; LOE rdx rcx rbx rbp rsi rdi ↙
    ↪ r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 ↙

```

```

    ↪ xmm13 xmm14 xmm15
.B1.3::                                ; Preds .B1.2
    cmp      rcx, rdx                    ↪
    ↪ ;9.11
    jbe     .B1.5                        ↪
    ↪ ;9.11
                                           ; LOE rdx rcx rbx rbp rsi rdi ↪
    ↪ r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 ↪
    ↪ xmm13 xmm14 xmm15
.B1.4::                                ; Preds .B1.3
    mov     r10, r8                      ↪
    ↪ ;9.11
    mov     r9, rcx                      ↪
    ↪ ;9.11
    shl    r10, 5                        ↪
    ↪ ;9.11
    lea    rax, QWORD PTR [r8*4]        ↪
    ↪ ;9.11
    sub    r9, rdx                       ↪
    ↪ ;9.11
    sub    r10, rax                      ↪
    ↪ ;9.11
    cmp    r9, r10                      ↪
    ↪ ;9.11
    jge    just_copy2                   ; Prob 50% ↪
    ↪ ;9.11
                                           ; LOE rdx rcx rbx rbp rsi rdi ↪
    ↪ r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 ↪
    ↪ xmm13 xmm14 xmm15
.B1.5::                                ; Preds .B1.3 .B1.4
    cmp    rdx, rcx                      ↪
    ↪ ;9.11
    jbe    just_copy                    ; Prob 50% ↪
    ↪ ;9.11
                                           ; LOE rdx rcx rbx rbp rsi rdi ↪
    ↪ r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 ↪
    ↪ xmm13 xmm14 xmm15
.B1.6::                                ; Preds .B1.5
    mov    r9, rdx                       ↪
    ↪ ;9.11
    lea    rax, QWORD PTR [r8*8]        ↪
    ↪ ;9.11
    sub    r9, rcx                       ↪
    ↪ ;9.11
    lea    r10, QWORD PTR [rax+r8*4]    ↪
    ↪ ;9.11
    cmp    r9, r10                      ↪

```

```

    ↪ ;9.11
    jnl just_copy ; Prob 50% ↪
    ↪ ;9.11
    ; LOE rdx rcx rbx rbp rsi rdi ↪
    ↪ r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 xmm12 ↪
    ↪ xmm13 xmm14 xmm15
just_copy2:: ; Preds .B1.4 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
    xor r10d, r10d ↪
    ↪ ;8.2
    xor r9d, r9d ↪
    ↪ ;
    xor eax, eax ↪
    ↪ ;
    ; LOE rax rdx rcx rbx rbp rsi ↪
    ↪ rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 ↪
    ↪ xmm11 xmm12 xmm13 xmm14 xmm15
.B1.8:: ; Preds .B1.8 just_copy2
    mov r11d, DWORD PTR [rax+rdx] ↪
    ↪ ;3.6
    inc r10 ↪
    ↪ ;8.2
    mov DWORD PTR [r9+rcx], r11d ↪
    ↪ ;3.6
    add r9, 12 ↪
    ↪ ;8.2
    add rax, 28 ↪
    ↪ ;8.2
    cmp r10, r8 ↪
    ↪ ;8.2
    jbe .B1.8 ; Prob 82% ↪
    ↪ ;8.2
    jmp exit ; Prob 100% ↪
    ↪ ;8.2
    ; LOE rax rdx rcx rbx rbp rsi ↪
    ↪ rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 ↪
    ↪ xmm11 xmm12 xmm13 xmm14 xmm15
just_copy:: ; Preds .B1.2 .B1.5 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
    xor r10d, r10d ↪
    ↪ ;8.2
    xor r9d, r9d ↪
    ↪ ;

```

```

    xor     eax, eax                                ↗
    ↪ ;
                                ; LOE rax rdx rcx rbx rbp rsi ↗
    ↪ rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 ↗
    ↪ xmm11 xmm12 xmm13 xmm14 xmm15
.B1.11::
                                ; Preds .B1.11 just_copy
    mov     r11d, DWORD PTR [rax+rdx]              ↗
    ↪ ;3.6
    inc     r10                                    ↗
    ↪ ;8.2
    mov     DWORD PTR [r9+rcx], r11d              ↗
    ↪ ;3.6
    add     r9, 12                                 ↗
    ↪ ;8.2
    add     rax, 28                                ↗
    ↪ ;8.2
    cmp     r10, r8                                ↗
    ↪ ;8.2
    jb     .B1.11                                  ↗
    ↪ ;8.2
                                ; LOE rax rdx rcx rbx rbp rsi ↗
    ↪ rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10 ↗
    ↪ xmm11 xmm12 xmm13 xmm14 xmm15
exit::
                                ; Preds .B1.11 .B1.8 .B1.1
    ret
    ↪ ;10.1

```

First, there are some decisions taken, than one of the routines is executed. Supposedly, it is checked, if arrays are intersected. This is very well known way of optimizing memory blocks copy routines. But copy routines are the same! This is probably error of Intel C++ optimizer, which still produce workable code, though.

I'm adding such example code to my book so reader would understand that compilers output is weird at times, but still correct, because when compiler was tested, tests were passed.

## 13.1 Conclusion

Rough skeleton of loop from 2 to 9 inclusive:

Listing 13.11: 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

```

Increment operation may be represented as 3 instructions in non-optimized code:

Listing 13.12: 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 loop body is short, a whole register can be dedicated to the counter variable:

Listing 13.13: 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 loop may be generated by compiler in different order:

Listing 13.14: x86

```

    MOV [counter], 2 ; initialization
    JMP label_check
label_increment:
    ADD [counter], 1 ; increment
label_check:
    CMP [counter], 10
    JGE exit
    ; loop body

```

```

; do something here
; use counter variable in local stack
JMP label_increment
exit:

```

Usually condition is checked *before* loop body, but compiler may rearrange loop in that way so condition will be checked *after* loop body. This is when compiler is sure that condition is always *true* on first iteration, so loop body will be executed at least once:

Listing 13.15: 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 LOOP instruction. This is rare, compilers are not using it. So when you see it, it's a right sign this piece of code is hand-written:

Listing 13.16: x86

```

; count from 10 to 1
MOV ECX, 10
body:
; loop body
; use ECX, bit do not modify it!
LOOP body

```

ARM. A R4 register is dedicated to counter variable:

Listing 13.17: 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

```

## 13.2 Exercises

### 13.2.1 Exercise #1

Why LOOP instruction is not used by modern compilers anymore?

### 13.2.2 Exercise #2

Take a loop example from this section ([13.0.2](#)), compile it in your favorite OS and compiler and modify (patch) executable file, so the loop range will be [6..20].

### 13.2.3 Exercise #3

What this code does?

Listing 13.18: Optimizing MSVC 2010

```

$SG2795 DB      '%d', 0aH, 00H

_main PROC
    push     esi
    push     edi
    mov     edi, DWORD PTR __imp__printf
    mov     esi, 100
    npad    3 ; align next label
$LL3@main:
    push     esi
    push     OFFSET $SG2795 ; '%d'
    call    edi
    dec     esi
    add     esp, 8
    test    esi, esi
    jg     SHORT $LL3@main
    pop     edi
    xor     eax, eax
    pop     esi
    ret     0
_main ENDP

```

Listing 13.19: Non-optimizing Keil 6/2013 (ARM mode)

```

main PROC
    PUSH     {r4,lr}
    MOV      r4,#0x64
|L0.8|
    MOV      r1,r4
    ADR      r0,|L0.40|

```

```

    BL      __2printf
    SUB     r4,r4,#1
    CMP     r4,#0
    MOVLE   r0,#0
    BGT     |L0.8|
    POP     {r4,pc}
    ENDP

|L0.40|
    DCB     "%d\n",0

```

Listing 13.20: Non-optimizing Keil 6/2013 (thumb mode)

```

main PROC
    PUSH   {r4,lr}
    MOVS   r4,#0x64
|L0.4|
    MOVS   r1,r4
    ADR    r0,|L0.24|
    BL     __2printf
    SUBS   r4,r4,#1
    CMP    r4,#0
    BGT    |L0.4|
    MOVS   r0,#0
    POP    {r4,pc}
    ENDP

    DCW    0x0000
|L0.24|
    DCB    "%d\n",0

```

Answer: [F.1.5](#).

## 13.2.4 Exercise #4

What this code does?

Listing 13.21: Optimizing MSVC 2010

```

$SG2795 DB      '%d', 0aH, 00H

_main PROC
    push     esi
    push     edi
    mov     edi, DWORD PTR __imp__printf
    mov     esi, 1
    npad    3 ; align next label
$LL3@main:

```

```

    push    esi
    push    OFFSET $SG2795 ; '%d'
    call   edi
    add     esi, 3
    add     esp, 8
    cmp     esi, 100
    jl     SHORT $LL3@main
    pop     edi
    xor     eax, eax
    pop     esi
    ret     0
_main    ENDP

```

Listing 13.22: Non-optimizing Keil 6/2013 (ARM mode)

```

main PROC
    PUSH    {r4,lr}
    MOV     r4,#1
|L0.8|
    MOV     r1,r4
    ADR     r0,|L0.40|
    BL     __2printf
    ADD     r4,r4,#3
    CMP     r4,#0x64
    MOVGE  r0,#0
    BLT     |L0.8|
    POP     {r4,pc}
    ENDP

|L0.40|
    DCB     "%d\n",0

```

Listing 13.23: Non-optimizing Keil 6/2013 (thumb mode)

```

main PROC
    PUSH    {r4,lr}
    MOVS   r4,#1
|L0.4|
    MOVS   r1,r4
    ADR     r0,|L0.24|
    BL     __2printf
    ADDS   r4,r4,#3
    CMP     r4,#0x64
    BLT     |L0.4|
    MOVS   r0,#0
    POP     {r4,pc}
    ENDP

```

L0.24	DCW	0x0000
	DCB	"%d\n", 0

Answer: [F.1.6.](#)

# Chapter 14

## Simple C-strings processings

### 14.1 strlen()

Now let's talk about loops one more time. Often, `strlen()` function<sup>1</sup> is implemented using `while()` statement. Here is how it is done in 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!");
};
```

#### 14.1.1 x86

##### Non-optimizing MSVC

Let's compile:

```
_eos$ = -4 ; size = 4
```

---

<sup>1</sup>counting characters in string in C language

```

_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

```

Two new instructions here: **MOVSX** and **TEST**.

About first: **MOVSX** is intended to take byte from a point in memory and store value in a 32-bit register. **MOVSX** meaning *MOV with Sign-Extent*. Rest bits starting at 8th till 31th **MOVSX** will set to 1 if source byte in memory has *minus* sign or to 0 if *plus*.

And here is why all this.

By default, *char* type as signed in MSVC and GCC. If we have two values, one is *char* and another is *int*, (*int* is signed too), and if first value contain  $-2$  (it is coded as  $0xFE$ ) and we just copying this byte into *int* container, there will be  $0x000000FE$ , 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 need to transfer  $0xFE$  value from variable of *char* type to *int*, we need to identify its sign and extend it. That is what **MOVSX** does.



Also read about it in “*Signed number representations*” section (35).

I’m not sure if the compiler needs to store *char* variable in the EDX, it could take 8-bit register part (let’s say DL). Apparently, compiler’s [register allocator](#) works like that.

Then we see TEST EDX, EDX. About TEST instruction, read more in section about bit fields (18). But here, this instruction just checking value in the EDX, if it is equals to 0.

## Non-optimizing GCC

Let’s try GCC 4.4.1:

```

strlen          public 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 almost the same as MSVC did, but here we see MOVZX instead of MOVSX. MOVZX means *MOV with Zero-Extent*. This instruction copies 8-bit or 16-bit value into 32-bit register and sets the rest bits to 0. In fact, this instruction is convenient only since it enable us to replace two instructions at once: `xor eax, eax / mov al, [...]`.

On the other hand, it is obvious to us the compiler could produce the code: `mov al, byte ptr [eax] / test al, al` –it is almost the same, however, the highest EAX register bits will contain random noise. But let’s think it is compiler’s drawback –it cannot produce more understandable code. Strictly speaking, compiler is not obliged to emit understandable (to humans) code at all.

Next new instruction for us is `SETNZ`. Here, if AL contain not zero, `test al, al` will set 0 to the ZF flag, but `SETNZ`, if `ZF==0` (*NZ* means *not zero*) will set 1 to the AL. Speaking in natural language, *if AL is not zero, let’s jump to loc\_80483F0*. Compiler emitted slightly redundant code, but let’s not forget the optimization is turned off.

## Optimizing MSVC

Now let’s compile all this in MSVC 2012, with optimization turned on (`/Ox`):

Listing 14.1: 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. But it is needless to say the compiler could use registers such efficiently only in small functions with small number of local variables.

`INC/DEC` – are [increment/decrement](#) instruction, in other words: add 1 to variable or subtract.

## Optimizing MSVC + OllyDbg

We may try this (optimized) example in OllyDbg. Here is a very first iteration: [fig.14.1](#). We see that OllyDbg found a loop and, for convenience, *wrapped* its instructions in bracket. By clicking right button on EAX, we can choose “Follow in Dump” and the memory window position will scroll to the right place. We can see here a string “hello!” in memory. There are at least one zero byte after it and then random garbage. If OllyDbg sees a register with a valid address in it, pointing to

some string, it will show a string.

Let's press F8 (step over) enough number of times, so the current address will be at the loop body begin again: fig.14.2. We see that EAX contain address of the second character in the string.

We will press F8 enough number of times in order to escape from the loop: fig.14.3. We will see that EAX now contain address of zeroth byte, placed right after the string. Meanwhile, EDX wasn't changed, so it still pointing to the string begin. Difference between these two addresses will be calculated now.

SUB instruction was just executed: fig.14.4. Difference of pointers in the EAX register now  $-7$ . Indeed, the "hello!" string length is 6, but with zeroth byte included  $-7$ . But the `strlen()` must return number of non-zero characters in the string. So the decrement will processed now and then return from the function.

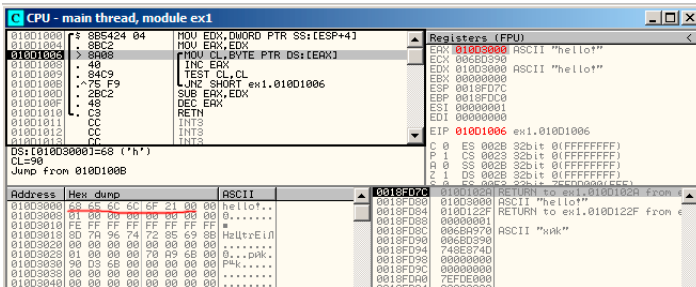


Figure 14.1: OllyDbg: first iteration begin

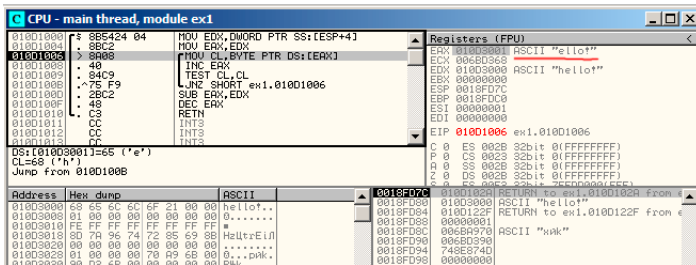


Figure 14.2: OllyDbg: second iteration begin

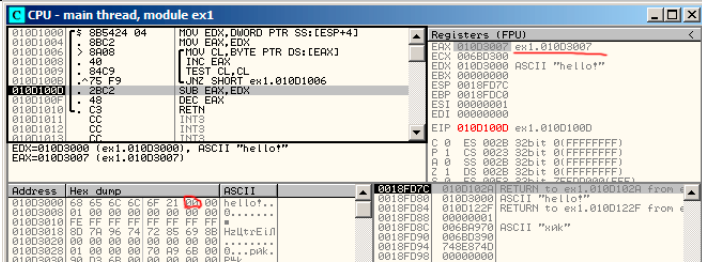


Figure 14.3: OllyDbg: pointers difference to be calculated now

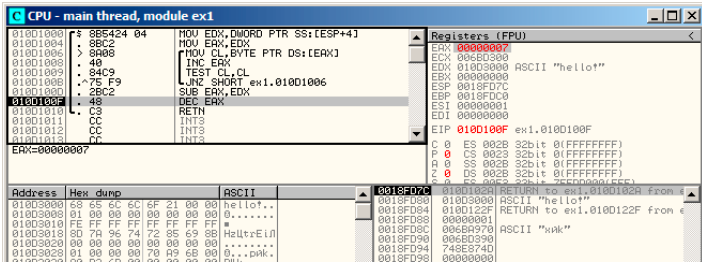


Figure 14.4: OllyDbg: EAX to be decremented now

### Optimizing GCC

Let's check GCC 4.4.1 with optimization turned on (-O3 key):

```

strlen
public 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 of MOVZX presence.

However, MOVZX could be replaced here to `mov dl, byte ptr [eax]`.

Probably, it is simpler for GCC compiler's code generator to *remember* the whole 32-bit EDX register is allocated for *char* variable and it can be sure the highest bits will not contain any noise at any point.

After, we also see new instruction NOT. This instruction inverts all bits in operand. It can be said, it is synonym to the `XOR ECX, 0fffffffh` instruction. NOT and following ADD calculating pointer difference and subtracting 1. At the beginning ECX, where pointer to *str* is stored, inverted and 1 is subtracted from it.

See also: "Signed number representations" (35).

In other words, at the end of 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 GCC decided it would be better? I cannot be sure. But I'm sure the both variants are effectively equivalent in efficiency sense.

## 14.1.2 ARM

### 32-bit ARM

#### Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 14.2: 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 ; deallocate 8 bytes for local variables
    BX    LR

```

Non-optimizing LLVM generates too much code, however, here we can see how 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](#), I renamed *var\_8* and *var\_4* into *eos* and *str* manually.

So, first instructions are just saves input value into both *str* and *eos*.

Loop body is beginning at *loc\_2CB8* label.

First three instructions in loop body (LDR, ADD, STR) loads *eos* value into R0, then value is [incremented](#) and it is saved back into *eos* local variable located in the stack.

The next `LDRSB R0, [R0]` (*Load Register Signed Byte*) instruction loading byte from memory at R0 address and sign-extends it to 32-bit<sup>2</sup>. This is similar to MOVSX instruction in x86. The compiler treating this byte as signed since *char* type in C standard is signed. I already wrote about it ([14.1.1](#)) in this section, but in relation to x86.

It should be noted, it is impossible in ARM to use 8-bit part or 16-bit part of 32-bit register separately of the whole register, as it is in x86. Apparently, it is because x86 has a huge history of compatibility with its ancestors up to 16-bit 8086 and even 8-bit 8080, but ARM was developed from scratch as 32-bit RISC-processor. Consequently, in order to process separate bytes in ARM, one has to use 32-bit registers anyway.

So, LDRSB loads symbol from string into R0, one by one. The next CMP and BEQ instructions check, if loaded symbol is 0. If not 0, control passed to loop body begin. And if 0, loop is finishing.

<sup>2</sup>Keil compiler treat *char* type as signed, just like MSVC and GCC.

At the end of function, a difference between *eos* and *str* is calculated, 1 is also subtracting, and resulting value is returned via R0.

N.B. Registers was not saved in this function. That's because by ARM calling convention, R0-R3 registers are “scratch registers”, they are intended for arguments passing, its values may not be restored upon function exit since calling function will not use them anymore. Consequently, they may be used for anything we want. Other registers are not used here, so that is why we have nothing to save on the stack. Thus, control may be returned back to calling function by simple jump (BX), to address in the LR register.

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

Listing 14.3: Optimizing Xcode 4.6.3 (LLVM) (thumb mode)

```

_strlen
    MOV        R1, R0

loc_2DF6
    ; CODE XREF: _strlen+8
    LDRB.W    R2, [R1],#1
    CMP       R2, #0
    BNE       loc_2DF6
    MVNS     R0, R0
    ADD      R0, R1
    BX       LR

```

As optimizing LLVM concludes, space on the stack for *eos* and *str* may not be allocated, and these variables may always be stored right in registers. Before loop body beginning, *str* will always be in R0, and *eos*—in R1.

``LDRB.W R2, [R1],#1`` instruction loads byte from memory at the address R1 into R2, sign-extending it to 32-bit value, but not just that. #1 at the instruction's end calling “Post-indexed addressing”, this means, 1 is to be added to the R1 after byte load.

Read more about it: [33.1](#).

Then one may spot CMP and BNE<sup>3</sup> in loop body, these instructions continue operation until 0 will be met in string.

MVNS<sup>4</sup> (inverting all bits, NOT in x86) instructions and ADD computes  $eos - str - 1$ . In fact, these two instructions computes  $R0 = str + eos$ , which is effectively equivalent to what was in source code, and why it is so, I already described here ([14.1.1](#)).

Apparently, LLVM, just like GCC, concludes this code will be shorter, or faster.

<sup>3</sup>(PowerPC, ARM) Branch if Not Equal

<sup>4</sup>MoVe Not

**Optimizing Keil 6/2013 (ARM mode)**

Listing 14.4: Optimizing Keil 6/2013 (ARM mode)

```

_strlen
        MOV     R1, R0

loc_2C8 ; CODE XREF: _strlen+14
        LDRB   R2, [R1],#1
        CMP    R2, #0
        SUBEQ  R0, R1, R0
        SUBEQ  R0, R0, #1
        BNE   loc_2C8
        BX    LR

```

Almost the same what we saw before, with the exception the *str - eos - 1* expression may be computed not at the function's end, but right in loop body. *-EQ* suffix, as we may recall, means the instruction will be executed only if operands in executed before *CMP* were equal to each other. Thus, if 0 will be in the *R0* register, both *SUBEQ* instructions are to be executed and result is leaved in the *R0* register.

**ARM64****Optimizing GCC (Linaro) 4.9**

```

my_strlen:
        mov     x1, x0
        ; X1 is now temporary pointer (eos), like cursor
.L58:
        ; load byte from X1 to W2, increment X1 by 1 (post-↵
↵ index)
        ldrb   w2, [x1],1
        ; Compare and Branch if NonZero: compare W2 with 0, ↵
↵ jump to .L58 if it's not
        cbnz   w2, .L58
        ; calculate difference between original X0 pointer and ↵
↵ current value of X1
        sub    x0, x1, x0
        ; decrement lowest 32-bit of result
        sub    w0, w0, #1
        ret

```

The algorithm is the same as in [14.1.1](#): find a zero byte, then calculate difference between pointers, then decrement result. I added some comments. The only thing worth noting is that my example is somewhat broken: *my\_strlen()* function returns 32-bit *int*, while it should return *size\_t* or other 64-bit type. The reason



for that is that, theoretically, `strlen()` can be called for huge blocks in memory, exceeding 4GB, so it must be able to return 64-bit value on 64-bit platform. So because of my mistake, the last SUB instruction operates on 32-bit part of register, while penultimate SUB instruction works on full 64-bit parts (it calculates pointer difference). It's my mistake, but I decided to leave it as is, as an example of what code could be in such case.

### Non-optimizing GCC (Linaro) 4.9

```

my_strlen:
; function epilogue
    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 register always contain 32-bit 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 prologue
    add    sp, sp, 32
    ret

```

It's more verbose. Variables are often tossed here to and from memory (local stack). The same mistake here: decrement operation is happen on 32-bit register part.

## 14.2 Strings trimming

Another very common task is to remove some characters on begin and/or end.

In this example, we will work with a function which removes all newline characters (CR<sup>5</sup>/LF<sup>6</sup>) at the input string end:

```
#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 it's 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,
    // 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")));
```

<sup>5</sup>Carriage return (13 or '\r' in C/C++)

<sup>6</sup>Line feed (10 or '\n' in C/C++)

```

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")));
};

```

Input argument is always returned on exit, this is convenient when you need to chain string processing functions, like it was done here in the `main()` function.

The second part of `for()` (`str_len>0 && (c=s[str_len-1])`) is so called “short-circuit” in C/C++ and is very convenient [Yur13, p. 1.3.8]. C/C++ compilers guarantee evaluation sequence from left to right. So if the first clause is false after evaluation, second will never be evaluated.

## 14.2.1 x64: Optimizing MSVC 2013

Listing 14.5: Optimizing MSVC 2013 x64

```

s$ = 8
str_trim PROC

; RCX is the first function argument and it always holds ↗
↳ pointer to the string

; this is strlen() function inlined right here:
; set RAX to 0xFFFFFFFFFFFFFFFF (-1)
or     rax, -1
$LL14@str_trim:
inc     rax
cmp     BYTE PTR [rcx+rax], 0
jne     SHORT $LL14@str_trim
; is string length zero? exit then
test    eax, eax
$LN18@str_trim:
je     SHORT $LN15@str_trim
; RAX holds string length
; here is probably disassembler (or disassembler printing ↗
↳ routine) error,
; LEA RDX... should be here instead of LEA EDX...
lea     edx, DWORD PTR [rax-1]
; idle instruction: EAX will be reset at the next instruction's ↗
↳ execution
mov     eax, edx
; load character at s[str_len-1]
movzx  eax, BYTE PTR [rdx+rcx]
; save also pointer to the last character to R8
lea     r8, QWORD PTR [rdx+rcx]
cmp     al, 13 ; is it '\r'?

```

```

    je     SHORT $LN2@str_trim
    cmp    al, 10 ; is it '\n'?
    jne    SHORT $LN15@str_trim
$LN2@str_trim:
; store 0 to that place
    mov    BYTE PTR [r8], 0
    mov    eax, edx
; check character for 0, but conditional jump is above...
    test   edx, edx
    jmp    SHORT $LN18@str_trim
$LN15@str_trim:
; return s
    mov    rax, rcx
    ret    0
str_trim ENDP

```

First, MSVC inlined `strlen()` function code right in the code, because it concludes this will be faster than usual `strlen()` work + cost of calling it and returning from it. This is called inlining: 29.

First instruction of inlined `strlen()` is `OR RAX, 0xFFFFFFFFFFFFFFFF`. I don't know, why MSVC uses `OR` instead of `MOV RAX, 0xFFFFFFFFFFFFFFFF`, but it does this often. And of course, it is equivalent: all bits are just set, and all bits set is -1 in two's complement arithmetics: 35.

Why would -1 number be used in `strlen()`, one might ask? Due to optimizations, of course. Here is the code MSVC did:

Listing 14.6: 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    label
; RAX = string length

```

Try to write shorter if you want to initialize counter at 0! Here is my attempt:

Listing 14.7: My 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
```

I failed. We ought to use additional JMP instruction anyway!

So what MSVC 2013 compiler did is moved INC instruction to the place before actual character load. If the very first character is 0, that's OK, RAX is 0 at this moment, so the resulting string length is 0.

All the rest in the function is seems easy to understand. Another trick is at the end. If not to count `strlen()` inlined code, there are only 3 conditional jumps in function. There should be 4: 4th is to be located at the function end, checking, if the character is zero. But there are unconditional jump to the "\$LN18@str\_trim" label, where we see JE, which was first used to check if the input string is empty, right after `strlen()` finish. So the code uses JE instruction at this place for two purposes! This may be overkill, but nevertheless, MSVC did it.

Read more, why it's important to do the job without conditional jumps, if possible: [38.1](#).

## 14.2.2 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
```

```

    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 to c
    ↪ 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

```

I added my comments. After `strlen()` execution, control is passed to `L2` label, and there are two clauses are checked, one after one. Second will never be checked, if first (`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`

**14.2.3 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 it's 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 it's zero
    cmp    cl, 10
    je     .L4
    cmp    cl, 13            ; exit if it's not '\n' and
    ↪ not '\r'
    jne    .L9
.L4:
; this is weird instruction. we need RSI=s-1 here.
; it's possible to get it by MOV RSI, EBX / DEC RSI
; but this is two instructions instead of one
    sub    rsi, rax
; RSI = s+str_len-1-str_len = s-1
; main loop begin
.L12:
    test   rdx, rdx
; store zero at 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 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. Code before loop body begin executed only once, but it has CR/LF characters check too! What this code duplication is for?

Common way to implement main loop is probably this:

- (loop begin) check for CR/LF characters, make decisions
- store zero character

But GCC 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 very good for modern CPUs.

The code doesn't use `str_len` variable, but `str_len-1`. So this is more like index in buffer. Apparently, GCC notices that `str_len-1` statement is used twice. So it's better to allocate a variable which is always holds a value lesser of current string length by one, and decrement it (this is the same effect as decrementing `str_len` variable).

### 14.2.4 ARM64: Non-optimizing GCC (Linaro) 4.9

This implementation is straightforward:

Listing 14.8: 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
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 10      ; is it '\n'?
        bne     .L4        ; goto exit if it's 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]
; is 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 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?
        cmp     w0, wzr
; goto exit, if it's zero or to L5 if it's not
        bne     .L5
.L4:
; return s

```

```
ldr    x0, [x29,24]
ldp   x29, x30, [sp], 48
ret
```

## 14.2.5 ARM64: Optimizing GCC (Linaro) 4.9

This is more advanced optimization. First character is loaded at the beginning, and compared against 10 (LF character). Characters also loaded in the main loop, for characters after first. This is somewhat similar to [14.2.3](#) example.

Listing 14.9: 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
    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 X19+X1=s+str_len-1
    ↪ -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 for 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   wzr, [x2,1]     ; store 0 byte at 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 X19+X1=↪
    ↪ s+str_len-1
    cmp    w2, 10           ; is it '\n'?
    cbz    w2, .L9           ; jump to exit, if it's zero
```

```

    beq     .L12          ; jump to begin loop, if it's '\n'
↳ 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

```

## 14.2.6 ARM: Optimizing Keil 6/2013 (ARM mode)

And again, compiler took advantage of ARM mode conditional instructions, so the code is much more compact.

Listing 14.10: Optimizing Keil 6/2013 (ARM mode)

```

str_trim PROC
    PUSH    {r4,lr}
; R0=s
    MOV     r4,r0
; R4=s
    BL     strlen      ; it 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↵
↳ 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 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

```

## 14.2.7 ARM: Optimizing Keil 6/2013 (thumb mode)

There are less number of conditional instructions in Thumb mode, so the code is more ordinary. But there are one really weird thing with 0x20 and 0x19 offsets. Why Keil compiler did so? Honestly, I have no idea. Probably, this is a quirk of Keil optimization process. Nevertheless, the code will work correctly.

Listing 14.11: Optimizing Keil 6/2013 (thumb mode)

```

str_trim PROC
        PUSH    {r4,lr}
        MOVS   r4,r0
; R4=s
        BL     strlen      ; it takes s value from R0
; R0=str_len
        MOVS   r3,#0
; R3 will always hold zero
        B     |L0.24|
|L0.12|
        CMP    r1,#0xd     ; is loaded byte '\r'?
        BEQ    |L0.20|
        CMP    r1,#0xa     ; is loaded byte '\n'?
        BNE    |L0.38|     ; jump to exit, if no
|L0.20|
        SUBS   r0,r0,#1    ; R0-- or str_len--
        STRB   r3,[r2,#0x1f] ; store 0 at R2+0x1F=s+str_len-0↵
        ↵ x20+0x1F=s+str_len-1
|L0.24|
        CMP    r0,#0       ; str_len==0?
        BEQ    |L0.38|     ; yes, jump to exit
        ADDS   r2,r4,r0    ; R2=R4+R0=s+str_len
        SUBS   r2,r2,#0x20 ; R2=R2-0x20=s+str_len-0x20
        LDRB   r1,[r2,#0x1f] ; load byte at R2+0x1F=s+str_len↵
        ↵ -0x20+0x1F=s+str_len-1 to R1
        CMP    r1,#0       ; is loaded byte 0?
        BNE    |L0.12|     ; jump to loop begin, if it's ↵
        ↵ not 0
|L0.38|
; return s

```

```

MOVS    r0,r4
POP     {r4,pc}
ENDP

```

## 14.3 Exercises

### 14.3.1 Exercise #1

What this code does?

Listing 14.12: Optimizing MSVC 2010

```

_s$ = 8
_f    PROC
      mov     edx, DWORD PTR _s$[esp-4]
      mov     cl, BYTE PTR [edx]
      xor     eax, eax
      test    cl, cl
      je     $LN2@f
      npad   4 ; align next label
$LL4@f:
      cmp     cl, 32
      jne    $LN3@f
      inc     eax
$LN3@f:
      mov     cl, BYTE PTR [edx+1]
      inc     edx
      test    cl, cl
      jne    $LL4@f
$LN2@f:
      ret     0
_f    ENDP

```

Listing 14.13: GCC 4.8.1 -O3

```

f:
.LFB24:
      push    ebx
      mov     ecx, DWORD PTR [esp+8]
      xor     eax, eax
      movzx   edx, BYTE PTR [ecx]
      test    dl, dl
      je     .L2
.L3:
      cmp     dl, 32
      lea    ebx, [eax+1]

```

```

        cmove    eax, ebx
        add     ecx, 1
        movzx   edx, BYTE PTR [ecx]
        test    dl, dl
        jne     .L3
.L2:
        pop     ebx
        ret

```

Listing 14.14: Optimizing Keil 6/2013 (ARM mode)

```

f PROC
MOV     r1,#0
|L0.4|
LDRB    r2,[r0,#0]
CMP     r2,#0
MOVEQ   r0,r1
BXEQ    lr
CMP     r2,#0x20
ADDEQ   r1,r1,#1
ADD     r0,r0,#1
B       |L0.4|
ENDP

```

Listing 14.15: Optimizing Keil 6/2013 (thumb mode)

```

f PROC
MOVS    r1,#0
B       |L0.12|
|L0.4|
CMP     r2,#0x20
BNE     |L0.10|
ADDS    r1,r1,#1
|L0.10|
ADDS    r0,r0,#1
|L0.12|
LDRB    r2,[r0,#0]
CMP     r2,#0
BNE     |L0.4|
MOVS    r0,r1
BX      lr
ENDP

```

Answer [F.1.7](#).

# Chapter 15

## Replacing arithmetic instructions to other ones

While pursuing the goal of optimization, one instruction may be replaced by others, or even with group of instructions.

LEA instruction is also often used for simple arithmetic calculations: [A.6.2](#).

ADD and SUB may replace each other. For example, line 18 in listing [17.31](#).

### 15.1 Multiplication

#### 15.1.1 Multiplication using addition

Here is a simple example:

Listing 15.1: Optimizing MSVC 2010

```
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 code will be faster.

```
_TEXT SEGMENT
_a$ = 8 ; size ↵
    ↵ = 4
_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

```

## 15.1.2 Multiplication using shifting

Multiplication and division instructions by numbers in form  $2^n$  are often replaced by shift instructions.

```

unsigned int f(unsigned int a)
{
    return a*4;
};

```

Listing 15.2: Non-optimizing MSVC 2010

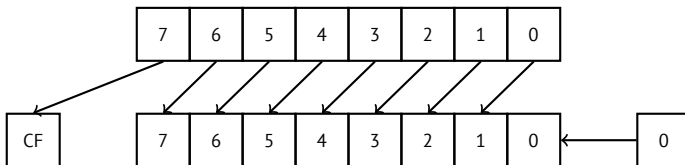
```

_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, while inserting 2 zero bits at right (as the last two bits). It is just like to multiply 3 by 100 – we need just to add two zeroes at the right.

That's how shift left instruction works:



Added bits at right—always zeroes.

Multiplication by 4 in ARM:

Listing 15.3: Non-optimizing Keil 6/2013 (ARM mode)

```

f PROC

```



```

LSL    r0,r0,#2
BX     lr
ENDP

```

### 15.1.3 Multiplication using shifting/subtracting/adding

It's still possible to get rid of multiplication operation when you multiply by numbers like 7 or 17 and still use shifting. Relatively easy mathematics is used here.

#### 32-bit

```

#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};

```

Listing 15.4: 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

```

Keil generating for ARM mode took advantage of second operand's shift modifiers:

Listing 15.5: 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()` function:

Listing 15.6: 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

```

## 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;
};

```

Listing 15.7: 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 shift modifiers:

Listing 15.8: 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

```

## 15.2 Division

### 15.2.1 Division using shifts

For example:

```

unsigned int f(unsigned int a)
{
    return a/4;
};

```

We got (MSVC 2010):

Listing 15.9: MSVC 2010

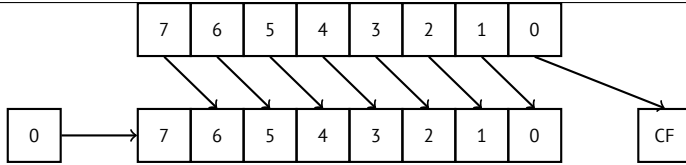
```

_a$ = 8 ; size ↙
    ↙ = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    shr    eax, 2
    ret    0
_f    ENDP

```

SHR (*SHift Right*) instruction in this example is shifting a number by 2 bits right. Two freed bits at left (e.g., two most significant bits) are set to zero. Two least significant bits are dropped. In fact, these two dropped bits –division operation remainder.

SHR instruction works just like as SHL but in other direction.



It can be easily understood if to imagine decimal numeral system and number 23. 23 can be easily divided by 10 just by dropping last digit (3 – is division remainder). 2 is leaving after operation as a [quotient](#).

So remainder is dropped, but that's OK, we work on integer values anyway, these are not floating point numbers!

Division by 4 in ARM:

Listing 15.10: Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
    LSR    r0,r0,#2
    BX    lr
ENDP
```

## 15.3 Division by 9

Very simple function:

```
int f(int a)
{
    return a/9;
};
```

### 15.3.1 x86

...is compiled in a very predictable way:

Listing 15.11: 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 64-bit number stored in the EDX : EAX register pair by value in the ECX register. As a result, EAX will contain [quotient](#), and EDX –remainder. Result is returning from the `f()` function in the EAX register, so, the value is not moved anymore after division operation, it is in right place already. Since IDIV requires value in the EDX : EAX register pair, CDQ instruction (before IDIV) extending value in the EAX to 64-bit value taking value sign into account, just as MOVSX does. If we turn optimization on (`/Ox`), we got:

Listing 15.12: 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 ; 0000001FH
    add     eax, edx
    ret     0
_f  ENDP

```

This is –division by multiplication. Multiplication operation works much faster. And it is possible to use the trick <sup>1</sup> to produce a code which is effectively equivalent and faster.

This is also called “strength reduction” in compiler optimization.

GCC 4.4.1 generates almost the same code even without additional optimization flags, just like MSVC with optimization turned on:

Listing 15.13: 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

```

<sup>1</sup>Read more about division by multiplication in [[War02](#), pp. 10-3]

```

    sar    edx, 1
    mov    eax, ecx
    sar    eax, 1Fh
    mov    ecx, edx
    sub    ecx, eax
    mov    eax, ecx
    pop    ebp
    retn
f      endp

```

## 15.3.2 ARM

ARM processor, just like in any other "pure" RISC-processors, lacks division instruction. It also lacks a single instruction for multiplication by 32-bit constant (recall that 32-bit constant cannot fit into 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 (18).

Here is an example of 32-bit number division by 10 from [Ltd94, 3.3 Division by a Constant]. Quotient and remainder on output.

```

; 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 mostly the same to what was generated by optimizing MSVC and GCC. Apparently, LLVM use the same algorithm for constants generating.

Observant reader may ask, how MOV writes 32-bit value in register, while this is not possible in ARM mode. It is impossible indeed, but, as we see, there are 8 bytes per instruction instead of standard 4, in fact, there are two instructions. First instruction loading 0x8E39 value into low 16 bit of register and second instruction is in fact MOVT, it loading 0x383E into high 16-bit of register. IDA is fully aware of such sequences, and for the sake of compactness, reduced it to one single “pseudo-instruction”.

SMMUL (*Signed Most Significant Word Multiply*) instruction multiply numbers treating them as signed numbers, and leaving high 32-bit part of result in the R0 register, dropping low 32-bit part of result.

``MOV R1, R0, ASR#1`` instruction is arithmetic shift right by one bit.

``ADD R0, R1, R0, LSR#31`` is  $R0 = R1 + R0 \gg 31$

As a matter of fact, there is no separate shifting instruction in ARM mode. Instead, an instructions like (MOV, ADD, SUB, RSB)<sup>2</sup> may be supplied by suffix, is the second operand must be shifted, if yes, by what value and how. ASR meaning *Arithmetic Shift Right*, LSR—*Logical Shift Right*.

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

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 code we saw before in this section, but inserts a call to library function `__divsi3` instead.

What about Keil: it inserts call to library function `__aeabi_idivmod` in all cases.

## 15.3.3 How it works

That’s how division can be replaced by multiplication and division by  $2^n$  numbers:

$$result = \frac{input}{divisor} = \frac{input \cdot \frac{2^n}{divisor}}{2^n} = \frac{input \cdot M}{2^n}$$

<sup>2</sup>These instructions are also called “data processing instructions”

Where  $M$  is *magic*-coefficient.

That's how  $M$  can be computed:

$$M = \frac{2^n}{\text{divisor}}$$

So these code snippets are usually have this form:

$$\text{result} = \frac{\text{input} \cdot M}{2^n}$$

Division by  $2^n$  is usually done by simple right bit shift. If  $n < 32$ , then low part of **product** is shifted (in EAX or RAX). If  $n \geq 32$ , then the high part of **product** is shifted (in EDX or RDX).

$n$  is chosen in order to minimize error.

When doing signed division, sign of multiplication result also added to the output result.

Take a look at the difference:

```
int f3_32_signed(int a)
{
    return a/3;
};

unsigned int f3_32_unsigned(unsigned int a)
{
    return a/3;
};
```

In the unsigned version of function, *magic*-coefficient is 0xAAAAAAB and multiplication result is divided by  $2^{33}$ .

In the signed version of function, *magic*-coefficient is 0x55555556 and multiplication result is divided by  $2^{32}$ . There are no division instruction though: result is just taken from EDX.

Sign is also taken from multiplication result: high 32 bits of result is shifted by 31 (leaving sign in least significant bit of EAX). 1 is added to the final result if sign is negative, for result correction.

Listing 15.14: Optimizing MSVC 2012

```
_f3_32_unsigned PROC
    mov     eax, -1431655765      ; aaaaaaabH
    mul    DWORD PTR _a$[esp-4] ; unsigned multiply
; EDX=(input*0xaaaaaab)/2^32
    shr    edx, 1
; EDX=(input*0xaaaaaab)/2^33
    mov    eax, edx
    ret    0
_f3_32_unsigned ENDP
```

```

_f3_32_signed PROC
    mov     eax, 1431655766          ; 55555556H
    imul   DWORD PTR _a$(esp-4) ; signed multiply
; take high part of product
; it is just the same as if to shift product by 32 bits right ↵
↳ or to divide it by 2^32
    mov     eax, edx                ; EAX=EDX=(input*0x55555556)↵
↳ /2^32
    shr     eax, 31                 ; 0000001fH
    add     eax, edx                ; add 1 if sign is negative
    ret     0
_f3_32_signed ENDP

```

Read more about it in [War02, pp. 10-3].

## 15.3.4 Getting divisor

### Variant #1

Often, the code has a form of:

```

    mov     eax, MAGICAL CONSTANT
    imul   input value
    sar     edx, SHIFTING COEFFICIENT ; signed division by ↵
↳ 2^x using arithmetic shift right
    mov     eax, edx
    shr     eax, 31
    add     eax, edx

```

Let's denote 32-bit *magic*-coefficient as  $M$ , shifting coefficient by  $C$  and divisor by  $D$ .

The divisor we need to get is:

$$D = \frac{2^{32+C}}{M}$$

For example:

Listing 15.15: Optimizing MSVC 2012

```

    mov     eax, 2021161081          ; ↵
↳ 78787879H
    imul   DWORD PTR _a$(esp-4)
    sar     edx, 3
    mov     eax, edx
    shr     eax, 31                 ; ↵
↳ 0000001fH
    add     eax, edx

```

This is:

$$D = \frac{2^{32+3}}{2021161081}$$

Numbers are larger than 32-bit ones, so I use Wolfram Mathematica for convenience:

Listing 15.16: Wolfram Mathematica

```
In[1]:=N[2^(32+3)/2021161081]
Out[1]:=17.
```

So the divisor from the code I used for example is 17.

As of x64 division, things are the same, but  $2^{64}$  should be used instead of  $2^{32}$ :

```
uint64_t f1234(uint64_t a)
{
    return a/1234;
};
```

Listing 15.17: Optimizing MSVC 2012 x64

```
f1234 PROC
    mov     rax, 7653754429286296943      ; 6↵
    ↵ a37991a23aead6fH
    mul     rcx
    shr     rdx, 9
    mov     rax, rdx
    ret     0
f1234 ENDP
```

Listing 15.18: Wolfram Mathematica

```
In[1]:=N[2^(64+9)/16^^6a37991a23aead6f]
Out[1]:=1234.
```

## Variant #2

A variant with omitted arithmetic shift is also exist:

```
mov     eax, 55555556h ; 1431655766
imul   ecx
mov     eax, edx
shr     eax, 1Fh
```

The method of getting divisor is simplified:

$$D = \frac{2^{32}}{M}$$

As of my example, this is:

$$D = \frac{2^{32}}{1431655766}$$

And again I use Wolfram Mathematica:

Listing 15.19: Wolfram Mathematica

```
In[1]:=N[2^32/16^55555556]
Out[1]:=3.
```

The divisor is 3.

## 15.4 Exercises

### 15.4.1 Exercise #1

What this code does?

Listing 15.20: Optimizing MSVC 2010

```
_a$ = 8
_f PROC
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, -968154503 ; c64b2279H
    imul   ecx
    add     edx, ecx
    sar     edx, 9
    mov     eax, edx
    shr     eax, 31          ; 0000001fH
    add     eax, edx
    ret     0
_f ENDP
```

Answer [F.1.8](#).

### 15.4.2 Exercise #2

What this code does?

Listing 15.21: Optimizing MSVC 2010

```
_a$ = 8
```

```
_f PROC
    mov     ecx, DWORD PTR _a$[esp-4]
    lea    eax, DWORD PTR [ecx*8]
    sub    eax, ecx
    ret    0
_f ENDP
```

Listing 15.22: Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
    RSB    r0,r0,r0,LSL #3
    BX    lr
    ENDP
```

Listing 15.23: Non-optimizing Keil 6/2013 (thumb mode)

```
f PROC
    LSLS   r1,r0,#3
    SUBS   r0,r1,r0
    BX    lr
    ENDP
```

Answer [F.1.8](#).

## Chapter 16

# Floating-point unit

**FPU**<sup>1</sup>— is a device within main **CPU** specially designed to deal with floating point numbers.

It was called coprocessor in past. It stay aside of the main **CPU** and looks like programmable calculator in some way.

It is worth to study stack machines<sup>2</sup> before **FPU** studying, or learn Forth language basics<sup>3</sup>.

It is interesting to know that coprocessor in past (before 80486 CPU) was a separate chip and it was not always settled on motherboard. It was possible to buy it separately and install <sup>4</sup>.

Starting at 80486 DX CPU, FPU is always present in it.

**FWAIT** instruction may remind us that fact—it switches **CPU** to waiting state, so it can wait until **FPU** finishes its work. Another rudiment is the fact that **FPU**-instruction opcodes are started with so called “escape”-opcodes (D8 . . DF), i.e., opcodes passed into separate coprocessor.

FPU has a stack capable to hold 8 80-bit registers, each register can hold a number in IEEE 754<sup>5</sup>format. They are ST(0)..ST(7). For brevity, IDA and OllyDbg shows

---

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

<sup>2</sup>[http://en.wikipedia.org/wiki/Stack\\_machine](http://en.wikipedia.org/wiki/Stack_machine)

<sup>3</sup>[http://en.wikipedia.org/wiki/Forth\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Forth_(programming_language))

<sup>4</sup>For example, John Carmack used fixed-point arithmetic values in his Doom video game, stored in 32-bit **GPR** registers (16 bit for intergral part and another 16 bit for fractional part), so the Doom could work on 32-bit computer without FPU, i.e., 80386 and 80486 SX

<sup>5</sup>[http://en.wikipedia.org/wiki/IEEE\\_754-2008](http://en.wikipedia.org/wiki/IEEE_754-2008)

ST(0) as ST, which is represented in some textbooks and manuals as “Stack Top”.

Standard C/C++ languages offer at least two floating number types, *float* (single-precision<sup>6</sup>, 32 bits)<sup>7</sup> and *double* (double-precision<sup>8</sup>, 64 bits).

GCC also supports *long double* type (extended precision<sup>9</sup>, 80 bit) but MSVC is not.

*float* type requires the same number of bits as *int* type in 32-bit environment, but number representation is completely different.

Number in IEEE 754 format consisting of *sign*, *significand* (also called *fraction*) and *exponent*.

## 16.1 Simple example

Let’s consider 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));
};
```

### 16.1.1 x86

#### MSVC

Compile it in MSVC 2010:

Listing 16.1: MSVC 2010: f()

```
CONST    SEGMENT
__real@4010666666666666 DQ 0401066666666666r    ; 4.1
CONST    ENDS
```

<sup>6</sup>[http://en.wikipedia.org/wiki/Single-precision\\_floating-point\\_format](http://en.wikipedia.org/wiki/Single-precision_floating-point_format)

<sup>7</sup>single precision float numbers format is also addressed in the *Working with the float type as with a structure* (19.6.2) section

<sup>8</sup>[http://en.wikipedia.org/wiki/Double-precision\\_floating-point\\_format](http://en.wikipedia.org/wiki/Double-precision_floating-point_format)

<sup>9</sup>[http://en.wikipedia.org/wiki/Extended\\_precision](http://en.wikipedia.org/wiki/Extended_precision)



```

CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_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 3.13

    fld     QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b; ST(1) = result of _a divided
↳ by 3.13

    fmul   QWORD PTR __real@401066666666666666

; current stack state: ST(0) = result of _b * 4.1; ST(1) =
↳ result of _a divided by 3.13

    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 load the number into the ST(0) register, automatically converting it into internal 80-bit format (*extended precision*).

FDIV divides value in the ST(0) register by number stored at address `__real@40091e`. 3.14 value is encoded there. Assembler syntax missing floating point numbers, so, what we see here is hexadecimal representation of 3.14 number in 64-bit IEEE 754 format.

After FDIV execution, ST(0) will hold [quotient](#).

By the way, there is also FDIVP instruction, which divides ST(1) by ST(0), popping both these values from stack and then pushing result. If you know Forth

language<sup>10</sup>, you will quickly understand that this is stack machine<sup>11</sup>.

The next FLD instruction pushing  $b$  value into stack.

After that, quotient is placed to the ST(1) register, and the ST(0) will hold  $b$  value.

The next FMUL instruction do multiplication:  $b$  from the ST(0) register by value at `__real@4010666666666666` (4.1 number is there) and leaves result in the ST(0) register.

The very last FADDP instruction adds two values at top of stack, storing result to the ST(1) register and then popping value at ST(1), hereby leaving result at top of stack in the ST(0).

The function must return result in the ST(0) register, so there are no any other instructions except of function epilogue after FADDP.

## MSVC + OllyDbg

I marked by red 2 pairs of 32-bit words in stack. Each pair is double-number in IEEE 754 format passed from `main()`. We see how first FLD loads a value (1.2) from stack and put it into ST(0) register: fig.16.1. Because of unavoidable conversion errors from 64-bit IEEE 754 float point number into 80-bit (used internally in FPU), we see here 1.999..., which is close to 1.2. EIP right now is pointing to the next instruction (FDIV), which loads double-number (a constant) from memory. For convenience, OllyDbg shows its value: 3.14.

Let's trace more. FDIV executed, now ST(0) contain 0.382... (quotient): fig.16.2.

Third step: the next FLD executed, loading 3.4 into ST(0) (we see here approximated value 3.39999...): fig.16.3. At the same time, quotient pushed into ST(1). Right now, EIP points to the next instruction: FMUL. It loads 4.1 constant from memory, so OllyDbg shows it here.

Next: FMUL was executed, now product is in ST(0): fig.16.4.

Next: FADDP was executed, now result of addition is in ST(0), and ST(1) is cleared: fig.16.5.

Result is left in ST(0), because the function returns its value in ST(0). `main()` will take this value from the register soon.

We also see something unusual: 13.93... value is now located in ST(7). Why?

As I wrote before, FPU registers is stack: 16. But this is simplification. Just imagine if it would be implemented *in hardware* as it's described, then all the rest 7

<sup>10</sup>[http://en.wikipedia.org/wiki/Forth\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Forth_(programming_language))

<sup>11</sup>[http://en.wikipedia.org/wiki/Stack\\_machine](http://en.wikipedia.org/wiki/Stack_machine)

register's contents must be moved (or copied) to adjacent registers during pushing and popping, and that's a lot of work. In reality, **FPU** has just 8 registers and a pointer (called TOP) which has register number, which is current "top of stack". When value is pushed into stack, TOP register is changing and pointing to a next available register, and then a value is written to it. The procedure is reversed if value is popped, however, register which was freed is not cleared (it could be cleared, but this is another work which may degrade performance). So that's what we see here. It can be said, **FADDP** saved sum in stack, and then popped one element. But in fact, this instruction saved sum and then shifted TOP register. More precisely, **FPU** registers is circular buffer.

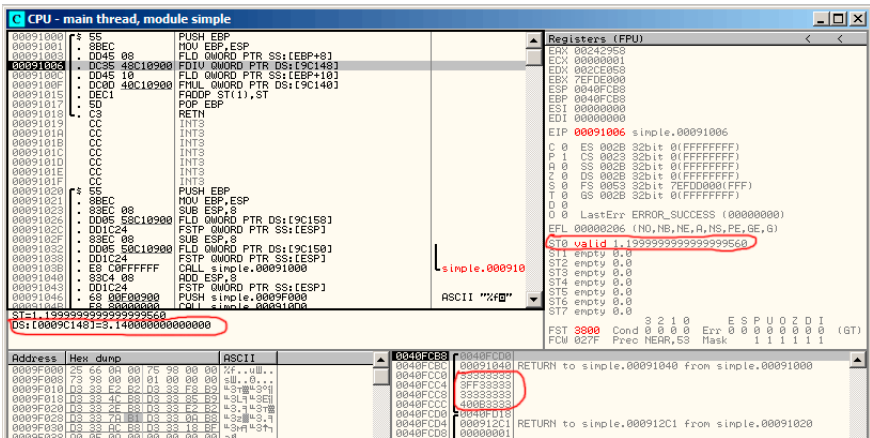


Figure 16.1: OllyDbg: first FLD executed

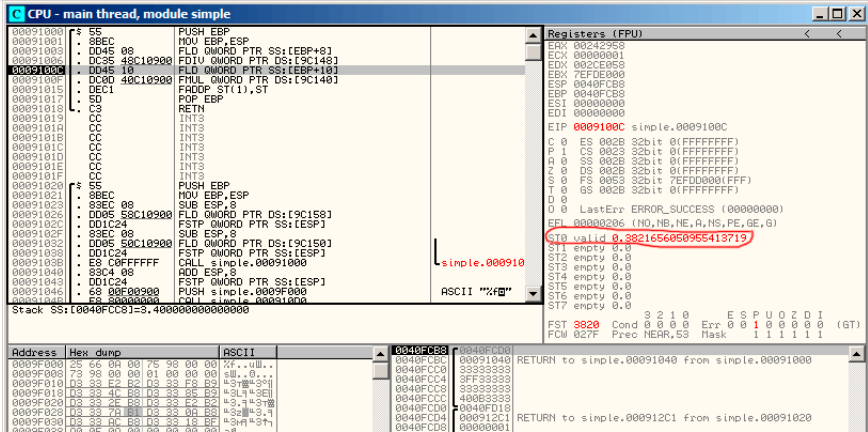


Figure 16.2: OllyDbg: FDIV executed

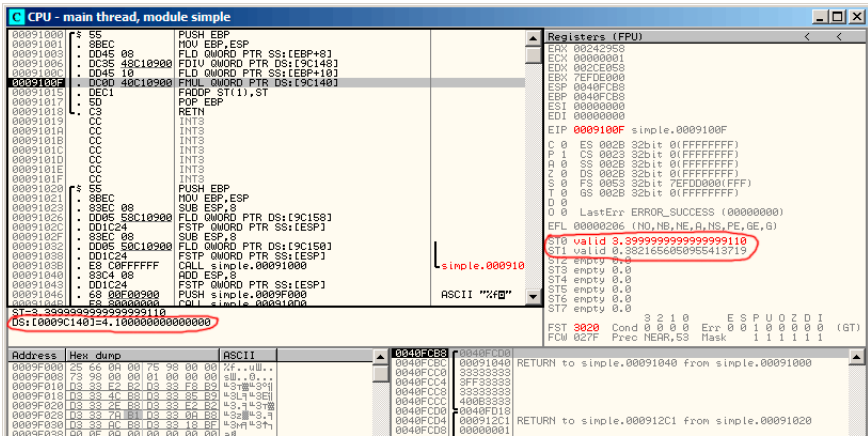


Figure 16.3: OllyDbg: second FLD executed

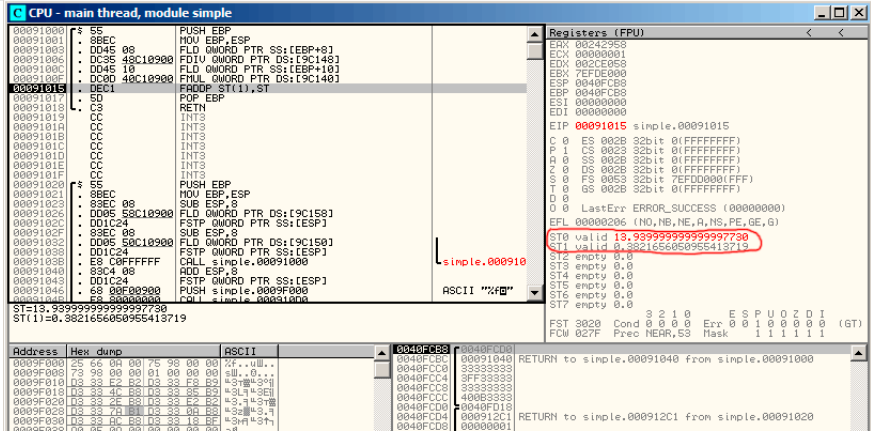


Figure 16.4: OllyDbg: FMUL executed

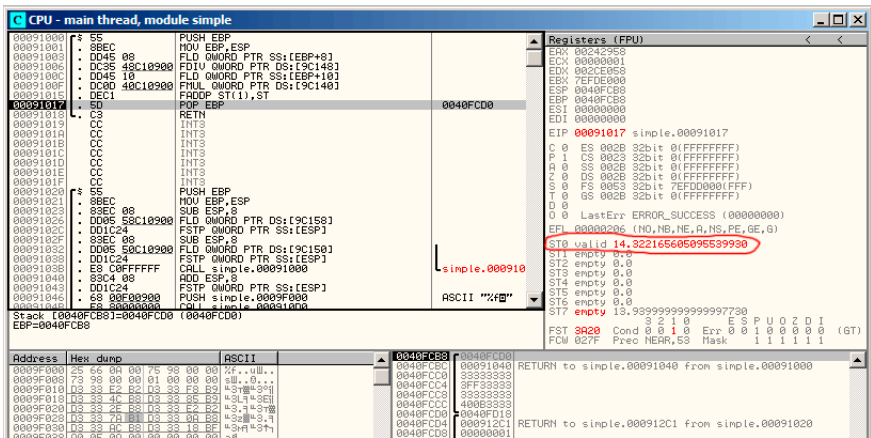


Figure 16.5: OllyDbg: FADDP executed

**GCC**

GCC 4.4.1 (with -O3 option) emits the same code, however, slightly different:

Listing 16.2: 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) = 3.13

        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

f      retn
        endp

```

The difference is that, first of all, 3.14 is pushed to stack (into  $ST(0)$ ), and then value in  $arg\_0$  is divided by value in the  $ST(0)$  register.

**FDIVR** meaning *Reverse Divide* –to divide with divisor and dividend swapped with each other. There is no likewise instruction for multiplication since multiplication is commutative operation, so we have just **FMUL** without its **-R** counterpart.

**FADDP** adding two values but also popping one value from stack. After that operation,  $ST(0)$  holds the sum.

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

Until ARM has floating standardized point support, several processor manufacturers may add their own instructions extensions. Then, **VFP** (*Vector Floating Point*) was standardized.

One important difference from x86, there you working with FPU-stack, but here, in ARM, there are no any stack, you work just with registers.

```

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
dbl_2C98	DCFD 3.14	; DATA XREF: f
dbl_2CA0	DCFD 4.1	; DATA XREF: f+10

So, we see here new registers used, with D prefix. These are 64-bit registers, there are 32 of them, and these 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, they are intended to be used for single precision floating pointer numbers (float). It is easy to remember: D-registers are intended for double precision numbers, while S-registers –for single precision numbers. More about it: [B.3.3](#).

Both (3.14 and 4.1) constants are stored in memory in IEEE 754 form.

VLDR and VMOV instructions, as it can be easily deduced, are analogous to the LDR and MOV instructions, but they works with D-registers. It should be noted that these instructions, just like D-registers, are intended not only for floating point numbers, but can be also used for SIMD (NEON) operations and this will also be revealed soon.

Arguments are passed to function in common way, via R-registers, however, each number having double precision has size 64-bits, so, for passing each, two R-registers are needed.

``VMOV D17, R0, R1`` at the very beginning, composing two 32-bit values from R0 and R1 into one 64-bit value and saves it to D17.

``VMOV R0, R1, D16`` is inverse operation, what was in D16 leaving in two R0 and R1 registers, since double-precision number, needing 64 bits for storage, is returning in the R0 and R1 registers pair.

VDIV, VMUL and VADD, are instruction for floating point numbers processing, computing, [quotient](#), [product](#) and sum, respectively.

The code for thumb-2 is same.

### 16.1.3 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

```

MOVSW R0, R7
MOVSW R1, R4
BL    __aeabi_dmul
MOVSW R7, R0
MOVSW R4, R1
LDR   R2, =0x51EB851F ; 3.14
LDR   R3, =0x40091EB8
MOVSW R0, R5
MOVSW R1, R6
BL    __aeabi_ddiv
MOVSW R2, R7
MOVSW 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 for processor without FPU or NEON support. So, double-precision floating 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 emulates multiplication, division and addition floating-point numbers. Of course, that is slower than FPU-coprocessor, but it is better than nothing.

By the way, similar FPU-emulating libraries were very popular in x86 world when coprocessors were rare and expensive, and were installed only on expensive computers.

FPU-coprocessor emulating called *soft float* or *armel* in ARM world, while using coprocessor's FPU-instructions called *hard float* or *armhf*.

### 16.1.4 ARM64: Optimizing GCC (Linaro) 4.9

Very compact code:

```

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

```

### 16.1.5 ARM64: Non-optimizing GCC (Linaro) 4.9

```

f:
    sub     sp, sp, #16
    str    d0, [sp,8]      ; store a in Register Save Area
    str    d1, [sp]       ; store 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
    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

; redundant code:
        fmov    x0, d0
        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 are a lot of unnecessary value shuffling, including clearly redundant code (last two FMOV instructions). Probably, GCC 4.9 is not yet good on generating ARM64 code. What is worth to note is that ARM64 has 64-bit registers, and D-registers are 64-bit ones as well. So the compiler is free to save values of *double* type in [GPR's](#) instead of local stack. This wasn'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.

## 16.2 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;
}

```

### 16.2.1 x86

Let's see what we got in (MSVC 2010):

Listing 16.3: 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 place for the first variable
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub     esp, 8 ; allocate place 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
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP

```

FLD and FSTP are moving variables from/to data segment to FPU stack. `pow()`<sup>12</sup> taking both values from FPU-stack and returns result in the ST(0) register. `printf()` takes 8 bytes from local stack and interpret them as *double* type variable.

By the way, pair of MOV instructions could be used here for moving values from memory into stack: because 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 ARM example: [16.2.2](#).

## 16.2.2 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

```

<sup>12</sup>standard C function, raises a number to the given power (exponentiation)

```

                                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 I wrote before, 64-bit floating pointer numbers are passed in R-registers pairs. This code is redundant for a little (certainly because optimization is turned off), since it is actually possible to load values into R-registers directly without touching D-registers.

So, as we see, `_pow` function receiving first argument in R0 and R1, and the second one in R2 and R3. Function leaves result in R0 and R1. Result of `_pow` is moved into D16, then in R1 and R2 pair, from where `printf()` will take the resulting number.

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

```

_main
                                STMFDP   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
                                ADR        R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\
↳ n"
                                BL        __2printf
                                MOV        R0, #0
                                LDMFDP   SP!, {R4-R6,PC}

```

```

y          DCD 0xA3D70A4          ; DATA XREF: _main+4
dword_520 DCD 0x3FF8A3D7         ; DATA XREF: _main+8
; double x
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, only R-register pairs are used.

## 16.2.4 ARM64 + 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"

```

Constants are loaded into D0 and D1: pow() function will take them there. Result is in D0 after execution of pow(). It is passed into printf() without any modification and moving, because printf() takes arguments of [integral types](#) and pointers from X-registers, and floating pointer arguments from D-registers.

## 16.3 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));
};

```

Despite simplicity of the function, it will be harder to understand how it works.

### 16.3.1 x86

#### Non-optimizing MSVC

MSVC 2010 generated:

Listing 16.4: 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 loading `_b` into the `ST(0)` register.

FCOMP compares the value in the `ST(0)` register with what is in `_a` value and set `C3/C2/C0` bits in FPU status word register. This is 16-bit register reflecting current state of FPU.

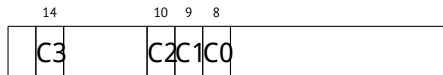
After bits are set, the FCOMP instruction also popping one variable from stack. This is what distinguishes it from FCOM, which is just comparing values, leaving the stack at the same state.

Unfortunately, CPU before Intel P6<sup>13</sup> has not any conditional jumps instructions which are checking `C3/C2/C0` bits. Probably, it is a matter of history (remember: FPU was separate chip in past). Modern CPU starting at Intel P6 has FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions – which does the same, but modifies CPU flags `ZF/PF/CF`.

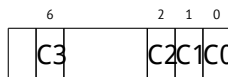
So the FNSTSW instruction copies FPU status word register to the AX. Bits `C3/C2/C0` are placed at positions 14/10/8, they will be at the same positions in the AX register and all they are placed in high part of the AX – AH.

- If  $b > a$  in our example, then `C3/C2/C0` bits will be set as following: 0, 0, 0.
- If  $a > b$ , then bits will be set: 0, 0, 1.
- If  $a = b$ , then bits will be set: 1, 0, 0.
- If result is unordered (in case of error), then bits will be set: 1, 1, 1.

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:



<sup>13</sup>Intel P6 is Pentium Pro, Pentium II, etc

After test ah, 5 execution<sup>14</sup>, only C0 and C2 bits (on 0 and 2 position) will be considered, all other bits will be ignored.

Now let's talk about *parity flag*. Another notable epoch rudiment.

This flag is to be set to 1 if number of ones in last calculation result is even. And to 0 if odd.

Let's look into Wikipedia <sup>15</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 can not 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 and let's see how.

The PF flag will be set to 1 if both C0 and C2 are set to 0 or both are 1. And then following JP (*jump if PF==1*) will be triggered. If we recall values of the C3/C2/C0 for various cases, we will see the conditional jump JP will be triggered in two cases: if  $b > a$  or  $a = b$  (C3 bit is not considering here since it was cleared while execution of the test ah, 5 instruction).

It is all simple thereafter. If conditional jump was triggered, FLD will load the b value to the ST(0) register, and if it is not triggered, the value of the a variable will be loaded.

### What about C2 flag checking?

C2 flag is set in case of error (NaN, etc), but our code doesn't check it. If programmer is aware about FPU errors, he/she must add additional checks.

### First OllyDbg example: a=1.2 and b=3.4

Let's load the example into OllyDbg: fig.16.6. Current function arguments are:  $a = 1.2$  and  $b = 3.4$  (We can see them in stack: two pairs of 32-bit values).  $b$  (3.4) already loaded in ST(0). FCOMP will be executed now. OllyDbg show the second FCOMP argument, which is in stack right now.

<sup>14</sup>5=1001b

<sup>15</sup>[http://en.wikipedia.org/wiki/Parity\\_flag](http://en.wikipedia.org/wiki/Parity_flag)



FCOMP is executed: fig.16.7. We see **FPU** condition flags state: all zeroes. Popped value is reflected as ST(7), I wrote earlier about reason of this: 16.1.1.

**FNSTSW** is executed: fig.16.8. We see that **AX** register contain zeroes: indeed, all condition flags has zeroes. (OllyDbg disassembles **FNSTSW** instruction as **FSTSW** – they are synonyms).

**TEST** is executed: fig.16.9. **PF** flag is one. Indeed: number of bits set in 0 is 0 and 0 is even number. OllyDbg disassembles **JP** as **JPE<sup>16</sup>** – they are synonyms. And it will be triggered right now.

**JPE** triggered, **FLD** loads *b* (3.4) value into **ST(0)**: fig.16.10. The function finishes its work.

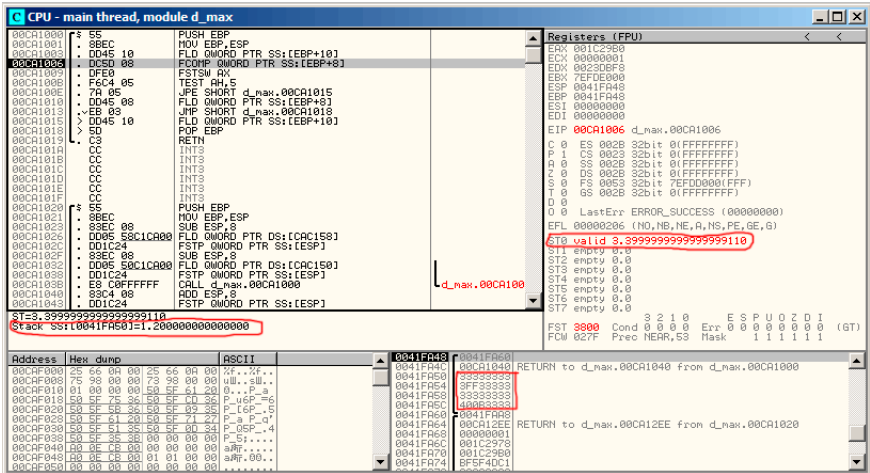


Figure 16.6: OllyDbg; first **FLD** is executed

<sup>16</sup>Jump Parity Even (x86 instruction)

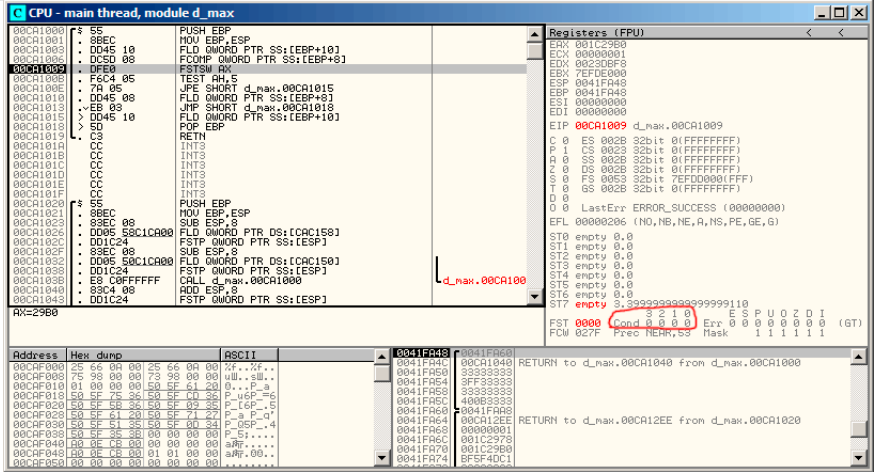


Figure 16.7: OllyDbg: FCOMP is executed

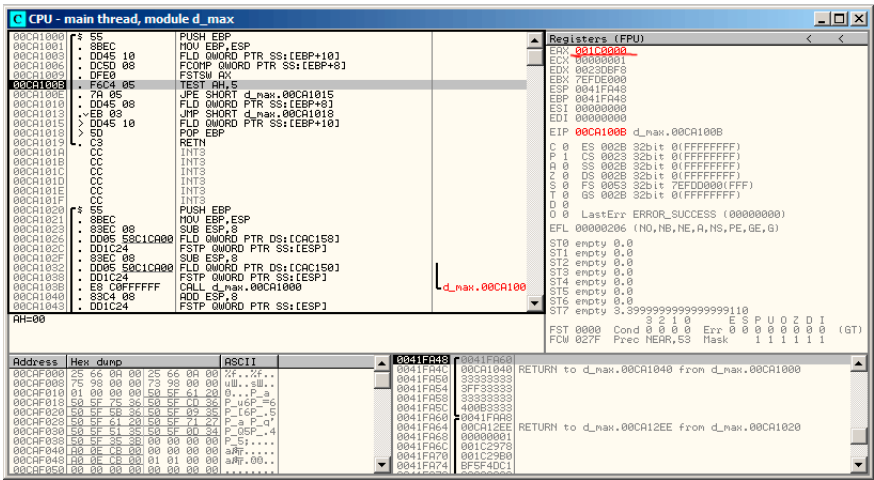


Figure 16.8: OllyDbg: FNSTSW is executed

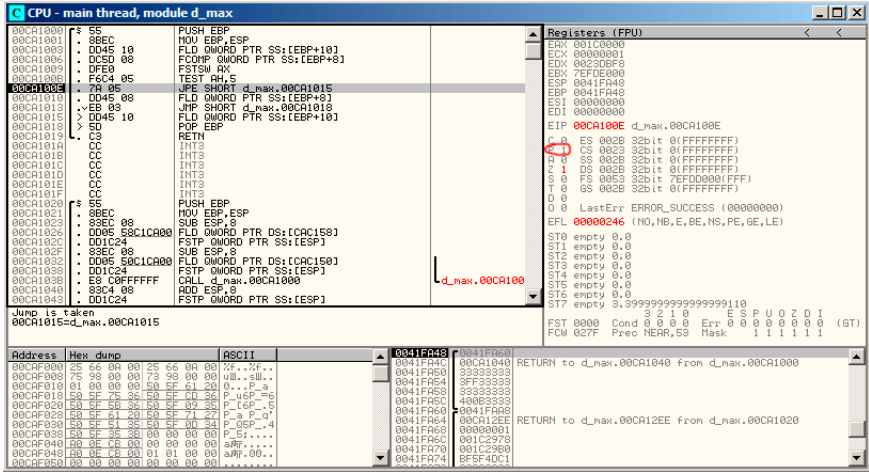


Figure 16.9: OllyDbg: TEST is executed

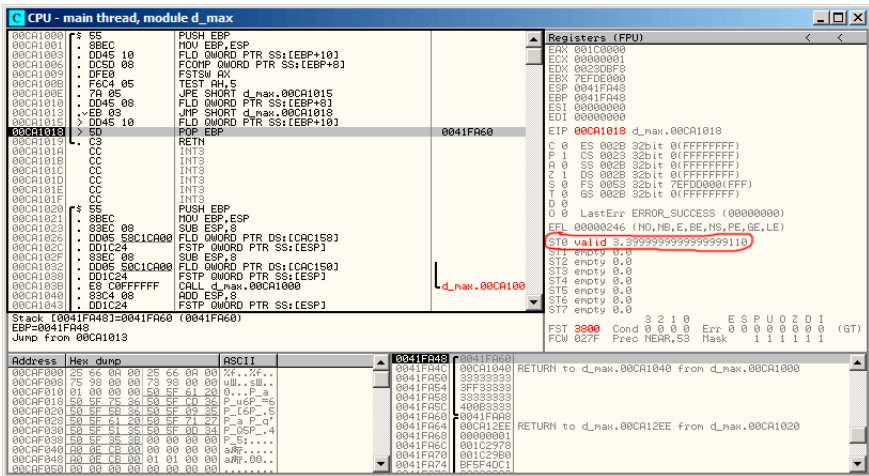


Figure 16.10: OllyDbg: second FLD is executed

### Second OllyDbg example: a=5.6 and b=-4

Let's load example into OllyDbg: fig.16.11. Current function arguments are:  $a = 5.6$  and  $b = -4$ .  $b (-4)$  is already loaded into `ST(0)`. `FCOMP` will be executed now. OllyDbg shows second `FCOMP` argument which is in stack right now.

FCOMP executed: fig.16.12. We see FPU condition flags state: all zeroes except of C0.

FNSTSW executed: fig.16.13. We see that AX register contain 0x100: C0 flag now at the place of 16th bit.

TEST executed: fig.16.14. PF flag is cleared. Indeed: count of bits set in 0x100 is 1 and 1 is odd number. JPE will not be triggered now.

JPE wasn't triggered, FLD loads *a* value (5.6) into ST(0): fig.16.15. The function finishes its work.

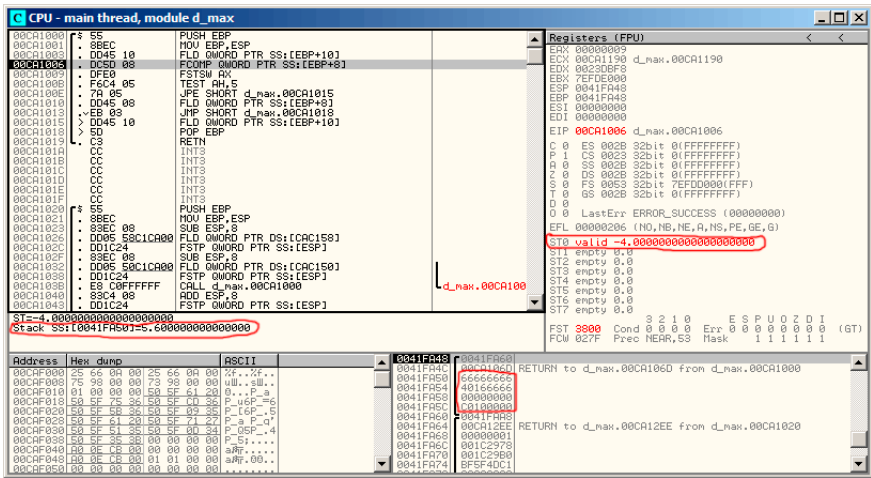


Figure 16.11: OllyDbg: first FLD executed

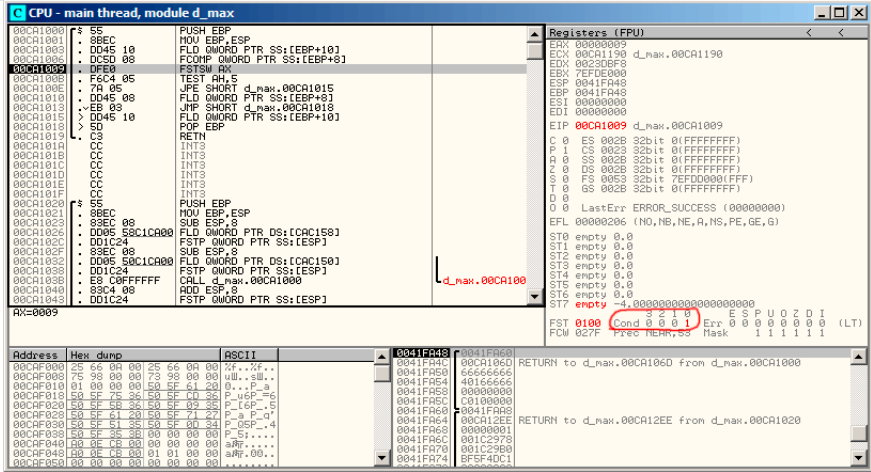


Figure 16.12: OllyDbg: FCOMP executed

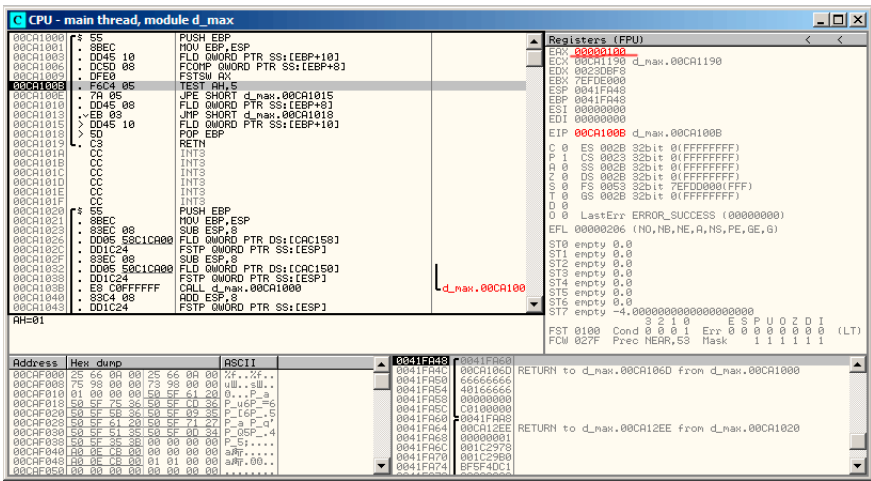


Figure 16.13: OllyDbg: FNSTSW executed

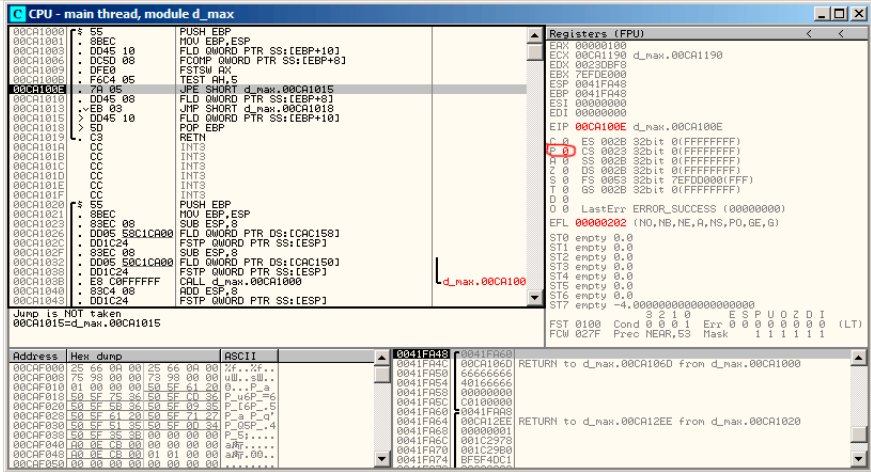


Figure 16.14: OllyDbg: TEST executed

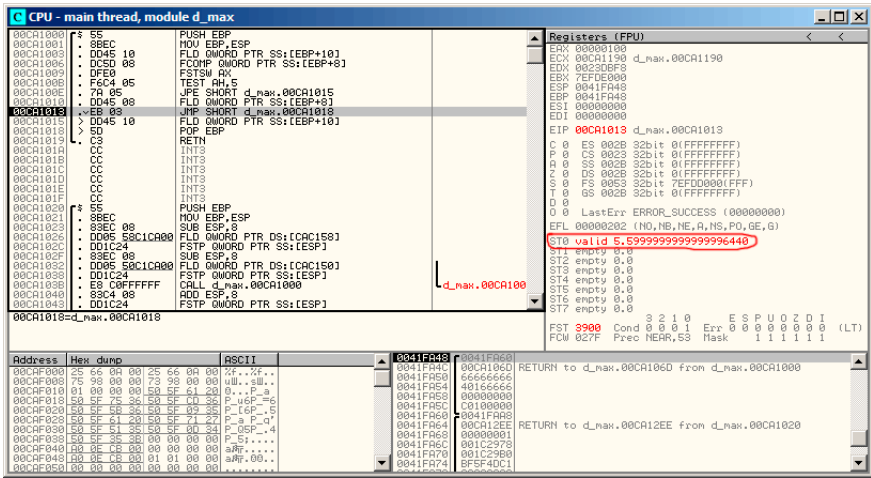


Figure 16.15: OllyDbg: second FLD executed

### Optimizing MSVC 2010

Listing 16.5: 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
    fstp   ST(1) ; copy ST(0) to ST(1) and pop register, ↗
    ↪ leave (_a) on top

; current stack state: ST(0) = _a

    ret    0
$LN5@d_max:
    fstp   ST(0) ; copy ST(0) to ST(0) and pop register, ↗
    ↪ leave (_b) on top

; current stack state: ST(0) = _b

    ret    0
_d_max    ENDP

```

FCOM is distinguished from FCOMP in that sense that it just comparing values and leaves FPU stack in the same state. Unlike previous example, operands here in reverse order. And that is why result of comparison in the C3/C2/C0 will be different:

- If  $a > b$  in our example, then C3/C2/C0 bits will be set as: 0, 0, 0.
- If  $b > a$ , then bits will be set as: 0, 0, 1.
- If  $a = b$ , then bits will be set as: 1, 0, 0.

It can be said, `test ah, 65` instruction just leaves two bits –C3 and C0. Both will be zeroes if  $a > b$ : in that case JNE jump will not be triggered. Then FSTP ST(1) is following –this instruction copies value in the ST(0) into operand and popping one value from FPU stack. In other words, the instruction copies ST(0) (where `_a` value is now) into the ST(1). After that, two values of the `_a` are at the top of stack now. After that, one value is popping. After that, ST(0) will contain `_a` and function is finished.

Conditional JNE jump is triggered in two cases: if  $b > a$  or  $a = b$ . ST(0) will be copied into ST(0), it is just like idle (NOP) operation, then one value is popping from stack and top of stack (ST(0)) will contain what was in the ST(1) before (that is `_b`). Then function finishes. The reason this instruction is used here probably is because FPU has no other instruction to pop value from the stack and discard it.

### First OllyDbg example: a=1.2 and b=3.4

Both FLD executed: fig.16.16. FCOMP being executed: OllyDbg shows contents of ST(0) and ST(1), for convenience.

FCOM is done: fig.16.17. C0 is set, all other condition flags are cleared.

FNSTSW is done, AX=0x3100: fig.16.18.

TEST is executed: fig.16.19. ZF=0, conditional jump will trigger now.

FSTP ST (or FSTP ST(0)) executed: 1.2 was popped from the stack, and 3.4 was left on top of it: fig.16.20. We see that the FSTP ST instruction works just like popping one value from FPU-stack.

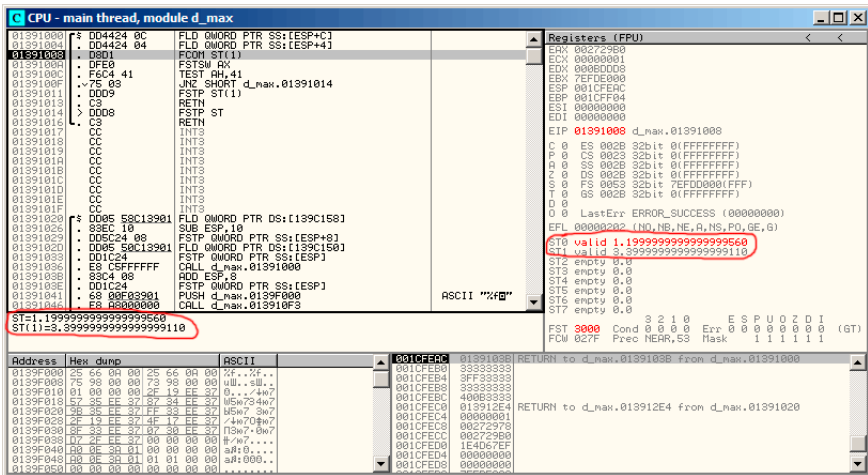


Figure 16.16: OllyDbg: both FLD are executed



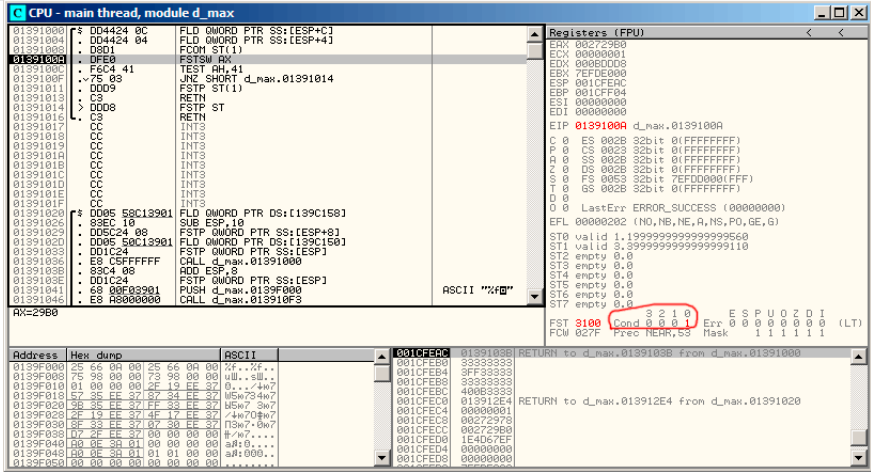


Figure 16.17: OllyDbg: FCOM is executed

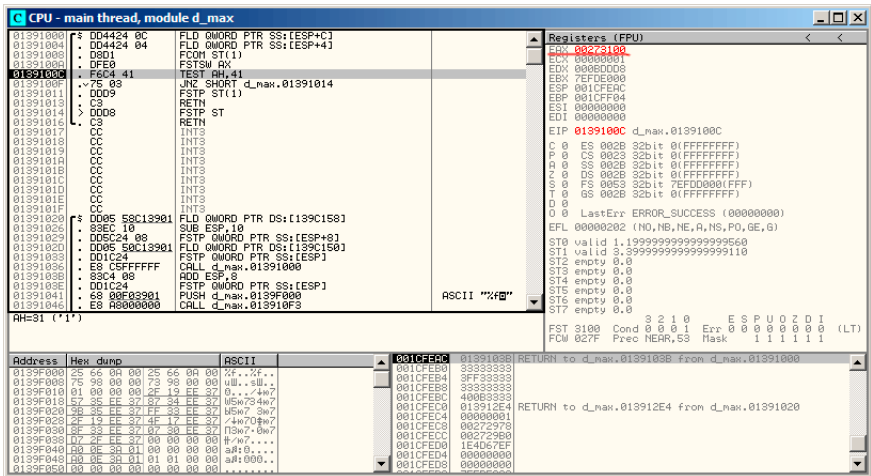


Figure 16.18: OllyDbg: FNSTSW is executed

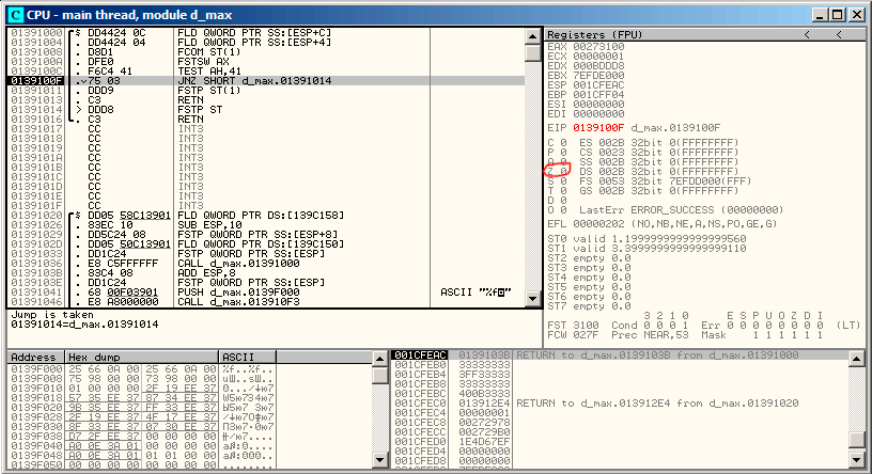


Figure 16.19: OllyDbg: TEST is executed

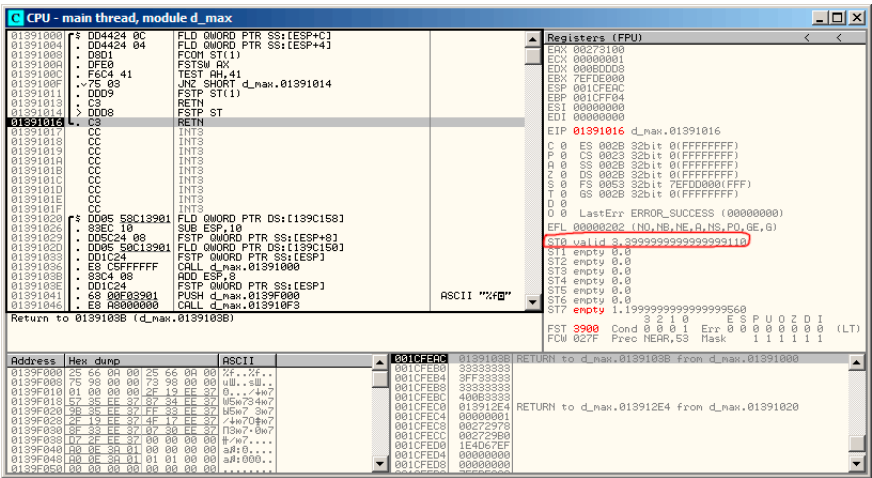


Figure 16.20: OllyDbg: FSTP is executed

Second OllyDbg example: a=5.6 and b=-4

Both FLD executed: fig.16.21. FCOMP being executed.  
 FCOM done: fig.16.22. All condition-flags are cleared.  
 FNSTSW done, AX=0x3000: fig.16.23.  
 TEST is done: fig.16.24. ZF=1, jump will not be triggered now.

FSTP ST(1) was executed: a value of 5.6 is now at the top of FPU-stack. fig.16.25. We now see that FSTP ST(1) instruction works as follows: it leaves what was at the top of stack, but clears ST(1) register.

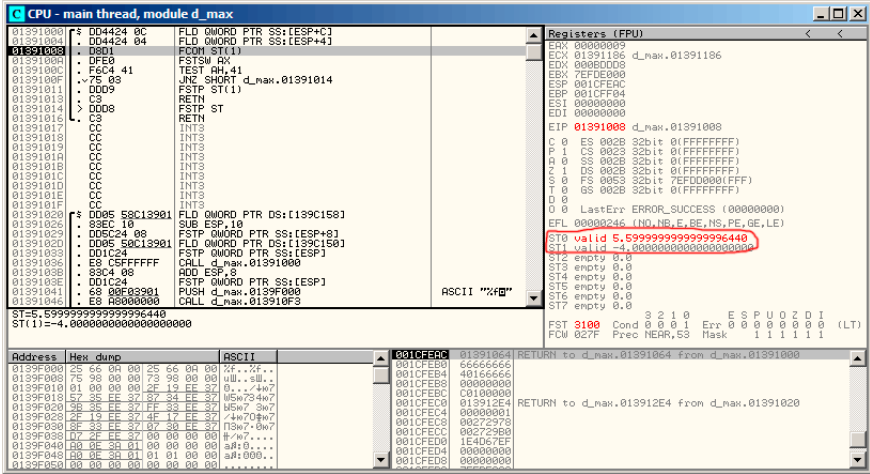


Figure 16.21: OllyDbg: both FLD are executed

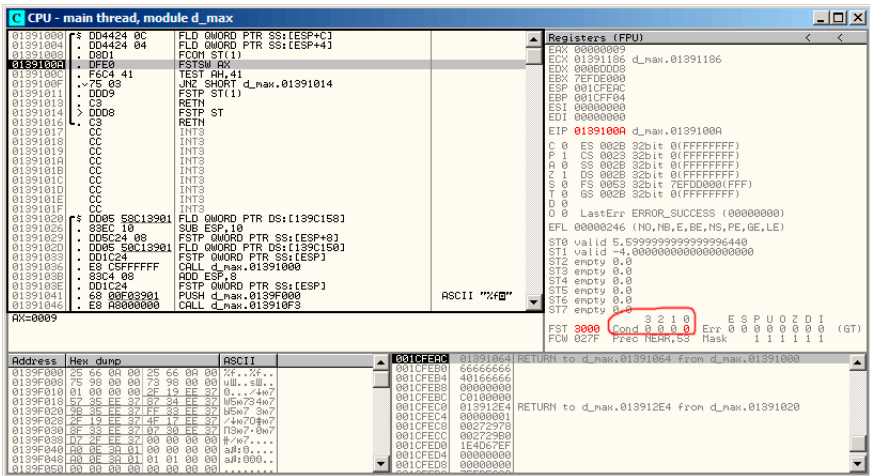


Figure 16.22: OllyDbg: FCOM is executed

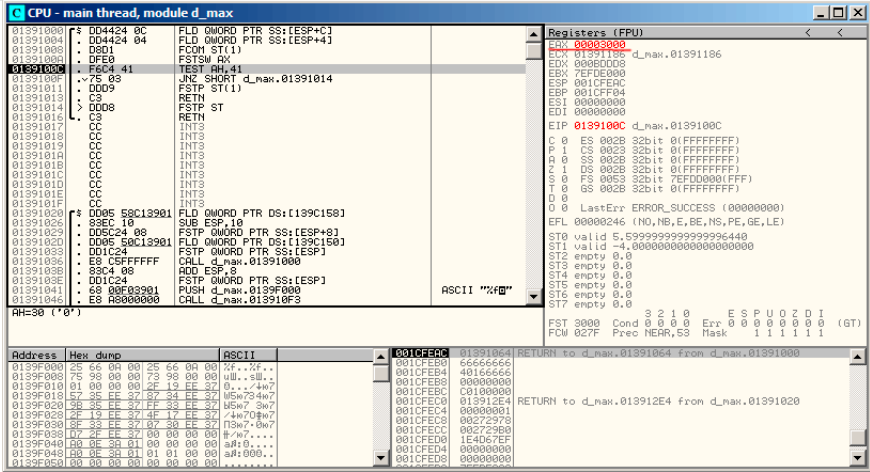


Figure 16.23: OllyDbg: FNSTSW is executed

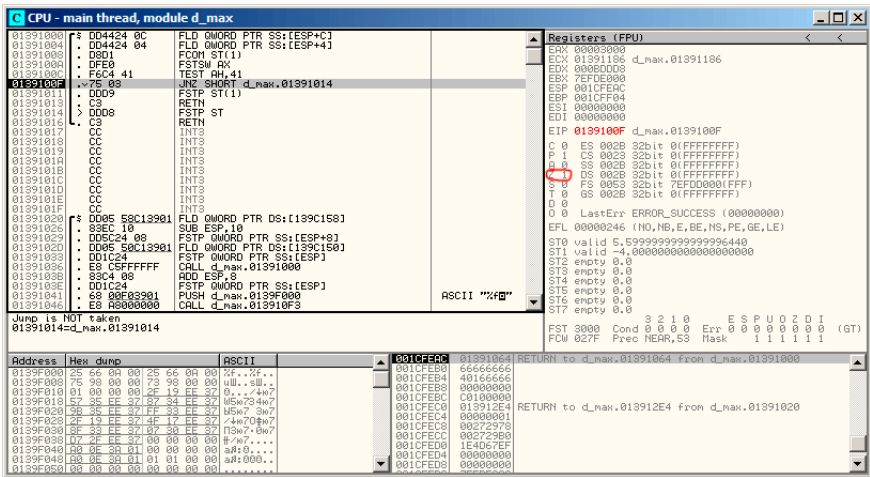


Figure 16.24: OllyDbg: TEST is executed

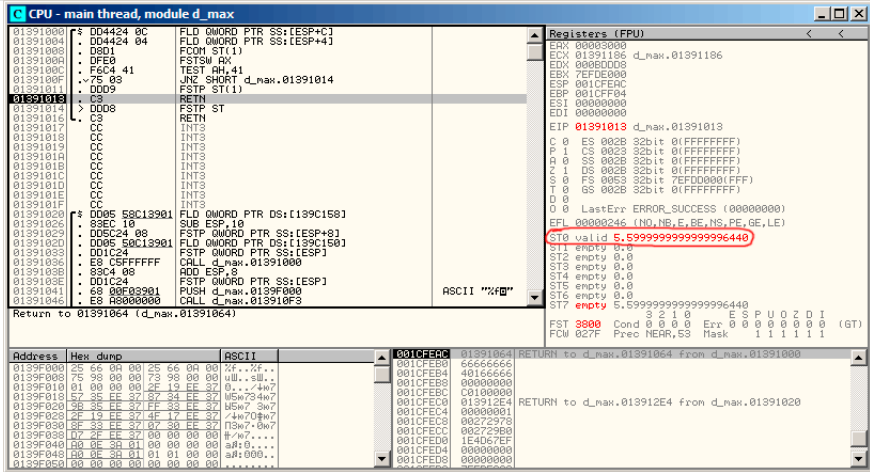


Figure 16.25: OllyDbg: FSTP is executed

### GCC 4.4.1

Listing 16.6: 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
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

    fucomp ; 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 popping both values from stack and handling “not-a-numbers” differently.

More about *not-a-numbers*:

FPU is able to deal with a special values which are *not-a-numbers* or NaNs <sup>17</sup>. These are infinity, result of dividing by 0, etc. Not-a-numbers can be “quiet” and “signaling”. It is possible to continue to work with “quiet” NaNs, but if one try to do any operation with “signaling” NaNs –an exception will be raised.

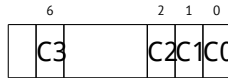
FCOM will raise exception if any operand is NaN. FUCOM will raise exception only if any operand is signaling NaN (SNaN).

The following instruction is SAHF (*Store AH into Flags*) –this is rare instruction in the code not related to FPU. 8 bits from AH are moving into lower 8 bits of CPU flags in the following order:



<sup>17</sup><http://en.wikipedia.org/wiki/NaN>

Let's recall the FNSTSW is moving interesting for us bits C3/C2/C0 into the AH and they will be in positions 6, 2, 0 in the AH register:



In other words, `fnstsw ax / sahf` instruction pair is moving C3/C2/C0 into ZF, PF, CF CPU flags.

Now let's also recall, what values in the C3/C2/C0 bits will be set:

- If  $a$  is greater than  $b$  in our example, then C3/C2/C0 bits will be set as: 0, 0, 0.
- if  $a$  is less than  $b$ , then bits will be set as: 0, 0, 1.
- If  $a = b$ , then: 1, 0, 0.

In other words, after three FUCOMPP/FNSTSW/SAHF instructions, we will have these CPU flags states:

- If  $a > b$ , CPU flags will be set as: ZF=0, PF=0, CF=0.
- If  $a < b$ , then flags will be set as: ZF=0, PF=0, CF=1.
- If  $a = b$ , then: ZF=1, PF=0, CF=0.

It is depends of CPU flags, will SETNBE instruction store 1 to AL or 0. It is almost JNBE instruction counterpart, with the exception that SETcc<sup>18</sup> is storing 1 or 0 to the AL, but Jcc do actual jump or not. SETNBE store 1 only if CF=0 and ZF=0. If it is not true, 0 will be stored into AL.

Only in one case both CF is 0 and ZF is also 0 simultaneously: if  $a > b$ .

Then 1 will be stored to the AL and the following JZ will not be triggered and function will return `_a`. In all other cases, `_b` will be returned.

## Optimizing GCC 4.4.1

Listing 16.7: Optimizing GCC 4.4.1

```

d_max      public d_max
           proc near

arg_0      = qword ptr 8
arg_8      = qword ptr 10h

           push    ebp

```

<sup>18</sup>cc is condition code

```

        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(0), 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 one: JA is used after SAHF. Actually, conditional jump instructions checking “larger”, “lesser” or “equal” for unsigned number comparison (these are JA, JAE, JBE, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) are checking only CF and ZF flags.

Let’s recall how C3/C2/C0 bits are located in the AH register after FSTSW/FNSTSW execution:



Let’s also recall, how bits from AH are stored into CPU flags after SAHF execution:



C3 and C0 bits after comparison are moving into ZF and CF flags exactly, so conditional jumps will be able to work here. JA will work if both CF and ZF are zero.



Thereby, conditional jumps instructions listed here can be used after FNSTSW/SAHF instructions pair.

Apparently, FPU C3/C2/C0 status bits was placed there intentionally, so to easily map them to base CPU flags without additional permutations. But I'm not sure.

### GCC 4.8.1 with -O3 optimization turned on

Some new FPU instructions were appeared in P6 Intel family<sup>19</sup>. These are FUCOMI (compare operands and set flags of the main CPU) and FCMOVcc (works like CMOVcc, but on FPU registers). Apparently, GCC maintainers decided to drop support of pre-P6 Intel CPUs (early Pentiums, etc).

And also, FPU is no longer separate unit in P6 Intel family, so now it is possible to modify/check flags of main CPU from FPU.

So what we got is:

Listing 16.8: 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)
; move 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
```

I'm not sure why FXCH (swap operands) is here. It's possible to get rid of it easily by swapping two first FLD instructions or by replacing FCMOVBE (*below or equal*) by FCMOVA (*above*). Probably, compiler's inaccuracy.

So FUCOMI compares ST(0) (*a*) and ST(1) (*b*) and then sets main CPU flags. FCMOVBE checks flags and copying ST(1) (*b* here at the moment) to ST(0) (*a* here) if  $ST0(a) \leq ST1(b)$ . Otherwise ( $a > b$ ), it leaves *a* in ST(0).

The last FSTP leaves ST(0) on top of stack discarding ST(1) contents.

Let's trace this function in GDB:

Listing 16.9: 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 Copyright (C) 2013 Free Software Foundation, Inc.
```

<sup>19</sup>Starting at Pentium Pro, Pentium-II, etc

```

5 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/↵
  ↳ licenses/gpl.html>
6 This is free software: you are free to change and redistribute ↵
  ↳ it.
7 There is NO WARRANTY, to the extent permitted by law. Type "↵
  ↳ show copying"
8 and "show warranty" for details.
9 This GDB was configured as "i686-linux-gnu".
10 For bug reporting instructions, please see:
11 <http://www.gnu.org/software/gdb/bugs/>...
12 Reading symbols from /home/dennis/polygon/d_max...(no debugging↵
  ↳ symbols found)...done.
13 (gdb) b d_max
14 Breakpoint 1 at 0x80484a0
15 (gdb) run
16 Starting program: /home/dennis/polygon/d_max
17
18 Breakpoint 1, 0x080484a0 in d_max ()
19 (gdb) ni
20 0x080484a4 in d_max ()
21 (gdb) disas $eip
22 Dump of assembler code for function d_max:
23   0x080484a0 <+0>:   fldl   0x4(%esp)
24 => 0x080484a4 <+4>:   fldl   0xc(%esp)
25   0x080484a8 <+8>:   fxch  %st(1)
26   0x080484aa <+10>:  fucomi %st(1),%st
27   0x080484ac <+12>:  fcmovbe %st(1),%st
28   0x080484ae <+14>:  fstp  %st(1)
29   0x080484b0 <+16>:  ret
30 End of assembler dump.
31 (gdb) ni
32 0x080484a8 in d_max ()
33 (gdb) info float
34   R7: Valid   0x3fff999999999999800 +1.19999999999999956
35 =>R6: Valid   0x4000d99999999999800 +3.39999999999999911
36   R5: Empty   0x0000000000000000000
37   R4: Empty   0x0000000000000000000
38   R3: Empty   0x0000000000000000000
39   R2: Empty   0x0000000000000000000
40   R1: Empty   0x0000000000000000000
41   R0: Empty   0x0000000000000000000
42
43 Status Word:      0x3000
44                  TOP: 6
45 Control Word:    0x037f   IM DM ZM OM UM PM
46                  PC: Extended Precision (64-bits)
47                  RC: Round to nearest

```

```

48 Tag Word:                0x0fff
49 Instruction Pointer:    0x73:0x080484a4
50 Operand Pointer:      0x7b:0xbffff118
51 Opcode:                0x0000
52 (gdb) ni
53 0x080484aa in d_max ()
54 (gdb) info float
55 R7: Valid    0x4000d99999999999800 +3.39999999999999911
56 =>R6: Valid  0x3fff999999999999800 +1.19999999999999956
57 R5: Empty   0x00000000000000000000
58 R4: Empty   0x00000000000000000000
59 R3: Empty   0x00000000000000000000
60 R2: Empty   0x00000000000000000000
61 R1: Empty   0x00000000000000000000
62 R0: Empty   0x00000000000000000000
63
64 Status Word:          0x3000
65                      TOP: 6
66 Control Word:        0x037f  IM DM ZM OM UM PM
67                      PC: Extended Precision (64-bits)
68                      RC: Round to nearest
69 Tag Word:            0x0fff
70 Instruction Pointer:  0x73:0x080484a8
71 Operand Pointer:    0x7b:0xbffff118
72 Opcode:              0x0000
73 (gdb) disas $eip
74 Dump of assembler code for function d_max:
75   0x080484a0 <+0>:    fldl   0x4(%esp)
76   0x080484a4 <+4>:    fldl   0xc(%esp)
77   0x080484a8 <+8>:    fxch  %st(1)
78 => 0x080484aa <+10>: fucomi %st(1),%st
79   0x080484ac <+12>: fcmovbe %st(1),%st
80   0x080484ae <+14>: fstp  %st(1)
81   0x080484b0 <+16>: ret
82 End of assembler dump.
83 (gdb) ni
84 0x080484ac in d_max ()
85 (gdb) info registers
86 eax            0x1            1
87 ecx            0xbffff1c4      -1073745468
88 edx            0x8048340       134513472
89 ebx            0xb7fbf000      -1208225792
90 esp            0xbffff10c      0xbffff10c
91 ebp            0xbffff128      0xbffff128
92 esi            0x0            0
93 edi            0x0            0
94 eip            0x080484ac      0x080484ac <d_max+12>

```

```

95  eflags          0x203    [ CF IF ]
96  cs              0x73     115
97  ss              0x7b     123
98  ds              0x7b     123
99  es              0x7b     123
100 fs             0x0      0
101 gs             0x33     51
102 (gdb) ni
103 0x080484ae in d_max ()
104 (gdb) info float
105   R7: Valid    0x4000d99999999999800 +3.39999999999999911
106 =>R6: Valid    0x4000d99999999999800 +3.39999999999999911
107   R5: Empty    0x0000000000000000000
108   R4: Empty    0x0000000000000000000
109   R3: Empty    0x0000000000000000000
110   R2: Empty    0x0000000000000000000
111   R1: Empty    0x0000000000000000000
112   R0: Empty    0x0000000000000000000
113
114 Status Word:      0x3000
115                  TOP: 6
116 Control Word:     0x037f  IM DM ZM OM UM PM
117                  PC: Extended Precision (64-bits)
118                  RC: Round to nearest
119 Tag Word:         0x0fff
120 Instruction Pointer: 0x73:0x080484ac
121 Operand Pointer:   0x7b:0xbffff118
122 Opcode:           0x0000
123 (gdb) disas $eip
124 Dump of assembler code for function d_max:
125   0x080484a0 <+0>:   fldl   0x4(%esp)
126   0x080484a4 <+4>:   fldl   0xc(%esp)
127   0x080484a8 <+8>:   fxch  %st(1)
128   0x080484aa <+10>: fucomi %st(1),%st
129   0x080484ac <+12>: fcmovbe %st(1),%st
130 => 0x080484ae <+14>: fstp  %st(1)
131   0x080484b0 <+16>: ret
132 End of assembler dump.
133 (gdb) ni
134 0x080484b0 in d_max ()
135 (gdb) info float
136 =>R7: Valid    0x4000d99999999999800 +3.39999999999999911
137   R6: Empty    0x4000d99999999999800
138   R5: Empty    0x0000000000000000000
139   R4: Empty    0x0000000000000000000
140   R3: Empty    0x0000000000000000000
141   R2: Empty    0x0000000000000000000

```

```

142  R1: Empty  0x00000000000000000000
143  R0: Empty  0x00000000000000000000
144
145  Status Word:      0x3800
146                    TOP: 7
147  Control Word:    0x037f  IM DM ZM OM UM PM
148                    PC: Extended Precision (64-bits)
149                    RC: Round to nearest
150  Tag Word:        0x3fff
151  Instruction Pointer: 0x73:0x080484ae
152  Operand Pointer:  0x7b:0xbffff118
153  Opcode:          0x0000
154  (gdb) quit
155  A debugging session is active.
156
157      Inferior 1 [process 30194] will be killed.
158
159  Quit anyway? (y or n) y
160  dennis@ubuntuvm:~/polygon$

```

Using “ni”, let’s execute two first FLD instructions.

Let’s examine FPU registers (line 33).

As I wrote before, FPU registers set is circular buffer rather than stack (16.1.1). And GDB shows not STx registers, but internal FPU registers (Rx). Arrow (at line 35) points to the current stack top. You may also see TOP register contents in *Status Word* (line 44) it has 6 now, so stack top is now points to the internal register 6.

*a* and *b* values are swapped after FXCH is executed (line 54).

FUCOMI is executed (line 83). Let’s see flags: CF is set (line 95).

FCMOVBE is actually copied value of *b* (see line 104).

FSTP leaves one value at the top of stack (line 136). TOP value is now 7, so FPU stack top is points to internal register 7.

## 16.3.2 ARM

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

Listing 16.10: 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 b to D16
VMOV      R0, R1, D16
BX        LR

```

A very simple case. Input values are placed into the D17 and D16 registers and then compared with the help of VCMPE instruction. Just like in x86 coprocessor, ARM coprocessor has its own status and flags register, (FPSCR<sup>20</sup>), since there is a need to store coprocessor-specific flags. And just like in x86, there are no conditional jump instruction in ARM, which can check bits in coprocessor status register, so there is VMRS instruction, copying 4 bits (N, Z, C, V) from the coprocessor status word into bits of *general* status (APSR<sup>21</sup> register).

VMOVGT is analogue of MOVGT, instruction for D-registers, to be executed if one operand is greater than other while comparing (*GT*—*Greater Than*).

If it will be executed, *b* value will be written into D16, stored at the moment in D17.

Otherwise, *a* value will stay in the D16 register.

Penultimate instruction VMOV will prepare value in the D16 register for returning it via R0 and R1 registers pair.

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

Listing 16.11: 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	
VMOVGT.F64	D16, D17
VMOV	R0, R1, D16
BX	LR

Almost the same as in previous example, however slightly different. As we already know, many instructions in ARM mode can be supplied by condition predicate.

But there is no such thing in thumb mode. There is no place in 16-bit instructions for spare 4 bits in which condition can be encoded.

However, thumb-2 was extended to make possible to specify predicates to old thumb instructions.

Here, is the *IDA*-generated listing, we see VMOVGT instruction, the same as in previous example.

In fact, usual VMOV is encoded there, but *IDA* added -GT suffix to it, since there is ``IT GT`` instruction placed right before.

IT instruction defines so-called *if-then block*. After the instruction, it is possible to place up to 4 instructions, to which predicate suffix will be added. In our

<sup>20</sup>(ARM) Floating-Point Status and Control Register

<sup>21</sup>(ARM) Application Program Status Register

example, ``IT GT' ' meaning, the next instruction will be executed, if *GT* (*Greater Than*) condition is true.

Now more complex code fragment, by the way, from “Angry Birds” (for iOS):

Listing 16.12: Angry Birds Classic

```
ITE NE
VMOVNE      R2, R3, D16
VMOVEQ      R2, R3, D17
```

ITE meaning *if-then-else* and it encode suffixes for two next instructions. First instruction will execute if condition encoded in ITE (*NE, not equal*) will be true at the moment, and the second –if the condition will not be true. (Inverse condition of NE is EQ (*equal*)).

Slightly harder, and this fragment is also from “Angry Birds”:

Listing 16.13: Angry Birds Classic

```
ITTTT EQ
MOVEQ       R0, R4
ADDEQ       SP, SP, #0x20
POPEQ.W     {R8,R10}
POPEQ       {R4-R7,PC}
```

4 “T” symbols in instruction mnemonic means the 4 next instructions will be executed if condition is true. That’s why [IDA](#) added -EQ suffix to each 4 instructions.

And if there will be, for example, ITEEEE EQ (*if-then-else-else-else*), then suffixes will be set as follows:

```
-EQ
-NE
-NE
-NE
```

Another fragment from “Angry Birds”:

Listing 16.14: Angry Birds Classic

```
CMP.W       R0, #0xFFFFFFFF
ITTE LE
SUBLE.W     R10, R0, #1
NEGLE       R0, R0
MOVGT       R10, R0
```

ITTE (*if-then-then-else*) means the 1st and 2nd instructions will be executed, if LE (*Less or Equal*) condition is true, and 3rd—if inverse condition (GT—*Greater Than*) is true.

Compilers usually not generating all possible combinations. For example, it mentioned “Angry Birds” game (*classic* version for iOS) only these cases of IT instruction are used: IT, ITE, ITT, ITTE, ITTT, ITTTT. How I learnt this? It is possible in *IDA* to produce listing files, so I did it, but I also set in options to show 4 bytes of each opcodes . Then, knowing the high part of 16-bit opcode (IT is 0xBF), I did this using `grep`:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.↵
↵ lst
```

By the way, if to program in ARM assembly language manually for thumb-2 mode, with adding conditional suffixes, assembler will add IT instructions automatically, with respectable flags, where it is necessary.

### Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 16.15: 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 we already saw, but too much redundant code because of  $a$  and  $b$  variables stored in the local stack, as well as returning value.

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

Listing 16.16: 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 it will be supported on the target CPU, and it cannot be done by straightforward bitwise comparing. So there is called external library function for comparing: `__aeabi_cdrcmple`.

N.B. Comparison result is to be left in flags by this function, so the following `BCS` (*Carry set - Greater than or equal*) instruction may work without any additional code.

## 16.3.3 ARM64

### Optimizing GCC (Linaro) 4.9

```

d_max:
; D0 - a, D1 - b
    fcmpe    d0, d1
    fcsele  d0, d0, d1, gt
; now result in D0
    ret

```

ARM64 ISA now also have FPU-instructions which sets **APSR** CPU flags instead of **FPSCR**, for convenience. **FPU** is not separate device here anymore (at least, logically). That is **FCMPE**, it compares two values, passed here in **D0** and **D1** (which are first and second function arguments) and sets **APSR** flags (N, Z, C, V).

**FCSEL** (*Floating Conditional Select*) copies value of **D0** or **D1** into **D0** depending on condition (**GT** (*Greater Than*) here), and again, it uses flags in **APSR** register instead of **FPSCR**. This is much more convenient, if to compare to the instruction set in older CPUs.

If condition is true (**GT**) then value of **D0** is copied into **D0** (i.e., nothing happens). If condition is not true, 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
; 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, function saves input argument values in the local stack (*Register Save Area*). Then the code reloads these values into **X0/X1** registers and finally copies them into **D0/D1** for comparison using **FCMPE**. A lot of redundant code, but that is how non-optimizing compiler may work. **FCMPE** compare values and set **APSR** flags. At this moment, compiler is not yet thinking about more convenient **FCSEL** instruction, so it proceed to old methods: using **BLE**

instruction (*Branch if Less than or Equal*). In one case ( $a > b$ ),  $a$  value is reloaded into X0. In other case ( $a \leq b$ ),  $b$  value is placed in X0. Finally, value from X0 copied into D0, because returning value is leaved in this register.

## Exercise

As an exercise, you may try to optimize this piece of code manually by removing redundant instructions, but do not introduce new ones (including FCSEL).

### Optimizing GCC (Linaro) 4.9—float

I also rewrote this example, now *float* is used 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
    fcsl    s0, s0, s1, gt
; now result in S0
    ret
```

It is a very same code, but S-registers are used instead of D- ones. So numbers of *float* type is passed in 32-bit S-registers (which are in fact lower parts of 64-bit D-registers).

## 16.4 x64

How float point numbers are processed in x86-64, read more here: [25](#).

## 16.5 Exercises

### 16.5.1 Exercise #1

Eliminate FXCH instrucion in example [16.3.1](#) and test it.

**16.5.2 Exercise #2**

What this code does?

Listing 16.17: Optimizing MSVC 2010

```

__real@4014000000000000 DQ 0401400000000000r ; 5

_a1$ = 8 ; size = 8
_a2$ = 16 ; size = 8
_a3$ = 24 ; size = 8
_a4$ = 32 ; size = 8
_a5$ = 40 ; size = 8
_f PROC
    fld     QWORD PTR _a1$[esp-4]
    fadd   QWORD PTR _a2$[esp-4]
    fadd   QWORD PTR _a3$[esp-4]
    fadd   QWORD PTR _a4$[esp-4]
    fadd   QWORD PTR _a5$[esp-4]
    fdiv   QWORD PTR __real@4014000000000000
    ret    0
_f ENDP

```

Listing 16.18: Non-optimizing Keil 6/2013 (thumb mode / compiled for Cortex-R4F CPU)

```

f PROC
    VADD.F64 d0,d0,d1
    VMOV.F64 d1,#5.00000000
    VADD.F64 d0,d0,d2
    VADD.F64 d0,d0,d3
    VADD.F64 d2,d0,d4
    VDIV.F64 d0,d2,d1
    BX     lr
    ENDP

```

Answer [F.1.9](#).

# Chapter 17

## Arrays

Array is just a set of variables in memory, always lying next to each other, always has same type <sup>1</sup>.

### 17.1 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;
};
```

#### 17.1.1 x86

##### MSVC

Let's compile:

---

<sup>1</sup>AKA<sup>2</sup> "homogeneous container"

## Listing 17.1: 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$[ebp+ecx*4]
    push    edx
    mov     eax, DWORD PTR _i$[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: first is filling loop and second is printing loop. `shl ecx, 1` instruction is used for value multiplication by 2 in the ECX, more about below 15.2.1.

80 bytes are allocated on the stack for array, that is 20 elements of 4 bytes.

Let's try this example in OllyDbg.

We see how array gets filled: each element is 32-bit word of *int* type, step by 2: fig.17.1. Since this array is located in stack, we see all its 20 elements inside of stack.

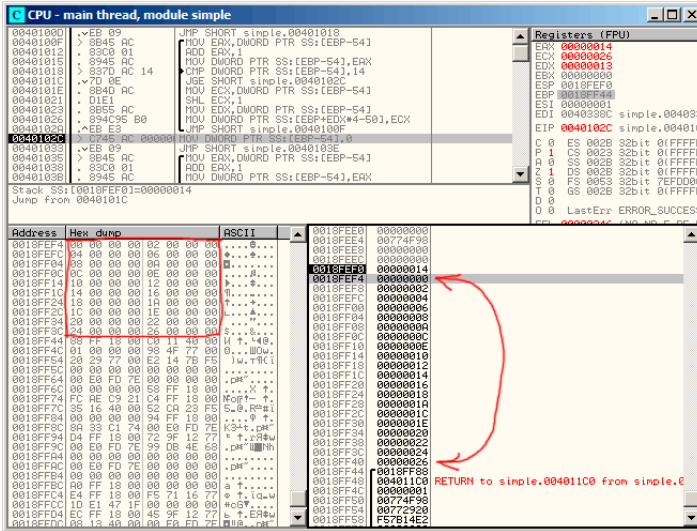


Figure 17.1: OllyDbg: after array filling

**GCC**

Here is what GCC 4.4.1 does:

Listing 17.2: GCC 4.4.1

```

main      public 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
    
```

```

    and    esp, 0FFFFFF0h
    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, *a* variable has *int\** type (the pointer to *int*) –you can try to pass a pointer to array to another function, but it much correctly to say the pointer to the first array element is passed (addresses of another element’s places are calculated in obvious way). If to index this pointer as *a[idx]*, *idx* just to be added to the pointer and the element placed there (to which calculated pointer is pointing) returned.

An interesting example: string of characters like “*string*” is array of characters and it has *const char[]* type. Index can also be applied to this pointer. And that is why it is possible to write like ``string'[i]` –this is correct C/C++ expression!



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

```

EXPORT _main
_main
    STMTFD    SP!, {R4,LR}
    SUB      SP, SP, #0x50      ; allocate place for 20 ↵
    ↵ int variables

; first loop

    MOV      R4, #0            ; i
    B        loc_494

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 place, ↵
    ↵ allocated for 20 int variables
    LDMFD   SP!, {R4,PC}

```

*int* type requires 32 bits for storage (or 4 bytes), so for storage of 20 *int* variables, 80 (0x50) bytes are needed. So that is why ``SUB SP, SP, #0x50``

instruction in function epilogue allocates exactly this amount of space in local stack.

In both first and second loops,  $i$  loop iterator will be placed in the R4 register.

A number to be written into array is calculating as  $i * 2$  which is effectively equivalent to shifting left by one bit, so ``MOV R0, R4, LSL#1`` instruction do this.

``STR R0, [SP, R4, LSL#2]`` writes R0 contents into array. Here is how a pointer to array element is to be calculated: SP pointing to array begin, R4 is  $i$ . So shift  $i$  left by 2 bits, that is effectively equivalent to multiplication by 4 (since each array element has size of 4 bytes) and add it to address of array begin.

The second loop has inverse ``LDR R2, [SP, R4, LSL#2]`` instruction, it loads from array value we need, and the pointer to it is calculated likewise.

### 17.1.3 ARM + 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

loc_1DC
    MOVS    R4, #0           ; i=0
    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 place, 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 value to be written into array and address of each element in array as well.

Compiler allocates slightly more space in local stack, however, last 4 bytes are not used.

## 17.2 Buffer overflow

### 17.2.1 Reading behind array bounds

So, array indexing is just `array[index]`. If you study generated code closely, you'll probably note missing index bounds checking, which could check index, *if it is less than 20*. What if index will be 20 or greater? That's the one C/C++ feature it is often blamed for.

Here is a code successfully compiling and working:

```

#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;
};

```

Compilation results (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

```

I'm running it, and I got:

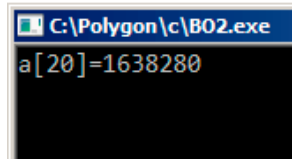


Figure 17.2: OllyDbg: console output

It is just *something*, occasionally lying in the stack near to array, 80 bytes from its first element.

Let's try to find out using OllyDbg, where this value came from. Let's load

and find a value located right after the last array element: fig.17.3. What is this? Judging by stack layout, this is saved EBP register value. Let's trace further and see, how it will be restored: fig.17.4.

Indeed, how it could be done differently? Compiler may generate some additional code for checking index value to be always in array's bound (like in higher-level programming languages<sup>3</sup>) but this makes running code slower.

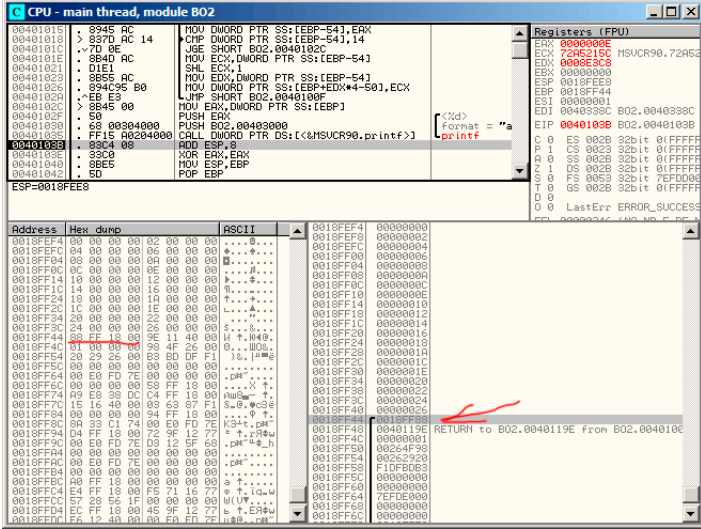


Figure 17.3: OllyDbg: reading of 20th element and execution of `printf()`

<sup>3</sup>Java, Python, etc

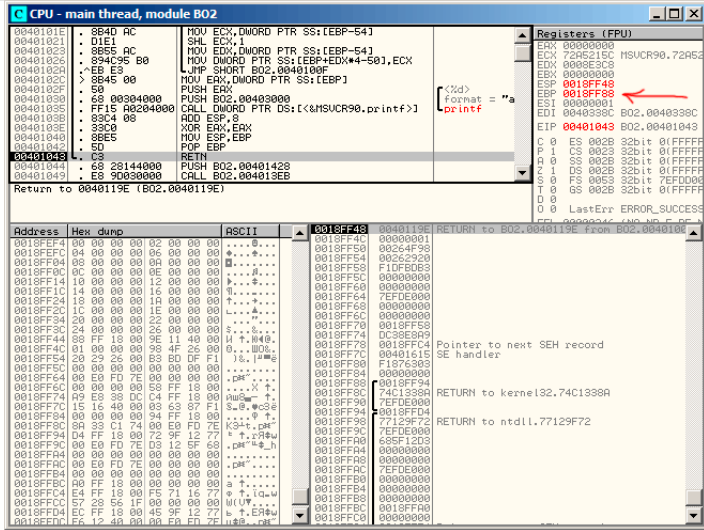


Figure 17.4: OllyDbg: restoring value of EBP register

## 17.2.2 Writing behind array bounds

OK, we read some values from the stack *illegally* but what if we could write something to it?

Here is what we will write:

```

#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
};
    
```

### MSVC

And what we've got:

```

_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

```

Run compiled program and its crashing. No wonder. Let's see, where exactly it is crashing.

Let's load into OllyDbg, trace until all 30 elements are written: [fig.17.5](#).

Trace until the function end: [fig.17.6](#).

Now please keep your eyes on registers.

EIP is 0x15 now. It is not legal address for code –at least for win32 code! We trapped there somehow against our will. It is also interesting fact the EBP register contain 0x14, ECX and EDX –0x1D.

Let's study stack layout more.

After control flow was passed into `main()`, the value in the EBP register was saved on the stack. Then, 84 bytes was allocated for array and `i` variable. That's  $(20+1)*sizeof(int)$ . The ESP pointing now to the `_i` variable in the local stack and after execution of next PUSH something, *something* will be appeared next to `_i`.

That's stack layout while control is inside `main()`:

ESP	4 bytes allocated for <i>i</i> variable
ESP+4	80 bytes allocated for a[20] array
ESP+84	saved EBP value
ESP+88	returning address

a[19]=something statement writes last *int* in array bounds (in bounds so far!)

a[20]=something statement writes *something* to the place where value of the EBP is saved.

Please take a look at registers state at the crash moment. In our case, number 20 was written to 20th element. By the function ending, function epilogue restores original EBP value. (20 in decimal base is 0x14 in hexadecimal). Then, RET instruction was executed, which is effectively equivalent to POP EIP instruction.

RET instruction taking returning address from the stack (that is the address inside of CRT), which was called main( ), and 21 was stored there (0x15 in hexadecimal). The CPU trapped at the address 0x15, but there is no executable code, so exception was raised.

Welcome! It is called *buffer overflow*<sup>4</sup>.

Replace *int* array by string (*char* array), create a long string deliberately, and pass it to the program, to the function which is not checking string length and copies it to short buffer, and you'll able to point to a program an address to which it must jump. Not that simple in reality, but that is how it was emerged<sup>5</sup>

<sup>4</sup>[http://en.wikipedia.org/wiki/Stack\\_buffer\\_overflow](http://en.wikipedia.org/wiki/Stack_buffer_overflow)

<sup>5</sup>Classic article about it: [One96].



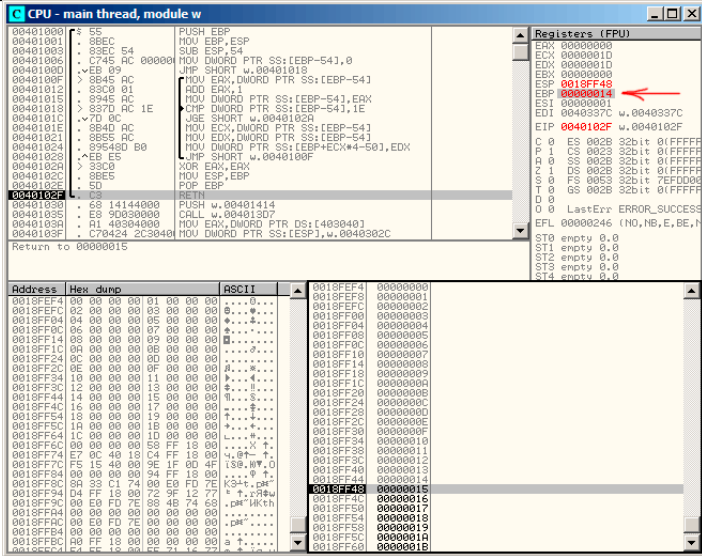


Figure 17.5: OllyDbg: after restoring value of EBP

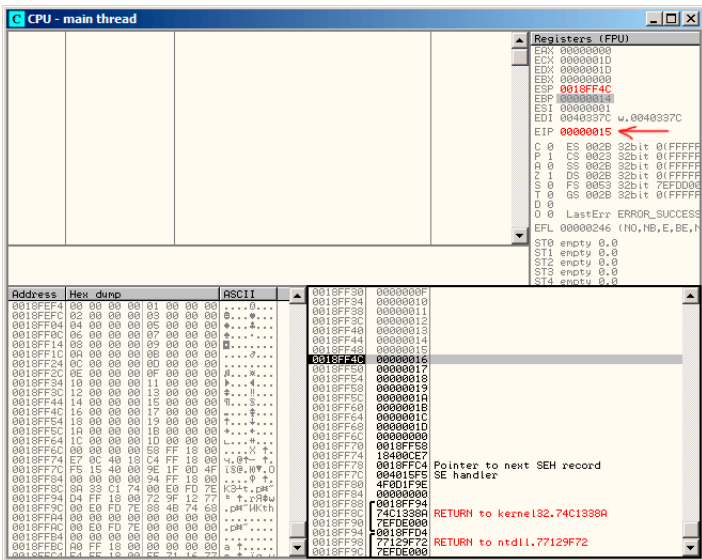


Figure 17.6: OllyDbg: EIP is restored, but OllyDbg can't disassemble at 0x15

**GCC**

Let's try the same code in GCC 4.4.1. We got:

```

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 GDB debugger, we getting this:

```

(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax                0x0          0
ecx                0xd2f96388   -755407992
edx                0x1d         29
ebx                0x26eff4    2551796
esp                0xbffff4b0   0xbffff4b0
ebp                0x15         0x15
esi                0x0          0
edi                0x0          0
eip                0x16         0x16
eflags            0x10202    [ IF RF ]
cs                 0x73         115
ss                 0x7b         123

```

ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x33	51
(gdb)		

Register values are slightly different then in win32 example since stack layout is slightly different too.

## 17.3 Buffer overflow protection methods

There are several methods to protect against it, regardless of C/C++ programmers' negligence. MSVC has options like<sup>6</sup>:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

One of the methods is to write random value among local variables to stack at function prologue and to check it in function epilogue before function exiting. And if value is not the same, do not execute last instruction RET, but halt (or hang). Process will hang, but that is much better then remote attack to your host.

This random value is called “canary” sometimes, it is related to miner’s canary<sup>7</sup>, they were used by miners in these 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 dead.

If to compile our very simple array example (17.1) in MSVC with RTC1 and RTCs option, you will see call to `@_RTC_CheckStackVars@8` function at the function end, checking “canary” correctness.

Let’s see how GCC handles this. Let’s take `alloca()` (4.2.4) 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__
```

<sup>6</sup>Wikipedia: compiler-side buffer overflow protection methods:

[http://en.wikipedia.org/wiki/Buffer\\_overflow\\_protection](http://en.wikipedia.org/wiki/Buffer_overflow_protection)

<sup>7</sup>[http://en.wikipedia.org/wiki/Domestic\\_Canary#Miner.27s\\_canary](http://en.wikipedia.org/wiki/Domestic_Canary#Miner.27s_canary)

```

    snprintf (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 will insert “canary” check into code:

Listing 17.3: 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]
    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

```

Random value is located in `gs : 20`. It is to be written on the stack and then, at the function end, value in the stack is compared with correct “canary” in `gs : 20`. If values are not equal to each other, `__stack_chk_fail` function will be called and we will see something like that in console (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)[0x
↳ xb7606e5c]
/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)[0x
↳ xb75ac935]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2x
↳ _1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2x
↳ _1
0804a000-0804b000 rw-p 00001000 08:01 2097586 /home/dennis/2x
↳ _1
094d1000-094f2000 rw-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-x
↳ linux-gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602 /lib/i386-x
↳ linux-gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602 /lib/i386-x
↳ linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-x
↳ linux-gnu/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781 /lib/i386-x
↳ linux-gnu/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781 /lib/i386-x
↳ 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 [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794 /lib/i386-x
↳ linux-gnu/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794 /lib/i386-x
↳ linux-gnu/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794 /lib/i386-x
↳ linux-gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0 [stack]
Aborted (core dumped)

```

gs—is so-called segment register, these registers were used widely in MS-DOS and DOS-extenders times. Today, its function is different. If to say briefly, the gs register in Linux is always pointing to the TLS (50) –various information specific to thread is stored there (by the way, in win32 environment, the fs register plays the same role, it pointing to TIB<sup>8</sup> 9).

More information can be found in Linux source codes (at least in 3.11 version), in *arch/x86/include/asm/stackprotector.h* file this variable is described in comments.

### 17.3.1 Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)

Let's back to our simple array example (17.1), again, now we can see how LLVM will check “canary” correctness:

```

_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

```

<sup>8</sup>Thread Information Block

<sup>9</sup>[https://en.wikipedia.org/wiki/Win32\\_Thread\\_Information\\_Block](https://en.wikipedia.org/wiki/Win32_Thread_Information_Block)

```

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]
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          ; 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 made loop “unrolled” and all values are written into array one-by-one, already calculated since LLVM concluded it will be faster. By the way, ARM mode instructions may help to do this even faster, and finding this way could be your homework.

At the function end we see “canaries” comparison –that laying in local stack and correct one, to which the R8 register pointing. If they are equal to each other, 4-instruction block is triggered by `ITTTT EQ`, it is writing 0 into R0, function epilogue and exit. If “canaries” are not equal, block will not be executed, and jump to `___stack_chk_fail` function will be occurred, which, as I suppose, will halt execution.

## 17.4 One more word about arrays

Now we understand, why it is impossible to write something like that in C/C++ code [10](#):

<sup>10</sup>However, it is possible in C99 standard[ISO07, pp. 6.7.5/2]: GCC is actually do this by allocating array dynamically on the stack (like `alloca()` (4.2.4))



```
void f(int size)
{
    int a[size];
    ...
};
```

That's just because compiler must know exact array size to allocate space for it in local stack layout on compiling stage.

If you need array of arbitrary size, allocate it by `malloc()`, then access allocated memory block as array of variables of type you need. Or use C99 standard feature [ISO07, pp. 6.7.5/2], but it looks like `alloca()` (4.2.4) internally.

## 17.5 Array of pointers to strings

Here is also example of array of pointers.

Listing 17.4: 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];
};
```

### 17.5.1 x64

## Listing 17.5: 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
$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
    movsxd 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 MOVSSXD instruction copies 32-bit value from ECX (where *month* argument is passed) to RAX with sign-extension (because *month* argument has *int* type). The reason of extension laying in the fact that this 32-bit value will be used in calculation among with other 64-bit values. Hence, it should be promoted to 64-bit value <sup>11</sup>.

<sup>11</sup>It is somewhat weird issue, but negative array index could be passed here (negative array indices will be explained later: 17.8). And if it will, negative input *int* value will be sign-extended correctly and the corresponding element before table will be picked. It will not work correctly without sign-extension.

- Then the address of the whole pointers table is loaded into RCX.
- Finally, the input value (*month*) is multiplied by 8 and added to the address. Indeed: we are in 64-bit environment and all address (or pointers) require exactly 64 bits (or 8 bytes) for storage. Hence, each table element has width of 8 bytes. And that's why to pick element of specific number, *month*\*8 bytes should be skipped from the start. That's what MOV does. But it also loads element at this address. For 1 it would be pointer to the string containing "February" string, etc.

Optimizing GCC 4.9 can do the job even better <sup>12</sup>:

Listing 17.6: 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 32-bit MSVC compiler:

Listing 17.7: 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
```

Input value not needed to be extended to 64-bit value, so it is used as is. And it's multiplied by 4, because table elements has width of 32-bit or 4 bytes.

## 17.5.2 32-bit ARM

### ARM in ARM mode

Listing 17.8: Optimizing Keil 6/2013 (ARM mode)

```
get_month1 PROC
    LDR    r1, |L0.100|
    LDR    r0, [r1, r0, LSL #2]
    BX    lr
    ENDP
```

<sup>12</sup>"0+" left in listing because GCC assembler output is not tidy enough for eliminating it. It's displacement, and it's zero here.

```

|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

```

Table address is loaded into R1. All the rest is done using only one LDR instruction. Then input *month* value is shifted left by 2 (which is the same as multiplying by 4), this value added to R1 (where address of table is) and then a value is loaded at this address. Then 32-bit value is loaded into R0 from the table.

### ARM in Thumb mode

The code is mostly the same, but less dense, because LSL suffix cannot be specified in LDR instruction here:

```

get_month1 PROC
    LSLS   r0,r0,#2
    LDR    r1,|L0.64|
    LDR    r0,[r1,r0]
    BX     lr

```

```
ENDP
```

## 17.5.3 ARM64

Listing 17.9: Optimizing GCC 4.9 ARM64

```
get_month1:
    adrp    x1, .LANCHOR0
    add     x1, x1, :lo12:LANCHOR0
    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"
```

Table address is loaded into X1 using ADRP/ADD pair. Then corresponding element is picked using only one instruction LDR, which takes W0 (register where input *month* argument is), shifts it 3 bits left (which is the same as multiplying by 8), sign-extends it (this is what “sxtw” suffix mean) and adds to X0. Then 64-bit value is loaded into X0 from the table.

### Array overflow

Our function accepts values in range of 0..11, but what if 12 will be passed? There are no element in table present at this place. So, the function will load some value which happened to be located there, and return it. Soon after, some other function will try to get a string at this address and may crash.

I compiled the example in MSVC for win64 and opened it in IDA to see what linker placed after the table:

Listing 17.10: Executable 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 after all, so there are no much data to pack in the data segment, so these are names. But I should to note that there might be really *anything* what linker decided to put by chance.

So what if 12 will be passed to the function? 13th table element will be returned. Let's see how CPU will treat the bytes there as 64-bit value:

Listing 17.11: Executable 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, string processing) will try to read bytes at this address expecting C-string there. Most likely it will crash, because this value is don'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 are also a good “fail early and fail loudly” or “fail-fast” philosophy, which teaches to report problems as early as possible and stop.

One of such methods in C/C++ is assertions. We can modify our program to fail if incorrect value is passed:

Listing 17.12: assert() added

```
const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
};
```

Assertion macro will check for valid values at each function start and fail if it's not.

Listing 17.13: Optimizing MSVC 2013 x64

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00
↳ H, '.', 00H
      DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00
↳ H, '<', 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 $LN3@get_month1
    lea     rdx, OFFSET FLAT:$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 condition, then pass also line number and file name to another function which will show this information to user.

Here we see that both file name and condition was encoded in UTF-16. Line number is also passed (it's 29).

This mechanism is same in probably all compilers. Here is what GCC does:



Listing 17.14: 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 it also pass function name for convenience.

Nothing came for free: sanitizing check as well. It makes your program slower, especially if `assert()` macros used in small time-critical functions. So MSVC, for example, leaves checks in Debug builds, but in Release builds they all are disappear.

Microsoft [Windows NT](#) kernels are also came in “checked” and “free” builds <sup>13</sup>. First has validation checks (hence, “checked”), second doesn’t (hence, “free” of checks).

## 17.6 Multidimensional arrays

Internally, multidimensional array is essentially the same thing as linear array.

Since computer memory in linear, it is one-dimensional array. But this one-dimensional array can be easily represented as multidimensional for convenience.

For example, that is how `a[3][4]` array elements will be placed in one-dimensional array of 12 cells:

<sup>13</sup>[http://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)

[0][0]
[0][1]
[0][2]
[0][3]
[1][0]
[1][1]
[1][2]
[1][3]
[2][0]
[2][1]
[2][2]
[2][3]

Table 17.1: Two-dimensional array represented in memory as one-dimensional

Here is how each cell of 3\*4 array will be placed in memory space:

0	1	2	3
4	5	6	7
8	9	10	11

Table 17.2: Memory addresses of each cell of two-dimensional array

So, in order to calculate an address of elements we need, first multiply first index by 4 (matrix width) and then add second index. That's called *row-major order*, and this method of arrays and matrices representation is used in at least C/C++ and Python. *row-major order* term in plain English language means: "first, write elements of first row, then second row ...and finally elements of last row".

Another method of representation called *column-major order* (array indices 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 elements of first column, then second column ...and finally elements of last column".

### 17.6.1 Two-dimensional array example

We will work with array of *char* type, meaning that each element require only one byte in memory.

#### Row filling example

Let's fill the second row with values: 0 . . . 3:

Listing 17.15: 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++)
            a[x][y]=0;

    // fill second row by 0..3:
    for (y=0; y<4; y++)
        a[1][y]=y;
};
```

I marked all three rows with red. We see that second row now has values 0, 1, 2 and 3:

Address	Hex dump	ASCII
01303370	00 00 00 00 00 01 02 03	.....00*
01303378	00 00 00 00 00 00 00 00	.....
01303380	02 00 00 00 3A 3D C4 9E	@...:=-H
01303388	3A 3D C4 9E 00 00 00 00	:=-H....
01303390	00 00 00 00 00 00 00 00	.....

Figure 17.7: OllyDbg: array is filled

### Column filling example

Let's fill the third column with values: 0...2.

Listing 17.16: 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;
};

```

I also marked three rows by red here. We see that in each row, at third position, these values are written: 0, 1 and 2.

Address	Hex dump	ASCII
00B13370	00 00 00 00 00 00 00 01 00	.....0.
00B13378	00 00 02 00 00 00 00 00 00	..0.....
00B13380	02 00 00 00 93 61 C8 E6	0...Ya <u>4</u>
00B13388	93 61 C8 E6 00 00 00 00	Ya <u>4</u> ....
00B13390	00 00 00 00 00 00 00 00	.....

Figure 17.8: OllyDbg: array is filled

## 17.6.2 Access two-dimensional array as one-dimensional

I can easily show how to access two-dimensional array as one-dimensional array in at least two other ways:

```

#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
    return array[a][b];
};

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 it and run: it will show correct values.

What did MSVC 2013 is fascinating, all three routines are just the same!

Listing 17.17: Optimizing MSVC 2013 x64

```

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=array
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+array
        movzx   eax, BYTE PTR [rax+r9*4]
; AL=load at RAX+R9*4=b+array+a*4=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:

Listing 17.18: Optimizing GCC 4.9 x64

```

; RDI=array
; RSI=a
; RDX=b

get_by_coordinates1:
; sign-extend input 32-bit int values a,b to 64-bit
    movsx  rsi, esi
    movsx  rdx, edx
    lea    rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=array+a*4
    movzx  eax, BYTE PTR [rax+rdx]
; AL=load at RAX+RDX=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 at RDI+RAX=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,b to 64-bit
    movsx  rdx, edx
    movsx  rsi, esi
    add    rdi, rsi
; RDI=RDI+RSI=array+a*4
    movzx  eax, BYTE PTR [rdi+rdx]
; AL=load at RDI+RDX=array+a*4+b
    ret

```

### 17.6.3 Three-dimensional array example

Same thing about multidimensional arrays.

Now we will work with array of *int* type: each element require 4 bytes in memory.

Let's see:

Listing 17.19: 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 got (MSVC 2010):

Listing 17.20: MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8           ; size = 4
_y$ = 12          ; size = 4
_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 multiplying by formula  $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$  to represent array as multidimensional. Do not forget the *int* type is 32-bit (4 bytes), so all coefficients must be multiplied by 4.

Listing 17.21: GCC 4.4.1

```

insert      public 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

```

GCC compiler does it differently. For one of operations calculating  $(30y)$ , GCC produced a code without multiplication instruction. 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  $30y$  calculation, only one addition operation used, one bitwise shift operation and one subtraction operation. That works faster.



**ARM + Non-optimizing Xcode 4.6.3 (LLVM) (thumb mode)**

Listing 17.22: Non-optimizing Xcode 4.6.3 (LLVM) (thumb mode)

```

_insert

value  = -0x10
z      = -0xC
y      = -8
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]
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 place 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, however, it is redundant. Address of array element is calculated by formula we already figured out.

**ARM + Optimizing Xcode 4.6.3 (LLVM) (thumb mode)**

Listing 17.23: 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^2
    ↪ *15
ADD     R9, PC           ; R9 = pointer to a array
LDR.W   R9, [R9]
MLA.W   R0, R0, R12, R9 ; R0 = x, R12 = 2400, R9 = pointer to a
    ↪ 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

```

Tricks for replacing multiplication by shift, addition and subtraction which we already considered are also present here.

Here we also see new instruction for us: *RSB (Reverse Subtract)*. It works just as *SUB*, but it swaps its operands with each other before. Why? *SUB* and *RSB* are those instructions, to the second operand of which shift coefficient may be applied: (*LSL#4*). But this coefficient may be applied only to second operand. That's fine for commutative operations like addition or multiplication (operands may be swapped there without result affecting). But subtraction is non-commutative operation, so *RSB* exist for these cases.

```LDR.W R9, [R9]``` instruction works like *LEA (A.6.2)* in x86, but it does nothing here, it is redundant. Apparently, compiler not optimized it.

## 17.6.4 More examples

Computer screen is represented as 2D array, but video-buffer is linear 1D array. We talk about it here: [68.2](#).

## 17.7 Pack of strings as two-dimensional array

Let's revisit function which returns name of month: [listing.17.4](#). As you may see, at least one memory load operation is needed to prepare a pointer to the string consisting of month name. Will it be possible to get rid of this memory load operation? In fact yes, if to represent 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 },
    { '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, 0 },
    { 'J','u','n','e', 0, 0, 0, 0, 0, 0 },
    { 'J','u','l','y', 0, 0, 0, 0, 0, 0 },
    { 'A','u','g','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 got:

Listing 17.24: 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
    movsxd rax, ecx
    lea    rcx, QWORD PTR [rax+rax*4]
; RCX=month+month*4=month*5
    lea    rax, OFFSET FLAT:month2
; RAX=pointer to the 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 access at all. All this function do is calculating point at which the first character of month name is: *pointer\_to\_the\_table* + *month* \* 10. There are also two LEA instructions which are effectively working as several MUL and MOV instructions.

Width of the array is 10 bytes. Indeed, longest string here is “September” (9 bytes) plus terminating zero is 10 bytes. Other 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 are started at points which can be easily calculated.

GCC 4.9 can do it even shorter:

Listing 17.25: Optimizing GCC 4.9 x64

```
movsx   rdi, edi
lea     rax, [rdi+rdi*4]
lea     rax, month2[rax+rax]
ret
```

LEA is also used for multiplication by 10 here.

Non-optimizing compilers do multiplication differently.

Listing 17.26: 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 use IMUL instruction:

Listing 17.27: 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
    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 to add multiplication by zero and adding zero to the final result? I don't know, this looks like compiler code generator quirk, which wasn't caught by compiler's tests (resulting code is correct after all). I intentionally add such pieces of code so the reader would understand, that sometimes one shouldn't puzzle over such compiler's artifacts.

### 17.7.1 32-bit ARM

Optimizing Keil for Thumb mode use multiplication instruction MULS:

Listing 17.28: 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

```

Optimizing Keil for ARM mode use add and shift operations:

Listing 17.29: 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

```

## 17.7.2 ARM64

Listing 17.30: 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 64-bit one and storing it in X0. ADRP/ADD pair is used for loading address of the table. ADD instructions also has LSL suffix, which helps with multiplications.

## 17.7.3 Conclusion

This is a bit old-school technique to store strings. You may find a lot of it in Oracle RDBMS, for example. But I don't really know if it's worth to do it on modern computers. Nevertheless, it was a good example of arrays, so I added it to this book.

## 17.8 Negative array indices

It's possible to address a space *before* array by supplying negative index, e.g., `array[-1]`.

It's very hard to say, why one should need it, I know probably only one practical application of this technique. C/C++ array elements indices are started at 0, but some PLs has first index at 1 (at least FORTRAN). Programmers may still have this habit, so using this little trick, it's possible to address 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 17.31: 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 $LN1@main:
35     mov     ecx, DWORD PTR _fakearray$[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 $SG2753 ; 'last element %d'
54     call  _printf
55     add   esp, 8
56     ; subtract 4, 3, 2 and 1 from pointer to array[0] in ↵
    ↪ order to find values before array[]
57     lea   ecx, DWORD PTR _array$[ebp]
58     movzx  edx, BYTE PTR [ecx-4]
59     push   edx
60     lea   eax, DWORD PTR _array$[ebp]
61     movzx  ecx, BYTE PTR [eax-3]
62     push   ecx

```



```

63     lea     edx, DWORD PTR _array$[ebp]
64     movzx  eax, BYTE PTR [edx-2]
65     push   eax
66     lea     ecx, DWORD PTR _array$[ebp]
67     movzx  edx, BYTE PTR [ecx-1]
68     push   edx
69     push   OFFSET $SG2754 ; 'array[-1]=%02X, array[-2]=%02X,
    ↪ X, array[-3]=%02X, array[-4]=%02X'
70     call   _printf
71     add    esp, 20
72     xor    eax, eax
73     mov    esp, ebp
74     pop    ebp
75     ret    0
76 _main  ENDP

```

So we have `array[]` of ten elements, filled with 0...9 bytes. Then we have `fakearray[]` pointer which points one byte before `array[]`. `fakearray[1]` pointing exactly to `array[0]`. But we still curious, what is before `array[]`? I added `random_value` before `array[]` and set it to 0x11223344. Non-optimizing compiler allocated variables in the order they were declared, so yes, 32-bit `random_value` is right before `array`.

I run it, and:

```

first element 0
second element 1
last element 9
array[-1]=11, array[-2]=22, array[-3]=33, array[-4]=44

```

Stack fragment I copyasted from OllyDbg stack window (with my comments):

Listing 17.32: Non-optimizing MSVC 2010

```

CPU Stack
Address  Value
001DFBCC /001DFBD3 ; fakearray pointer
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

```

Pointer to the `fakearray[]` (0x001DFBD3) is indeed address of `array[]` in stack (0x001DFBD4), but minus 1 byte.

It's still very hackish and dubious trick, I doubt anyone should use it in production code, but as a demonstration, it fits perfectly here.

## 17.9 Conclusion

Array is a pack of values in memory located adjacently. It's true for any element type, including structures. Access to specific array element is just calculation of its address.

### 17.10 Exercises

#### 17.10.1 Exercise #1

What this code does?

Listing 17.33: MSVC 2010 + /O1

```

_a$ = 8      ; size = 4
_b$ = 12     ; size = 4
_c$ = 16     ; size = 4
?s@@YAXPAN00@Z PROC      ; s, COMDAT
    mov     eax, DWORD PTR _b$[esp-4]
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _c$[esp-4]
    push   esi
    push   edi
    sub    ecx, eax
    sub    edx, eax
    mov    edi, 200      ; 000000c8H
$LL6@s:
    push   100          ; 00000064H
    pop    esi
$LL3@s:
    fld    QWORD PTR [ecx+eax]
    fadd   QWORD PTR [eax]
    fstp   QWORD PTR [edx+eax]
    add    eax, 8
    dec    esi
    jne    SHORT $LL3@s
    dec    edi
    jne    SHORT $LL6@s
    pop    edi
    pop    esi
    ret    0
?s@@YAXPAN00@Z ENDP      ; s

```

(/O1: minimize space).

Listing 17.34: Optimizing Keil 6/2013 (ARM mode)

```

        PUSH    {r4-r12,lr}
        MOV     r9,r2
        MOV     r10,r1
        MOV     r11,r0
        MOV     r5,#0
|L0.20|
        ADD     r0,r5,r5,LSL #3
        ADD     r0,r0,r5,LSL #4
        MOV     r4,#0
        ADD     r8,r10,r0,LSL #5
        ADD     r7,r11,r0,LSL #5
        ADD     r6,r9,r0,LSL #5
|L0.44|
        ADD     r0,r8,r4,LSL #3
        LDM     r0,{r2,r3}
        ADD     r1,r7,r4,LSL #3
        LDM     r1,{r0,r1}
        BL      __aeabi_dadd
        ADD     r2,r6,r4,LSL #3
        ADD     r4,r4,#1
        STM     r2,{r0,r1}
        CMP     r4,#0x64
        BLT     |L0.44|
        ADD     r5,r5,#1
        CMP     r5,#0xc8
        BLT     |L0.20|
        POP     {r4-r12,pc}

```

Listing 17.35: Optimizing Keil 6/2013 (thumb mode)

```

        PUSH    {r0-r2,r4-r7,lr}
        MOVS   r4,#0
        SUB     sp,sp,#8
|L0.6|
        MOVS   r1,#0x19
        MOVS   r0,r4
        LSLS   r1,r1,#5
        MULS   r0,r1,r0
        LDR     r2,[sp,#8]
        LDR     r1,[sp,#0xc]
        ADDS   r2,r0,r2
        STR     r2,[sp,#0]
        LDR     r2,[sp,#0x10]
        MOVS   r5,#0
        ADDS   r7,r0,r2
        ADDS   r0,r0,r1
        STR     r0,[sp,#4]
|L0.32|

```

```

LSLS    r6,r5,#3
ADDS    r0,r0,r6
LDM     r0!,{r2,r3}
LDR     r0,[sp,#0]
ADDS    r1,r0,r6
LDM     r1,{r0,r1}
BL      __aeabi_dadd
ADDS    r2,r7,r6
ADDS    r5,r5,#1
STM     r2!,{r0,r1}
CMP     r5,#0x64
BGE     |L0.62|
LDR     r0,[sp,#4]
B       |L0.32|
|L0.62|
ADDS    r4,r4,#1
CMP     r4,#0xc8
BLT     |L0.6|
ADD     sp,sp,#0x14
POP     {r4-r7,pc}

```

Answer [F.1.10](#).

## 17.10.2 Exercise #2

What this code does?

Listing 17.36: MSVC 2010 + /O1

```

tv315 = -8           ; size = 4
tv291 = -4           ; size = 4
_a$ = 8              ; size = 4
_b$ = 12             ; size = 4
_c$ = 16             ; size = 4
?m@@YAXPAN00@Z PROC ; m, COMDAT
    push    ebp
    mov     ebp, esp
    push    ecx
    push    ecx
    mov     edx, DWORD PTR _a$[ebp]
    push    ebx
    mov     ebx, DWORD PTR _c$[ebp]
    push    esi
    mov     esi, DWORD PTR _b$[ebp]
    sub     edx, esi
    push    edi
    sub     esi, ebx
    mov     DWORD PTR tv315[ebp], 100 ; 00000064H

```

```

$LL9@m:
    mov     eax, ebx
    mov     DWORD PTR tv291[ebp], 300 ; 0000012cH
$LL6@m:
    fldz
    lea     ecx, DWORD PTR [esi+eax]
    fstp   QWORD PTR [eax]
    mov     edi, 200 ; 000000c8H
$LL3@m:
    dec     edi
    fld     QWORD PTR [ecx+edx]
    fmul   QWORD PTR [ecx]
    fadd   QWORD PTR [eax]
    fstp   QWORD PTR [eax]
    jne     HORT $LL3@m
    add     eax, 8
    dec     DWORD PTR tv291[ebp]
    jne     SHORT $LL6@m
    add     ebx, 800 ; 00000320H
    dec     DWORD PTR tv315[ebp]
    jne     SHORT $LL9@m
    pop     edi
    pop     esi
    pop     ebx
    leave
    ret     0
?m@YAXPAN00@Z ENDP ; m

```

(/O1: minimize space).

Listing 17.37: Optimizing Keil 6/2013 (ARM mode)

```

PUSH     {r0-r2,r4-r11,lr}
SUB      sp,sp,#8
MOV      r5,#0
|L0.12|
LDR      r1,[sp,#0xc]
ADD      r0,r5,r5,LSL #3
ADD      r0,r0,r5,LSL #4
ADD      r1,r1,r0,LSL #5
STR      r1,[sp,#0]
LDR      r1,[sp,#8]
MOV      r4,#0
ADD      r11,r1,r0,LSL #5
LDR      r1,[sp,#0x10]
ADD      r10,r1,r0,LSL #5
|L0.52|
MOV      r0,#0
MOV      r1,r0

```

```

ADD    r7,r10,r4,LSL #3
STM    r7,{r0,r1}
MOV    r6,r0
LDR    r0,[sp,#0]
ADD    r8,r11,r4,LSL #3
ADD    r9,r0,r4,LSL #3
|L0.84|
LDM    r9,{r2,r3}
LDM    r8,{r0,r1}
BL     __aeabi_dmul
LDM    r7,{r2,r3}
BL     __aeabi_dadd
ADD    r6,r6,#1
STM    r7,{r0,r1}
CMP    r6,#0xc8
BLT   |L0.84|
ADD    r4,r4,#1
CMP    r4,#0x12c
BLT   |L0.52|
ADD    r5,r5,#1
CMP    r5,#0x64
BLT   |L0.12|
ADD    sp,sp,#0x14
POP    {r4-r11,pc}

```

Listing 17.38: Optimizing Keil 6/2013 (thumb mode)

```

PUSH   {r0-r2,r4-r7,lr}
MOVS   r0,#0
SUB    sp,sp,#0x10
STR    r0,[sp,#0]
|L0.8|
MOVS   r1,#0x19
LSLS   r1,r1,#5
MULS   r0,r1,r0
LDR    r2,[sp,#0x10]
LDR    r1,[sp,#0x14]
ADDS   r2,r0,r2
STR    r2,[sp,#4]
LDR    r2,[sp,#0x18]
MOVS   r5,#0
ADDS   r7,r0,r2
ADDS   r0,r0,r1
STR    r0,[sp,#8]
|L0.32|
LSLS   r4,r5,#3
MOVS   r0,#0
ADDS   r2,r7,r4

```

```

        STR     r0,[r2,#0]
        MOVS   r6,r0
        STR     r0,[r2,#4]
|L0.44|
        LDR     r0,[sp,#8]
        ADDS   r0,r0,r4
        LDM    r0!,{r2,r3}
        LDR     r0,[sp,#4]
        ADDS   r1,r0,r4
        LDM    r1,{r0,r1}
        BL     __aeabi_dmul
        ADDS   r3,r7,r4
        LDM    r3,{r2,r3}
        BL     __aeabi_dadd
        ADDS   r2,r7,r4
        ADDS   r6,r6,#1
        STM    r2!,{r0,r1}
        CMP    r6,#0xc8
        BLT    |L0.44|
        MOVS   r0,#0xff
        ADDS   r5,r5,#1
        ADDS   r0,r0,#0x2d
        CMP    r5,r0
        BLT    |L0.32|
        LDR     r0,[sp,#0]
        ADDS   r0,r0,#1
        CMP    r0,#0x64
        STR     r0,[sp,#0]
        BLT    |L0.8|
        ADD    sp,sp,#0x1c
        POP    {r4-r7,pc}

```

Answer [F.1.10](#).

### 17.10.3 Exercise #3

What this code does?

Try to determine array dimensions, at least partially.

Listing 17.39: Optimizing MSVC 2010

```

_array$ = 8
_x$ = 12
_y$ = 16
_f      PROC
        mov     eax, DWORD PTR _x$[esp-4]
        mov     edx, DWORD PTR _y$[esp-4]
        mov     ecx, eax

```

```

    shl     ecx, 4
    sub     ecx, eax
    lea    eax, DWORD PTR [edx+ecx*8]
    mov    ecx, DWORD PTR _array$[esp-4]
    fld    QWORD PTR [ecx+eax*8]
    ret    0
_f      ENDP

```

Listing 17.40: Non-optimizing Keil 6/2013 (ARM mode)

```

f PROC
    RSB    r1,r1,r1,LSL #4
    ADD    r0,r0,r1,LSL #6
    ADD    r1,r0,r2,LSL #3
    LDM    r1,{r0,r1}
    BX    lr
    ENDP

```

Listing 17.41: Non-optimizing Keil 6/2013 (thumb mode)

```

f PROC
    MOVS   r3,#0xf
    LSLS   r3,r3,#6
    MULS   r1,r3,r1
    ADDS   r0,r1,r0
    LSLS   r1,r2,#3
    ADDS   r1,r0,r1
    LDM    r1,{r0,r1}
    BX    lr
    ENDP

```

Answer [F.1.10](#)

### 17.10.4 Exercise #4

What this code does?

Try to determine array dimensions, at least partially.

Listing 17.42: Optimizing MSVC 2010

```

_array$ = 8
_x$ = 12
_y$ = 16
_z$ = 20
_f      PROC
    mov    eax, DWORD PTR _x$[esp-4]
    mov    edx, DWORD PTR _y$[esp-4]
    mov    ecx, eax

```



```

    shl     ecx, 4
    sub     ecx, eax
    lea    eax, DWORD PTR [edx+ecx*4]
    mov    ecx, DWORD PTR _array$[esp-4]
    lea    eax, DWORD PTR [eax+ecx*4]
    shl    eax, 4
    add    eax, DWORD PTR _z$[esp-4]
    mov    eax, DWORD PTR [ecx+eax*4]
    ret    0
_f      ENDP

```

Listing 17.43: Non-optimizing Keil 6/2013 (ARM mode)

```

f PROC
    RSB     r1,r1,r1,LSL #4
    ADD     r1,r1,r1,LSL #2
    ADD     r0,r0,r1,LSL #8
    ADD     r1,r2,r2,LSL #2
    ADD     r0,r0,r1,LSL #6
    LDR     r0,[r0,r3,LSL #2]
    BX     lr
    ENDP

```

Listing 17.44: Non-optimizing Keil 6/2013 (thumb mode)

```

f PROC
    PUSH    {r4,lr}
    MOVS    r4,#0x4b
    LSLS    r4,r4,#8
    MULS    r1,r4,r1
    ADDS    r0,r1,r0
    MOVS    r1,#0xff
    ADDS    r1,r1,#0x41
    MULS    r2,r1,r2
    ADDS    r0,r0,r2
    LSLS    r1,r3,#2
    LDR     r0,[r0,r1]
    POP     {r4,pc}
    ENDP

```

Answer [F.1.10](#)

### 17.10.5 Exercise #5

What this code does?

Listing 17.45: Optimizing MSVC 2012 /GS-

```

COMM      _tbl:DWORD:064H

tv759 = -4      ; size = 4
_main PROC
    push    ecx
    push    ebx
    push    ebp
    push    esi
    xor     edx, edx
    push    edi
    xor     esi, esi
    xor     edi, edi
    xor     ebx, ebx
    xor     ebp, ebp
    mov     DWORD PTR tv759[esp+20], edx
    mov     eax, OFFSET _tbl+4
    npad   8 ; align next label
$LL6@main:
    lea     ecx, DWORD PTR [edx+edx]
    mov     DWORD PTR [eax+4], ecx
    mov     ecx, DWORD PTR tv759[esp+20], 3
    add     DWORD PTR [eax+8], ecx
    lea     ecx, DWORD PTR [edx*4]
    mov     DWORD PTR [eax+12], ecx
    lea     ecx, DWORD PTR [edx*8]
    mov     DWORD PTR [eax], edx
    mov     DWORD PTR [eax+16], ebp
    mov     DWORD PTR [eax+20], ebx
    mov     DWORD PTR [eax+24], edi
    mov     DWORD PTR [eax+32], esi
    mov     DWORD PTR [eax-4], 0
    mov     DWORD PTR [eax+28], ecx
    add     eax, 40
    inc     edx
    add     ebp, 5
    add     ebx, 6
    add     edi, 7
    add     esi, 9
    cmp     eax, OFFSET _tbl+404
    jl     SHORT $LL6@main
    pop     edi
    pop     esi
    pop     ebp
    xor     eax, eax
    pop     ebx
    pop     ecx

```

```

ret      0
_main   ENDP

```

Listing 17.46: Non-optimizing Keil 6/2013 (ARM mode)

```

main PROC
    LDR    r12,|L0.60|
    MOV    r1,#0
|L0.8|
    ADD    r2,r1,r1,LSL #2
    MOV    r0,#0
    ADD    r2,r12,r2,LSL #3
|L0.20|
    MUL    r3,r1,r0
    STR    r3,[r2,r0,LSL #2]
    ADD    r0,r0,#1
    CMP    r0,#0xa
    BLT    |L0.20|
    ADD    r1,r1,#1
    CMP    r1,#0xa
    MOVGE  r0,#0
    BLT    |L0.8|
    BX     lr
    ENDP

|L0.60|
    DCD    ||.bss||

    AREA  ||.bss||, DATA, NOINIT, ALIGN=2

tbl
    %      400

```

Listing 17.47: Non-optimizing Keil 6/2013 (thumb mode)

```

main PROC
    PUSH   {r4,r5,lr}
    LDR    r4,|L0.40|
    MOVS   r1,#0
|L0.6|
    MOVS   r2,#0x28
    MULS   r2,r1,r2
    MOVS   r0,#0
    ADDS   r3,r2,r4
|L0.14|
    MOVS   r2,r1
    MULS   r2,r0,r2
    LSLS   r5,r0,#2

```

```
    ADDS    r0,r0,#1
    CMP     r0,#0xa
    STR     r2,[r3,r5]
    BLT     |L0.14|
    ADDS    r1,r1,#1
    CMP     r1,#0xa
    BLT     |L0.6|
    MOVS    r0,#0
    POP     {r4,r5,pc}
    ENDP

    DCW     0x0000
|L0.40|
    DCD     ||.bss||

    AREA   ||.bss||, DATA, NOINIT, ALIGN=2

tbl
    %      400
```

Answer [F.1.10](#)

# Chapter 18

## Working with specific bits

A lot of functions defining input flags in arguments using bit fields. Of course, it could be substituted by *bool*-typed variables set, but it is not frugally.

### 18.1 Specific bit checking

#### 18.1.1 x86

Win32 API example:

```
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, ↵
↳ FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL ↵
↳ , NULL);
```

We got (MSVC 2010):

Listing 18.1: 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 into WinNT.h:

Listing 18.2: 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 is value is used as the second argument for `CreateFile()`<sup>1</sup> function.

How `CreateFile()` will check flags?

Let's take a look into `KERNEL32.DLL` in Windows XP SP3 x86 and we'll find this fragment of code in the function `CreateFileW`:

Listing 18.3: `KERNEL32.DLL` (Windows XP SP3 x86)

```
.text:7C83D429          test     byte ptr [ebp+var_4
    ↪ dwDesiredAccess+3], 40h
.text:7C83D42D          mov     [ebp+var_8], 1
.text:7C83D434          jz     short loc_7C83D417
.text:7C83D436          jmp    loc_7C810817
```

Here we see `TEST` instruction, it takes, however, not the whole second argument, but only most significant byte (`ebp+dwDesiredAccess+3`) and checks it for `0x40` flag (meaning `GENERIC_WRITE` flag here)

`TEST` is merely the same instruction as `AND`, but without result saving (recall the fact `CMP` instruction is merely the same as `SUB`, but without result saving (6.6.1)).

This fragment of code logic is as follows:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

If `AND` instruction leaving this bit, `ZF` flag is to be cleared and `JZ` conditional jump will not be triggered. Conditional jump will be triggered only if `0x40000000` bit is absent in the `dwDesiredAccess` variable – then `AND` result will be 0, `ZF` flag will be set and 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;
```

<sup>1</sup>[MSDN: CreateFile function](#)

```

        handle=open ("file", O_RDWR | O_CREAT);
};

```

We got:

Listing 18.4: GCC 4.4.1

```

main                public 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

```

Let's take a look into `open()` function in the `libc.so.6` library, but there is only `syscall`:

Listing 18.5: `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, `open()` bit fields apparently checked somewhere in Linux kernel.

Of course, it is easy to download both Glibc and Linux kernel source code, but we are interesting to understand the matter without it.

So, as of Linux 2.6, when `sys_open` syscall is called, control eventually passed into `do_sys_open` kernel function. From there –to the `do_filp_open()` function (this function located in kernel source tree in the file `fs/namei.c`).

N.B. Aside from common passing arguments via stack, there is also a method of passing some of them via registers. This is also called `fastcall` (49.3). This works faster since CPU not needed to access a stack in memory to read argument values.

GCC has option *regparm*<sup>2</sup>, and it is possible to set a number of arguments which might be passed via registers.

Linux 2.6 kernel compiled with *-mregparm=3* option<sup>3 4</sup>.

What it means to us, the first 3 arguments will be passed via EAX, EDX and ECX registers, the rest ones via stack. Of course, if arguments number is less than 3, only part of registers are to be used.

So, let's download Linux Kernel 2.6.31, compile it in Ubuntu: make *vmlinux*, open it in *IDA*, find the *do\_filp\_open()* function. At the beginning, we will see (comments are mine):

Listing 18.6: *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
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5th arg)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1th arg)
                mov     [ebp+var_7C], edx ; pathname (2th arg)
                mov     [ebp+var_78], ecx ; open_flag (3th arg)
                jnz     short loc_C01EF684
                mov     ebx, ecx          ; ebx <- open_flag
```

GCC saves first 3 arguments values in local stack. Otherwise, if compiler would not touch these registers, it would be too tight environment for compiler's *register allocator*.

Let's find this fragment of code:

Listing 18.7: *do\_filp\_open()* (linux kernel 2.6.31)

```
loc_C01EF6B4:                                     ; CODE XREF: ↗
↳ do_filp_open+4F
                test    bl, 40h                ; O_CREAT
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
```

<sup>2</sup><http://ohse.de/uwe/articles/gcc-attributes.html#func-regparm>

<sup>3</sup>[http://kernelnewbies.org/Linux\\_2\\_6\\_20#head-042c62f290834eb1fe0a1942bbf5bb9a4a](http://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4a)

<sup>4</sup>See also `arch\x86\include\asm\calling.h` file in kernel tree



```

test    ebx, 10000h
jz      short loc_C01EF6D3
or      edi, 2

```

0x40 –is what `O_CREAT` macro equals to. `open_flag` checked for 0x40 bit presence, and if this bit is 1, next `JNZ` instruction is triggered.

## 18.1.2 ARM

`O_CREAT` bit is checked differently in Linux kernel 3.8.0.

Listing 18.8: 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);
    }
}

```

```
...
}
```

Here is how kernel compiled for ARM mode looks like in [IDA](#):

Listing 18.9: do\_last() (vmlinux)

```
...
.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, #0↵
    ↳ x200000
.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 analogical to a TEST instruction in x86.

We can “spot” visually this code fragment by the fact the `lookup_fast()` will be executed in one case and the `complete_walk()` in another case. This

is corresponding to the `do_last()` function source code.

`O_CREAT` macro is equals to `0x40` here too.

## 18.2 Specific bit setting/clearing

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);
};
```

### 18.2.1 x86

#### Non-optimizing MSVC

We got (MSVC 2010):

Listing 18.10: 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                ; fffffdfh
mov     DWORD PTR _rt$[ebp], edx
mov     eax, DWORD PTR _rt$[ebp]
mov     esp, ebp
pop     ebp
ret     0
_f     ENDP
    
```

OR instruction adds one more bit to value while ignoring the rest ones.

AND resetting one bit. It can be said, AND just copies all bits except one. Indeed, in the second AND operand only those bits are set, which are needed to be saved, except one bit we would not like to copy (which is 0 in bitmask). It is easier way to memorize the logic.

### OllyDbg

Let's try this example in OllyDbg. First, let's see binary form of constants we use:

0x200 (00000000000000000000000000000000**1**000000000) (i.e., 10th bit (counting from 1st)).

Inverted 0x200 is 0xFFFFFFF (111111111111111111111111**0**111111111).

0x4000 (00000000000000000000000000000000**1**0000000000000000)

Input value is: 0x12340678 (1001000110100000001100111100). We see how it's loaded: fig.18.1.

OR executed: fig.18.2. 15th bit is set: 0x1234**4**678 (10010001101000**1**0001100111100)

Value is reloaded again (because it's not optimizing compiler's mode): fig.18.3.

AND executed: fig.18.3. 10th bit is cleared (or, in other words, all bits are leaved instead of 10th) and final value now is 0x1234**4**78 (100100011010001000**1**0001111000)

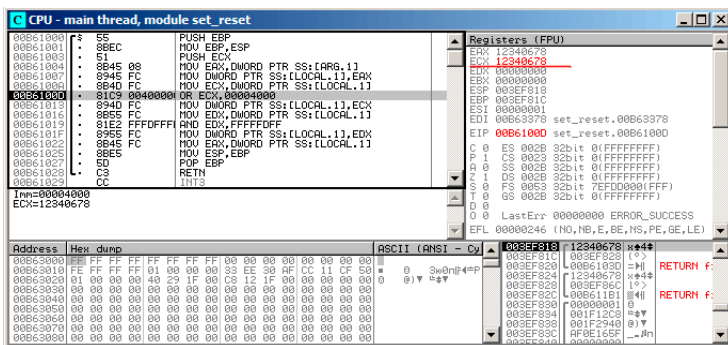


Figure 18.1: OllyDbg: value is loaded into ECX

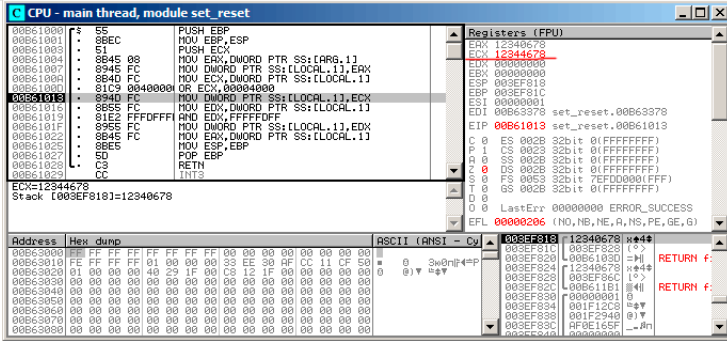


Figure 18.2: OllyDbg: OR executed

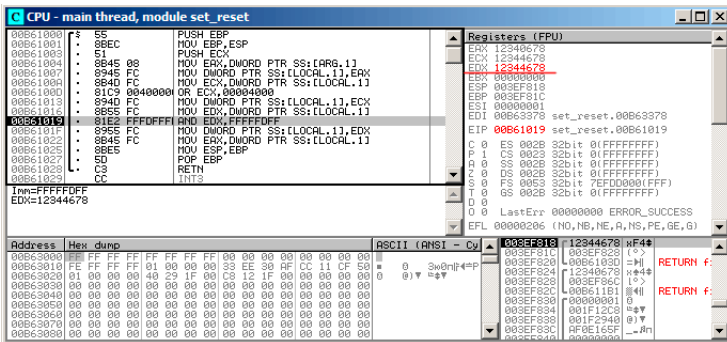


Figure 18.3: OllyDbg: value was reloaded into EDX

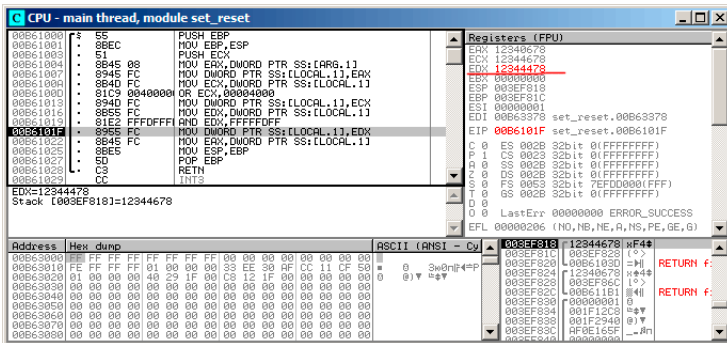


Figure 18.4: OllyDbg: AND executed

## Optimizing MSVC

If we compile it in MSVC with optimization turned on (/Ox), the code is even shorter:

Listing 18.11: Optimizing MSVC

```

_a$ = 8                ; size = 4
_f   PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513    ; ffffffffH
    or      eax, 16384   ; 00004000H
    ret     0
_f   ENDP

```

## Non-optimizing GCC

Let's try GCC 4.4.1 without optimization:

Listing 18.12: Non-optimizing GCC

```

f           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], 0FFFFFFDFh
           mov      eax, [ebp+var_4]
           leave
           retn
f           endp

```

There is a redundant code present, however, it is shorter than MSVC version without optimization.

Now let's try GCC with optimization turned on -O3:

## Optimizing GCC

Listing 18.13: 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
f          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 8th to 15th bits inclusive.

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

N.B. 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, accumulator was named EAX, but for the sake of compatibility, its *older parts* may be still accessed as AX/AH/AL registers.

Since all x86 CPUs are 16-bit 8086 CPU successors, these *older* 16-bit opcodes are shorter than newer 32-bit opcodes. That's why `or ah, 40h` instruction occupying 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 if register in first operand is not EAX).

### Optimizing GCC and regparm

It would be even shorter if to turn on `-O3` optimization flag and also set `reg-  
parm=3`.

Listing 18.14: Optimizing GCC

```

f          public f
          proc near
          push    ebp
          or      ah, 40h
          mov     ebp, esp
          and     ah, 0FDh
          pop     ebp

```

f	retn endp
---	--------------

Indeed –first argument is already loaded into EAX, so it is possible to work with it in-place. It is worth noting that both function prologue (``push ebp / mov ebp, esp'`) and epilogue (``pop ebp'`) can easily be omitted here, but GCC probably is not good enough for such code size optimizations. However, such short functions are better to be *inlined functions* (29).

## 18.2.2 ARM + Optimizing Keil 6/2013 (ARM mode)

Listing 18.15: 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 clearing specific bits. This is just like AND instruction, but with inverted operand. I.e., it's analogous to NOT+AND instruction pair.

ORR is “logical or”, analogical to OR in x86.

So far, so easy.

## 18.2.3 ARM + Optimizing Keil 6/2013 (thumb mode)

Listing 18.16: 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

Apparently, Keil concludes the code in thumb mode, making 0x200 from 0x4000, will be more compact than code, writing 0x200 to arbitrary register.

So that is why, with the help of ASRS (arithmetic shift right), this value is calculating as  $0x4000 \gg 5$ .

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

Listing 18.17: 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 was generated by LLVM, in source code form, in fact, could be looks like:

```
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

And it does exactly the same we need. But why 0x4200? Perhaps, that is the LLVM optimizer's artifact<sup>5</sup>. Probably, compiler's optimizer error, but generated code works correctly anyway.

More about compiler's anomalies, read here (75).

Optimizing Xcode 4.6.3 (LLVM) for thumb mode generates likewise code.

## 18.2.5 ARM: more about BIC instruction

If to rework example slightly:

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

Then optimizing Keil 5.03 in ARM mode will do:

```
f PROC
    BIC    r0,r0,#0x1000
    BIC    r0,r0,#0x234
    BX     lr
    ENDP
```

There are two BIC instructions, i.e., 0x1234 bits are cleared in two passes. This is because it's not possible to encode 0x1234 value in BIC instruction, but it's possible to encode 0x1000 or 0x234.

## 18.2.6 ARM64: Optimizing GCC (Linaro) 4.9

Optimizing GCC compiling for ARM64 can use AND instruction instead of BIC:

Listing 18.18: Optimizing GCC (Linaro) 4.9

```
f:
    and    w0, w0, -513    ; 0xFFFFFFFFFFFFDFF
    orr    w0, w0, 16384   ; 0x4000
    ret
```

<sup>5</sup>It was LLVM build 2410.2.00 bundled with Apple Xcode 4.6.3

## 18.2.7 ARM64: Non-optimizing GCC (Linaro) 4.9

Non-optimizing GCC generate more redundant code, but works just like optimized:

Listing 18.19: 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     ; 0xFFFFFFFFFFFFFFDF
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    add    sp, sp, 32
    ret
```

## 18.3 Shifts

Bit shifts in C/C++ are implemented using `<<` and `>>` operators.

x86 ISA has SHL (SHift Left) and SHR (SHift Right) instructions for this.

Shift instructions are often used in division and multiplications by power of two numbers:  $2^n$  (e.g., 1, 2, 4, 8, etc): [15.1.2](#), [15.2.1](#).

## 18.4 Counting bits set to 1

Here is a simple example of function, calculating number of bits set in input value.

This operation is also called “population count”<sup>6</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++)
```

<sup>6</sup>modern x86 CPUs (supporting SSE4) even have POPCNT instruction for it

```
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};

int main()
{
    f(0x12345678); // test
};
```

In this loop, iteration count value  $i$  counting from 0 to 31,  $1 \ll i$  statement will be 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 will consequently produce all possible bit positions in 32-bit number. 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

These constant numbers (bit masks) are very often appears in code and practicing reverse engineer should quickly spot them. You probably shouldn't memorize decimal numbers, but hexadecimal ones are very easy to memorize.

These constants are very often used for mapping flags to specific bits. For example, here is excerpt from `ssl_private.h` file from Apache 2.4.6 source code:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE (0)
```

```

#define SSL_OPT_RESET          (1<<0)
#define SSL_OPT_STDENVVARS    (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 back to our example.

IS\_SET macro is checking bit presence in the *a*.

The IS\_SET macro is in fact logical and operation (*AND*) and it returns 0 if specific bit is absent there, or bit mask, if the bit is present. *if()* operator triggered in C/C++ if expression in it is not a zero, it might be even 123456, that is why it always working correctly.

## 18.4.1 x86

### MSVC

Let's compile (MSVC 2010):

Listing 18.20: 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 1
    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?
    ↵ instructions                ; then skip next ↴

```

```

    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
    pop     ebp
    ret     0
_f        ENDP

```

## OllyDbg

Let's load this example into OllyDbg. Let's input value be 0x12345678.

For  $i = 1$ , we see how  $i$  is loaded into ECX: fig.18.5. EDX is 1. SHL is to be executed now.

SHL was executed: fig.18.6. EDX contain  $1 \ll 1$  (or 2). This is a bit mask.

AND sets ZF to 1, which is meaning that input value (0x12345678) ANDed with 2 resulting 0: fig.18.7. So, there are no corresponding bit in input value. The piece of code which **increments** counter will not be executed: JZ instruction will *bypass* it.

Now I traced some time further and  $i$  is now 4. SHL is to be executed now: fig.18.8.

EDX =  $1 \ll 4$  (or 0x10 or 16): fig.18.9. This is another bit mask.

AND is executed: fig.18.10. ZF is 0 because this bit is present in input value. Indeed,  $0x12345678 \& 0x10 = 0x10$ . This bit counts: jump will not trigger and bits counter will be **incremented** now.

The function returns 13. This is total bits set in 0x12345678 value.

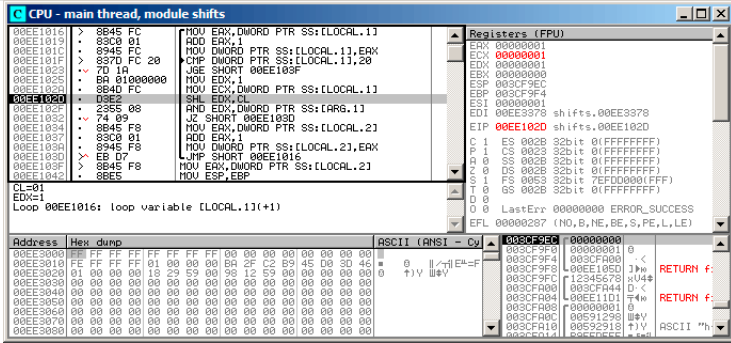


Figure 18.5: OllyDbg:  $i = 1$ ,  $i$  is loaded into ECX

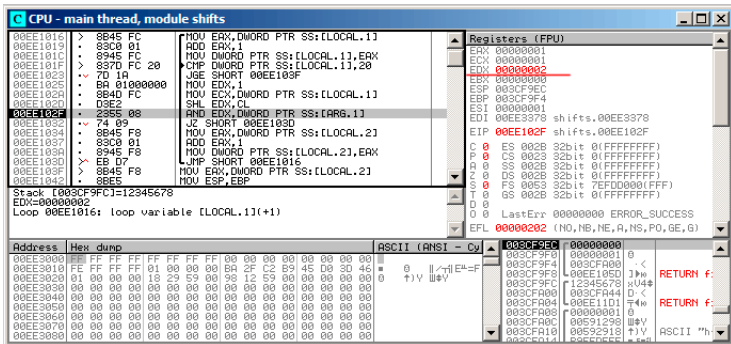


Figure 18.6: OllyDbg:  $i = 1$ ,  $EDX = 1 \ll 1 = 2$

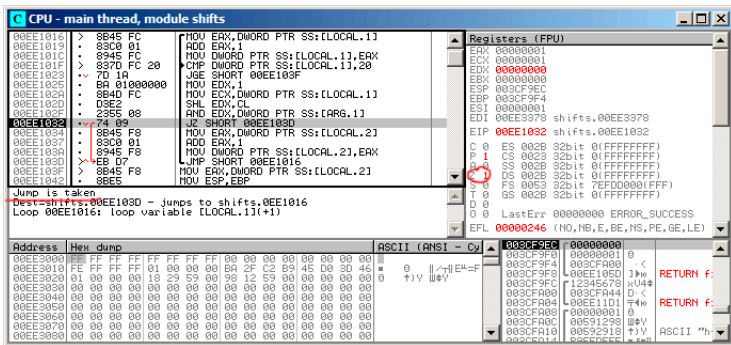


Figure 18.7: OllyDbg:  $i = 1$ , are there that bit in the input value? No. (ZF=1)

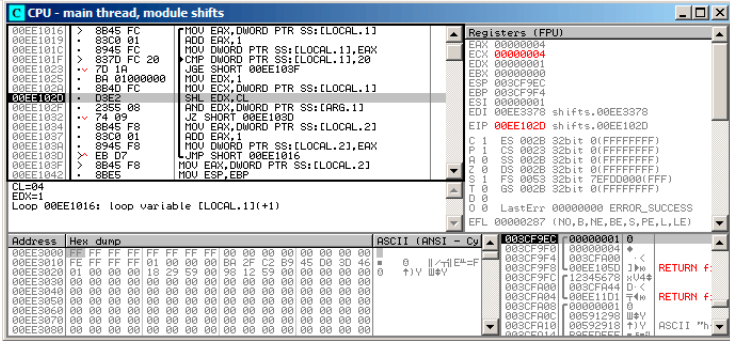


Figure 18.8: OllyDbg:  $i = 4$ ,  $i$  is loaded into ECX

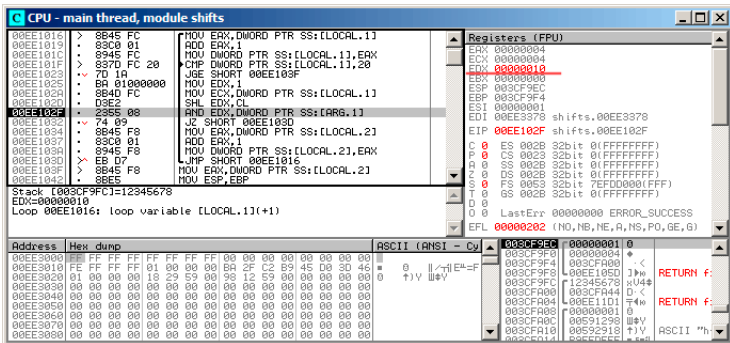


Figure 18.9: OllyDbg:  $i = 4$ ,  $EDX = 1 \ll 4 = 0x10$

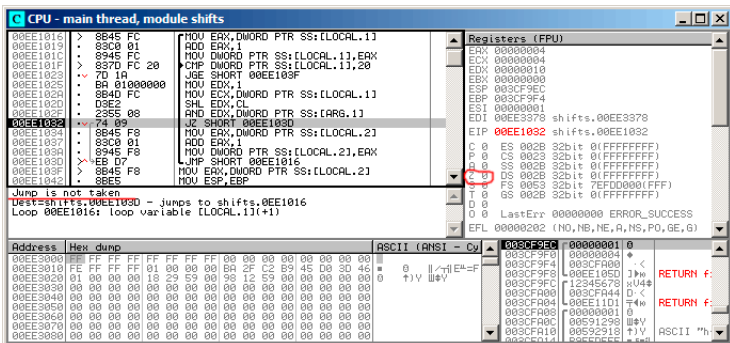


Figure 18.10: OllyDbg:  $i = 4$ , are there that bit in the input value? Yes. (ZF = 0)



**GCC**

Let's compile it in GCC 4.4.1:

Listing 18.21: GCC 4.4.1

```

f                public 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

loc_80483EF:
                cmp     [ebp+i], 1Fh
                jle    short loc_80483D0
                mov     eax, [ebp+rt]
                add     esp, 10h
                pop     ebx
                pop     ebp
                retn

f                endp

```

**18.4.2 x64**

I changed 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 18.22: 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's so.
    je     .L3
    add     DWORD PTR [rbp-12], 1 ; rt++
.L3:

```

```

.L2:    add     QWORD PTR [rbp-8], 1    ; i++
        cmp     QWORD PTR [rbp-8], 63 ; i<63?
        jbe     .L4                    ; jump to the loop body
        ↪ begin, if so
        mov     eax, DWORD PTR [rbp-12] ; return rt
        pop     rbp
        ret

```

## Optimizing GCC 4.8.2

Listing 18.23: Optimizing GCC 4.8.2

```

1  f:
2      xor     eax, eax                ; rt variable will be here
3      xor     ecx, ecx                ; i variable will be here
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', which will be written into
   ↪ rt variable, if the last bit is 1
8      shr     rsi, cl                 ; RSI=RSI>>CL
9      and     esi, 1                  ; ESI=ESI&1
10 ; the last bit is 1? If so, write `new version of rt' into EAX
11      cmovne eax, edx
12      add     rcx, 1                  ; RCX++
13      cmp     rcx, 64
14      jne     .L3
15      rep ret                          ; AKA fatret

```

This code is more terse, but also, has some quirk. While all examples we saw so far, incrementing “rt” value after comparing specific bit with one, the code here incrementing “rt” before (line 6), writing new value into EDX register. Then, if the last bit was 1, CMOVNE<sup>7</sup> instruction (which is synonymous to CMOVNZ<sup>8</sup>) *commits* new value of “rt” by moving EDX (“proposed rt value”) into EAX (“current rt” to be returned at the end). Hence, incrementing is done 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 loop) instead of two jumps (skipping “rt” value increment and at the end of loop). And that might work faster on the modern CPUs with branch predictors: [38.1](#).

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

<sup>7</sup>Conditional MOVE if Not Equal

<sup>8</sup>Conditional MOVE if Not Zero

AMD to be placed at the end of function, if RET goes right after conditional jump: [AMD13b, p.15]<sup>9</sup>.

## Optimizing MSVC 2010

Listing 18.24: 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
$LL4@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++
$LN3@f:
    rol    rdx, 1 ; RDX=RDX<<1
    dec    r8     ; R8--
    jne    SHORT $LL4@f
    fatret 0
f    ENDP

```

Here is the ROL instruction is used instead of SHL, which is in fact “rotate left” instead of “shift left”, but here, in this example, it will work just as SHL.

Read more about rotate instruction here: [A.6.3](#).

R8 here is counting from 64 to 0. It's just like inverted *i* variable.

Here is a table of some registers during execution:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

At the end we see FATRET instruction, which was described here: [18.4.2](#).

<sup>9</sup>More information on it: <http://repzret.org/p/repzret/>

**Optimizing MSVC 2012**

Listing 18.25: 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 loop count is now 32 instead of 64. To be honest, I don't know why. Some optimization trick? Maybe it's better for loop body to be slightly longer? Anyway, I add the code here intentionally to show that sometimes, compiler output may be really weird and illogical, but perfectly working, of course.

**18.4.3 ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)**

Listing 18.26: 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, R0++
    CMP                R3, #32
    BNE                loc_2E54
    BX                 LR

```

TST is the same things as TEST in x86.

As I mentioned before ([15.3.2](#)), 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>10</sup>.

These modifiers define, how to shift second operand and by how many bits. Thus ``TST R1, R2,LSL R3`` instruction works here as  $R1 \wedge (R2 \ll R3)$ .

### 18.4.4 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 single TST, because, in thumb mode, it is not possible to define LSL modifier right 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

```

### 18.4.5 ARM64 + Optimizing GCC 4.9

I took 64-bit example I already used: [18.4.2](#).

<sup>10</sup>These instructions are also called "data processing instructions"

## Listing 18.27: 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)
    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 generated for x64: [18.23](#).

CSEL instruction is “Conditional SElect”, it just choose one variable of two depending on flags set by TST and copy value into W2 register, which holds “rt” variable.

### 18.4.6 ARM64 + Non-optimizing GCC 4.9

Again, I use 64-bit example I already used: [18.4.2](#).

The code is more verbose, as usual.

## Listing 18.28: 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<<1
    ldr    x0, [sp,8]
; X0 = a
    and    x0, x1, x0
```

```

; X0 = X1&X0 = (1<<i) & a
; X0 is 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

```

## 18.5 CRC32 calculation example

This is very popular table-based CRC32 hash calculation technique<sup>11</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, 0
    ↪ x706af48f,
    0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0
    ↪ x97d2d988,
    0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0
    ↪ x6ab020f2,

```

<sup>11</sup>Source code was taken here: <http://burtleburtle.net/bob/c/crc.c>



```

0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0↵
  ↵ x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0↵
  ↵ x63066cd9,
0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0↵
  ↵ xa2677172,
0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0↵
  ↵ x42b2986c,
0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0↵
  ↵ xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfdf06116, 0x21b4f4b5, 0↵
  ↵ x56b3c423,
0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0↵
  ↵ xb10be924,
0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0↵
  ↵ x01db7106,
0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0↵
  ↵ xe8b8d433,
0x7807c9a2, 0xf00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0↵
  ↵ x086d3d2d,
0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0↵
  ↵ xf262004e,
0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0↵
  ↵ x12b7e950,
0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0↵
  ↵ xfbfd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0↵
  ↵ x3dd895d7,
0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0↵
  ↵ xda60b8d0,
0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0↵
  ↵ x270241aa,
0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0↵
  ↵ xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0↵
  ↵ x2eb40d81,
0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0↵
  ↵ x74b1d29a,
0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0↵
  ↵ x94643b84,
0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0↵
  ↵ x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0↵
  ↵ x806567cb,
0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0↵
  ↵ x67dd4acc,
0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0↵

```

```

    ↪ xa1d1937e,
    0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0↪
    ↪ x48b2364b,
    0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0↪
    ↪ xa867df55,
    0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0↪
    ↪ x5268e236,
    0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0↪
    ↪ xb2bd0b28,
    0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0↪
    ↪ x5bdeae1d,
    0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0↪
    ↪ xeb0e363f,
    0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0↪
    ↪ x0cb61b38,
    0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0↪
    ↪ xf1d4e242,
    0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0↪
    ↪ x18b74777,
    0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0↪
    ↪ xf862ae69,
    0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0↪
    ↪ x3903b3c2,
    0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0↪
    ↪ xd9d65adc,
    0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdeb9ec5, 0x47b2cf7f, 0↪
    ↪ x30b5ffe9,
    0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0↪
    ↪ xcdd70693,
    0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0↪
    ↪ x2a6f2b94,
    0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

```

```

/* how to derive the values in crctab[] from polynomial 0↪
    ↪ xeddb88320 */

```

```

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);
    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 -O 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 interesting in the `crc()` function only. By the way, pay attention to two loop initializers in the `for()` statement: `hash=len, i=0`. C/C++ standard allows this, of course. Emitted code will contain two operations in loop initialization part instead of usual one.

Let's compile it in MSVC with optimization (`/Ox`). For the sake of brevity, only `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
$LL3@crc:

```

```

; work with bytes using only 32-bit registers. byte from ↗
  ↘ address key+i we store into EDI

  movzx  edi, BYTE PTR [ecx+esi]
  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 it's 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:

```

crc                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 doesn't changing 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
        retn

crc
endp
\

```

GCC aligned loop start on a 8-byte boundary by adding NOP and `lea esi, [esi+0]` (that is the *idle operation* too). Read more about it in `npad` section (72).

## 18.6 Network address calculation example

As we know, TCP/IP address (IPv4) consists of four numbers in 0...255 range, i.e., four bytes. Four bytes can be fitted in 32-bit variable easily, so, IPv4 host address, network mask or network address can all be 32-bit integers.

From a user's point of view, network mask is defined in four numbers format like 255.255.255.0 or so, but network engineers use more compact notation ([CIDR](#)<sup>12</sup>), like /8, /16 or like that. This notation just defines number of bits mask has, starting at [MSB](#)<sup>13</sup>.

<sup>12</sup>Classless Inter-Domain Routing

<sup>13</sup>Most significant bit/byte

Mask	Hosts	Usable	Netmask	Hex mask	
/30	4	2	255.255.255.252	fffffffc	
/29	8	6	255.255.255.248	ffffff8	
/28	16	14	255.255.255.240	ffffff0	
/27	32	30	255.255.255.224	fffffe0	
/26	64	62	255.255.255.192	fffffc0	
/24	256	254	255.255.255.0	fffff00	class C network
/23	512	510	255.255.254.0	ffffe00	
/22	1024	1022	255.255.252.0	ffffc00	
/21	2048	2046	255.255.248.0	ffff800	
/20	4096	4094	255.255.240.0	ffff000	
/19	8192	8190	255.255.224.0	fffe000	
/18	16384	16382	255.255.192.0	fffc000	
/17	32768	32766	255.255.128.0	fff8000	
/16	65536	65534	255.255.0.0	fff0000	class B network
/8	16777216	16777214	255.0.0.0	ff00000	class A network

Here is a small example, which calculates network address by applying network mask to 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)
{
    uint32_t netmask=form_netmask(netmask_bits);
    uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
    uint32_t netw_adr;

    printf ("netmask=");
    print_as_IP (netmask);

    netw_adr=ip&netmask;

    printf ("network address=");
    print_as_IP (netw_adr);
};

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
};

```

### 18.6.1 calc\_network\_address()

calc\_network\_address() function is simplest one: it just ANDing host address with network mask, resulting network address.

Listing 18.29: 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 most important AND— here is network address is calculated.

## 18.6.2 form\_IP()

form\_IP() function just puts all 4 bytes into 32-bit value.

Here is how it is usually done:

- Allocate a variable for return value. Set it to 0.
- Take fourth (lowest) byte, apply OR operation to this byte and return value. Return value contain 4th byte now.
- Take third byte, shift it by 8 bits left. You'll get a value in form of 0x0000bb00 where bb is your third byte. Apply OR operation to the resulting value and return value. Return value contain 0x000000aa so far, so ORing values will produce value in form of 0x0000bbaa.
- Take second byte, shift it by 16 bits left. You'll get a value in form of 0x00cc0000 where cc is your second byte. Apply OR operation to the resulting value and return value. Return value contain 0x0000bbaa so far, so ORing values will produce value in form 0x00ccbbaa.



- Take first byte, shift it by 24 bits left. You'll get a value in form of 0xdd000000 where dd is your first byte. Apply OR operation to the resulting value and return value. Return value contain 0x00ccbbbaa so far, so ORing values will produce value in form of 0xddccbbbaa.

And that's how it's done by non-optimizing MSVC 2012:

Listing 18.30: 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$[ebp]
    ; EDX=000000bb
    shl     edx, 8
    ; EDX=0000bb00
    or      eax, edx
    ; EAX=ddccbb00
    movzx   ecx, BYTE PTR _ip4$[ebp]
    ; ECX=000000aa
    or      eax, ecx
    ; EAX=ddccbbaa
    pop     ebp
    ret     0
_form_IP ENDP

```

Well, order is different, but, of course, order of operations doesn't matters. Optimizing MSVC 2012 does essentially the same, but in different way:

Listing 18.31: 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
    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, each byte is written to the lowest 8 bits of returning value, and then returning value is shifting by one byte left at each step. Repeat 4 times for each input byte.

That's it! Unfortunately, there are probably no other ways to do so. I've never heard of CPUs or ISAs which has some instruction for composing a value from bits or bytes. It's all usually done by bit shifting and ORing.

### 18.6.3 print\_as\_IP()

print\_as\_IP() do inverse: split 32-bit value into 4 bytes.

Slicing works somewhat simpler: just shift input value by 24, 16, 8 or 0 bits, take bits from zeroth to seventh (lowest byte), and that's it:

Listing 18.32: 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
_print_as_IP ENDP

```

Optimizing MSVC 2012 does almost the same, but without unnecessary input value reloads:

Listing 18.33: 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=ddccbaa
shr     eax, 16
; EAX=0000ddcc
and     eax, 255
; EAX=000000cc
push   eax
; ECX=ddccbaa
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

```

### 18.6.4 form\_netmask() and set\_bit()

form\_netmask() makes network mask value from CIDR notation. Of course, it would be much effective to use there some kind of precalculated table, but I wrote it in this way intentionally, to demonstrate bit shifts. I also made separate function set\_bit(). It's a not very good idea to make a function for such primitive operation, but it would be easy to understand how it all works.

Listing 18.34: 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
$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: just shift 1 to number of bits we need and then OR-ing it with “input” value. `form_netmask()` has a loop: it will set as many bits (starting from **MSB**) as passed in `netmask_bits` argument

## 18.6.5 Summary

That’s it! I run it and got:

```

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

```

## 18.7 Conclusion

Analogous to the C/C++ shifting operators `<<` and `>>`, shift instructions in x86 are `SHR/SHL` (for unsigned values), `SAR/SHL` (for signed values).

ARM shift instructions are LSR/LSL (for unsigned values), ASR/LSL (for signed values). It's also possible to add shifting suffix to some instructions (which are called "data processing instructions").

### 18.7.1 Check for specific bit (known at compiling stage)

Test if the 1000000 bit (0x40) is present in register's value:

Listing 18.35: C/C++

```
if (input&0x40)
    ...
```

Listing 18.36: x86

```
TEST REG, 40h
JNZ is_set
; bit is not set
```

Listing 18.37: x86

```
TEST REG, 40h
JZ is_cleared
; bit is set
```

Listing 18.38: ARM (ARM mode)

```
TST REG, #0x40
BNE is_set
; bit is not set
```

Sometimes, AND is used instead of TEST, but flags are set just as the same.

### 18.7.2 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 18.39: C/C++

```
if ((value>>n)&1)
    ....
```

This is usually implemented in x86 code as:

Listing 18.40: 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 18.41: C/C++

```

if (value & (1<<n))
    ....

```

This is usually implemented in x86 code as:

Listing 18.42: x86

```

; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG

```

### 18.7.3 Set specific bit (known at compiling stage)

Listing 18.43: C/C++

```

value=value|0x40;

```

Listing 18.44: x86

```

OR REG, 40h

```

Listing 18.45: ARM (ARM mode) and ARM64

```

ORR R0, R0, #0x40

```

### 18.7.4 Set specific bit (specified at runtime)

Listing 18.46: C/C++

```

value=value|(1<<n);

```

This is usually implemented in x86 code as:

Listing 18.47: x86

```

; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG

```

## 18.7.5 Clear specific bit (known at compiling stage)

Just apply AND operation with inverted value:

Listing 18.48: C/C++

```
value=value&(~0x40);
```

Listing 18.49: x86

```
AND REG, 0FFFFFFBFh
```

Listing 18.50: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

This is actually leaving all bits set but one.

ARM in ARM mode has BIC instruction, which works like NOT+AND instruction pair:

Listing 18.51: ARM (ARM mode)

```
BIC R0, R0, #0x40
```

## 18.7.6 Clear specific bit (specified at runtime)

Listing 18.52: C/C++

```
value=value&(~(1<<n));
```

Listing 18.53: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

## 18.8 Exercises

### 18.8.1 Exercise #1

What this code does?



## Listing 18.54: Optimizing MSVC 2010

```

_a$ = 8
_f PROC
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, ecx
    mov     edx, ecx
    shl     edx, 16           ; 00000010H
    and     eax, 65280        ; 0000ff00H
    or      eax, edx
    mov     edx, ecx
    and     edx, 16711680    ; 00ff0000H
    shr     ecx, 16          ; 00000010H
    or      edx, ecx
    shl     eax, 8
    shr     edx, 8
    or      eax, edx
    ret     0
_f ENDP

```

## Listing 18.55: Optimizing Keil 6/2013 (ARM mode)

```

f PROC
    MOV     r1,#0xff0000
    AND     r1,r1,r0,LSL #8
    MOV     r2,#0xff00
    ORR     r1,r1,r0,LSR #24
    AND     r2,r2,r0,LSR #8
    ORR     r1,r1,r2
    ORR     r0,r1,r0,LSL #24
    BX     lr
    ENDP

```

## Listing 18.56: Optimizing Keil 6/2013 (thumb mode)

```

f PROC
    MOVS    r3,#0xff
    LSLS    r2,r0,#8
    LSLS    r3,r3,#16
    ANDS    r2,r2,r3
    LSRS    r1,r0,#24
    ORRS    r1,r1,r2
    LSRS    r2,r0,#8
    ASRS    r3,r3,#8
    ANDS    r2,r2,r3
    ORRS    r1,r1,r2
    LSLS    r0,r0,#24
    ORRS    r0,r0,r1

```

```

BX      lr
ENDP

```

Answer: [F.1.11](#).

## 18.8.2 Exercise #2

What this code does?

Listing 18.57: Optimizing MSVC 2010

```

_a$ = 8 ; size ↗
↳ = 4
_f PROC
    push    esi
    mov     esi, DWORD PTR _a$[esp]
    xor     ecx, ecx
    push    edi
    lea    edx, DWORD PTR [ecx+1]
    xor     eax, eax
    npad   3 ; align next label
$LL3@f:
    mov     edi, esi
    shr     edi, cl
    add     ecx, 4
    and     edi, 15
    imul   edi, edx
    lea    edx, DWORD PTR [edx+edx*4]
    add     eax, edi
    add     edx, edx
    cmp     ecx, 28
    jle    SHORT $LL3@f
    pop     edi
    pop     esi
    ret     0
_f ENDP

```

Listing 18.58: Optimizing Keil 6/2013 (ARM mode)

```

f PROC
    MOV     r3,r0
    MOV     r1,#0
    MOV     r2,#1
    MOV     r0,r1
|L0.16|
    LSR     r12,r3,r1
    AND     r12,r12,#0xf
    MLA     r0,r12,r2,r0

```

```

ADD    r1,r1,#4
ADD    r2,r2,r2,LSL #2
CMP    r1,#0x1c
LSL    r2,r2,#1
BLE    |L0.16|
BX     lr
ENDP

```

Listing 18.59: Optimizing Keil 6/2013 (thumb mode)

```

f PROC
    PUSH    {r4,lr}
    MOVS    r3,r0
    MOVS    r1,#0
    MOVS    r2,#1
    MOVS    r0,r1
|L0.10|
    MOVS    r4,r3
    LSRS    r4,r4,r1
    LSLS    r4,r4,#28
    LSRS    r4,r4,#28
    MULS    r4,r2,r4
    ADDS    r0,r4,r0
    MOVS    r4,#0xa
    MULS    r2,r4,r2
    ADDS    r1,r1,#4
    CMP    r1,#0x1c
    BLE    |L0.10|
    POP     {r4,pc}
ENDP

```

Answer: [F.1.11](#).

### 18.8.3 Exercise #3

Using [MSDN<sup>14</sup>](#) documentation, find out, which flags were used in `MessageBox()` `win32` function call.

Listing 18.60: Optimizing MSVC 2010

```

_main PROC
    push    278595             ; 00044043H
    push    OFFSET $SG79792   ; 'caption'
    push    OFFSET $SG79793   ; 'hello, world!'
    push    0
    call    DWORD PTR __imp__MessageBoxA@16

```

<sup>14</sup>Microsoft Developer Network

```

        xor     eax, eax
        ret     0
_main   ENDP

```

Answer: [F.1.11](#).

## 18.8.4 Exercise #4

What this code does?

Listing 18.61: Optimizing MSVC 2010

```

_m$ = 8      ; size = 4
_n$ = 12     ; size = 4
_f          PROC
        mov     ecx, DWORD PTR _n$[esp-4]
        xor     eax, eax
        xor     edx, edx
        test    ecx, ecx
        je     SHORT $LN2@f
        push   esi
        mov     esi, DWORD PTR _m$[esp]
$LL3@f:
        test   cl, 1
        je     SHORT $LN1@f
        add    eax, esi
        adc    edx, 0
$LN1@f:
        add    esi, esi
        shr    ecx, 1
        jne   SHORT $LL3@f
        pop    esi
$LN2@f:
        ret     0
_f          ENDP

```

Listing 18.62: Optimizing Keil 6/2013 (ARM mode)

```

f PROC
        PUSH   {r4,lr}
        MOV    r3,r0
        MOV    r0,#0
        MOV    r2,r0
        MOV    r12,r0
        B     |L0.48|
|L0.24|
        TST    r1,#1
        BEQ   |L0.40|

```

```

        ADDS    r0,r0,r3
        ADC     r2,r2,r12
|L0.40|
        LSL     r3,r3,#1
        LSR     r1,r1,#1
|L0.48|
        CMP     r1,#0
        MOVEQ   r1,r2
        BNE     |L0.24|
        POP     {r4,pc}
        ENDP

```

Listing 18.63: Optimizing Keil 6/2013 (thumb mode)

```

f PROC
        PUSH    {r4,r5,lr}
        MOVS   r3,r0
        MOVS   r0,#0
        MOVS   r2,r0
        MOVS   r4,r0
        B       |L0.24|
|L0.12|
        LSLS   r5,r1,#31
        BEQ    |L0.20|
        ADDS   r0,r0,r3
        ADCS   r2,r2,r4
|L0.20|
        LSLS   r3,r3,#1
        LSRS   r1,r1,#1
|L0.24|
        CMP    r1,#0
        BNE    |L0.12|
        MOVS   r1,r2
        POP    {r4,r5,pc}
        ENDP

```

Answer: [F.1.11](#).

# Chapter 19

## Structures

It can be defined that the C/C++ structure, with some assumptions, just a set of variables, always stored in memory together, not necessary of the same type <sup>1</sup>.

### 19.1 MSVC: SYSTEMTIME example

Let's take SYSTEMTIME<sup>2</sup> win32 structure describing time.

That's how it is defined:

Listing 19.1: 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 current time:

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

void main()
{
    SYSTEMTIME t;
```

---

<sup>1</sup>AKA "heterogeneous container"

<sup>2</sup>MSDN: [SYSTEMTIME structure](#)

```

GetSystemTime (&t);

printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

return;
};

```

We got (MSVC 2010):

Listing 19.2: 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', 0
    ↓ aH, 00H
    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 local stack – that is exactly `sizeof(WORD)*8` (there are 8 WORD variables in the structure).

Pay attention to the fact the structure beginning with wYear field. It can be said, a pointer to SYSTEMTIME structure is passed to the `GetSystemTime()`<sup>3</sup>,

<sup>3</sup>MSDN: SYSTEMTIME structure

but it is also can be said, pointer to the `wYear` field is passed, and that is the same! `GetSystemTime()` writes current year to the `WORD` pointer pointing to, then shifts 2 bytes ahead, then writes current month, etc, etc.

### 19.1.1 OllyDbg

Let's compile this example in MSVC 2010 with `/GS- /MD` keys and run it in OllyDbg. Let's open windows of data and stack at the address which is passed as the first argument into `GetSystemTime()` function, let's wait until it's executed and we see this: [fig.19.1](#).

Precise system time of function execution on my computer is 5 june 2014, 7:17:45: [fig.19.2](#).

So we see these 16 bytes in the data window:

```
DE 07 06 00 04 00 04 00
07 00 11 00 2D 00 8D 00
```

Each two bytes representing one structure field. Since [endianness](#) is *little endian*, we see low byte first and then high one. Hence, these are values which are currently stored in memory:

Hexadecimal number	decimal number	field name
0x07DE	2014	wYear
0x0006	6	wMonth
0x0004	4	wDayOfWeek
0x0005	5	wDay
0x0007	7	wHour
0x0011	17	wMinute
0x002D	45	wSecond
0x008D	141	wMilliseconds

The same values are seen in the stack window, but they are grouped as 32-bit values.

And then `printf()` just takes values it needs and outputs them to the console.

Some values `printf()` doesn't output (`wDayOfWeek` and `wMilliseconds`), but they are in memory right now, available for usage.



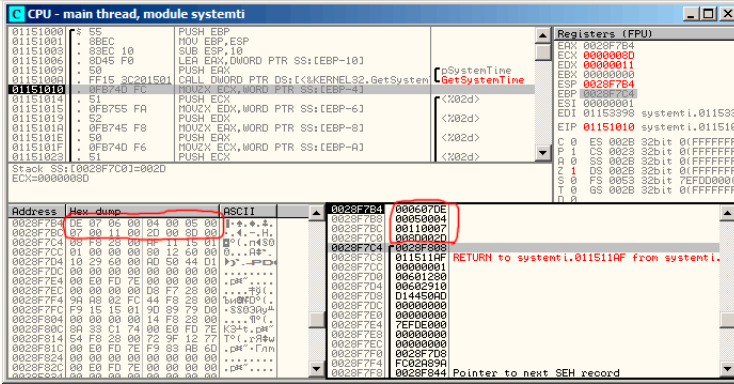


Figure 19.1: OllyDbg: GetSystemTime() just executed

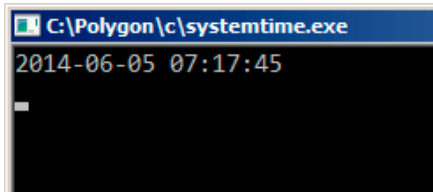


Figure 19.2: OllyDbg: printf() output

### 19.1.2 Replacing the structure by array

The fact the structure fields are just variables located side-by-side, I can demonstrate by the following way. Keeping in mind SYSTEMTIME structure description, I can 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;
};
```

Compiler will grumble for a little:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible ↵
    ↵ types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

But nevertheless, it will produce this code:

Listing 19.3: 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 fact the result in assembly form cannot be distinguished from the result of previous compilation. So by looking at this code, one cannot say for sure, was there structure declared, or an array.

Nevertheless, no one will do it in sane state of mind. Since it is not convenient. Also structure fields may be changed by developers, swapped, etc.

I'm not adding OllyDbg example here, because it will be just as the same as in the case with structure.

## 19.2 Let's allocate space for structure using malloc()

However, sometimes it is simpler to place structures not in local stack, but in [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 to easily see what we need.

Listing 19.4: 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`, that is exact number of bytes to be allocated by `malloc()`. It returns the pointer to freshly allocated memory block in the EAX register, which is then moved into the ESI register. `GetSystemTime()` win32 function undertake to save value in the ESI, and that is why it is not saved here and continue to be used after `GetSystemTime()` call.

New instruction –`MOVZX` (*Move with Zero extent*). It may be used almost in those cases as `MOVSX`, but, it clears other bits to 0. That's because `printf()` requires 32-bit *int*, but we got `WORD` in structure –that is 16-bit unsigned type. That's why by copying value from `WORD` into *int*, bits from 16 to 31 must also be cleared, because there will be random noise otherwise, left there from previous operations on registers.

In this example, I can represent structure as array of 8 `WORD`-s:

```

#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 got:

Listing 19.5: 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 cannot be distinguished from the previous. And again I should note, one should not do this in practice, unless you know what you do.

## 19.3 UNIX: struct tm

### 19.3.1 Linux

As of Linux, let's take tm structure from `time.h` 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 19.6: 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]
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 create local variable names in local stack. But since we already 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 the value in the EAX, but does not modify any flags. See also the relevant section about LEA ([A.6.2](#)).

## GDB

Let's try to load the example into GDB <sup>4</sup>:

Listing 19.7: 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
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

```

<sup>4</sup>I corrected the *date* result slightly for demonstration purposes. Of course, I wasn't able to run GDB that quickly in the same second.

```

This is free software: you are free to change and redistribute ↵
↳ it.
There is NO WARRANTY, to the extent permitted by law. Type "↵
↳ show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
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 19.8: 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;
};

```



Take a notice that 32-bit *int* here instead of WORD in SYSTEMTIME. So, each field occupies 32-bit word.

Here is a 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		
0xbffff10c:	0x00000001 wday	0x00000098 yday	0x00000001 ↵
↳	isdst0x00002a30		
0xbffff11c:	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 in case of SYSTEMTIME (19.1), there are also other fields available, but not used, like tm\_wday, tm\_yday, tm\_isdst.

### Structure as a set of values

In order to illustrate the structure is just variables laying side-by-side in one place, let's rework example, while looking at the *tm* structure definition again: listing.19.8.

```
#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 exactly `tm_sec` field is passed into `localtime_r`, i.e., to the first “structure” element.

Compiler will warn us:

#### Listing 19.9: 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, will generate this:

#### Listing 19.10: 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
        mov     ebp, esp
        and     esp, 0FFFFFF0h
        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 structure in original source code or just pack of variables.

And this works. However, it is not recommended to do this in practice. Usually, non-optimizing compiler allocated variables in local stack in the same order as they were declared in function. Nevertheless, there is no any guarantee.

By the way, some other compiler may warn the `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` variables, but not `tm_sec` are used without being initialized. Indeed, compiler is not aware these will be filled when calling to `localtime_r()`.

I chose exactly this example for illustration, since all structure fields has *int* type. This will not work if structure fields have 16-bit size (*WORD*), as in case of *SYSTEMTIME* structure—*GetSystemTime()* will fill them incorrectly (because local variables will be aligned on 32-bit border). Read more about it in next section: “Fields packing in structure” (19.4).

So, structure is just variables pack laying on one place, side-by-side. I could say the structure is a syntactic sugar, directing compiler to hold them in one place. However, I’m not programming languages expert, so, most likely, I’m wrong with this term. By the way, there were a times, in very early C versions (before 1972), in which there were no structures at all [Rit93].

I’m not adding debugger example here: because it will be just the same as you already saw.

### 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);
    };
};
```

I just cast pointer to structure to array of *int*'s. And that works! I run example at 23:51:45 26-July-2014.

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
```

0x00000001 (1)

Variables here are just in the same order as they are enumerated in structure definition: 19.8.

Here is how it was compiled:

Listing 19.11: 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?
                    jnz     short loc_80483D8 ; no - load ↵
↳ next value
                    lea     esp, [ebp-8]
                    pop     ebx
                    pop     esi
                    pop     ebp

```

```

main          retn
              endp

```

Indeed: the space in local stack is first treated as structure, then it's treated as array.

It's even possible to modify structure fields through this pointer.

And again, it's dubious hackish way to do things, which is not recommended to use in production code.

## Exercise

As an exercise, try to modify (increase by 1) current month number treating structure as array.

## Structure as an array of bytes

I can do even more. Let's cast the pointer to 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
```

I run this example also at 23:51:45 26-July-2014. The values are just the same as in previous dump (19.3.1), and of course, lower byte goes first, because this is little-endian architecture (36).

Listing 19.12: Optimizing GCC 4.8.1

```
main          proc near
              push    ebp
              mov     ebp, esp
              push    edi
              push    esi
              push    ebx
              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 newline character
              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

```

## 19.3.2 ARM + Optimizing Keil 6/2013 (thumb mode)

Same example:

Listing 19.13: 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}

```

### 19.3.3 ARM + Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)

IDA “get to know” tm structure (because IDA “knows” argument types of library functions like `localtime_r()`), so it shows here structure elements accesses and also names are assigned to them.

Listing 19.14: 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

```

## 19.4 Fields packing in structure

One important thing is fields packing in structures<sup>5</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);
};

```

<sup>5</sup>See also: [Wikipedia: Data structure alignment](#)

```

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};

```

As we see, we have two *char* fields (each is exactly one byte) and two more – *int* (each - 4 bytes).

### 19.4.1 x86

That's all compiling into:

Listing 19.15: 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

```

By the way, we pass a structure as a whole, but in fact, as we can see, the structure is being copied to the temporary one (a place in stack is allocated in line 10 for it, and then all 4 fields, one by one, is copied in lines 12 ... 19), then its pointer is to be passed (or address). The structure is copied, because it's unknown, will the `f()` function modify structure or not. If it's so, then the structure inside of `main()` should remain as the same. We could use pointers in C/C++, and resulting code might be almost the same, but without copying.

As we can see, each field's address is aligned on a 4-bytes border. That's why each `char` occupies 4 bytes here (like `int`). Why? Thus it is easier for CPU to access memory at aligned addresses and to cache data from it.

However, it is not very economical in size sense.

Let's try to compile it with option (`/Zp1`) (`/Zp[n]` *pack structures on n-byte boundary*).

Listing 19.16: 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 ↗
10     ↘ temporary structure
11     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 it give to us? Size economy. And as drawback –CPU will access these fields without full performance it could.

Structure is also copied in `main()`. Not by one-by-one field, but 10 bytes, using three pairs of `MOV`. Why not 4? Compiler decided that it's better to copy 10 bytes using 3 `MOV` pairs then to copy two 32-bit words and two bytes using 4 `MOV` pairs. By the way, such copy implementation using `MOV` instead of `memcpy()` function calling is widely used, because it's faster then to call `memcpy()`—if to consider short blocks, of course: [29.1.4](#).

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.

Aside from MSVC /Zp option which set how to align each structure field, here is also `#pragma pack` compiler option, it can be defined right in source code. It is available in both MSVC<sup>6</sup> and GCC<sup>7</sup>.

Let's back to the `SYSTEMTIME` structure consisting in 16-bit fields. How our compiler know to pack them on 1-byte alignment boundary?

WinNT.h file has this:

Listing 19.17: WinNT.h

```
#include "pshpack1.h"
```

And this:

Listing 19.18: WinNT.h

```
#include "pshpack4.h" // 4 byte packing is ↵
↳ the default
```

The file `PshPack1.h` looks like:

Listing 19.19: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _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
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

That's how compiler will pack structures defined after `#pragma pack`.

## 19.4.2 x86 + OllyDbg + fields are packed by default

Let's try our example (where fields are aligned by default (4 bytes)) in OllyDbg: fig.19.3.

We see our 4 fields in data window. But where from random bytes (0x30, 0x27) aside of first (a) and third (c) fields are came? By looking at our listing 19.15, we can see that first and third field has *char* type, therefore, only one byte is written, 1 and 3 respectively (lines 6 and 8). Other 3 bytes of 32-bit words

<sup>6</sup>MSDN: Working with Packing Structures

<sup>7</sup>Structure-Packing Pragma

are not being modified in memory! Hence, random garbage is there. This garbage influence `printf()` output in no way, because values for it are prepared using `MOVSB` instruction, which takes bytes, but not words: listing.19.15 (lines 34 and 38).

By the way, `MOVSB` (sign-extending) instruction is used here, because `char` type is signed by default in MSVC and GCC. If the type `unsigned char` or `uint8_t` be here, `MOVZB` instruction would be used instead.

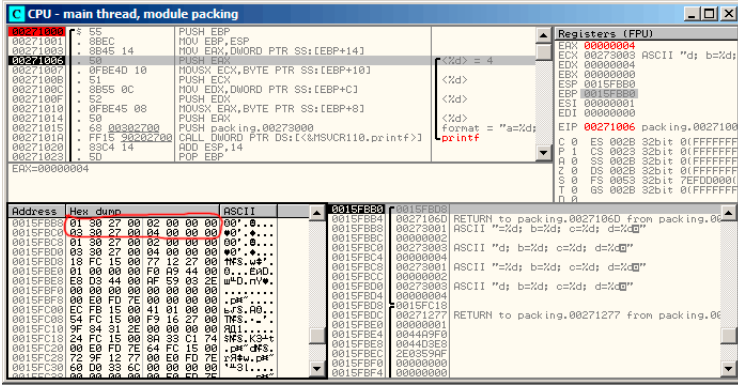


Figure 19.3: OllyDbg: Before `printf()` execution

### 19.4.3 x86 + OllyDbg + fields aligning by 1 byte boundary

Things are much clearer here: 4 fields occupies 10 bytes and values are stored side-by-side: fig.19.4.

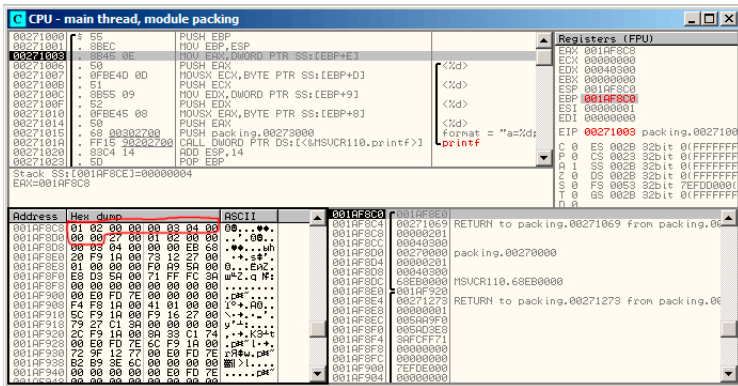


Figure 19.4: OllyDbg: Before `printf()` execution

## 19.4.4 ARM + Optimizing Keil 6/2013 (thumb mode)

Listing 19.20: 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 passed instead of pointer to structure, and since first 4 function arguments in ARM are passed via registers, so then structure fields are passed via R0-R3.

LDRB loads one byte from memory and extending it to 32-bit, taking into account its sign. This is akin to MOVSB instruction in x86. Here it is used for loading fields *a* and *c* from structure.

One more thing we spot easily, instead of function epilogue, here is jump to another function's epilogue! Indeed, that was quite different function, not related in any way to our function, 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 on addresses). Indeed, there is no difference, which epilogue to execute, if it works just as we need. Apparently, Keil decides to reuse a part of another function by a reason of economy. Epilogue takes 4 bytes while jump –only 2.

## 19.4.5 ARM + Optimizing Xcode 4.6.3 (LLVM) (thumb-2 mode)



Listing 19.21: 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 MOVSB in x86. All the rest – just the same.

## 19.5 Nested structures

Now what about situations when one structure is defined inside of another structure?

```

#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);
};

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 will be placed between `a,b` and `d,e` fields of `outer_struct`.

Let's compile (MSVC 2010):

Listing 19.22: Optimizing MSVC 2010 /Ob0

```

$SG2802 DB      'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00
↳ H

_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
    pop     ebx
    add     esp, 24
    ret     0
_main ENDP

```

One curious point 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 finally unfolds into *linear* or *one-dimensional* structure.

Of course, if to replace `struct inner_struct c;` declaration to `struct inner_struct *c;` (thus making a pointer here) situation will be quite different.

### 19.5.1 OllyDbg

Let's load the example into OllyDbg and take a look on `outer_struct` in memory: [fig.19.5](#).

That's how 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.

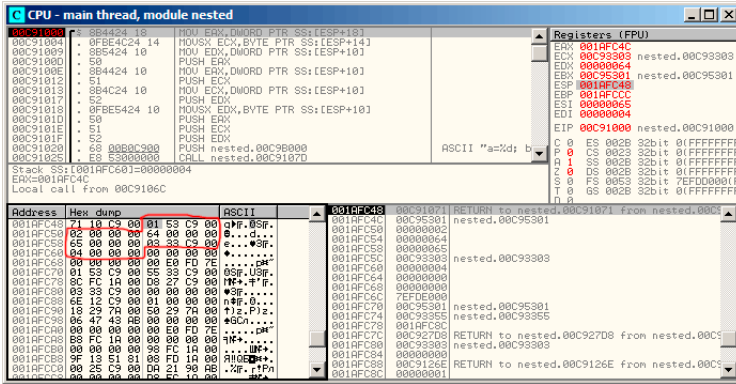


Figure 19.5: OllyDbg: Before `printf()` execution

## 19.6 Bit fields in structure

### 19.6.1 CPUID example

C/C++ language allow to define exact number of bits for each structure fields. It is very useful if one needs to save memory space. For example, one bit is enough for variable of *bool* type. But of course, it is not rational if speed is important.

Let's consider CPUID<sup>8</sup> instruction example. This instruction returning information about current CPU and its features.

If the EAX is set to 1 before instruction execution, CPUID will return 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>8</sup><http://en.wikipedia.org/wiki/CPUID>

MSVC 2010 has CPUID macro, but GCC 4.4.1 –has not. So let's make this function by yourself for GCC with the help of its built-in assembler<sup>9</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);
}
```

<sup>9</sup>[More about internal GCC assembler](#)

```

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 will fill EAX/EBX/ECX/EDX, these registers will be reflected in the `b[]` array. Then, we have a pointer to the `CPUID_1_EAX` structure and we point it to the value in the EAX from `b[]` array.

In other words, we treat 32-bit *int* value as a structure. Then we read from the structure.

## MSVC

Let's compile it in MSVC 2008 with `/Ox` option:

Listing 19.23: Optimizing MSVC 2008

```

_b$ = -16 ; size = 16
_main PROC
sub esp, 16
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

```

SHR instruction shifting value in the EAX register by number of bits must be *skipped*, e.g., we ignore some bits *at right*.

AND instruction clears bits not needed *at left*, or, in other words, leaves only those bits in the EAX register we need now.

## MSVC + OllyDbg

Let's load our example into OllyDbg and see, which values was set in EAX/EBX/ECX/EDX after execution of CPUID: [fig.19.6](#).

EAX has 0x000206A7 (my CPU is Intel Xeon E3-1220).  
 This is 000000000000010000011010100111 in binary form.  
 Here is how 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

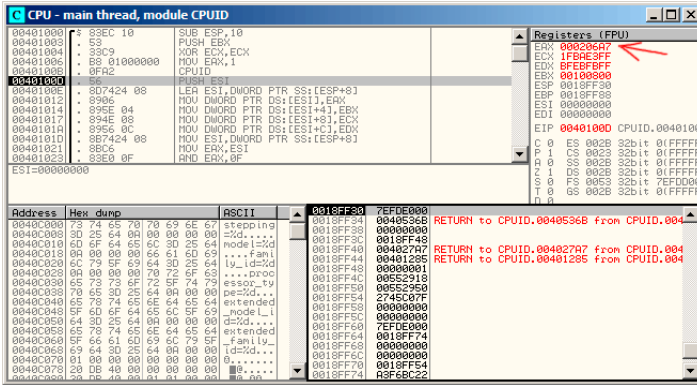


Figure 19.6: OllyDbg: After CPUID execution

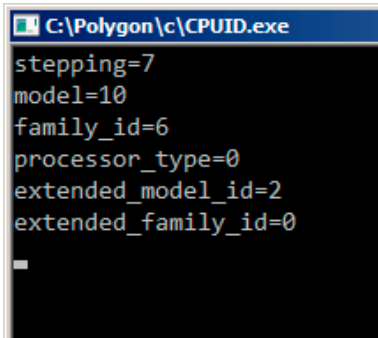


Figure 19.7: OllyDbg: Result



**GCC**

Let's try GCC 4.4.1 with -O3 option.

Listing 19.24: Optimizing GCC 4.4.1

```

main                proc near ; DATA XREF: _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    push    esi
    mov     esi, 1
    push    ebx
    mov     eax, esi
    sub     esp, 18h
    cpushd
    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

```

```

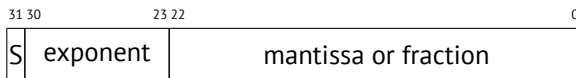
and    eax, 0Fh
and    esi, 0FFh
mov    [esp+8], eax
mov    dword ptr [esp+4], offset aExtended_model ; "\
↳ extended_model_id=%d\n"
mov    dword ptr [esp], 1
call   __printf_chk
mov    [esp+8], esi
mov    dword ptr [esp+4], offset unk_80486D0
mov    dword ptr [esp], 1
call   __printf_chk
add    esp, 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 the GCC somehow united calculation of `extended_model_id` and `extended_family_id` into one block, instead of calculating them separately, before corresponding each `printf()` call.

## 19.6.2 Working with the float type as with a structure

As it was already noted in section about FPU (16), both *float* and *double* types consisted of *sign*, *significand* (or *fraction*) and *exponent*. But will we able to work with these fields directly? Let's try with *float*.



(S—sign)

```

#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));
};

```

`float_as_struct` structure occupies as much space in memory as *float*, i.e., 4 bytes or 32 bits.

Now we setting negative sign in input value and also, by adding 2 to exponent, we thereby multiplicating the whole number by  $2^2$ , i.e., by 4.

Let's compile in MSVC 2008 without optimization turned on:

Listing 19.25: 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$[ebp]
shr     eax, 23          ; 00000017H - drop significand
and     eax, 255        ; 000000ffH - leave here only ↗
↳ exponent
add     eax, 2          ; add 2 to it
and     eax, 255        ; 000000ffH
shl     eax, 23        ; 00000017H - shift result to place ↗
↳ of bits 30:23
mov     ecx, DWORD PTR _t$[ebp]
and     ecx, -2139095041 ; 807ffffFH - drop exponent
or      ecx, eax        ; add original value without ↗
↳ exponent with new calculated exponent
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

```

Redundant for a bit. If it is compiled with `/Ox` flag there is no `memcpy()` call, `f` variable is used directly. But it is easier to understand it all by looking at unoptimized version.

What GCC 4.4.1 with `-O3` will do?

Listing 19.26: 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 significand and ↗
↪ exponent 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 ; add 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
    retn
main endp

```

The `f()` function is almost understandable. However, what is interesting, GCC was able to calculate `f(1.234)` result during compilation stage despite all this hodge-podge with structure fields and prepared this argument to the `printf()` as precalculated at compile time!

## 19.7 Exercises

### 19.7.1 Exercise #1

[http://beginners.re/exercises/per\\_chapter/struct\\_exercise\\_Linux8tar](http://beginners.re/exercises/per_chapter/struct_exercise_Linux8tar)<sup>10</sup>:

This Linux x86 program opens a file and prints some number. What this number is?

Answer: [F.1.12](#).

### 19.7.2 Exercise #2

This function takes some structure on input and do something. Try to reverse engineer structure field types. Function contents may be ignored so far.

Listing 19.27: Optimizing MSVC 2010

```

$SG2802 DB      '%f', 0aH, 00H
$SG2803 DB      '%c, %d', 0aH, 00H
$SG2805 DB      'error #2', 0aH, 00H
$SG2807 DB      'error #1', 0aH, 00H

__real@405ec00000000000 DQ 0405ec0000000000r    ; 123
__real@407bc00000000000 DQ 0407bc0000000000r    ; 444

_s$ = 8
_f PROC
    push     esi
    mov     esi, DWORD PTR _s$[esp]
    cmp     DWORD PTR [esi], 1000
    jle     SHORT $LN4@f
    cmp     DWORD PTR [esi+4], 10
    jbe     SHORT $LN3@f
    fld     DWORD PTR [esi+8]
    sub     esp, 8
    fmul    QWORD PTR __real@407bc00000000000
    fld     QWORD PTR [esi+16]
    fmul    QWORD PTR __real@405ec00000000000
    faddp   ST(1), ST(0)
    fstp    QWORD PTR [esp]
    push    OFFSET $SG2802 ; '%f'
    call    _printf
    movzx  eax, BYTE PTR [esi+25]
    movsx  ecx, BYTE PTR [esi+24]
    push   eax

```

<sup>10</sup>GCC 4.8.1 -03

```

    push    ecx
    push    OFFSET $SG2803 ; '%c, %d'
    call   _printf
    add    esp, 24
    pop    esi
    ret    0
$LN3@f:
    pop    esi
    mov    DWORD PTR _s$[esp-4], OFFSET $SG2805 ; 'error ↵
    ↵ #2'
    jmp    _printf
$LN4@f:
    pop    esi
    mov    DWORD PTR _s$[esp-4], OFFSET $SG2807 ; 'error ↵
    ↵ #1'
    jmp    _printf
_f      ENDP

```

Listing 19.28: Non-optimizing Keil 6/2013 (ARM mode)

```

f PROC
    PUSH    {r4-r6,lr}
    MOV     r4,r0
    LDR     r0,[r0,#0]
    CMP     r0,#0x3e8
    ADRL   r0,|L0.140|
    BLE     |L0.132|
    LDR     r0,[r4,#4]
    CMP     r0,#0xa
    ADRL   r0,|L0.152|
    BLS     |L0.132|
    ADD     r0,r4,#0x10
    LDM     r0,{r0,r1}
    LDR     r3,|L0.164|
    MOV     r2,#0
    BL     __aeabi_dmul
    MOV     r5,r0
    MOV     r6,r1
    LDR     r0,[r4,#8]
    LDR     r1,|L0.168|
    BL     __aeabi_fmuls
    BL     __aeabi_f2d
    MOV     r2,r5
    MOV     r3,r6
    BL     __aeabi_dadd
    MOV     r2,r0
    MOV     r3,r1
    ADR     r0,|L0.172|

```

```

        BL      __2printf
        LDRB   r2,[r4,#0x19]
        LDRB   r1,[r4,#0x18]
        POP    {r4-r6,lr}
        ADR    r0,|L0.176|
        B      __2printf
|L0.132|
        POP    {r4-r6,lr}
        B      __2printf
        ENDP

|L0.140|
        DCB    "error #1\n",0
        DCB    0
        DCB    0
|L0.152|
        DCB    "error #2\n",0
        DCB    0
        DCB    0
|L0.164|
        DCD    0x405ec000
|L0.168|
        DCD    0x43de0000
|L0.172|
        DCB    "%f\n",0
|L0.176|
        DCB    "%c, %d\n",0

```

Listing 19.29: Non-optimizing Keil 6/2013 (thumb mode)

```

f PROC
        PUSH   {r4-r6,lr}
        MOV    r4,r0
        LDR    r0,[r0,#0]
        CMP    r0,#0x3e8
        ADRL   r0,|L0.140|
        BLE    |L0.132|
        LDR    r0,[r4,#4]
        CMP    r0,#0xa
        ADRLS  r0,|L0.152|
        BLS    |L0.132|
        ADD    r0,r4,#0x10
        LDM    r0,{r0,r1}
        LDR    r3,|L0.164|
        MOV    r2,#0
        BL     __aeabi_dmul
        MOV    r5,r0
        MOV    r6,r1

```



```

    LDR    r0,[r4,#8]
    LDR    r1,|L0.168|
    BL     __aeabi_fmuls
    BL     __aeabi_f2d
    MOV    r2,r5
    MOV    r3,r6
    BL     __aeabi_dadd
    MOV    r2,r0
    MOV    r3,r1
    ADR    r0,|L0.172|
    BL     __2printf
    LDRB   r2,[r4,#0x19]
    LDRB   r1,[r4,#0x18]
    POP    {r4-r6,lr}
    ADR    r0,|L0.176|
    B      __2printf
|L0.132|
    POP    {r4-r6,lr}
    B      __2printf
    ENDP

|L0.140|
    DCB    "error #1\n",0
    DCB    0
    DCB    0
|L0.152|
    DCB    "error #2\n",0
    DCB    0
    DCB    0
|L0.164|
    DCD    0x405ec000
|L0.168|
    DCD    0x43de0000
|L0.172|
    DCB    "%f\n",0
|L0.176|
    DCB    "%c, %d\n",0

```

Answer: [F.1.12](#).

# Chapter 20

## Unions

### 20.1 Pseudo-random number generator example

If we need float random numbers from 0 to 1, the most simplest thing is to use [PRNG](#)<sup>1</sup> like Mersenne twister. It produces random 32-bit values in DWORD form. Then we can transform this value to *float* and then dividing it by `RAND_MAX` (`0xFFFFFFFF` in our case) —value we got will be in 0..1 interval.

But as we know, division operation is slow. Also, we would like to issue as small number of FPU operations as possible. Will it be possible to get rid of division?

Let's recall what float number consisted of: sign bit, significand bits and exponent bits. We need just to store random bits to all significand bits for getting random float number!

Exponent cannot be zero (number is denormalized in this case), so we will store `01111111` to exponent —this means exponent is 1. Then fill significand with random bits, set sign bit to 0 (which means positive number) and voilà. Generated numbers will be in 1 to 2 interval, so we also must subtract 1 from it.

Very simple linear congruential random numbers generator is used in my example<sup>2</sup>, produces 32-bit numbers. The PRNG initializing by current time in UNIX-style.

Then, *float* type represented as *union* —it is the C/C++ construction enabling us to interpret piece of memory as differently typed. In our case, we are able to create a variable of union type 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.

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

union uint32_t_float
```

<sup>1</sup>Pseudorandom number generator

<sup>2</sup>idea was taken from: <http://goo.gl/KiVQjZ>

```

{
    uint32_t i;
    float f;
};

// from the Numerical Recipes book
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;

int main()
{
    uint32_t_float tmp;

    uint32_t RNG_state=time(NULL); // initial seed
    for (int i=0; i<100; i++)
    {
        RNG_state=RNG_state*RNG_a+RNG_c;
        tmp.i=RNG_state & 0x007fffff | 0x3F800000;
        float x=tmp.f-1;
        printf ("%f\n", x);
    };
    return 0;
};

```

Listing 20.1: Optimizing MSVC 2010

```

$SG4232    DB    '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff000000000000r    ; 1

tv140 = -4                                ; size = 4
_tmp$ = -4                                ; size = 4
_main    PROC
    push    ebp
    mov     ebp, esp
    and     esp, -64                        ; ffffffff0H
    sub     esp, 56                          ; 00000038H
    push    esi
    push    edi
    push    0
    call    __time64
    add     esp, 4
    mov     esi, eax
    mov     edi, 100                          ; 00000064H
$LN3@main:
; let's generate random 32-bit number

```

```

    imul   esi, 1664525           ; 0019660dH
    add    esi, 1013904223       ; 3c6ef35fH
    mov    eax, esi

; leave bits for significand only

    and    eax, 8388607          ; 007ffffffH

; set exponent to 1

    or     eax, 1065353216       ; 3f800000H

; store this value as int

    mov    DWORD PTR _tmp$[esp+64], eax
    sub    esp, 8

; load this value as float

    fld    DWORD PTR _tmp$[esp+72]

; subtract one from it

    fsub   QWORD PTR __real@3ff0000000000000
    fstp   DWORD PTR tv140[esp+72]
    fld    DWORD PTR tv140[esp+72]
    fstp   QWORD PTR [esp]
    push   OFFSET $SG4232
    call   _printf
    add    esp, 12                ; 0000000cH
    dec    edi
    jne    SHORT $LN3@main
    pop    edi
    xor    eax, eax
    pop    esi
    mov    esp, ebp
    pop    ebp
    ret    0

_main    ENDP
_TEXT    ENDS
END

```

GCC produces very similar code.

# Chapter 21

## Pointers to functions

Pointer to function, as any other pointer, is just an address of function beginning in its code segment.

It is often used in callbacks <sup>1</sup>.

Well-known examples are:

- `qsort()`<sup>2</sup>, `atexit()`<sup>3</sup> from the standard C library;
- \*NIX OS signals<sup>4</sup>;
- thread starting: `CreateThread()` (win32), `pthread_create()` (POSIX);
- a lot of win32 functions, e.g. `EnumChildWindows()`<sup>5</sup>.
- a lot of places in Linux kernel, for example, filesystem driver functions are called via callbacks: <http://lxr.free-electrons.com/source/include/linux/fs.h?v=3.14#L1525>
- GCC plugin functions are also called via callbacks: <https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html#Plugin-API>

So, `qsort()` function is a C/C++ standard library quicksort implementation. The functions is able to sort anything, any types of data, as long as you have a function for these two elements comparison, and `qsort()` is able to call it.

The comparison function can be defined as:

```
int (*compare)(const void *, const void *)
```

<sup>1</sup>[http://en.wikipedia.org/wiki/Callback\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Callback_(computer_science))

<sup>2</sup>[http://en.wikipedia.org/wiki/Qsort\\_\(C\\_standard\\_library\)](http://en.wikipedia.org/wiki/Qsort_(C_standard_library))

<sup>3</sup><http://www.opengroup.org/onlinepubs/009695399/functions/atexit.html>

<sup>4</sup><http://en.wikipedia.org/wiki/Signal.h>

<sup>5</sup>[http://msdn.microsoft.com/en-us/library/ms633494\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms633494(VS.85).aspx)

Let's use slightly modified example I found [here](#):

```

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↵
23     ↵ [10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
24     int i;
25
26     /* Sort the array */
27     qsort(numbers,10,sizeof(int),comp) ;
28     for (i=0;i<9;i++)
29         printf("Number = %d\n",numbers[ i ] ) ;
30     return 0;
31 }

```

## 21.1 MSVC

Let's compile it in MSVC 2010 (I omitted some parts for the sake of brevity) with /Ox option:

Listing 21.1: 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
    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 ; ↗
↳ 00000028H
    push   esi
    push   OFFSET _comp
    push   4
    lea    eax, DWORD PTR _numbers$[esp+52]
    push   10 ; ↗
↳ 0000000aH
    push   eax
    mov    DWORD PTR _numbers$[esp+60], 1892 ; ↗
↳ 00000764H
    mov    DWORD PTR _numbers$[esp+64], 45 ; ↗
↳ 0000002dH
    mov    DWORD PTR _numbers$[esp+68], 200 ; ↗
↳ 000000c8H
    mov    DWORD PTR _numbers$[esp+72], -98 ; ↗
↳ ffffffff9eH
    mov    DWORD PTR _numbers$[esp+76], 4087 ; 00000↗
↳ ff7H
    mov    DWORD PTR _numbers$[esp+80], 5
    mov    DWORD PTR _numbers$[esp+84], -12345 ; ↗
↳ fffffcfc7H
    mov    DWORD PTR _numbers$[esp+88], 1087 ; ↗

```

```

↳ 0000043FH
   mov     DWORD PTR _numbers$[esp+92], 88           ; ↵
↳ 00000058H
   mov     DWORD PTR _numbers$[esp+96], -100000     ; ↵
↳ fffe7960H
   call    _qsort
   add     esp, 16                                   ; ↵
↳ 00000010H

...

```

Nothing surprising so far. As a fourth argument, an address of label `_comp` is passed, that is just a place where function `comp()` located, or, in other words, address of the very first instruction of this function.

How `qsort()` calling it?

Let's take a look into this function located in `MSVCR80.DLL` (a MSVC DLL module with C standard library functions):

Listing 21.2: MSVCR80.DLL

```

.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, ↵
↳ unsigned int, int (__cdecl *)(const 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:7816CCED          call   [esp+118h+comp]
.text:7816CCF2          add    esp, 8
.text:7816CCF5          test   eax, eax
.text:7816CCF7          jle    short loc_7816CD04
```

`comp`— is fourth function argument. Here the control is just passed to the address in the `comp` argument. Before it, two arguments 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 incorrect pointer to function, `qsort()` may pass control to incorrect point, a process may crash and this bug will be hard to find.

Second reason is the callback function types must comply strictly, calling wrong function with wrong arguments of wrong types may lead to serious problems, however, process crashing is not a big problem –big problem is to determine a reason of crashing –because compiler may be silent about potential trouble while compiling.

## 21.1.1 MSVC + OllyDbg

Let's load our example into OllyDbg and set breakpoint on `comp()` function.

How values are compared we can see at the very first `comp()` call: fig.21.1. OllyDbg shows compared values in the window under code window, for convenience. We can also see that the `SP` pointing to `RA` where the place in `qsort()` function is (actually located in `MSVCR100.DLL`).

By tracing (F8) until `RETN` instruction, and pressing F8 one more time, we returning into `qsort()` function: fig.21.2. That was a call to comparison function.

Here is also screenshot of the moment of the second call of `comp()`— now values to be compared are different: fig.21.3.

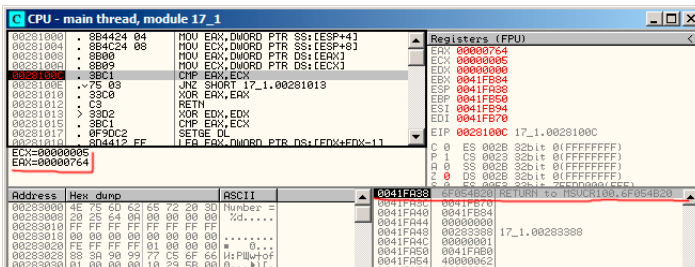


Figure 21.1: OllyDbg: first call of `comp()`

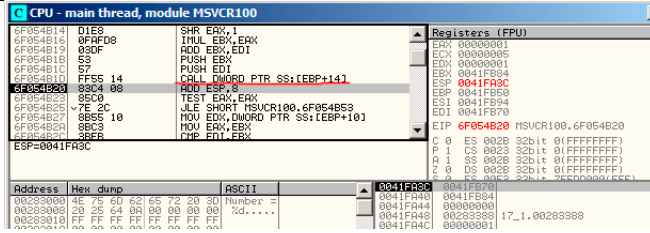


Figure 21.2: OllyDbg: the code in qsort() right after comp() call

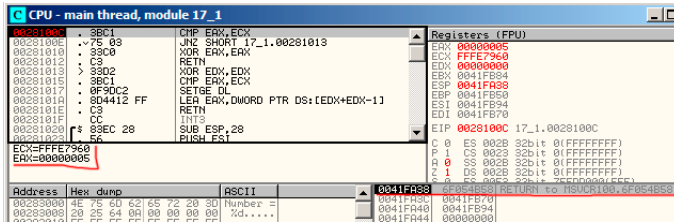


Figure 21.3: OllyDbg: second call of comp()

### 21.1.2 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.

I found the address of the first CMP instruction in comp(), it is 0x0040100C and I'm setting breakpoint on it:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

I'm getting information about registers at 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=0xfffe7960 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
...
```

I filtered out EAX and ECX and 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
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xffffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xffffe7960
EAX=0xffffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xffffe7960
EAX=0x000000c8 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, quick sort algorithm needs 34 comparison operations for sorting these 10 numbers.

### 21.1.3 MSVC + tracer (code coverage)

We can also use tracer's feature to collect all possible register's values and show them in [IDA](#).

Let's trace all instructions in `comp()` function:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

We getting `.idc-script` for loading into `IDA` and load it: [fig.21.4](#).

`IDA` gave the function name (`PtFuncCompare`) – it seems, because `IDA` sees that pointer to this function is passed into `qsort()`.

We see that `a` and `b` pointers are pointing to various places in array, but step between points is 4—indeed, 32-bit values are stored in the array.

We see that the instructions at `0x401010` and `0x401012` was never executed (so they leaved as white): indeed, `comp()` was never returned 0, because there no equal elements in array.

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare proc near ; DATA XREF: _main+5J0
.text:00401000
.text:00401000 arg_0 = dword ptr 4
.text:00401000 arg_4 = dword ptr 8
.text:00401000
.text:00401004 mov eax, [esp+arg_0] ; [ESP+4]=0x45F7ec..0x45F810(step=4), L"?\x0?
.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 ; 2F=false
.text:00401010 xor eax, eax
.text:00401012 retn
.text:00401013 ; -----
.text:00401013 loc_401013: ; CODE XREF: PtFuncCompare+E1J
.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:0040101f PtFuncCompare endp
.text:0040101f
```

Figure 21.4: tracer and IDA. N.B.: some values are cutted at right

## 21.2 GCC

Not a big difference:

Listing 21.3: 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], 0FFFFFFC7h
mov [esp+40h+var_C], 43Fh
```

```

mov    [esp+40h+var_8], 58h
mov    [esp+40h+var_4], 0FFFE7960h
mov    [esp+40h+var_34], offset comp
mov    [esp+40h+var_38], 4
mov    [esp+40h+var_3C], 0Ah
call   _qsort

```

comp() function:

```

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

```

qsort() implementation is located in the `libc.so.6` and it is in fact just a wrapper<sup>6</sup> for `qsort_r()`.

It will call then `quicksort()`, where our defined function will be called via passed pointer:

Listing 21.4: (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]
...

```

<sup>6</sup>a concept like [thunk function](#)

### 21.2.1 GCC + GDB (with source code)

Obviously, we have a C-source code of our example (21), so we can set breakpoint (b) on line number (11th—the line where first comparison is occurred). We also need to compile example with debugging information included (-g), so the table with addresses and corresponding line numbers is present. We can also print values by variable name (p): debugging information also has information about which register and/or local stack element contain which variable.

We can also see stack (bt) and find out that there are some intermediate function `msort_with_tmp()` used in Glibc.

Listing 21.5: GDB session

```
dennis@ubuntuv:~/polygon$ gcc 17_1.c -g
dennis@ubuntuv:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
  ↵ licenses/gpl.html>
This is free software: you are free to change and redistribute ↵
  ↵ it.
There is NO WARRANTY, to the extent permitted by law. Type "↵
  ↵ show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
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
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) 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=0xbffff104, _b=_b@entry=0xbffff108) at ↵
  ↵ 17_1.c:11
11      if (*a==*b)
(gdb) p *a
$3 = -98
```

```
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#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=0↵
    ↵ xbffff07c) 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=0↵
    ↵ xbffff07c) 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=0↵
    ↵ xbffff07c) at msort.c:45
#7  __GI_qsorth_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry↵
    ↵ =4, cmp=cmp@entry=0x804844d <cmp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsorth (b=0xbffff0f8, n=10, s=4, cmp=0↵
    ↵ x804844d <cmp>) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

## 21.2.2 GCC + GDB (no source code)

But often there are no source code at all, so we can disassemble `comp()` function (`disas`), find the very first `CMP` instruction and set breakpoint (`b`) at that address. At each breakpoint, we will dump all register contents (`info registers`). Stack information is also available (`bt`), but partial: there are no line number information for `comp()` function.

Listing 21.6: GDB session

```
dennis@ubuntuv:~/polygon$ gcc 17_1.c
dennis@ubuntuv:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/↵
    ↵ licenses/gpl.html>
This is free software: you are free to change and redistribute ↵
    ↵ it.
There is NO WARRANTY, to the extent permitted by law. Type "↵
    ↵ show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
```

For bug reporting instructions, please see:

<<http://www.gnu.org/software/gdb/bugs/>>...

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
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            0xbfffeeb8   0xbfffeeb8
ebp            0xbfffec8    0xbfffec8

```



```

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      0xffffffff9e   -98
ebx      0xb7fc0000     -1208221696
esp      0xbffffee58    0xbffffee58
ebp      0xbffffee68    0xbffffee68
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      0xffffffff9e   -98
ecx      0xbffff100     -1073745664
edx      0xc8           200
ebx      0xb7fc0000     -1208221696
esp      0xbffffeeb8    0xbffffeeb8
ebp      0xbffffeec8    0xbffffeec8
esi      0xbffff104     -1073745660
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) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b↵
    ↵ b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0↵
    ↵ xbffff07c) 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=0↵
    ↵ xbffff07c) 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=0↵
    ↵ xbffff07c) at msort.c:45
#7  __GI_qsorth_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry↵
    ↵ =4, cmp=cmp@entry=0x804844d <cmp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsorth (b=0xbffff0f8, n=10, s=4, cmp=0↵
    ↵ x804844d <cmp>) at msort.c:307
#9  0x0804850d in main ()

```

## Chapter 22

# 64-bit values in 32-bit environment

In the 32-bit environment [GPR](#)'s are 32-bit, so 64-bit values are stored and passed as 32-bit value pairs <sup>1</sup>.

### 22.1 Arguments passing, addition, subtraction

```
#include <stdint.h>

uint64_t f1 (uint64_t a, uint64_t b)
{
    return a+b;
};

void f1_test ()
{
#ifdef __GNUC__
    printf ("%lld\n", f1(12345678901234, 23456789012345));
#else
    printf ("%I64d\n", f1(12345678901234, 23456789012345));
#endif
};

uint64_t f2 (uint64_t a, uint64_t b)
{
    return a-b;
};
```

---

<sup>1</sup>By the way, 32-bit values are passed as pairs in 16-bit environment just as the same: [34.4](#)

## Listing 22.1: Optimizing MSVC 2012 /Ob1

```

_a$ = 8           ; size = 8
_b$ = 16          ; size = 8
_f1 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
_f1 ENDP

_f1_test PROC
    push   5461           ; 00001555H
    push   1972608889    ; 75939f79H
    push   2874          ; 00000b3aH
    push   1942892530    ; 73ce2ff2H
    call   _f1
    push   edx
    push   eax
    push   OFFSET $SG1436 ; '%I64d', 0aH, 00H
    call   _printf
    add     esp, 28
    ret     0
_f1_test ENDP

_f2 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
_f2 ENDP

```

We may see in the `f1_test()` function as each 64-bit value is passed by two 32-bit values, high part first, then low part.

Addition and subtraction occurring by pairs as well.

While addition, low 32-bit part are added first. If carry was occurred while addition, CF flag is set. The next ADC instruction adds high parts of values, but also adds 1 if  $CF = 1$ .

Subtraction is also occurred by pairs. The very first SUB may also turn CF flag on, which will be checked in the subsequent SBB instruction: if carry flag is on, then 1 will also be subtracted from the result.

In a 32-bit environment, 64-bit values are returned from a functions in EDX:EAX registers pair. It is easily can be seen how `f1()` function is then passed to `printf()`.

Listing 22.2: GCC 4.8.1 -O1 -fno-inline

```

_f1:
    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

_f1_test:
    sub     esp, 28
    mov     DWORD PTR [esp+8], 1972608889 ; 75939f79H
    mov     DWORD PTR [esp+12], 5461      ; 00001555H
    mov     DWORD PTR [esp], 1942892530  ; 73ce2ff2H
    mov     DWORD PTR [esp+4], 2874      ; 00000b3aH
    call    _f1
    mov     DWORD PTR [esp+4], eax
    mov     DWORD PTR [esp+8], edx
    mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\12\0"
    call    _printf
    add     esp, 28
    ret

_f2:
    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.

## 22.2 Multiplication, division

```

#include <stdint.h>

uint64_t f3 (uint64_t a, uint64_t b)
{
    return a*b;
};

uint64_t f4 (uint64_t a, uint64_t b)
{

```

```

    return a/b;
};

uint64_t f5 (uint64_t a, uint64_t b)
{
    return a % b;
};

```

Listing 22.3: Optimizing MSVC 2012 /Ob1

```

_a$ = 8           ; size = 8
_b$ = 16          ; size = 8
_f3 PROC
    push        DWORD PTR _b$[esp]
    push        DWORD PTR _b$[esp]
    push        DWORD PTR _a$[esp+8]
    push        DWORD PTR _a$[esp+8]
    call       __allmul ; long long multiplication
    ret        0
_f3 ENDP

_a$ = 8           ; size = 8
_b$ = 16          ; size = 8
_f4 PROC
    push        DWORD PTR _b$[esp]
    push        DWORD PTR _b$[esp]
    push        DWORD PTR _a$[esp+8]
    push        DWORD PTR _a$[esp+8]
    call       __aulldiv ; unsigned long long division
    ret        0
_f4 ENDP

_a$ = 8           ; size = 8
_b$ = 16          ; size = 8
_f5 PROC
    push        DWORD PTR _b$[esp]
    push        DWORD PTR _b$[esp]
    push        DWORD PTR _a$[esp+8]
    push        DWORD PTR _a$[esp+8]
    call       __aullrem ; unsigned long long remainder
    ret        0
_f5 ENDP

```

Multiplication and division is more complex operation, so usually, the compiler embedds calls to the library functions doing that.

These functions meaning are here: [D](#).

## Listing 22.4: Optimizing GCC 4.8.1 -fno-inline

```

_f3:
    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

_f4:
    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

_f5:
    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 doing almost the same, but multiplication code is inlined right in the function, thinking it could be more efficient. GCC has different library function names: C.

## 22.3 Shifting right

```
#include <stdint.h>

uint64_t f6 (uint64_t a)
{
    return a>>7;
};
```

Listing 22.5: Optimizing MSVC 2012 /Ob1

```
_a$ = 8          ; size = 8
_f6 PROC
    mov     eax, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    shrd   eax, edx, 7
    shr    edx, 7
    ret    0
_f6 ENDP
```

Listing 22.6: Optimizing GCC 4.8.1 -fno-inline

```
_f6:
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+4]
    shrd   eax, edx, 7
    shr    edx, 7
    ret
```

Shifting also occurring in two passes: first lower part is shifting, then higher part. But the lower part is shifting with the help of SHRD instruction, it shifting EDX value by 7 bits, but pulling new bits from EAX, i.e., from the higher part. Higher part is shifting using more popular SHR instruction: indeed, freed bits in the higher part should be just filled with zeroes.

## 22.4 Converting 32-bit value into 64-bit one

```
#include <stdint.h>

int64_t f7 (int64_t a, int64_t b, int32_t c)
{
    return a*b+c;
};

int64_t f7_main ()
```



```
{
    return f7(12345678901234, 23456789012345, 12345);
};
```

Listing 22.7: Optimizing MSVC 2012 /Ob1

```
_a$ = 8      ; size = 8
_b$ = 16     ; size = 8
_c$ = 24     ; size = 4
_f7 PROC
    push     esi
    push     DWORD PTR _b$[esp+4]
    push     DWORD PTR _b$[esp+4]
    push     DWORD PTR _a$[esp+12]
    push     DWORD PTR _a$[esp+12]
    call    __allmul ; long long multiplication
    mov     ecx, eax
    mov     eax, DWORD PTR _c$[esp]
    mov     esi, edx
    cdq     ; input: 32-bit value in EAX; output: 64-bit ↵
    ↵ value in EDX:EAX
    add     eax, ecx
    adc     edx, esi
    pop     esi
    ret     0
_f7 ENDP

_f7_main PROC
    push     12345      ; 00003039H
    push     5461       ; 00001555H
    push     1972608889 ; 75939f79H
    push     2874       ; 00000b3aH
    push     1942892530 ; 73ce2ff2H
    call    _f7
    add     esp, 20     ; 00000014H
    ret     0
_f7_main ENDP
```

Here we also run into necessity to extend 32-bit signed value from *c* into 64-bit signed. Unsigned values are converted straightforwardly: all bits in higher part must be set to 0. But it is not appropriate for signed data types: sign should be copied into higher part of resulting number. CDQ instruction doing that here, it takes input value in EAX, extending it to 64-bit and leaving it in the EDX:EAX registers pair. In other words, CDQ instruction gets number sign in EAX (by getting just most significant bit in EAX), and depending of it, setting all 32-bits in EDX to 0 or 1. Its operation is somewhat similar to the MOVSB instruction.

## Listing 22.8: Optimizing GCC 4.8.1 -fno-inline

```

_f7:
    push    edi
    push    esi
    push    ebx
    mov     esi, DWORD PTR [esp+16]
    mov     edi, DWORD PTR [esp+24]
    mov     ebx, DWORD PTR [esp+20]
    mov     ecx, DWORD PTR [esp+28]
    mov     eax, esi
    mul     edi
    imul   ebx, edi
    imul   ecx, esi
    mov     esi, edx
    add     ecx, ebx
    mov     ebx, eax
    mov     eax, DWORD PTR [esp+32]
    add     esi, ecx
    cdq    ; input: 32-bit value in EAX; output: 64-bit ↗
    ↙ value in EDX:EAX
    add     eax, ebx
    adc     edx, esi
    pop     ebx
    pop     esi
    pop     edi
    ret

_f7_main:
    sub     esp, 28
    mov     DWORD PTR [esp+16], 12345        ; 00003039H
    mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
    mov     DWORD PTR [esp+12], 5461        ; 00001555H
    mov     DWORD PTR [esp], 1942892530    ; 73ce2ff2H
    mov     DWORD PTR [esp+4], 2874        ; 00000b3aH
    call   _f7
    add     esp, 28
    ret

```

GCC generates just the same code as MSVC, but inlines multiplication code right in the function.

# Chapter 23

## SIMD

**SIMD**<sup>1</sup> is an acronym: *Single Instruction, Multiple Data*.

As it is said, it is multiple data processing using only one instruction.

As **FPU**, that **CPU** subsystem looks like separate processor inside x86.

SIMD began as MMX in x86. 8 new 64-bit registers appeared: MM0-MM7.

Each MMX register may 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 graphics editor, representing image as a two dimensional array. When user change image brightness, the editor must add a coefficient to each pixel value, or to subtract. For the sake of brevity, our image may be grayscale and each pixel defined by one 8-bit byte, then it is possible to change brightness of 8 pixels simultaneously.

When MMX appeared, these registers was actually located in FPU registers. It was possible to use either FPU or MMX at the same time. One might think, Intel saved on transistors, but in fact, the reason of such symbiosis is simpler —older **OS** may not aware of additional CPU registers would not save them at the context switching, but will save FPU registers. Thus, MMX-enabled CPU + old **OS** + process utilizing MMX features = that all will work together.

SSE—is extension of SIMD registers up to 128 bits, now separately from FPU.

AVX—another extension to 256 bits.

Now about practical usage.

Of course, memory copy routines (**memcpy**), memory comparing (**memcmp**) and so on.

One more example: we got DES encryption algorithm, it takes 64-bit block, 56-bit key, encrypt block and produce 64-bit result. DES algorithm may be considered as a very large electronic circuit, with wires and AND/OR/NOT gates.

---

<sup>1</sup>Single instruction, multiple data

Bitslice DES<sup>2</sup> – is an idea of processing group of blocks and keys simultaneously. Let's say, variable of type *unsigned int* on x86 may hold up to 32 bits, so, it is possible to store there intermediate results for 32 blocks-keys pairs simultaneously, using 64+56 variables of *unsigned int* type.

I wrote an utility to brute-force Oracle RDBMS passwords/ hashes (ones based on DES), slightly modified bitslice DES algorithm for SSE2 and AVX – now it is possible to encrypt 128 or 256 block-keys pairs simultaneously.

[http://conus.info/utils/ops\\_SIMD/](http://conus.info/utils/ops_SIMD/)

## 23.1 Vectorization

Vectorization<sup>3</sup>, for example, is when you have a loop taking couple of arrays at input and produces one array. Loop body takes values from input arrays, do something and put result into output array. It is important that there is only one single operation applied to each element. Vectorization – is to process several elements simultaneously.

Vectorization is not very fresh technology: author of this textbook saw it at least on Cray Y-MP supercomputer line from 1988 when played with its “lite” version Cray Y-MP EL<sup>4</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 save result into C.

If each array element we have is 32-bit *int*, then it is possible to load 4 elements from A into 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 count is 1024/4 instead of 1024, that is 4 times less and, of course, faster.

Some compilers can do vectorization automatically in a simple cases, e.g., Intel C++<sup>5</sup>.

I wrote tiny function:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
```

<sup>2</sup><http://www.darkside.com.au/bitslice/>

<sup>3</sup>[Wikipedia: vectorization](#)

<sup>4</sup>Remotely. It is installed in the museum of supercomputers: <http://www.cray-cyber.org>

<sup>5</sup>More about Intel C++ automatic vectorization: [Excerpt: Effective Automatic Vectorization](#)

```
{
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];

    return 0;
};
```

### 23.1.1 Intel C++

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 = ar1
and     edi, 0Fh          ; is ar1 16-byte aligned?
jz     short loc_9A       ; yes
test    edi, 3
jnz    loc_162
neg     edi
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]
    movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not ↙
    ↘ 16-byte aligned, so load it to xmm0
    padd    xmm1, xmm0
    movdqa xmmword ptr [eax+edi*4], xmm1
    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]
    padd    xmm0, xmmword ptr [esi+edi*4]
    movdqa 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]
    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

```

SSE2-related instructions are:

- **MOVDQU** (*Move Unaligned Double Quadword*)— it just load 16 bytes from memory into a XMM-register.
- **PADDQ** (*Add Packed Integers*)— adding 4 pairs of 32-bit numbers and leaving result in first operand. By the way, no exception raised in case of overflow and



no flags will be set, just low 32-bit of result will be stored. If one of PADD operands is address of value in memory, then address must be aligned on a 16-byte boundary. If it is not aligned, exception will be occurred <sup>6</sup>.

- **MOVDQA** (*Move Aligned Double Quadword*)— the same as **MOVDQU**, but requires address of value in memory to be aligned on a 16-bit border. If it is not aligned, exception will be raised. **MOVDQA** works faster than **MOVDQU**, but requires aforesaid.

So, these SSE2-instructions will be executed only in case if there are more 4 pairs to work on plus pointer `ar3` is aligned on a 16-byte boundary.

More than that, if `ar2` is aligned on a 16-byte boundary as well, this fragment of code will be executed:

```
movdqu xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
padd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

Otherwise, value from `ar2` will be loaded into **XMM0** using **MOVDQU**, it 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
padd   xmm1, xmm0
movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

In all other cases, non-SSE2 code will be executed.

### 23.1.2 GCC

GCC may also vectorize in a simple cases<sup>7</sup>, if to use `-O3` option and to turn on SSE2 support: `-msse2`.

What we got (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
```

<sup>6</sup>More about data aligning: [Wikipedia: Data structure alignment](http://en.cppreference.com/w/cpp/string/basic/basic_string_view)

<sup>7</sup>More about GCC vectorization support: <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>

```

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
        retn

        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

```

```

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
    padd  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++ doing it.

## 23.2 SIMD strlen() implementation

It should be noted the SIMD-instructions may be inserted into C/C++ code via special macros<sup>8</sup>. As of MSVC, some of them are located in the `intrin.h` file.

It is possible to implement `strlen()` function<sup>9</sup> using SIMD-instructions, working 2-2.5 times faster than common implementation. This function will load 16 characters into a XMM-register and check each against zero<sup>10</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;

    if (str_is_aligned==false)

```

<sup>8</sup>MSDN: MMX, SSE, and SSE2 Intrinsics

<sup>9</sup>`strlen()` – standard C library function for calculating string length

<sup>10</sup>The example is based on source code from: [http://www.strchr.com/sse2\\_optimised\\_strlen](http://www.strchr.com/sse2_optimised_strlen).

```

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 in MSVC 2010 with /Ox option:

Listing 23.1: 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         ; 0000000cH
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 ; 00000010H
    pcmpeqb xmm1, xmm0
    add     edx, 16 ; 00000010H
    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@@YAIPBD@Z ENDP ; strlen_sse2

```

First of all, we check `str` pointer, if it is aligned on a 16-byte boundary. If not, let's call generic `strlen()` implementation.

Then, load next 16 bytes into the XMM1 register using `MOVDQA` instruction.

Observant reader might ask, why `MOVDQU` cannot be used here since it can load data from the memory regardless pointer alignment?

Yes, it might be done in this way: if pointer is aligned, load data using `MOVDQA`, if not — use slower `MOVDQU`.

But here we are may stick into hard to notice caveat:

In [Windows NT](#) line of [OS](#) (but not limited to it), memory allocated by pages of 4 KiB (4096 bytes). Each win32-process has ostensibly 4 GiB, but in fact, only some parts of address space are connected to real physical memory. If the process accessing to the absent memory block, exception will be raised. That's how [VM](#)

works<sup>11</sup>.

So, a function loading 16 bytes at once, may step over a border of allocated memory block. Let's consider, OS allocated 8192 (0x2000) bytes at the 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., OS not allocated any memory there. Attempt to access a memory starting from the address will raise exception.

And let's consider, the program holding a string containing 5 characters almost at the end of 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 common conditions the program calling `strlen()` passing it a pointer to string 'hello' lying in memory at address 0x008c1ff8. `strlen()` will read one byte at a time until 0x008c1ffd, where zero-byte, and so here it will stop working.

Now if we implement our own `strlen()` reading 16 byte at once, starting at any address, will it be aligned or not, `MOVDQU` may attempt to load 16 bytes at once at address 0x008c1ff8 up to 0x008c2008, and then exception will be raised. That's the situation to be avoided, of course.

So then we'll work only with the addresses aligned on a 16 byte boundary, what in combination with a knowledge of OS page size is usually aligned on a 16-byte boundary too, give us some warranty our function will not read from unallocated memory.

Let's back to our function.

`_mm_setzero_si128()` – is a macro generating `pxor xmm0, xmm0` – instruction just clears the XMM0 register.

`_mm_load_si128()` – is a macro for `MOVDQA`, it just loading 16 bytes from the address into the XMM1 register.

`_mm_cmpeq_epi8()` – is a macro for `PCMPEQB`, is an instruction comparing two XMM-registers bitwise.

And if some byte was equals to other, there will be 0xff at this point in the result or 0 if otherwise.

For example.

XMM1: 11223344556677880000000000000000

<sup>11</sup>[http://en.wikipedia.org/wiki/Page\\_\(computer\\_memory\)](http://en.wikipedia.org/wiki/Page_(computer_memory))

XMM0: 11ab3444007877881111111111111111

After `pcmpeqb xmm1, xmm0` execution, the XMM1 register shall contain:

XMM1: ff0000ff0000ffff0000000000000000

In our case, this instruction comparing each 16-byte block with the block of 16 zero-bytes, was set in the XMM0 register by `pxor xmm0, xmm0`.

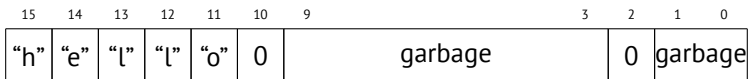
The next macro is `_mm_movemask_epi8()` –that is `PMOVMSKB` instruction. It is very useful if to use it with `PCMPEQB`.

`pmovmskb eax, xmm1`

This instruction will set first EAX bit into 1 if most significant bit of the first byte in the XMM1 is 1. In other words, if first byte of the XMM1 register is `0xff`, first EAX bit will be set to 1 too.

If second byte in the XMM1 register is `0xff`, then second EAX bit will be set to 1 too. In other words, the instruction is answering to the question “which bytes in the XMM1 are `0xff`?” And will prepare 16 bits in the EAX register. Other bits in the EAX register are to be cleared.

By the way, do not forget about this feature of our algorithm. There might be 16 bytes on input like:



It is a 'hello' string, terminating zero, and also a random noise in memory. If we load these 16 bytes into XMM1 and compare them with zeroed XMM0, we will get something like <sup>12</sup>:

XMM1: 0000ff00000000000000ff0000000000

This means, the instruction found two zero bytes, and that is not surprising.

`PMOVMSKB` in our case will prepare EAX like (in binary representation): `00100000001000`

Obviously, our function must take only first zero bit and ignore the rest ones.

The next instruction—`BSF` (*Bit Scan Forward*). This instruction find first bit set to 1 and stores its position into first operand.

EAX=0010000000100000b

After `bsf eax, eax` instruction execution, EAX will contain 5, this means, 1 found at 5th bit position (starting from zero).

MSVC has a macro for this instruction: `_BitScanForward`.

Now it is simple. If zero byte found, its position added to what we already counted and now we have ready to return result.

Almost all.

<sup>12</sup>I use here order from **MSB** to **LSB**<sup>13</sup>



By the way, it is also should be noted, MSVC compiler emitted two loop bodies side by side, for optimization.

By the way, SSE 4.2 (appeared in Intel Core i7) offers more instructions where these string manipulations might be even easier: [http://www.strchr.com/strcmp\\_and\\_strlen\\_using\\_sse\\_4.2](http://www.strchr.com/strcmp_and_strlen_using_sse_4.2)

# Chapter 24

## 64 bits

### 24.1 x86-64

It is a 64-bit extension to x86-architecture.

From the reverse engineer's perspective, most important changes are:

- Almost all registers (except FPU and SIMD) are extended to 64 bits and got R- prefix. 8 additional registers 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 to *older* register parts as usual. For example, it is possible to access lower 32-bit part of the RAX register using EAX:

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

New R8-R15 registers also has its *lower parts*: R8D-R15D (lower 32-bit parts), R8W-R15W (lower 16-bit parts), R8L-R15L (lower 8-bit parts).

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R8							
						R8D	
						R8W	
						R8L	

SIMD-registers number is doubled from 8 to 16: XMM0-XMM15.

- In Win64, function calling convention is slightly different, somewhat resembling fastcall (49.3). First 4 arguments stored in the RCX, RDX, R8, R9 registers, others –in the stack. Caller function must also allocate 32 bytes so the callee may save there 4 first arguments and use these registers for own needs. Short functions may use arguments just from registers, but larger may save their values on the stack.

System V AMD64 ABI (Linux, \*BSD, Mac OS X)[Mit13] also somewhat resembling fastcall, it uses 6 registers RDI, RSI, RDX, RCX, R8, R9 for the first 6 arguments. All the rest are passed in the stack.

See also section about calling conventions (49).

- C/C++ *int* type is still 32-bit for compatibility.
- All pointers are 64-bit now.

This provokes irritation sometimes: now one need twice as much memory for storing pointers, including, cache memory, despite the fact x64 CPUs can address only 48 bits of external RAM.

Since now register number is doubled, compilers has more space now for maneuvering called [register allocation](#). What it meanings for us, emitted code will contain less local variables.

For example, function calculating first S-box of DES encryption algorithm, it processing 32/64/128/256 values at once (depending on `DES_type` type (uint32, uint64, SSE2 or AVX)) using bitslice DES method (read more about this technique here (23)):

```

/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for ↵
 *   ↵ any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _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;
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 is a lot of local variables. Of course, not all those will be in local stack. Let's compile it with MSVC 2008 with /Ox option:

Listing 24.1: 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
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 ; 00000014H

```



```

    ret    0
_s1    ENDP

```

5 variables was allocated in local stack by compiler.  
 Now let's try the same thing in 64-bit version of MSVC 2008:

Listing 24.2: 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
$LN3:
    mov     QWORD PTR [rsp+24], rbx
    mov     QWORD PTR [rsp+32], rbp
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    push   rsi
    push   rdi
    push   r12
    push   r13
    push   r14
    push   r15
    mov     r15, QWORD PTR a5$[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
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    rbx, 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 allocated in local stack by compiler, x36 is synonym for a5.

By the way, there are CPUs with much more [GPR](#)'s, e.g. Itanium (128 registers).

## 24.2 ARM

64-bit instructions were appeared in ARMv8.

## 24.3 Float point numbers

How float point numbers are processed in x86-64, read more here: [25](#).

## Chapter 25

# Working with float point numbers using SIMD

Of course, [FPU](#) remained in x86-compatible processors, when [SIMD](#) extensions were added.

[SIMD](#)-extensions (SSE2) offers more easy way to work with float-point numbers. Number format remaining the same (IEEE 754).

So, modern compilers (including those generating for x86-64) usually uses [SIMD](#)-instructions instead of FPU ones.

It can be said, it's a good news, because it's easier to work with them.

We will reuse here examples from the FPU section: [16](#).

### 25.1 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));
};
```

#### 25.1.1 x64

## Listing 25.1: 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
f    ENDP

```

Input floating point values are passed in XMM0-XMM3 registers, all the rest—via stack<sup>1</sup>.

*a* is passed in XMM0, *b*—via XMM1. XMM-registers are 128-bit (as we know from the section about [SIMD: 23](#)), but *double* values—64 bit ones, so only lower register half is used.

DIVSD is SSE-instruction, meaning “Divide Scalar Double-Precision Floating-Point Values”, it just divides one value of *double* type by another, stored in the lower halves of operands.

Constants are encoded by compiler in IEEE 754 format.

MULSD and ADDSD works just as the same, but doing multiplication and addition.

The result of *double* type the function leaves in XMM0 register.

That is how non-optimizing MSVC works:

## Listing 25.2: MSVC 2012 x64

```

__real@4010666666666666 DQ 0401066666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14

a$ = 8
b$ = 16
f    PROC
    movsdx   QWORD PTR [rsp+16], xmm1
    movsdx   QWORD PTR [rsp+8], xmm0
    movsdx   xmm0, QWORD PTR a$[rsp]
    divsd    xmm0, QWORD PTR __real@40091eb851eb851f
    movsdx   xmm1, QWORD PTR b$[rsp]
    mulsd    xmm1, QWORD PTR __real@4010666666666666
    addsd    xmm0, xmm1
    ret      0
f    ENDP

```

<sup>1</sup>[MSDN: Parameter Passing](#)

Slightly redundant. Input arguments are saved in “shadow space” (7.2.1), but only lower register halves, i.e., only 64-bit values of *double* type.

GCC produces very same code.

## 25.1.2 x86

I also compiled this example for x86. Despite the fact of generating for x86, MSVC 2012 use SSE2-instructions:

Listing 25.3: 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 25.4: Optimizing MSVC 2012 x86

```

tv67 = 8      ; size = 8
_a$ = 8       ; size = 8
_b$ = 16      ; size = 8
_f          PROC
    movsd   xmm1, QWORD PTR _a$[esp-4]
    divsd   xmm1, QWORD PTR __real@40091eb851eb851f
    movsd   xmm0, QWORD PTR _b$[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 couple differences related to calling conventions: 1) arguments are passed not in XMM registers, but in stack, like in FPU examples (16); 2) function result is returned in ST(0) – in order to do so, it's copied (through local variable tv) from one of XMM-registers into ST(0).

Let's try optimized example in OllyDbg: fig.25.1, fig.25.2, fig.25.3, fig.25.4, fig.25.5.

We see that OllyDbg shows XMM-registers as *double* number pairs, but only *lower* part is used. Apparently, OllyDbg shows them in that format because SSE2-instructions (suffixed with -SD) are executed right now. But of course, it's possible to switch register format and to see its contents as 4 *float*-numbers or just as 16 bytes.

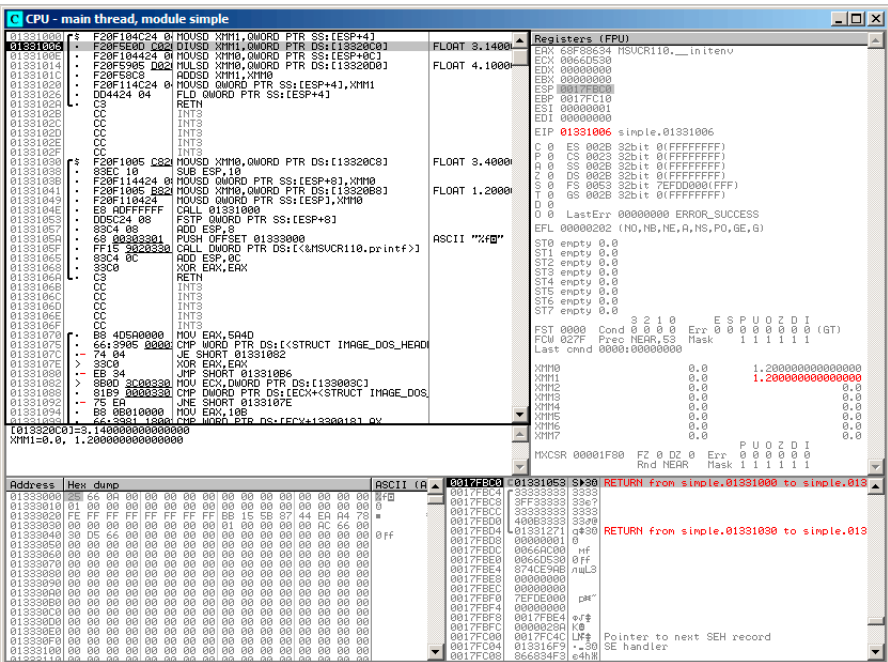


Figure 25.1: OllyDbg: MOVSD loads value of *a* into XMM1



The screenshot displays the OllyDbg interface with the following components:

- Assembly Window:** Shows assembly instructions such as `F20F10424 01 HOUNO XMM0, OMWORD PTR SS:[ESP+4]`, `F20F58C ADDSD XMM1, XMM0`, and `DD424 04 FLD OMWORD PTR SS:[ESP+4]`. It also shows control flow instructions like `CC INT3` and `CC INT3`.
- Registers (FPU) Window:** Shows the state of floating-point registers. XMM0 contains the value `0.382155609955414`. Other registers like XMM1 through XMM7 are empty.
- Memory Dump Window:** Shows a hex dump of memory starting at address `01133000`. The dump contains several zero bytes and some non-zero values, with ASCII characters like `0` and `0ff` visible.

Figure 25.2: OllyDbg: DIVSD calculated quotient and stored it in XMM1

The screenshot shows the OllyDbg interface with the following components:

- Assembly Window:** Displays assembly instructions such as `F20F10424 01 HOUND XMM1, OMWORD PTR SS:[ESP+4]`, `F20F5E00 C021 D1USD XMM1, OMWORD PTR DS:[13320C0]`, and `F20F10424 01 HOUND XMM0, OMWORD PTR SS:[ESP+0C]`. It also shows control flow instructions like `DD424 04 FLD OMWORD PTR SS:[ESP+4]` and `3C0D JNE SHORT 01331062`.
- Registers (FPU) Window:** Shows the state of floating-point registers. `MM0` contains the value `13.94000000000000`, which is the product of the calculation. Other registers like `MM1` through `MM7` are empty.
- Memory Dump Window:** Shows a hex dump of memory starting at address `01331000`. The dump contains several null bytes (`00`) and some non-zero values, including the ASCII string `RETURN from simple.01331000 to simple.01331000` and `RETURN from simple.01331030 to simple.01331030`.

Figure 25.3: OllyDbg: MULSD calculated product and stored it in XMM0

The screenshot displays the OllyDbg interface with three main panes:

- Assembly Window:** Shows assembly instructions. A key instruction is `ADDSD [EAX], XMM0` at address `01331053`. Other instructions include `F20F114424 0 MOVSD XMM0, OMWORD PTR DS:[ESP+4]`, `F20F1095 C32 MOVSD XMM0, OMWORD PTR DS:[13320C]`, and `F20F114424 0 MOVSD XMM0, OMWORD PTR SS:[ESP+0], XMM0`.
- Registers (FPU) Window:** Shows the state of floating-point registers. `XMM0` contains the value `14.32216560509554`. Other registers like `XMM1` through `XMM7` are empty or zero.
- Memory Dump Window:** Shows the memory at address `0017FBC4` containing the value `14.32216560509554`.

Figure 25.4: OllyDbg: ADDSD adds value in XMM0 to XMM1

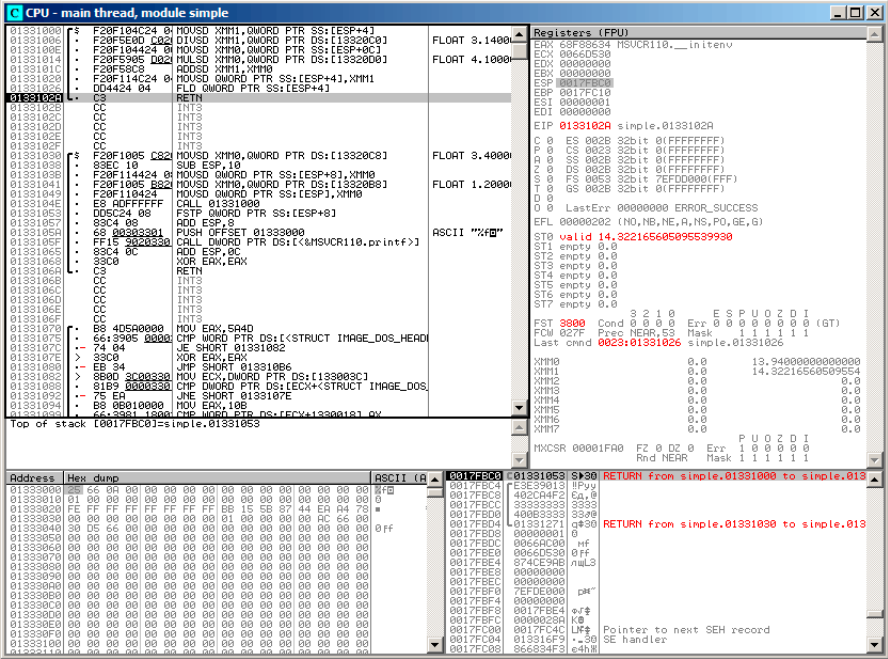


Figure 25.5: OllyDbg: FLD left function result in ST(0)

## 25.2 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 lower halves of the XMM0-XMM3 registers.

Listing 25.5: 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
    movsdx xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
    movsdx 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 are no MOVSDX instruction in Intel [Int13] and AMD [AMD13a] manuals, it is called there just MOVSD. So there are two instructions sharing the same name in x86 (about other: A.6.2). Apparently, Microsoft developers wanted to get rid of mess, so they renamed it into MOVSDX. It just loads a value into lower half of XMM-register.

`pow()` takes arguments from XMM0 and XMM1, and returning result in XMM0. It is then moved into RDX for `printf()`. Why? Honestly speaking, I don't know, maybe because `printf()`—is a variable arguments function?

Listing 25.6: 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 makes more clear result. Value for `printf()` is passed in `XMM0`. By the way, here is a case when 1 is written into `EAX` for `printf()`—this mean that one argument will be passed in vector registers, just as the standard requires [Mit13].

## 25.3 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));
};
```

### 25.3.1 x64

Listing 25.7: 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 25.8: 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 optimizing and used the MAXSD (“Return Maximum Scalar Double-Precision Floating-Point Value”) instruction, which just choose maximal value!

Listing 25.9: Optimizing GCC 4.4.6 x64

```

d_max:
    maxsd   xmm0, xmm1
    ret

```

## 25.3.2 x86

Let’s compile this example in MSVC 2012 with optimization turned on:

Listing 25.10: 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 values of  $a$  and  $b$  are taken from stack and function result is left in ST(0).

If to load this example in OllyDbg, we will see how COMISD instruction compares values and set/clear CF and PF flags: fig.25.6.

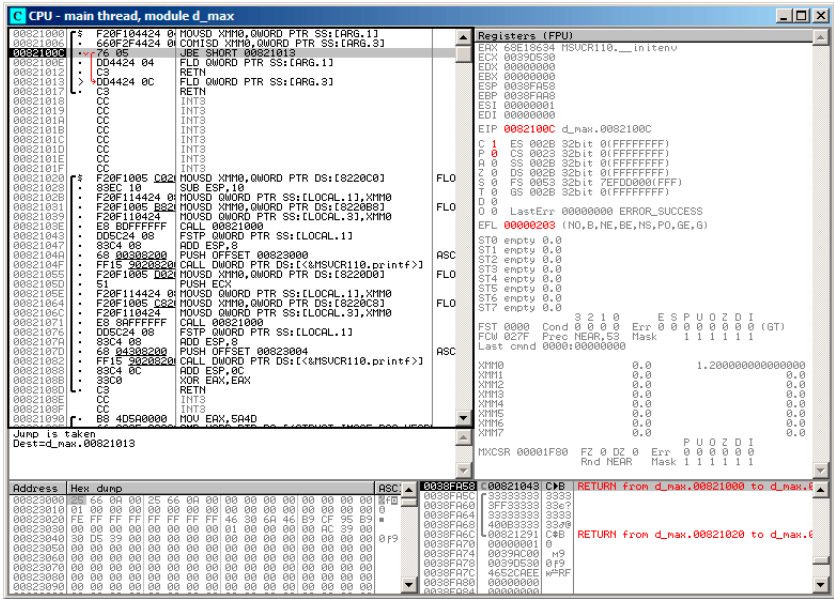


Figure 25.6: OllyDbg: COMISD changed CF and PF flags

## 25.4 Summary

Only lower half of XMM-registers are used in all examples here, a number in IEEE 754 format is stored there.

Essentially, all instructions prefixed by *-SD* (“Scalar Double-Precision”)—are instructions working with float point numbers in IEEE 754 format stored in the lower 64-bit half of XMM-register.

And it is easier than FPU, probably because SIMD extensions were evolved not as chaotic as FPU in the past. Stack register model is not used.

If you would try to replace *double* to *float* in these examples, the same instructions will be used, but prefixed with *-SS* (“Scalar Single-Precision”), for example, MOVSS, COMISS, ADDSS, etc.

“Scalar” mean that SIMD-register will contain only one value instead of several. Instructions working with several values in a register simultaneously, has “Packed” in the name.

Needless to say that SSE2-instructions works with 64-bit IEEE 754 numbers (*double*), while internal representation of float-point-numbers in FPU — 80-bit numbers. Hence, FPU may produce less round-off errors, as a consequence, FPU may



give more precise calculation results.

# Chapter 26

## Temperature converting

Another very popular example in programming books for beginners, is a small program converting Fahrenheit temperature to Celsius or back.

$$C = \frac{5 \cdot (F - 32)}{9}$$

I also added simple error handling: 1) we should check if user enters correct number; 2) we should check if Celsius temperature is not below  $-273$  number (which is below absolute zero, as we may remember from school physics lessons).

`exit()` function terminates program instantly, without returning to the [caller](#) function.

### 26.1 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");
        exit(0);
    };
    printf ("Celsius: %d\n", celsius);
};

```

## 26.1.1 Optimizing MSVC 2012 x86

Listing 26.1: 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                        ; ↗
↳ 0000000cH
    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                        ; ↗
↳ ffffffff0H
    lea    ecx, DWORD PTR [eax+eax*4]

```

```

    mov     eax, 954437177                ; 38↵
↵ e38e39H
    imul   ecx
    sar    edx, 1
    mov    eax, edx
    shr    eax, 31                       ; ↵
↵ 0000001fH
    add    eax, edx
    cmp    eax, -273                     ; ↵
↵ fffffeefH
    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 - at least by C99 standard
    xor    eax, eax
    pop    esi
    pop    ecx
    ret    0
$LN8@main:
_main   ENDP

```

What we can say about it:

- Address of `printf()` is first loaded into ESI register, so the subsequent `printf()` calls are processed just by `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 are free register which can be used for this.
- We see `ADD EAX, -32` instruction at the place where 32 should be subtracted from the value.  $EAX = EAX + (-32)$  is equivalent to  $EAX = EAX - 32$  and somehow, compiler decide to use `ADD` instead of `SUB`. Maybe it's worth it, but I'm not sure.
- `LEA` instruction is used when value should be multiplied by 5: `lea ecx, DWORD PTR [eax+eax*4]`. Yes,  $i + i * 4$  is equivalent to  $i * 5$  and `LEA` works faster then `IMUL`. By the way, `SHL EAX, 2 / ADD EAX, EAX`

instructions pair could be also used here instead— some compilers do it in this way.

- Division by multiplication trick (15.3) is also used here.
- `main()` function returns 0 while we haven't return 0 at its end. C99 standard tells us [ISO07, p. 5.1.2.2.3] that `main()` will return 0 in case of return statement absence. This rule works only for `main()` function. Though, MSVC doesn't officially support C99, but maybe its support partial?

## 26.1.2 Optimizing MSVC 2012 x64

The code is almost the same, but I've found `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 functions which never can return <sup>1</sup>, so if it does, something really odd happens and it's time to load debugger.

## 26.2 Float 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);
    }
}
```

<sup>1</sup>another popular one is `longjmp()`

```
};
printf ("Celsius: %lf\n", celsius);
};
```

MSVC 2010 x86 use [FPU](#) instructions...

Listing 26.2: 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@c071100000000000 DQ 0c071100000000000r ; -273
__real@4022000000000000 DQ 04022000000000000r ; 9
__real@4014000000000000 DQ 04014000000000000r ; 5
__real@4040000000000000 DQ 04040000000000000r ; 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 ; ↗
↳ 0000000cH
    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 ; ↗
↳ 0000000cH
    ; return 0
    xor eax, eax
    pop esi
    add esp, 8
    ret 0
$LN10@main:
_main ENDP

```

...but MSVC from year 2012 use [SIMD](#) instructions instead:

Listing 26.3: 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
__real@c071100000000000 DQ 0c07110000000000r ; -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                  ; ↵
↳ 0000000cH
    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                  ; ↵
↳ 0000000cH
    ; return 0
    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 use them.

We may also notice that  $-273$  value is loaded into XMM0 register too early. And that's OK, because, compiler may emit instructions not in the order they are in source code.

# Chapter 27

## Fibonacci numbers

Another often used in programming textbooks example is a recursive function generating Fibonacci numbers<sup>1</sup>. The sequence is very simple: each consecutive number is a sum of two previous. First two numbers are 1's or 0, 1 and 1.

The beginning of the sequence is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...

### 27.1 Example #1

Implementation is simple. This program generates a sequence till 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);
};
```

---

<sup>1</sup><http://oeis.org/A000045>

## Listing 27.1: MSVC 2010 x86

```

_TEXT SEGMENT
_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 $SG2750 ; "%d"
    call   DWORD PTR __imp__printf
    add     esp, 8
    mov     ecx, DWORD PTR _limit$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp]
    add     edx, DWORD PTR _b$[ebp]
    push    edx
    mov     eax, DWORD PTR _b$[ebp]
    push    eax
    call   _fib
    add     esp, 12
    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
_main ENDP

```

So I wanted to illustrate stack frames by this. Let's load the example to OllyDbg and trace to the latest call of `f()` function: [fig.27.1](#).

Let's investigate stack more closely. I added some comments to it <sup>2</sup>:

```

0035F940  00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F944  00000008 argument #1: a
0035F948  0000000D argument #2: b
0035F94C  00000014 argument #3: limit
0035F950  /0035F964 saved EBP register
0035F954  |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F958  |00000005 argument #1: a
0035F95C  |00000008 argument #2: b
0035F960  |00000014 argument #3: limit
0035F964  ]0035F978 saved EBP register
0035F968  |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F96C  |00000003 argument #1: a
0035F970  |00000005 argument #2: b
0035F974  |00000014 argument #3: limit
0035F978  ]0035F98C saved EBP register
0035F97C  |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F980  |00000002 argument #1: a
0035F984  |00000003 argument #2: b
0035F988  |00000014 argument #3: limit
0035F98C  ]0035F9A0 saved EBP register
0035F990  |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F994  |00000001 argument #1: a
0035F998  |00000002 argument #2: b
0035F99C  |00000014 argument #3: limit
0035F9A0  ]0035F9B4 saved EBP register
0035F9A4  |00FD105C RETURN to fib.00FD105C from fib.00FD1000
0035F9A8  |00000001 argument #1: a
0035F9AC  |00000001 argument #2: b | prepared in ↗
    ↘ main() for f1( )
0035F9B0  |00000014 argument #3: limit /
0035F9B4  ]0035F9F8 saved EBP register
0035F9B8  |00FD11D0 RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC  |00000001 main() argument #1: argc \
0035F9C0  |006812C8 main() argument #2: argv | prepared in ↗
    ↘ CRT for main( )
0035F9C4  |00682940 main() argument #3: envp /

```

The function is recursive <sup>3</sup>, hence stack looks like “sandwich”. We see that *limit* argument is always the same (0x14 or 20), but *a* and *b* arguments are different for each call. There are also *RA*-s and saved EBP values. OllyDbg is able to determine EBP-based frames, so it draws these brackets. Values inside of each bracket are **stack frame**, in other words, stack area which each function incarnation can use

<sup>2</sup>By the way, it's possible to select several entries in OllyDbg and copy them to clipboard (Ctrl-C). That's what I just did.

<sup>3</sup>i.e., calling itself

as scratch space. We can also say that each function incarnation must not access stack elements beyond frame boundaries (excluding function arguments), although it's technically possible. It's usually true, unless function has bugs. Each saved EBP value is an address of previous stack frame: it is a reason why some debuggers can easily divide stack by frames and dump each function's arguments.

As we see here, each function incarnation prepares arguments for the next function call.

At the very end we see a 3 arguments for main(). argc is 1 (yes, indeed, the program I run without command-line arguments).

It's easy to do stack overflow: just remove (or comment) limit check and it will crash with exception 0xC00000FD (stack overflow).

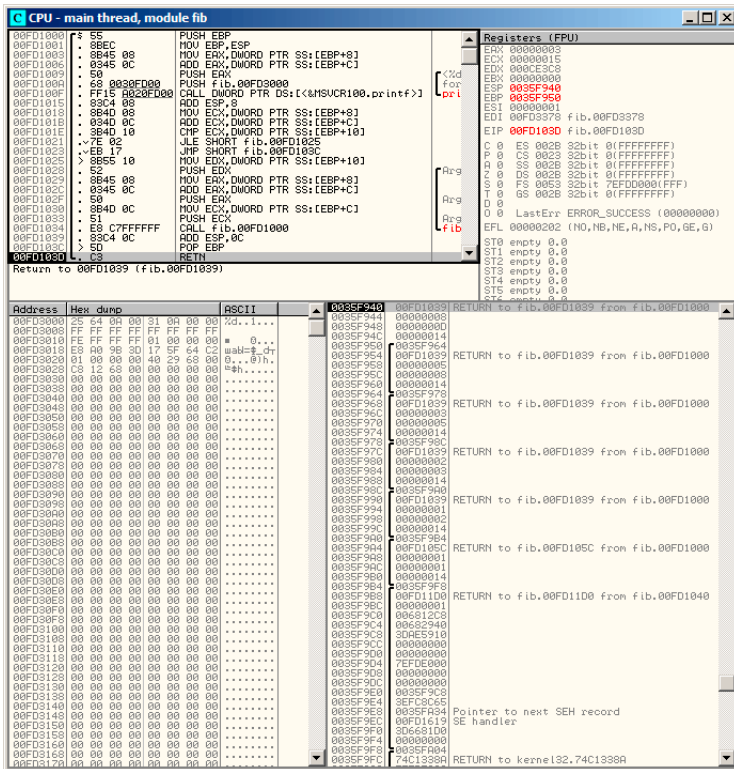


Figure 27.1: OllyDbg: last call of f()

## 27.2 Example #2

My function has some amount of redundancy, so let's add new local variable *next* and replace all "a+b" by it:

```
#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 output of non-optimizing MSVC, so *next* variable is actually allocated in local stack:

Listing 27.2: 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
_main   ENDP

```

Let's load OllyDbg again: [fig.27.2](#). Now *next* variable is present in each frame. Let's investigate the stack more closely. I added my comments again:

```

0029FC14  00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC18  00000008  argument #1: a
0029FC1C  0000000D  argument #2: b
0029FC20  00000014  argument #3: limit
0029FC24  0000000D  "next" variable
0029FC28  /0029FC40  saved EBP register
0029FC2C  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC30  |00000005  argument #1: a
0029FC34  |00000008  argument #2: b
0029FC38  |00000014  argument #3: limit
0029FC3C  |00000008  "next" variable
0029FC40  ]0029FC58  saved EBP register
0029FC44  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC48  |00000003  argument #1: a
0029FC4C  |00000005  argument #2: b
0029FC50  |00000014  argument #3: limit

```

```

0029FC54 |00000005 "next" variable
0029FC58 ]0029FC70 saved EBP register
0029FC5C |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC60 |00000002 argument #1: a
0029FC64 |00000003 argument #2: b
0029FC68 |00000014 argument #3: limit
0029FC6C |00000003 "next" variable
0029FC70 ]0029FC88 saved EBP register
0029FC74 |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC78 |00000001 argument #1: a \
0029FC7C |00000002 argument #2: b | prepared in f1 ↵
    ↵ () for next f1()
0029FC80 |00000014 argument #3: limit /
0029FC84 |00000002 "next" variable
0029FC88 ]0029FC9C saved EBP register
0029FC8C |00E0106C RETURN to fib2.00E0106C from fib2.00E01000
0029FC90 |00000001 argument #1: a \
0029FC94 |00000001 argument #2: b | prepared in ↵
    ↵ main() for f1()
0029FC98 |00000014 argument #3: limit /
0029FC9C ]0029FCE0 saved EBP register
0029FCA0 |00E011E0 RETURN to fib2.00E011E0 from fib2.00E01050
0029FCA4 |00000001 main() argument #1: argc \
0029FCA8 |000812C8 main() argument #2: argv | prepared in ↵
    ↵ CRT for main()
0029FCAC |00082940 main() argument #3: envp /

```

Here we see it: *next* value is calculated in each function incarnation, then passed as *b* argument to the next incarnation.



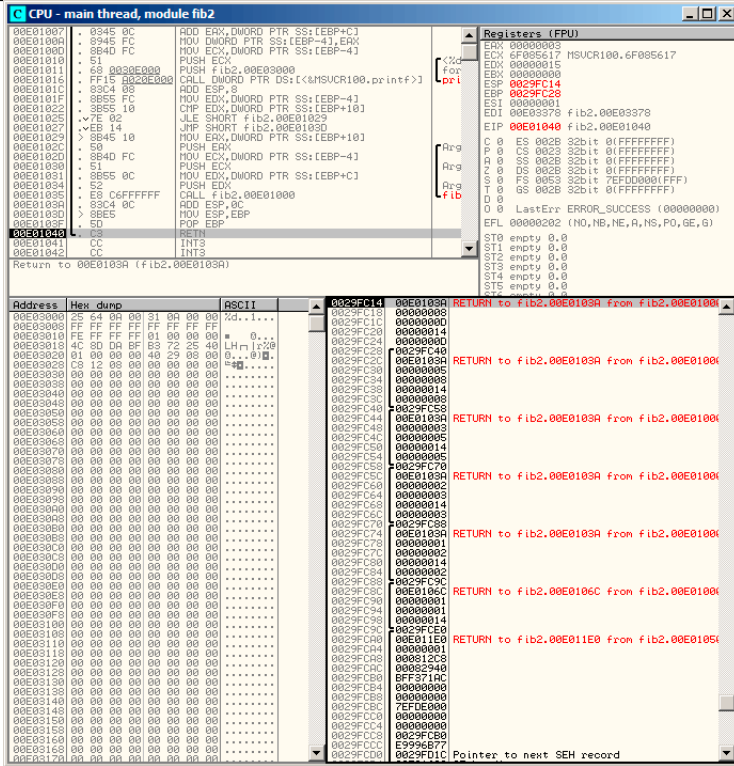


Figure 27.2: OllyDbg: last call of f()

## 27.3 Summary

Recursive functions are aesthetically nice, but technically may degrade performance because of heavy stack usage. One who write performance critical code pieces, probably, should avoid recursion there.

# Chapter 28

## C99 restrict

Here is a reason why FORTRAN programs, in some cases, works 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: pointer to `update_me` array could be a pointer to `sum` array, `product` array, or even `sum_product` array—since it is not a crime in it, right?

Compiler is fully aware about it, so it generates a code with four stages in 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 already calculated `sum[i]` and `product[i]`

Is it possible to optimize the last stage? Since already calculated `sum[i]` and `product[i]` are not necessary to load from memory again, because we already

calculated them. Yes, but compiler is not sure that nothing was overwritten on 3rd stage! This is called “pointer aliasing”, a situation, when compiler cannot be sure that a memory to which a pointer is pointing, was not changed.

*restrict* in C99 standard[ISO07, pp. 6.7.3/1] is a promise, given by programmer to compiler that the function arguments marked by this keyword will always be pointing to different memory locations and never be crossed.

If 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 argument-pointer:

```
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 28.1: GCC x64: f1()

```
f1:
    push    r15 r14 r13 r12 rbp rdi rsi rbx
    mov     r13, QWORD PTR 120[rsi]
    mov     rbp, QWORD PTR 104[rsi]
    mov     r12, QWORD PTR 112[rsi]
    test    r13, r13
    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 28.2: 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 compiled `f1()` and `f2()` function is as follows: in `f1()`, `sum[i]` and `product[i]` are reloaded in the middle of loop, and in `f2()` there are no such thing, already calculated values are used, since we “promised” to compiler, that no one and nothing will change values in `sum[i]` and `product[i]` while execution of loop body, so it is “sure” the value from memory may not be loaded again. Obviously, second example will work faster.

But what if pointers in function arguments will be crossed somehow? This will be on programmer’s conscience, and results may be incorrect.

Let’s 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 may generate faster code in these cases.

How practical is it? In the cases when function works with several big blocks in memory. There are a lot of such in linear algebra, for instance. A lot of linear algebra used on supercomputers/[HPC<sup>1</sup>](#), probably, that is why, traditionally, FORTRAN is still used there [[Loh10](#)].

But when a number of iterations is not very big, certainly, speed boost will not be significant.

---

<sup>1</sup>High-Performance Computing

## Chapter 29

# Inline functions

Inlined code is when compiler, instead of placing call instruction to a small or tiny function, just placing its body right in-place.

Listing 29.1: 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 to turn on GCC optimization (-O3), we'll see:

Listing 29.2: 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

```

(Here division is done by multiplication(15.3).)

Yes, our small function `celsius_to_fahrenheit()` was just placed before `printf()` call. Why? It may be faster than executing this function's code plus calling/returning overhead.

In past, such function must be marked with "inline" keyword in function's declaration, however, in modern times, these functions are automatically chosen by compiler.

## 29.1 Strings and memory functions

Another very common automatic optimization tactic is inlining of string functions like `strcpy()`, `strcmp()`, `strlen()`, `memcpy()`, `memcopy()`, etc.

Sometimes it's faster then to call separate function.

These are very frequent patterns, which are highly advisable to reverse engineers to learn to detect automatically.

### 29.1.1 strcmp()

Listing 29.3: `strcmp()` example

```

bool is_bool (char *s)
{
    if (strcmp (s, "true")==0)
        return true;
    if (strcmp (s, "false")==0)
        return false;

    assert(0);
};

```

## Listing 29.4: 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
    ret

.L3:
    add    esp, 20
    mov    eax, 1
    pop    esi
    pop    edi
    ret

```

## Listing 29.5: 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
$LN13@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

```

## 29.1.2 strlen()

Listing 29.6: strlen() example

```

int strlen_test(char *s1)
{
    return strlen(s1);
};

```

Listing 29.7: 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

```

### 29.1.3 strcpy()

Listing 29.8: strcpy() example

```

void strcpy_test(char *s1, char *outbuf)
{
    strcpy(outbuf, s1);
};

```

Listing 29.9: Optimizing MSVC 2010

```

_s1$ = 8      ; size = 4
_outbuf$ = 12 ; size = 4
_strlen_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
    test    cl, cl
    jne     SHORT $LL3@strcpy_tes
    ret     0
_strlen_test ENDP

```

### 29.1.4 memcpy()

#### Short blocks

Short block copy routine is often implemented as pack of MOV instructions.

Listing 29.10: memcpy() example

```

void memcpy_7(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 7);
}

```

};

Listing 29.11: 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 29.12: 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 16-bit word (if needed), then the last byte (if needed).

Structures are also copied using `MOV`: [19.4.1](#).

By the way, that remind us unrolled loops: [13.0.2](#).

## Long blocks

Compilers behave differently here.

Listing 29.13: `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);
};
```

While copying 128 bytes, MSVC can do this with single MOVSD instruction (because 128 divides evenly by 4):

Listing 29.14: 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 123 bytes are copying, 30 32-byte words are copied first using instruction MOVSD (that's 120 bytes), then 2 bytes are copied using MOVSW, then one more byte using MOVSB.

Listing 29.15: 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, working for any block size:

Listing 29.16: 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
    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 are usually works as follows: calculate, how many 32-bit words can be copied, then copy then by MOVSD, then copy remaining bytes.

More complex copy functions uses [SIMD](#) instructions and also take memory aligning into consideration.

### 29.1.5 memcmp()

Listing 29.17: memcmp() example

```

void memcpy_1235(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 1235);
};

```

For any block size, MSVC 2010 inserts the same universal function:

Listing 29.18: Optimizing MSVC 2010

```

_buf1$ = 8      ; size = 4
_buf2$ = 12     ; size = 4
_memcmp_1235 PROC
    mov     edx, DWORD PTR _buf2$[esp-4]
    mov     ecx, DWORD PTR _buf1$[esp-4]
    push   esi
    push   edi
    mov     esi, 1235
    add     edx, 10
$LL4@memcmp_123:
    mov     eax, DWORD PTR [edx]
    cmp     eax, DWORD PTR [ecx]
    jne     SHORT $LN10@memcmp_123
    sub     esi, 4
    add     ecx, 4
    add     edx, 4
    cmp     esi, 4
    jae     SHORT $LL4@memcmp_123
$LN10@memcmp_123:
    movzx   edi, BYTE PTR [ecx]
    movzx   eax, BYTE PTR [edx]
    sub     eax, edi
    jne     SHORT $LN7@memcmp_123
    movzx   eax, BYTE PTR [edx+1]

```

```

    movzx    edi, BYTE PTR [ecx+1]
    sub     eax, edi
    jne     SHORT $LN7@memcmp_123
    movzx    eax, BYTE PTR [edx+2]
    movzx    edi, BYTE PTR [ecx+2]
    sub     eax, edi
    jne     SHORT $LN7@memcmp_123
    cmp     esi, 3
    jbe     SHORT $LN6@memcmp_123
    movzx    eax, BYTE PTR [edx+3]
    movzx    ecx, BYTE PTR [ecx+3]
    sub     eax, ecx
$LN7@memcmp_123:
    sar     eax, 31
    pop     edi
    or     eax, 1
    pop     esi
    ret     0
$LN6@memcmp_123:
    pop     edi
    xor     eax, eax
    pop     esi
    ret     0
_memcmp_1235 ENDP

```

## 29.1.6 IDA script

I wrote small [IDA](https://github.com/yurichev/IDA_scripts) script for searching and folding such very frequently seen pieces of inline code:

[https://github.com/yurichev/IDA\\_scripts](https://github.com/yurichev/IDA_scripts).



## Chapter 30

# Incorrectly disassembled code

Practicing reverse engineers often dealing with incorrectly disassembled code.

### 30.1 Disassembling started incorrectly (x86)

Unlike ARM and MIPS (where any instruction has length of 2 or 4 bytes), x86 instructions has variable size, so, any disassembler, starting at the middle of x86 instruction, may produce incorrect results.

As an example:

```

add     [ebp-31F7Bh], c1
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]
push   eax
call    sub_407663
pop     ecx
test   eax, eax
jnz    short loc_402D7B

```

There are incorrectly disassembled instructions at the beginning, but eventually, disassembler finds right track.

## 30.2 How random noise looks disassembled?

Common properties which can be easily spotted are:

- Unusually big instruction dispersion. Most frequent x86 instructions are PUSH, MOV, CALL, but here we will see instructions from any instruction group: FPU instructions, IN/OUT instructions, rare and system instructions, everything messed up in one single place.
- Big and random values, offsets and immediates.
- Jumps having incorrect offsets often jumping into the middle of another instructions.

Listing 30.1: 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
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 30.2: random noise (x86-64)

```

lea     esi, [rax+rdx*4+43558D29h]
loc_AF3: ; CODE XREF: seg000:0000000000000B46
rcl     byte ptr [rsi+rax*8+29BB423Ah], 1
lea     ecx, cs:0FFFFFFFB2A6780Fh
mov     al, 96h
mov     ah, 0CEh
push    rsp
lods   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
lods   [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

loc_B3D: ; CODE XREF: seg000:0000000000000B5F
mov    ds:93CA685DF98A90F9h, eax
jnz    short near ptr loc_AF3+6
out    dx, eax
cwde
mov    bh, 5Dh ; ']'
movsb
pop    rbp

```

Listing 30.3: random noise (ARM (ARM mode))

```

BLNE   0xFE16A9D8
BGE    0x1634DOC
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
SVCGET   0x9B0362
SVCGET   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 30.4: random noise (ARM (thumb mode))

```

LSRS     R3, R6, #0x12
LDRH     R1, [R7,#0x2C]
SUBS     R0, #0x55 ; 'U'
ADR      R1, loc_3C
LDR      R2, [SP,#0x218]
CMP      R4, #0x86
SXTB     R7, R4
LDR      R4, [R1,#0x4C]
STR      R4, [R4,R2]

```

```

STR    R0, [R6,#0x20]
BGT    0xFFFFFFFF72
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    0xFFFFFFFF70
LDR    R4, [SP,#0x1D4]
STR    R5, [R5,#0x40]
ORRS   R5, R7

```

```

loc_3C ; DATA XREF: ROM:00000006
B      0xFFFFFFFF98

```

Listing 30.5: 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)
bgtz1  $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)
sd     $fp, -0x6474($t8)
jal    0x78C0050
ori    $v0, $s2, 0xC634
blez   $gp, 0xFFFEA9D4
sw     $t8, -0x2CD4($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 decrypting code (including self-modifying) may look like noise as well, nevertheless, it executes correctly.



# Chapter 31

## C++

### 31.1 Classes

#### 31.1.1 Simple example

Internally, C++ classes representation is almost the same as structures representation.

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 `main()` function looks like translated into assembly language:

Listing 31.1: 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$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    lea  ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP

```

So what's going on. For each object (instance of class `c`) 8 bytes allocated, that is exactly size of 2 variables storage.

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 object (*this* in C++ terminology) is passed in the ECX register. This is called `thiscall` (31.1.1) – a pointer to object passing method.

MSVC doing it using the ECX register. Needless to say, it is not a standardized method, other compilers could do it differently, e.g., via first function argument (like GCC).

Why these functions has so odd names? That's [name mangling](#).

C++ class may contain several methods sharing the same name but having different arguments – that is polymorphism. And of course, different classes may own methods sharing the same name.

*Name mangling* enable us to encode class name + method name + all method argument types in one ASCII-string, which is to be used as internal function name. That's all because neither linker, nor DLL OS loader (mangled names may be among DLL exports as well) knows nothing about C++ or OOP<sup>1</sup>.

`dump()` function is called two times.

Now let's see constructors' code:

Listing 31.2: 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]

```

<sup>1</sup>Object-Oriented Programming

```

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
pop ebp
ret 8
??0c@@QAE@HH@Z ENDP ; c::c

```

Constructors are just functions, they use pointer to structure in the ECX, copying the pointer into own local variable, however, it is not necessary.

From the C++ standard [ISO13, p. 12.1] we know that constructors should not return any values. In fact, internally, constructors are returning pointer to the newly created object, i.e., *this*.

Now `dump()` method:

Listing 31.3: 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()` taking pointer to the structure containing two *int*'s in the ECX, takes two values from it and passing it into `printf()`.

The code is much shorter if compiled with optimization (`/Ox`):

Listing 31.4: 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@
    call _printf
    add  esp, 12
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump

```

That's all. One more thing to say is the [stack pointer](#) was not corrected with `add esp, X` after constructor called. Withal, constructor has `ret 8` instead of the `RET` at the end.

This is all because thiscall ([31.1.1](#)) calling convention here used, the method of passing values through the stack, which is, together with `stdcall` ([49.2](#)) method, offers to correct stack to [callee](#) rather than to [caller](#). `ret x` instruction adds `X` to the value in the `ESP`, then passes control to the [caller](#) function.

See also section about calling conventions ([49](#)).

It is also should be noted the compiler deciding when to call constructor and destructor –but that is we already know from C++ language basics.

## MSVC–x86-64

As we already know, first 4 function arguments in x86-64 are passed in `RCX`, `RDY`, `R8`, `R9` registers, all the rest—via stack. Nevertheless, *this* pointer to the object is passed in `RCX`, first method argument—in `RDY`, etc. We can see this in the `c(int`

a, int b) method internals:

Listing 31.5: Optimizing MSVC 2012 x64

```

; void dump()

?dump@@@QEAAXZ 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@@@QEAAXZ 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

```

*int* data type is still 32-bit in x64 <sup>2</sup>, so that is why 32-bit register's parts are used here.

We also see `JMP printf` instead of `RET` in the `dump()` method, that *hack* we already saw earlier: [12.1.1](#).

## GCC—x86

It is almost the same story in GCC 4.4.1, with a few exceptions.

Listing 31.6: GCC 4.4.1

```

    public main
main proc near

var_20 = dword ptr -20h

```

<sup>2</sup>Apparently, for easier porting of C/C++ 32-bit code to x64

```

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>3</sup> It is also can be noted the pointer to object is passed as first function argument – invisibly for programmer, of course.

First constructor:

```

_ZN1cC1Ev      public _ZN1cC1Ev ; weak
               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

```

<sup>3</sup>There is a good document about various compilers name mangling conventions: [\[Fog14\]](#).

```
_ZN1cC1Ev      endp
```

What it does is just writes two numbers using pointer passed in first (and single) argument.

Second constructor:

```
_ZN1cC1Eii      public _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]
                 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, analog of which could be looks like:

```
void ZN1cC1Eii (int *obj, int a, int b)
{
    *obj=a;
    *(obj+1)=b;
};
```

...and that is completely predictable.

Now dump() function:

```
_ZN1c4dumpEv    public _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"
        ↵ n"
        call    _printf
        leave
        retn
_ZN1c4dumpEv  endp

```

This function in its *internal representation* has sole argument, used as pointer to the object (*this*).

This function could be rewritten into C:

```

void ZN1c4dumpEv (int *obj)
{
    printf ("%d; %d\n", *obj, *(obj+1));
};

```

Thus, if to base our judgment on these simple examples, the difference between MSVC and GCC is style of function names encoding (*name mangling*) and passing pointer to 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, R9 [Mit13] registers, and the pointer to *this* via first one (RDI) and that is what we see here. *int* data type is also 32-bit here. *JMP* instead of *RET hack* is also used here.

Listing 31.7: GCC 4.4.6 x64

```

; 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

```

### 31.1.2 Class inheritance

It can be said about inherited classes that it is simple structure we already considered, but extending in inherited classes.

Let's take 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\n",
                ↵ , 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);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};

```

Let's investigate generated code of the `dump()` functions/methods and also `object::print_color()`, let's see memory layout for structures-objects (as of 32-bit code).

So, `dump()` methods for several classes, generated by MSVC 2008 with `/Ox` and `/Ob0` options <sup>4</sup>

Listing 31.8: 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@

```

<sup>4</sup>`/Ob0` options means inline expansion disabling since function inlining right into the code where the function is called will make our experiment harder

```

    call _printf
    add esp, 8
    ret 0
?print_color@object@@QAEXXZ ENDP ; object::print_color

```

Listing 31.9: 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@_ODG@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 31.10: Optimizing MSVC 2008 /Ob0

```

?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
    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@_OCF@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 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 31.11: 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$[esp+40]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea ecx, DWORD PTR _s$[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

```

Inherited classes must always add their fields after base classes' fields, so to make possible for base class methods to work with their own fields.

When `object::print_color()` method is called, a pointers to both *box* object and *sphere* object are passed as `this`, it can work with these objects easily since *color* field in these objects is always at the pinned address (at `+0x0` offset).

It can be said, `object::print_color()` method is agnostic in relation to input object type as long as fields will be *pinned* at the same addresses, and this condition is always true.

And if you create inherited class of the e.g. *box* class, compiler will add new fields after *depth* field, leaving *box* class fields at the pinned addresses.

Thus, `box::dump()` method will work fine accessing *color/width/height/depths* fields always pinned on known addresses.

GCC-generated code is almost likewise, with the sole exception of `this` pointer passing (as it was described above, it passing as first argument instead of the ECX registers).

### 31.1.3 Encapsulation

Encapsulation is data hiding in the *private* sections of class, e.g. to allow access to them only from this class methods, but no more than.

However, are there any marks in code about the very fact that some field is private and some other – not?

No, there are no such marks.

Let's try 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 let's see `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@_ODG@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 access from any other functions, but, knowing this layout, can we create a code modifying these fields?

So I added `hack_ooop_encapsulation()` function, which, if has the body as follows, will not compile:

```
void hack_ooop_encapsulation(class box * o)
{
    o->width=1; // that code can't be compiled:
                // "error C2248: 'box::width' : cannot access ↵
    ↵ private member declared in class 'box'"
};
```

Nevertheless, if to cast `box` type to *pointer to int array*, and if to modify array of the *int*-s we got, then we have success.

```
void hack_ooop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int ↵
    ↵ *(o);
```

```
ptr_to_object[1]=123;
};
```

This functions' code is very simple –it can be said, the function taking pointer to array of the *int*-s on input and writing 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, encapsulation is just class fields protection only on compiling stage. C++ compiler will not allow to generate a code modifying protected fields straightforwardly, nevertheless, it is possible with the help of *dirty hacks*.

### 31.1.4 Multiple inheritance

Multiple inheritance is a class creation which inherits fields and methods from two or more classes.

Let's write 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();
    printf ("%d\n", sb.get_weight());

    return 0;
};

```

Let's compile it in MSVC 2008 with /Ox and /Ob0 options and let's see `box::dump()`, `solid_object::dump()` and `solid_box::dump()` methods code:

Listing 31.12: 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@_OCM@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 31.13: 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@_OCC@KICFJINL@this?5is?5solid_object?4?5↵
    ↵ density?5DN?5CFd@
    call _printf
    add  esp, 8
    ret  0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

## Listing 31.14: 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@_ODO@HNCNIHNN@this?5is?5solid_box?4?5width↵
    ↵ ?5DN?5CFd?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

It can be said, *solid\_box* class memory layout will be *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 31.15: 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 31.16: 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 31.17: 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 calling two methods, but for `get_volume()` it just passing pointer to `this`, and for `get_density()` it passing pointer to `this` shifted by 12 (or 0xC) bytes, and there, in the `solid_box` class memory layout, fields of the `solid_object` class are beginning.

Thus, `solid_object::get_density()` method will believe it is dealing with usual `solid_object` class, and `box::get_volume()` method will work with its three fields, believing this is usual object of the `box` class.

Thus, we can say, an object of a class, inheriting from several other classes, representing in memory *united* class, containing all inherited fields. And each inherited method called with a pointer to corresponding structure's part passed.

### 31.1.5 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 virtual method `dump()`, being replaced in the *box* and *sphere* class-inheritors.

If in an environment, where it is not known what type has object, as in the `main()` function in example, a virtual method `dump()` is called, somewhere, the information about its type must be stored, so to call relevant virtual method.

Let's compile it in MSVC 2008 with `/Ox` and `/Ob0` options and let's see `main()` function code:

```

_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$[esp+48]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea ecx, DWORD PTR _s$[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

```

Pointer to the `dump()` function is taken somewhere from object. Where the address of new method would be written there? Only somewhere in constructors: there is no other place since nothing more is called in the `main()` function.<sup>5</sup>

Let's see `box` class constructor's code:

```

??_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 0fffffffH
    DD 00H
    DD 040H
    DD FLAT:??_R3box@@@8

??_R2box@@@8 DD FLAT:??_R1A?0A@EA@box@@@8 ; box::`RTTI Base ↵
    ↳ Class Array'

```

<sup>5</sup>About pointers to functions, read more in relevant section:(21)

```

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
??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 slightly different memory layout: the first field is a pointer to some table `box::`vftable'` (name was set by MSVC compiler).

In this table we see a link to the table named `box::`RTTI Complete Object Locator'` and also a link to the `box::dump()` method. So this is named virtual methods table and [RTTI](#)<sup>6</sup>. Table of virtual methods contain addresses of methods and [RTTI](#) table contain information about types. By the way, [RTTI](#)-tables are the

<sup>6</sup>Run-time type information



tables enumerated while calling to *dynamic\_cast* and *typeid* in C++. You can also see here class name as plain text string. Thus, a method of base *object* class may call virtual method *object::dump()*, which in turn, will call a method of inherited class since that information is present right in the object's structure.

Some additional CPU time needed for enumerating these tables and finding right virtual method address, thus virtual methods are widely considered as slightly slower than common methods.

In GCC-generated code [RTTI](#)-tables constructed slightly differently.

## 31.2 ostream

Let's start again with a "hello world" example, but now will use ostream:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Almost any C++ textbook tells that << operation can be replaced (overloaded) for other types. That is what is done in ostream. We see that operator<< is called for ostream:

Listing 31.18: 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@@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
```

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 31.19: 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 to replace operator<< by f(), that code can be rewritten as:

```
f(f(std::cout, "Hello, "), "world!")
```

GCC generates almost the same code as MSVC.

## 31.3 References

In C++, references are pointers (9) as well, but they are called *safe*, because it is harder to make a mistake while dealing with them [ISO13, p. 8.3.2]. For example, reference must always be pointing to the object of corresponding type and cannot be NULL [Cli, p. 8.6]. Even more than that, reference cannot be changed, it is impossible to point it to another object (reset) [Cli, p. 8.5].

If we will try to change the pointers example (9) to use references instead of pointers:

```

void f2 (int x, int y, int & sum, int & product)
{
    sum=x+y;
    product=x*y;
};

```

Then we'll figure out the compiled code is just the same as in pointers example (9):

Listing 31.20: Optimizing MSVC 2010

```

_x$ = 8 ; size ↗
↳ = 4
_y$ = 12 ; size ↗
↳ = 4
_sum$ = 16 ; size ↗
↳ = 4
_product$ = 20 ; size ↗
↳ = 4
?f2@@YAXHHAH0@Z PROC ; f2
    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@@YAXHHAH0@Z ENDP ; f2

```

(A reason why C++ functions has such strange names, is described here: [31.1.1](#).)

## 31.4 STL

N.B.: all examples here were checked only in 32-bit environment. x64 wasn't checked.

## 31.4.1 `std::string`

### Internals

Many string libraries [Yur13, p. 2.2] implements structure containing pointer to the buffer containing string, a variable always containing current string length (that is very convenient for many functions: [Yur13, p. 2.2.1]) and a variable containing current buffer size. A string in buffer is usually terminated with zero: in order to be able to pass a pointer to a buffer into the functions taking usual C ASCIIZ-string.

It is not specified in the C++ standard [ISO13] how `std::string` should be implemented, however, it is usually implemented as described above.

C++ strings is not a class (as `QString` in `Qt`, for instance) but template (basic\_string), this is done in order to support various character types: at least `char` and `wchar_t`.

So, `std::string` is a class with `char` as base type. And `std::wstring` is a class with `wchar_t` as base type.

### MSVC

MSVC implementation may store buffer in place instead of pointer to buffer (if the string is shorter than 16 symbols).

This mean that short string will occupy at least  $16 + 4 + 4 = 24$  bytes in 32-bit environment or at least  $16 + 8 + 8 = 32$  bytes in 64-bit, and if the string is longer than 16 characters, add also length of the string itself.

Listing 31.21: example for MSVC

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

struct std_string
{
    union
    {
        char buf[16];
        char* ptr;
    } u;
    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.

Couple notes:

If the string is shorter than 16 symbols, a buffer for the string will not be allocated in the [heap](#). This is convenient because in practice, large amount of strings are short indeed. Apparently, Microsoft developers chose 16 characters as a good balance.

Very important thing here is in the end of main() functions: I'm not using c\_str() method, nevertheless, if to compile the code and run, both strings will be appeared in the console!

This is why it works.

The string is shorter than 16 characters and buffer with the string is located in the beginning of std::string object (it can be treated just as structure). printf() treats pointer as a pointer to the null-terminated array of characters, hence it works.

Second string (longer than 16 characters) printing is even more dangerous: it is 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 left unnoticed for a long span of time: until a longer string will appear there, then a process will crash.

## GCC

GCC implementation of a structure has one more variable—reference count.

One interesting fact is that a pointer to std::string instance of class points not to beginning of the structure, but to the pointer to buffer. In libstdc++-v3\include\bits\basic\_string.h we may read that it was made for convenient debugging:

```

* The reason you want _M_data pointing to the character %2
  ↳ array and
* not the _Rep is so that the debugger can see the string
* contents. (Probably we should add a non-inline member to %2
  ↳ get

```

```
* the _Rep for the debugger to use, so users can check the ↵
↳ actual
* string length.)
```

[basic\\_string.h source code](#)

I considering this in my example:

Listing 31.22: example for GCC

```
#include <string>
#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-sizeof(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 that 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 should be also used to imitate mistake I already wrote above because GCC has stronger type checking, nevertheless, `printf()` works here without `c_str()` as well.

### More complex 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 31.23: 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@@QAEAAV12@PBBDI@Z ; std::basic_string<
    ↪ char,std::char_traits<char>,std::allocator<char> >::
    ↪ assign

    push  7
    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@PBBDI@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@@0@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
jnb SHORT $LN119@main
push DWORD PTR _s3$[esp+72]
call ???@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
jnb SHORT $LN151@main
push DWORD PTR _s2$[esp+72]
call ???@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
jnb SHORT $LN195@main
push DWORD PTR _s1$[esp+72]
call ???@YAXPAX@Z ; operator delete
add esp, 4
$LN195@main:
xor eax, eax
add esp, 72
ret 0

```



```
_main ENDP
```

Compiler not constructing strings statically: how it is possible anyway if buffer should be located in the [heap](#)? Usual [ASCIIZ](#) strings are stored in the data segment instead, and later, at the moment of execution, with the help of “assign” method, `s1` and `s2` strings are constructed. With the help of operator `+`, `s3` string is constructed.

Please note that there are no call to `c_str()` method, because, its code is tiny enough so compiler inlined it right here: if the string is shorter than 16 characters, a pointer to buffer is left in `EAX` register, and an address of the string buffer located in the [heap](#) is fetched otherwise.

Next, we see calls to the 3 destructors, and they are called if string is longer than 16 characters: then a buffers in the [heap](#) should be freed. Otherwise, since all three `std::string` objects are stored in the stack, they are freed automatically, upon function finish.

As a consequence, short strings processing is faster because of lesser [heap](#) accesses.

`GCC` code is even simpler (because `GCC` way, as I mentioned above, is not to store shorter string right in the structure):

Listing 31.24: `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 _ZN5SsC1EPKcRKSaIcE

mov  DWORD PTR [esp+4], edi
mov  DWORD PTR [esp], ebx

call _ZN5SsC1ERKSs

mov  DWORD PTR [esp+4], esi
mov  DWORD PTR [esp], ebx

call _ZN5Ss6appendERKSs

; 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 _ZN5Ss4_Rep10_M_disposeERKSAIcE
mov  eax, DWORD PTR [esp+24]
mov  DWORD PTR [esp+4], ebx
sub  eax, 12
mov  DWORD PTR [esp], eax
call _ZN5Ss4_Rep10_M_disposeERKSAIcE
mov  eax, DWORD PTR [esp+20]
mov  DWORD PTR [esp+4], ebx
sub  eax, 12
mov  DWORD PTR [esp], eax
call _ZN5Ss4_Rep10_M_disposeERKSAIcE
lea  esp, [ebp-12]
xor  eax, eax
pop  ebx
pop  esi
pop  edi
pop  ebp
ret

```

It can be seen that not a pointer to object is passed to destructors, but rather a place 12 bytes (or 3 words) before, i.e., pointer to the real start of the structure.

**std::string as a global variable**

Experienced C++ programmers may argue: a global variables of [STL<sup>7</sup>](#) types are in fact can be defined.

Yes, indeed:

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

std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};
```

In fact, this variable will be initialized even before `main()` start.

Listing 31.25: MSVC 2012: here is global variable 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@PBIDI@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 31.26: MSVC 2012: here 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
```

<sup>7</sup>(C++) Standard Template Library: [31.4](#)

```

    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 31.27: 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 $LN23@dynamic
    push esi
    mov esi, DWORD PTR ?s@@3V?$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@@QAE@XPAD@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 31.28: 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 not create a separate functions for this, each destructor is passed to `atexit()`, one by one.

## 31.4.2 `std::list`

This is a well-known doubly-linked list: each element has two pointers, to the previous and the next elements.

This means that a 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 “next” and “previous” pointers to your existing structure you wish to unite into a list.

Let’s work out an example with a simple 2-variable structure we want to store in the list.

Although C++ standard [ISO13] does not tell how to implement it, MSVC and GCC implementations are straightforward and similar to each other, 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 doesn't 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;
    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 begin:\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.

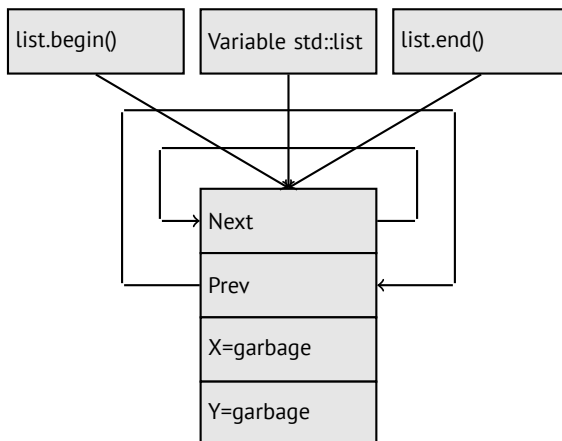
When we run the example, we'll see a long dump, let's work with it part by part.

```

* 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* variables. Both “next” and “prev” pointers are pointing to the self node:



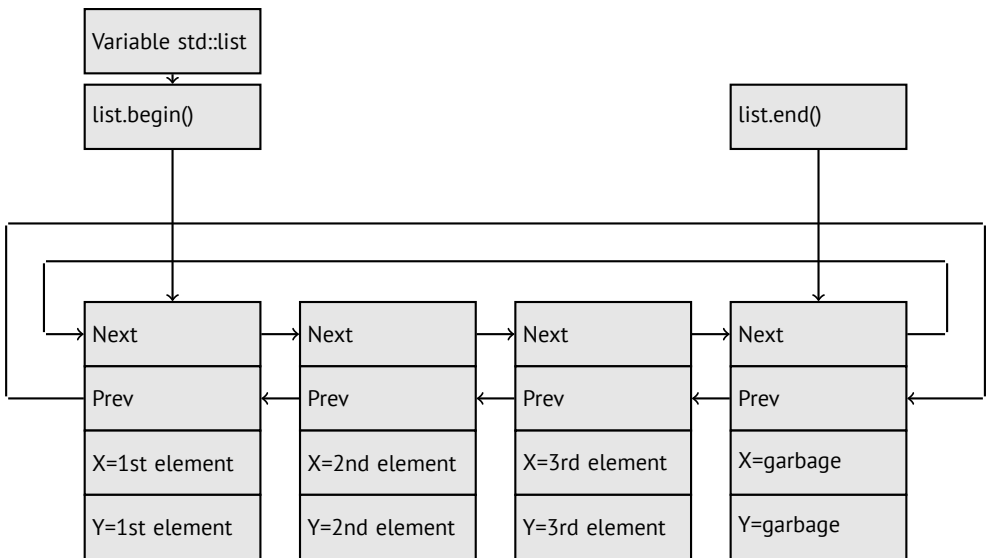


That's is the moment when `.begin` and `.end` iterators are equal to each other. Let's push 3 elements, and the list internally will be:

```
* 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 will not be moved until list disposal. It still contain random garbage in `x` and `y` fields (5 and 6). By occasion, these values are the same as in the last element, but it doesn't mean they are meaningful.

Here is how these 3 elements are stored in memory:



The variable `l` always points to the first node.

`.begin()` and `.end()` iterators are not variables, but functions, when called, returning pointers to a 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.

Iterator in fact is just a pointer to a node. `list.begin()` and `list.end()` are just returning 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 the last element has a pointer to the first and first element has a pointer to the last one, remind us circular list.

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 need to traverse whole list. Inserting element at the list end is also quick, thanks to this feature.

operator `--` and operator `++` are just set current iterator value to the `current_node` or `current_node->next` values. Reverse iterators (`.rbegin`, `.rend`) works just as the same, but in inverse way.

operator `*` of iterator just returns pointer to the point in the node structure, where user's structure is beginning, i.e., pointer to the very first structure element ( $x$ ).

List insertion and deletion is trivial: just allocate new node (or deallocate) and fix all pointers to be valid.

That's why iterator may become invalid after element deletion: it may still point to the node already deallocated. This is also called *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 current list size: this implies slow `.size()` method: it should traverse the whole list for counting elements, because it doesn't have any other way to get the information. This mean this operation is  $O(n)$ , i.e., it is as slow, as how many elements present in the list.

Listing 31.29: 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*)
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 begin:"
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
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 a1stElementDD ; "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_baseI1aSaISO_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 31.30: The whole output

```

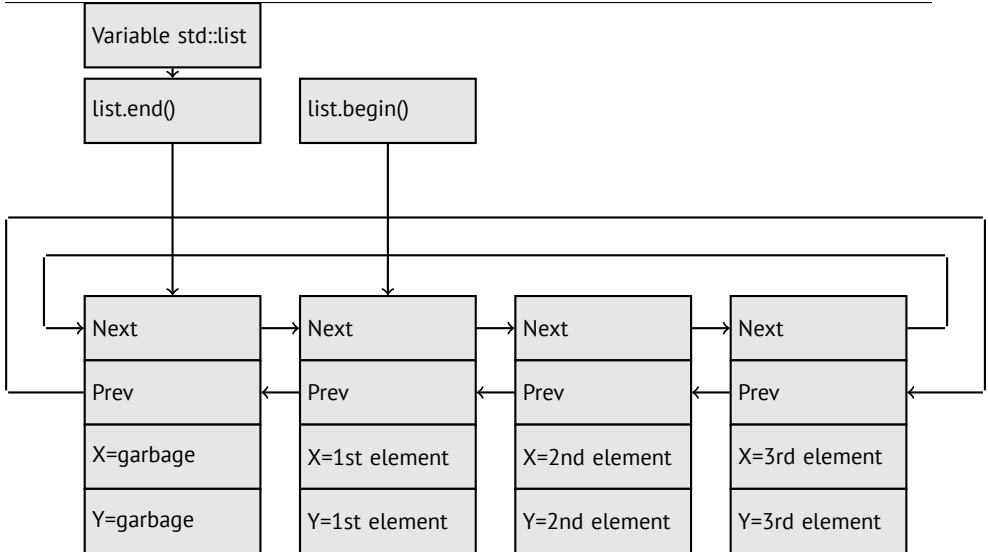
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 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 begin:
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 implementation (2012) is just the same, but it also stores current list size. This mean, `.size()` method is very fast ( $O(1)$ ): just read one value from memory. On the other way, size variable must be corrected at each insertion/deletion.

MSVC implementation is also slightly different in a way it arrange nodes:



GCC has its dummy element at the end of the list, while MSVC at the beginning of it.

Listing 31.31: 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@$0A@U?$_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?↵
↳ $_List_node@Ua@@PAX@1@PAU21@OABUa@@@Z ; std::_List_buy<a,↵
↳ std::allocator<a> >::_Buynode<a const &
mov DWORD PTR [esi+4], eax
mov ecx, DWORD PTR [eax+4]
mov DWORD PTR _t1$[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$[esp+44], 4 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↵
↳ $allocator@Ua@@@std@@@std@@QAEPAU?↵
↳ $_List_node@Ua@@PAX@1@PAU21@OABUa@@@Z ; std::_List_buy<a,↵
↳ std::allocator<a> >::_Buynode<a const &
mov DWORD PTR [esi+4], eax
mov ecx, DWORD PTR [eax+4]
mov DWORD PTR _t1$[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@OABUa@@@Z ; std::_List_buy<a,↵
↳ std::allocator<a> >::_Buynode<a const &
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
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 ???@YAXPAX@Z ; operator delete
    add  esp, 4
    mov  eax, esi
    cmp  esi, ebx
    jne  SHORT $LL414@main
$LN412@main:
    push ebx
    call ???@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 code allocates dummy element at the function start with the help of “Buynode” function, it is also used for the rest nodes allocations ( GCC code allocates the very first element in the local stack).

Listing 31.32: The whole output

```

* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=
↳ =4522072
* 3-elements list:
_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 begin:
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 “next” field at each node. It requires smaller memory footprint, but also doesn't offer a feature to traverse list back.

### 31.4.3 std::vector

I would call std::vector “safe wrapper” of PODT<sup>8</sup> C array. Internally, it is somewhat similar to std::string (31.4.1): it has a pointer to the allocated buffer, pointer to the end of array, and a pointer to the end of allocated buffer.

Array elements just lie in memory adjacently to each other, just like in usual array (17). In C++11 there are new method .data() appeared, returning a pointer to the buffer, akin to .c\_str() in std::string.

Buffer allocated in heap may be larger than array itself.

Both MSVC and GCC implementations are similar, just structure field names are slightly different<sup>9</sup>, so here is one source code working for both compilers. Here is again a C-like code for dumping std::vector structure:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;
```

<sup>8</sup>(C++) Plain Old Data Type

<sup>9</sup>GCC internals: <http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01371.html>

```

    // GCC structure is the same, names are: _M_start, ↵
    ↵ _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    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)); // bounds checking
    printf ("%d\n", c[8]); // operator[], no bounds checking
};

```

Here is a sample output if 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
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158

```

As it can be seen, there is no allocated buffer at the `main()` function start yet. After first `push_back()` call, buffer is allocated. And then, after each `push_back()` call, both array size and buffer size (*capacity*) are increased. But buffer address is changed as well, because `push_back()` function reallocates the buffer in the [heap](#) each time. It is costly operation, that's why it is very important to predict future array size and reserve a space for it with `.reserve()` method. The very last number is a garbage: there are no array elements at this point, so random number is printed. This is illustration to the fact that operator `[]` of `std::vector` is not checking if the index in the array bounds. Slower `.at()` method, however, does checking and throw `std::out_of_range` exception in case of error.

Let's see the code:

## Listing 31.33: 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 $LN1@at
    push OFFSET ??_C@_OBM@NMJKDPPO@invalid?5vector?$DMT?$DO?5↵
    ↵ subscript?5AA@
    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$[ebp+4], 0 ; Mylast
mov  DWORD PTR _c$[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$$QAH@Z ; std::vector<int,std↵
↵ ::allocator<int> >::push_back
lea  edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T5[ebp], 2
lea  eax, DWORD PTR $T5[ebp]
push eax
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?↵
↵ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↵
↵ ::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$$QAH@Z ; std::vector<int,std↵
↵ ::allocator<int> >::push_back
lea  eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T3[ebp], 4
lea  ecx, DWORD PTR $T3[ebp]
push ecx
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?↵
↵ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↵
↵ ::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$$QAH@Z ; std::vector<int,std
↳ ::allocator<int> >::push_back
lea edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov DWORD PTR $T1[ebp], 6
lea eax, DWORD PTR $T1[ebp]
push eax
lea ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV? ↵
↳ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std
↳ ::allocator<int> >::push_back
lea ecx, DWORD PTR _c$[ebp]
push ecx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
push 5
lea ecx, DWORD PTR _c$[ebp]
call ?at@?$vector@HV?$allocator@H@std@@@std@@QAEEAHI@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 `.at()` method check bounds and throw exception in case of error. The number of the last `printf()` call is just to be taken from a memory, without any checks.

One may ask, why not to use variables like “size” and “capacity”, like it was done in `std::string`. I suppose, that was done for the faster bounds checking. But I’m not sure.

The code GCC generates is almost the same on the whole, but `.at()` method is inlined:

Listing 31.34: 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, 0FFFFFF0h
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&)
    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 *)
    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() method is inlined as well. It calls new() if buffer is too small

for new size, call `memmove()` to copy buffer contents, and call `delete()` to free old buffer.

Let's also see what the compiled program outputs if compiled by GCC:

```
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_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
_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
_Myfirst=0x8257040, _Mylast=0x8257058, _Myend=0x8257058
size=6, 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 buffer size grows in different way that in MSVC.

Simple experimentation shows that MSVC implementation buffer grows by ~50% each time it needs to be enlarged, while GCC code enlarges it by 100% each time, i.e., doubles it each time.

### 31.4.4 `std::map` and `std::set`

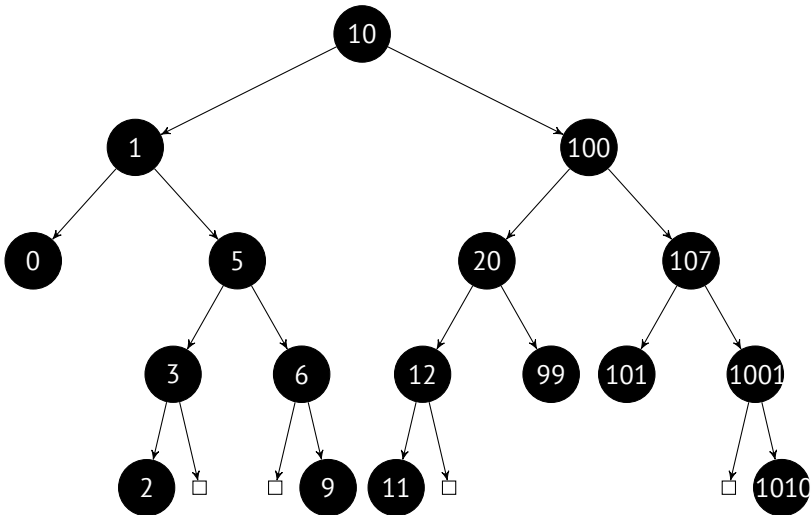
Binary tree is another fundamental data structure. As it states, this is a tree, but each node has at most 2 links to other nodes. Each node have key and/or value.

Binary trees are usually the structure used in “dictionaries” of key-values (AKA “associative arrays”) implementations.

There are at least three important properties binary trees has:

- All keys are stored in always sorted form.
- Keys of any types can be stored easily. Binary tree algorithms are unaware of key type, only key comparison function is required.
- Finding specific key is relatively fast in comparison with lists and arrays.

Here is a very simple example: let's store these numbers in binary tree: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.



All keys lesser than node key value is stored on the left side. All keys greater than node key value is stored on the right side.

Hence, lookup algorithm is straightforward: if the value you looking for is lesser than current node's key value: move left, if it is greater: move right, stop if the value required is equals to the node's key value. That is why searching algorithm may search for numbers, text strings, etc, as long as key comparison function is provided.

All keys has unique values.

Having that, one need  $\approx \log_2 n$  steps in order to find a key in the balanced binary tree of  $n$  keys. It is  $\approx 10$  steps for  $\approx 1000$  keys, or  $\approx 13$  steps for  $\approx 10000$  keys. Not bad, but tree should always be balanced for this: i.e., keys should be distributed evenly on all tiers. Insertion and removal operations do some maintenance to keep tree in balanced state.

There are several popular balancing algorithms available, including AVL tree and red-black tree. The latter extends a node by a “color” value for simplifying balancing process, hence, each node may be “red” or “black”.

Both GCC and MSVC `std::map` and `std::set` template implementations use red-black trees.

`std::set` contain only keys. `std::map` is “extended” version of `set`: it also has a value at each node.

## MSVC

```
#include <map>
#include <set>
#include <string>
#include <iostream>

// structure is not packed!
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;
    size_t Mysize;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool ↵
    ↵ traverse)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d↵
    ↵ Isnil=%d\n",
```





```

    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, ↵
    ↵ m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    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 31.35: MSVC 2012

```

dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0↵
↳ x005BB580 Color=1 Isn1=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0↵
↳ x005BB440 Color=1 Isn1=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0↵
↳ x005BB520 Color=1 Isn1=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0↵
↳ x005BB3A0 Color=1 Isn1=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0↵
↳ x005BB4E0 Color=0 Isn1=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0↵
↳ x005BB3A0 Color=1 Isn1=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0↵
↳ x005BB3A0 Color=0 Isn1=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0↵
↳ x005BB5C0 Color=1 Isn1=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0↵
↳ x005BB3A0 Color=0 Isn1=0
first=9 second=[nine]

```

```

ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0↵
  ↳ x005BB480 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0↵
  ↳ x005BB500 Color=0 Isnil=0
first=20 second=[twenty]
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0↵
  ↳ x005BB3A0 Color=1 Isnil=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0↵
  ↳ x005BB3A0 Color=0 Isnil=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0↵
  ↳ x005BB3A0 Color=1 Isnil=0
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0↵
  ↳ x005BB560 Color=0 Isnil=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0↵
  ↳ x005BB3A0 Color=1 Isnil=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0↵
  ↳ x005BB580 Color=1 Isnil=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0↵
  ↳ x005BB3A0 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=0↵

```

```

    ↪ x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
m.end():
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0↵
    ↪ x005BB580 Color=1 Isnil=1

dumping s as set:
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0↵
    ↪ x005BB6A0 Color=1 Isnil=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0↵
    ↪ x005BB620 Color=1 Isnil=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0↵
    ↪ x005BB680 Color=1 Isnil=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0↵
    ↪ x005BB5E0 Color=0 Isnil=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0↵
    ↪ x005BB5E0 Color=0 Isnil=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0↵
    ↪ x005BB6A0 Color=1 Isnil=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0↵
    ↪ x005BB5E0 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=0↵
    ↪ x005BB5E0 Color=0 Isnil=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0↵
    ↪ x005BB6A0 Color=1 Isnil=1

```

Structure is not packed, so both *char* type values occupy 4 bytes each.

As for `std::map`, `first` and `second` can be viewed as a single value of `std::pair` type. `std::set` has only one value at this point in the structure instead.

Current size of tree is always present, as in case of `std::list` MSVC implementation (31.4.2).

As in case of `std::list`, iterators are just pointers to the nodes. `.begin()` iterator pointing to the minimal key. That pointer is not stored somewhere (as in lists), minimal key of tree is to be found each time. `operator--` and `operator++` moves pointer to the current node to predecessor and successor respectively, i.e., nodes which has previous and next key. The algorithms for all these operations are described in [Cor+09].

`.end()` iterator pointing to the root node, it has 1 in `Isnll`, meaning, the node has no key and/or value. So it can be viewed as a “landing zone” in HDD<sup>10</sup>.

## 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)
{
```

<sup>10</sup>Hard disk drive

```

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;

if (n->M_left)
    dump_tree_node (n->M_left, is_set, traverse, ↵
↳ dump_keys_and_values);
if (n->M_right)
    dump_tree_node (n->M_right, is_set, traverse, ↵
↳ dump_keys_and_values);
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node↵
↳ );

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)↵
↳ point_after_struct;
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)

```

```

    {
        printf ("%.*sL-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left, is_set);
    };
    if (n->M_right)
    {
        printf ("%.*sR-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, ↵
    ↵ M_node_count=%d\n",
           m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    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, 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 *) (void*)&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 31.36: 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=0 ↵
↳ x007A4B80 M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0 ↵
↳ x007A4C60 M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0 ↵
↳ x00000000 M_color=1
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0 ↵
↳ x007A4C20 M_color=0

```

```

key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0 ↵
    ↵ x00000000 M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0 ↵
    ↵ x00000000 M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0 ↵
    ↵ x007A4D00 M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0 ↵
    ↵ x00000000 M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0 ↵
    ↵ x007A4BC0 M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0 ↵
    ↵ x007A4C40 M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0 ↵
    ↵ x00000000 M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0 ↵
    ↵ x00000000 M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0 ↵
    ↵ x00000000 M_color=1
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0 ↵
    ↵ x007A4CA0 M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0 ↵
    ↵ x00000000 M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0 ↵
    ↵ x007A4CC0 M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0 ↵
    ↵ x00000000 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=0 ↵
  ↵ x00000000 M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0 ↵
  ↵ x007A4CC0 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=0 ↵
  ↵ x01D5D850 M_color=1
key=123
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0 ↵
  ↵ x01D5D8B0 M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0 ↵
  ↵ x00000000 M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0 ↵
  ↵ x00000000 M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0 ↵
  ↵ x01D5D8D0 M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0 ↵
  ↵ x00000000 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=0 ↵
    ↵ x00000000 M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0 ↵
    ↵ x01D5D8D0 M_color=0

```

GCC implementation is very similar <sup>11</sup>. The only difference is absence of `Isnil` field, so the structure occupy slightly less space in memory than as it is implemented in MSVC. Root node is also used as a place `.end()` iterator pointing to and also has no key and/or value.

### Rebalancing demo (GCC)

Here is also a demo showing us how tree is rebalanced after insertions.

Listing 31.37: 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;
};

```

<sup>11</sup>[http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl\\_tree\\_8h-source.html](http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl_tree_8h-source.html)



```

s.insert(4);
s.insert(7);
printf ("\n");
printf ("667, 1, 4, 7 are inserted\n");
dump_map_and_set ((struct tree_struct *) (void *)&s);
printf ("\n");
};

```

Listing 31.38: GCC 4.8.1

```

123, 456 are inserted
root----123
      R-----456

11, 12 are inserted
root----123
      L-----11
            R-----12
      R-----456

100, 1001 are inserted
root----123
      L-----12
            L-----11
            R-----100
      R-----456
            R-----1001

667, 1, 4, 7 are inserted
root----12
      L-----4
            L-----1
            R-----11
                  L-----7
      R-----123
            L-----100
            R-----667
                  L-----456
                  R-----1001

```

## Chapter 32

# Obfuscation

Obfuscation is an attempt to hide the code (or its meaning) from reverse engineer.

### 32.1 Text strings

As I revealed in (41) text strings may be utterly helpful. Programmers who aware of this, may try to hide them resulting unablensess to find the string in [IDA](#) or any hex editor.

Here is the simpliest method.

That is how the string may 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 is also can be compared with another like:

```
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 hex editor.

By the way, this is a way to work with the strings when it is impossible to allocate them in data segment, for example, in [PIC](#) or in shellcode.

Another method I once saw is to use `sprintf()` for constructing:

```

sprintf(buf, "%s%c%s%c%s", "hel", 'l', "o w", 'o', "rld");

```

The code looks weird, but as a simplest anti-reversing measure, it may be helpful.

Text strings may also be present in encrypted form, then all string usage will precede string decrypting routine. For example: [63.2](#).

## 32.2 Executable code

### 32.2.1 Inserting garbage

Executable code obfuscation mean inserting random garbage code between real one, which executes but not doing anything useful.

Simple example is:

Listing 32.1: original code

```

add    eax, ebx
mul    ecx

```

Listing 32.2: 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 garbage code uses registers which are not used in the real code (ESI and EDX). However, intermediate results produced by the real code may be used by garbage instructions for extra mess—why not?



### 32.2.2 Replacing instructions to bloated equivalents

- `MOV op1, op2` can be replaced by `PUSH op2 / POP op1` pair.
- `JMP label` can be replaced by `PUSH label / RET` pair. [IDA](#) will not show references to the label.
- `CALL label` can be replaced by `PUSH label_after_CALL_instruction / PUSH label / RET` triplet.
- `PUSH op` may also be replaced by `SUB ESP, 4 (or 8) / MOV [ESP], op` pair.

### 32.2.3 Always executed/never executed code

If the developer is sure that `ESI` at the point is always 0:

```
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:
```

Reverse engineer need some time to get into it.

This is also called *opaque predicate*.

Another example ( and again, developer is sure that `ESI`—is always zero):

```
add     eax, ebx           ; real code
mul     ecx                ; real code
add     eax, esi           ; opaque predicate.
```

### 32.2.4 Making a lot of mess

```
instruction 1
instruction 2
instruction 3
```

Can be replaced to:

```
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:

```

### 32.2.5 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.

## 32.3 Virtual machine / pseudo-code

Programmer may construct his/her own [PL](#) or [ISA](#) and interpreter for it. (Like pre-5.0 Visual Basic, .NET, Java machine). Reverse engineer will have to spend some time to understand meaning and details of all [ISA](#) instructions. Probably, he/she will also need to write a disassembler/decompiler of some sort.

## 32.4 Other thing to mention

My own (yet weak) attempt to patch Tiny C compiler to produce obfuscated code: <http://blog.yurichev.com/node/58>.

Using MOV instruction for really complicated things: [Dol13].

## 32.5 Exercises

### 32.5.1 Exercise #1

This is very short program, compiled using patched Tiny C compiler <sup>1</sup>. Try to find out, what it does.

[http://beginners.re/exercises/per\\_chapter/obfuscation.exe](http://beginners.re/exercises/per_chapter/obfuscation.exe).

Answer: F.1.13.

---

<sup>1</sup><http://blog.yurichev.com/node/58>

# Chapter 33

## More about ARM

### 33.1 Addressing modes

This instruction is possible in ARM64:

```
ldr    x0, [x29,24]
```

This mean, add 24 to value in X29 and load value at this address. Please note that 24 is inside brackets. Meaning is different if number is outside brackets:

```
ldr    w4, [x1],28
```

This mean, load a value at address in X1, then add 28 to X1.

ARM allows to add some constant to the address used for loading, or to subtract. And it's possible both after loading and before.

There is no such addressing mode in x86, but it is present in some other processors, even on PDP-11. There is a legend the pre-increment, post-increment, pre-decrement and post-decrement modes in PDP-11, were "guilty" in appearance such C language (which developed on PDP-11) constructs as `*ptr++`, `*++ptr`, `*ptr--`, `*--ptr`. By the way, this is one of hard to memorize C feature. This is how it is:

C term	ARM term	C statement	how it works
Post-increment	post-indexed addressing	<code>*ptr++</code>	use <code>*ptr</code> value, then <b>increment</b> <code>ptr</code> pointer
Post-decrement	post-indexed addressing	<code>*ptr--</code>	use <code>*ptr</code> value, then <b>decrement</b> <code>ptr</code> pointer
Pre-increment	pre-indexed addressing	<code>*++ptr</code>	<b>increment</b> <code>ptr</code> pointer, then use <code>*ptr</code> value
Pre-decrement	post-indexed addressing	<code>*--ptr</code>	<b>decrement</b> <code>ptr</code> pointer, then use <code>*ptr</code> value

Pre-indexing marked as exclamation mark in ARM assembly language. For example, see line 2 in listing.2.15.

Dennis Ritchie (one of C language creators) mentioned that it is, probably, was invented by Ken Thompson (another C creator) because this processor feature was present in PDP-7 [Rit86][Rit93]. Thus, C language compilers may use it, if it is present on target processor.

That's very convenient for array processings.

## 33.2 Loading constants into register

### 33.2.1 32-bit ARM

As we already know, all instructions has length of 4 bytes in ARM mode and 2 bytes in Thumb mode. How to load 32-bit value into register, if it's not possible to encode it inside one instruction?

Let's try:

```
unsigned int f()
{
    return 0x12345678;
};
```

Listing 33.1: GCC 4.6.3 -O3 ARM mode

```
f:
    ldr    r0, .L2
    bx    lr
.L2:
    .word 305419896 ; 0x12345678
```

So, the 0x12345678 value just stored aside in memory and loads if it needs. But it's possible to get rid of additional memory access.

Listing 33.2: GCC 4.6.3 -O3 -march=armv7-a (ARM mode)

```
movw   r0, #22136      ; 0x5678
movt   r0, #4660       ; 0x1234
bx     lr
```

We see that value is loaded into register by parts, lower part first (using MOVW instruction), then higher (using MOVT).

It means, 2 instructions are necessary in ARM mode for loading 32-bit value into register. It's not a real problem, because in fact there are not much constants in the real code (except of 0 and 1). Does it mean it executes slower than one instruction, as two instructions? 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
```

### 33.2.2 ARM64

```
uint64_t f()
{
    return 0x12345678ABCDEF01;
};
```

Listing 33.3: 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** means “**MOV Keep**”, i.e., it writes 16-bit value into register, not touching other bits at the same time. **LSL** suffix shifts value left by 16, 32 and 48 bits at each step. Shifting done before loading. This means, 4 instructions are necessary to load 64-bit value into register.

#### Storing floating number into register

It's possible to store a floating number into D-register using only one instruction.

For example:

```
double a()
{
    return 1.5;
};
```

Listing 33.4: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:
 0: 1e6f1000      fmov    d0, #1.5000000000000000e+000
 4: d65f03c0      ret
```

1.5 number was indeed encoded in 32-bit instruction. But how? In ARM64, there are 8 bits in **FMOV** instruction for encoding some float point numbers. The algorithm is called **VFPEExpandImm()** in [\[ARM13a\]](#). I tried different: compiler is able to encode 30.0 and 31.0, but it couldn't encode 32.0, an 8 bytes should be allocated to this number in IEEE 754 format:

```
double a()
{
    return 32;
};
```

Listing 33.5: GCC 4.9.1 -O3

```
a:
    ldr    d0, .LC0
    ret

.LC0:
    .word  0
    .word  1077936128
```

### 33.3 Relocs in ARM64

As we know, there are 4-byte instructions in ARM64, so it is impossible to write large number into register using single instruction. Nevertheless, image may be loaded at random address in memory, so that's why relocs exists. Read more about them (in relation to Win32 PE): [53.2.6](#).

Address is formed using ADRP and ADD instructions pair in ARM64. The first loads 4Kb-page address and the second adding remainder. I compiled example from "Hello, world!" (listing.5) in GCC (Linaro) 4.9 under win32:

Listing 33.6: 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_LO12_NC .rodata
0000000000000010 R_AARCH64_CALL26 printf

```

So there are 3 relocs in this object file.

- The very first writes 21-bit page address into ADRP instruction bit fields.
- Second–12 bit of address relative to page start, into ADD instruction bit fields.
- Last, 26-bit one, is applied to the instruction at 0x10 address where the jump to the `printf()` function is. Since it's not possible in ARM64 (and in ARM in ARM mode) to jump to the address not multiple of 4, so the available address space is not 26 bits, but 28.

There are no such relocs in the executable file: because, it's known, where the “Hello!” string is located, in which page, and `puts()` function address is also known. So there are values already set in the ADRP, ADD and BL instructions (linker set it while linking):

Listing 33.7: objdump of executable file

```

000000000400590 <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!..

```

More about ARM64-related relocs: [[ARM13b](#)].



# Chapter 34

## Windows 16-bit

16-bit Windows program are rare nowadays, but in the sense of retrocomputing, or dongle hacking (63), I sometimes digging into these.

16-bit Windows versions were up to 3.11. 96/98/ME also support 16-bit code, as well as 32-bit versions of [Windows NT](#) line. 64-bit versions of [Windows NT](#) line are not support 16-bit executable code at all.

The code is resembling MS-DOS one.

Executable files has NE-type (so-called “new executable”).

All examples considered here were compiled by OpenWatcom 1.9 compiler, using these switches:

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows  
example.c
```

### 34.1 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.

## 34.2 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
    ↵ world"
    mov    ax, offset aHelloWorld ; 0x18. "hello, ↵
    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: PASCAL calling convention dictates passing the last argument first (MB\_YESNOCANCEL), and the first argument—last (NULL). This convention also tells *callee* to restore *stack pointer*: hence RETN instruction has 0Ah argument, meaning pointer should be shifted above by 10 bytes upon function exit.

Pointers are passed by pairs: a segment of data is first passed, then the pointer inside of segment. Here is only one segment in this example, so DS is always pointing to data segment of executable.

### 34.3 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
        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.

## 34.4 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
        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                     ; 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
        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 (long data type mean 32-bit, while *int* is fixed on 16-bit data type) in 16-bit code (both MS-DOS and Win16) are passed by pairs. It is just like 64-bit values are used in 32-bit environment (22).

`sub_B2` here is a library function written by compiler developers, doing “long multiplication”, i.e., multiplies two 32-bit values. Other compiler functions doing the same are listed here: [D](#), [C](#).

ADD/ADC instruction pair is used for addition of compound values: ADD may set/clear CF carry flag, ADC will use it. SUB/SBB instruction pair is used for subtraction: SUB may set/clear CF flag, SBB will use it.

32-bit values are returned from functions in DX:AX register pair.

Constant also passed by pairs in `WinMain()` here.

*int*-typed 123 constant is first converted respecting its sign into 32-bit value using CWD instruction.

## 34.5 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++;
    };
};
```



```

};

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
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
        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 so-called “near” pointers and “far” pointers: another weird artefact of segmented memory in 16-bit 8086.

Read more about it: [78](#).

“near” pointers are those which points within current data segment. Hence, `string_compare()` function takes only two 16-bit pointers, and accesses data as it is located in the segment DS pointing to (`mov a1, [bx]` instruction actually works like `mov a1, ds:[bx]`—DS is implicitly used here).

“far” pointers are those which may point to data in another segment memory. Hence `string_compare_far()` takes 16-bit pair as a pointer, loads high part of it to ES segment register and accessing data through it (`mov al, es:[bx]`). “far” pointers are also used in my `MessageBox()` win16 example: 34.2. Indeed, Windows kernel is not aware which data segment to use when accessing text strings, so it need more complete information.

The reason for this distinction is that compact program may use just one 64kb data segment, so it doesn’t need to pass high part of the address, which is always the same. Bigger program may use several 64kb data segments, so it needs to specify each time, in which segment data is located.

The same story for code segments. Compact program may have all executable code within one 64kb-segment, then all functions will be called in it using CALL NEAR instruction, and code flow will be returned using RETN. But if there are several code segments, then the address of the function will be specified by pair, it will be called using CALL FAR instruction, and the code flow will be returned using RETF.

This is what to be set in compiler by specifying “memory model”.

Compilers targeting MS-DOS and Win16 has specific libraries for each memory model: they were differ by pointer types for code and data.

## 34.6 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
        call  MESSAGEBOX
        xor     ax, ax
        mov    sp, bp
        pop    bp
        retn   0Ah
WinMain endp

```

UNIX time is 32-bit value, so it is returned in DX:AX register pair and stored into two local 16-bit variables. Then a pointer to the pair is passed to `localtime()` function. The `localtime()` function has `struct tm` allocated somewhere in guts of the C library, so only pointer to it is returned. By the way, this also means 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: first four arguments are passed in AX, DX, BX and CX registers, all the rest arguments are via stack. Functions used this convention are also marked by underscore at the end of name.

`sprintf()` does not use PASCAL calling convention, nor Watcom one, so the arguments are passed in usual `cdecl` way (49.1).

### 34.6.1 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
    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

```



```
WinMain      pop    bp
              retn  0Ah
              endp
```

`t` will not be used, but compiler emitted the code which stores the value. Because it is not sure, maybe that value will be eventually used somewhere.

## **Part II**

# **Important fundamentals**

## Chapter 35

# Signed number representations

There are several methods of representing signed numbers<sup>1</sup>, but “two’s complement” is most popular in computers.

binary	hexadecimal	unsigned	signed (2’s complement)
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
...			
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 0x0000002 as unsigned, then first number (4294967294) is bigger than second (2). If to represent them both as signed, first will be -2, and it is lesser than second (2). That is the reason why conditional jumps (11) are present both for signed (e.g. JG, JL) and unsigned (JA, JBE) operations.

For the sake of simplicity, that is what one need to know:

- Number can be signed or unsigned.

<sup>1</sup>[http://en.wikipedia.org/wiki/Signed\\_number\\_representations](http://en.wikipedia.org/wiki/Signed_number_representations)

- C/C++ signed types: *int* (-2147483646..2147483647 or 0x80000000..0x7FFFFFFF), *char* (-127..128 or 0x7F..0x80). Unsigned: *unsigned int* (0..4294967295 or 0..0xFFFFFFFF), *unsigned char* (0..255 or 0..0xFF), *size\_t*.
- Signed types has sign in the most significant bit: 1 mean “minus”, 0 mean “plus”.
- Addition and subtraction operations are working 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.
- More instructions working with signed numbers: *CBW/CWD/CWDE/CDQ/CDQE* (A.6.3), *MOVSX* (14.1.1), *SAR* (A.6.3).

## 35.1 Integer overflow

It is worth noting that incorrect representation of number can lead integer overflow vulnerability.

For example, we have a network service, it receives network packets. In the packets there is also a field where subpacket length is coded. It is 32-bit value. After network packet received, service checking the field, and if it is larger than, e.g. some *MAX\_PACKET\_SIZE* (let's say, 10 kilobytes), the packet is rejected as incorrect. Comparison is signed. Intruder set this value to the 0xFFFFFFFF. While comparison, this number is considered as signed -1 and it is lesser than 10 kilobytes. No error here. Service would like to copy the subpacket to another place in memory and call *memcpy* (*dst*, *src*, 0xFFFFFFFF) function: this operation, rapidly garbling a lot of inside of process memory.

More about it: [\[ble02\]](#).

# Chapter 36

## Endianness

Endianness is a way of representing values in memory.

### 36.1 Big-endian

A 0x12345678 value will be 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.

### 36.2 Little-endian

A 0x12345678 value will be represented in memory as:

address in memory	byte value
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Little-endian CPUs include Intel x86.

## 36.3 Bi-endian

CPUs which may switch between endianness are ARM, PowerPC, SPARC, MIPS, IA64<sup>1</sup>, etc.

## 36.4 Converting data

TCP/IP network data packets use big-endian conventions, so that is why a program working on little-endian architecture should convert values using `htonl()` and `htons()` functions.

In TCP/IP, big-endian is also called “network byte order”, while little-endian—“host byte order”.

The BSWAP instruction can also be used for conversion.

---

<sup>1</sup>Intel Architecture 64 (Itanium): 77

# Chapter 37

## Memory

There are 3 main types of memory:

- Global memory. AKA “static memory allocation”. No need to allocate explicitly, allocation is done just by declaring variables/arrays globally. This is global variables residing in data or constant segments. Available globally (hence, considered as [anti-pattern](#)). Not convenient for buffers/arrays, because must have fixed size. Buffer overflows occurring here usually overwriting variable or buffer residing next in memory. Example in this book: [6.5](#).
- Stack. AKA “allocate on stack”. Allocation is done just by declaring variables/arrays locally in the function. This is usually local to function variables. Sometimes these local variable are also available to descending functions (if one passing pointer to variable to the function to be executed). Allocation and deallocation are very fast, [SP](#) only needs to be shifted. But also not convenient for buffers/arrays, because buffer size should be fixed at some length, unless `alloca()` ([4.2.4](#)) (or variable-length array) is used. Buffer overflow usually overwrites important stack structures: [17.2](#).
- Heap. AKA “dynamic memory allocation”. Allocation occurring by calling `malloc()/free()` or `new/delete` in C++. Most convenient method: block size may be set at runtime. Resizing is possible (using `realloc()`), but may be slow. This is slowest way to allocate memory: memory allocator must support and update all control structures while allocating and deallocating. Buffer overflows are usually overwrites these structures. Heap allocations is also source of memory leak problem: each memory block should be deallocated explicitly, but one may forgot about it, or do it incorrectly. Another problem is “use after free”—using a memory block after `free()` was called on it, which is very dangerous. Example in this book: [19.2](#).

# Chapter 38

## CPU

### 38.1 Branch predictors

Some modern compilers try to get rid of conditional jump instructions. Examples in this book are: [11.1.2](#), [11.2](#), [18.4.2](#).

This is because because branch predictor is not always perfect, so compilers try to do without conditional jumps, if possible.

Conditional instructions in ARM (like `ADRcc`) is one way, another is `CMOVcc` x86 instruction.

### 38.2 Data dependencies

Modern CPUs are able to execute instructions simultaneously ([OOE<sup>1</sup>](#)), but in order to do so, results of one instructions in group must not influence execution of others. Hence, compiler endeavor to use instructions with minimal influence to the CPU state.

That's why `LEA` instruction is so popular, because it do not modify CPU flags, while other arithmetic instructions do this.

---

<sup>1</sup>Out-of-order execution



## **Part III**

# **Finding important/interesting stuff in the code**

---

Minimalism it is not a prominent feature of modern software.

But not because programmers are writing a lot, but in a reason that all libraries are commonly linked statically to executable files. If all external libraries were shifted into external DLL files, the world would be different. (Another reason for C++ –STL and other template libraries.)

Thus, it is very important to determine origin of a function, if it is from standard library or well-known library (like Boost<sup>2</sup>, libpng<sup>3</sup>), and which one –is related to what we are trying to find in the code.

It is just absurd to rewrite all code to C/C++ to find what we're looking for.

One of the primary reverse engineer's task is to find quickly the code he/she needed.

IDA disassembler allow us search among text strings, byte sequences, constants. It is even possible to export the code into .lst or .asm text file and then use `grep`, `awk`, etc.

When you try to understand what a code is doing, this easily could be some open-source library like libpng. So when you see some constants or text strings which looks familiar, it is always worth to *google* it. And if you find the opensource project where it is used, then it will be enough just to compare the functions. It may solve some part of the problem.

For example, if program use a XML files, the first step may be determining, which XML-library is used for processing, since standard (or well-known) libraries are usually used instead of self-made one.

For example, once upon a time I tried to understand how SAP 6.0 network packets compression/decompression was working. It is a huge software, but a detailed .PDB with debugging information is present, and that is cozily. I finally came to idea that one of the functions doing decompressing of network packet called `CsDecomprLZC()`. Immediately I tried to google its name and I quickly found the function named as the same is used in MaxDB (it is open-source SAP project)<sup>4</sup>.

<http://www.google.com/search?q=CsDecomprLZC>

Astoundingly, MaxDB and SAP 6.0 software shared likewise code for network packets compression/decompression.

---

<sup>2</sup><http://www.boost.org/>

<sup>3</sup><http://www.libpng.org/pub/png/libpng.html>

<sup>4</sup>More about it in relevant section (65.1)

# Chapter 39

## Identification of executable files

### 39.1 Microsoft Visual C++

MSVC versions and DLLs which may be imported:

Marketing version	Internal version	CL.EXE version	DLLs may be imported
6	6.0	12.00	msvcrt.dll, msvcp60.dll
.NET (2002)	7.0	13.00	msvcr70.dll, msvcp70.dll
.NET 2003	7.1	13.10	msvcr71.dll, msvcp71.dll
2005	8.0	14.00	msvcr80.dll, msvcp80.dll
2008	9.0	15.00	msvcr90.dll, msvcp90.dll
2010	10.0	16.00	msvcr100.dll, msvcp100.dll
2012	11.0	17.00	msvcr110.dll, msvcp110.dll
2013	12.0	18.00	msvcr120.dll, msvcp120.dll

msvcp\*.dll contain C++-related functions, so, if it is imported, this is probably C++ program.

#### 39.1.1 Name mangling

Names are usually started with ? symbol.

Read more about MSVC [name mangling](#) here: [31.1.1](#).

### 39.2 GCC

Aside from \*NIX targets, GCC is also present in win32 environment: in the form of Cygwin and MinGW.

### 39.2.1 Name mangling

Names are usually started with `_Z` symbols.

Read more about GCC [name mangling](#) here: [31.1.1](#).

### 39.2.2 Cygwin

`cygwin1.dll` is often imported.

### 39.2.3 MinGW

`msvcrt.dll` may be imported.

## 39.3 Intel FORTRAN

`libifcoremd.dll`, `libifportmd.dll` and `libiomp5md.dll` (OpenMP support) may be imported.

`libifcoremd.dll` has a lot of functions prefixed with `for_`, meaning FORTRAN.

## 39.4 Watcom, OpenWatcom

### 39.4.1 Name mangling

Names are usually started with `W` symbol.

For example, that is how method named “method” of the class “class” not having arguments and returning `void` is encoded to:

```
W?method$_class$n__v
```

## 39.5 Borland

Here is an example of Borland Delphi and C++Builder [name mangling](#):

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiit1iiiiii
@TrueBitmap@$bctr$qpcl
```

```
@TrueBitmap@$bctr$qpv1
@TrueBitmap@$bctr$qi111
```

Names are always started with @ symbol, then class name came, method name, and encoded method argument types.

These names can be in .exe imports, .dll exports, debug data, etc.

Borland Visual Component Libraries (VCL) are stored in .bpl files instead of .dll ones, for example, vcl50.dll, rtl60.dll.

Other DLL might be imported: BORLNDMM.DLL.

### 39.5.1 Delphi

Almost all Delphi executables has “Boolean” text string at the very beginning of code segment, along with other type names.

This is a very typical beginning of CODE segment of a Delphi program, this block came right after 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 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@.$M@.|
00000580 28 4d 40 00 2c 4d 40 00 20 4d 40 00 68 4a 40 00 |(↵
    ↳ M@.,M@. M@.hJ@.|
00000590 84 4a 40 00 c0 4a 40 00 07 54 4f 62 6a 65 63 74 |..↵
    ↳ J@..J@..TObject|
000005a0 a4 11 40 00 07 07 54 4f 62 6a 65 63 74 98 11 40 |..↵
    ↳ @...TObject..@|
000005b0 00 00 00 00 00 00 00 06 53 79 73 74 65 6d 00 00 ↵
    ↳ |.....System..|
000005c0 c4 11 40 00 0f 0a 49 49 6e 74 65 72 66 61 63 65 |..↵
    ↳ @...IInterface|
000005d0 00 00 00 00 01 00 00 00 00 00 00 00 00 c0 00 00 ↵
    ↳ |.....|
000005e0 00 00 00 00 46 06 53 79 73 74 65 6d 03 00 ff ff ↵
    ↳ |...F.System...|
000005f0 f4 11 40 00 0f 09 49 44 69 73 70 61 74 63 68 c0 |..↵
    ↳ @...IDispatch.|
00000600 11 40 00 01 00 04 02 00 00 00 00 c0 00 00 00 |.@↵
    ↳ .....|
00000610 00 00 00 46 06 53 79 73 74 65 6d 04 00 ff ff 90 ↵
    ↳ |...F.System....|
00000620 cc 83 44 24 04 f8 e9 51 6c 00 00 83 44 24 04 f8 |..↵
    ↳ D$...Q1...D$..|
00000630 e9 6f 6c 00 00 83 44 24 04 f8 e9 79 6c 00 00 cc |..↵
    ↳ o1...D$...y1...|
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 8d 40 00 |FA↵

```

```

    ↪ .@.....@.|
00000670  bc 12 40 00 4d 12 40 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 ↵
    ↪ |.....|
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 |1~↵
    ↪ @..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 00 03 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.|

```

First 4 bytes of the data segment (DATA) may be 00 00 00 00, 32 13 8B C0 or FF FF FF FF. This information may be useful when dealing with packed Delphi executables.

## 39.6 Other known DLLs

- vcomp\*.dll—Microsoft implementation of OpenMP.

## Chapter 40

# 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 may save time.

Files and registry access: for the very basic analysis, Process Monitor<sup>1</sup> utility from SysInternals may help.

For the basic analysis of network accesses, Wireshark<sup>2</sup> may help.  
But then you will need to look inside anyway.

The first thing to look for is which functions from OS API<sup>3</sup> and standard libraries are used.

If the program is divided into main executable file and a group of DLL-files, sometimes, these function's names can be helpful.

If we are interested in exactly what can lead to the `MessageBox()` call with specific text, we can try to find this text in data segment, find references to it and find the points from which a 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 `rand()` function or its replacements (like Mersenne twister algorithm) and find the places from which those functions are called, and more important: how the results are used. One example: 60.

But if it is not a game, and `rand()` is used, it is also interesting to know why. There are cases of unexpected `rand()` usage in data compression algorithm (for encryption imitation): <http://blog.yurichev.com/node/44>.

---

<sup>1</sup><http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>

<sup>2</sup><http://www.wireshark.org/>

<sup>3</sup>Application programming interface



## 40.1 Often used functions in Windows API

These functions may be among imported. It is worth to note that not every function might be used by the code written by author. A lot of functions might be called from library functions and CRT code.

- Registry access (advapi32.dll): RegEnumKeyEx<sup>4 5</sup>, RegEnumValue<sup>6 5</sup>, RegGetValue<sup>7 5</sup>, RegOpenKeyEx<sup>8 5</sup>, RegQueryValueEx<sup>9 5</sup>.
- Access to text .ini-files (kernel32.dll): GetPrivateProfileString<sup>10 5</sup>.
- Dialog boxes (user32.dll): MessageBox<sup>11 5</sup>, MessageBoxEx<sup>12 5</sup>, SetDlgItemText<sup>13 5</sup>, GetDlgItemText<sup>14 5</sup>.
- Resources access(53.2.8): (user32.dll): LoadMenu<sup>15 5</sup>.
- TCP/IP-network (ws2\_32.dll): WSARcv<sup>16</sup>, WSASend<sup>17</sup>.
- File access (kernel32.dll): CreateFile<sup>18 5</sup>, ReadFile<sup>19</sup>, ReadFileEx<sup>20</sup>, WriteFile<sup>21</sup>, WriteFileEx<sup>22</sup>.

<sup>4</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724862\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724862(v=vs.85).aspx)

<sup>5</sup>May have -A suffix for ASCII-version and -W for Unicode-version

<sup>6</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724865\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724865(v=vs.85).aspx)

<sup>7</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724868\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724868(v=vs.85).aspx)

<sup>8</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724897\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724897(v=vs.85).aspx)

<sup>9</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724911\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724911(v=vs.85).aspx)

<sup>10</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724353\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724353(v=vs.85).aspx)

<sup>11</sup>[http://msdn.microsoft.com/en-us/library/ms645505\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms645505(VS.85).aspx)

<sup>12</sup>[http://msdn.microsoft.com/en-us/library/ms645507\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms645507(v=vs.85).aspx)

<sup>13</sup>[http://msdn.microsoft.com/en-us/library/ms645521\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms645521(v=vs.85).aspx)

<sup>14</sup>[http://msdn.microsoft.com/en-us/library/ms645489\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms645489(v=vs.85).aspx)

<sup>15</sup>[http://msdn.microsoft.com/en-us/library/ms647990\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms647990(v=vs.85).aspx)

<sup>16</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms741688\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms741688(v=vs.85).aspx)

<sup>17</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms742203\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms742203(v=vs.85).aspx)

<sup>18</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx)

<sup>19</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/aa365467\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx)

<sup>20</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/aa365468\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365468(v=vs.85).aspx)

<sup>21</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/aa365747\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx)

<sup>22</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/aa365748\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365748(v=vs.85).aspx)

- High-level access to the Internet (wininet.dll): WinHttpOpen <sup>23</sup>.
- Check digital signature of a executable file (wintrust.dll): WinVerifyTrust <sup>24</sup>.
- Standard MSVC library (in case of dynamic linking) (msvcr\*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

## 40.2 tracer: Intercepting all functions in specific module

There is INT3-breakpoints in [tracer](#), triggered only once, however, they can be set to all functions in specific DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Or, let's set INT3-breakpoints to all functions with xml prefix in name:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

On the other side of 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 function's arguments.

Nevertheless, this feature is very useful when you know the program uses a DLL, but do not know which functions are actually used. And there are a lot of functions.

For example, let's see, what uptime cygwin utility uses:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Thus we may see all cygwin1.dll library functions which were called at least once, and where from:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from ↵
↳ uptime.exe!OEP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from ↵
↳ uptime.exe!OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from ↵
↳ uptime.exe!OEP+0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from ↵
↳ uptime.exe!OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from ↵
↳ uptime.exe!OEP+0xcbe (0x401cbe))
```

<sup>23</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/aa384098\(vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa384098(vs.85).aspx)

<sup>24</sup><http://msdn.microsoft.com/library/windows/desktop/aa388208.aspx>

```
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from ↗
↳ uptime.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from ↗
↳ uptime.exe!OEP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from ↗
↳ uptime.exe!OEP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from ↗
↳ uptime.exe!OEP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.↗
↳ exe!OEP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from ↗
↳ uptime.exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime↗
↳ .exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.↗
↳ exe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from ↗
↳ uptime.exe!OEP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from ↗
↳ uptime.exe!OEP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from ↗
↳ uptime.exe!OEP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from ↗
↳ uptime.exe!OEP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from ↗
↳ uptime.exe!OEP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.↗
↳ exe!OEP+0x4c3 (0x4014c3))
```

# Chapter 41

## Strings

### 41.1 Text strings

#### 41.1.1 C/C++

Usual C-strings are zero-terminated ([ASCIIZ](#)-strings).

The reason why C string format is as it is (zero-terminating) is apparently historical. In [\[Rit79\]](#) we can 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 as it is:

```
int main()
{
    printf ("Hello, world!\n");
};
```

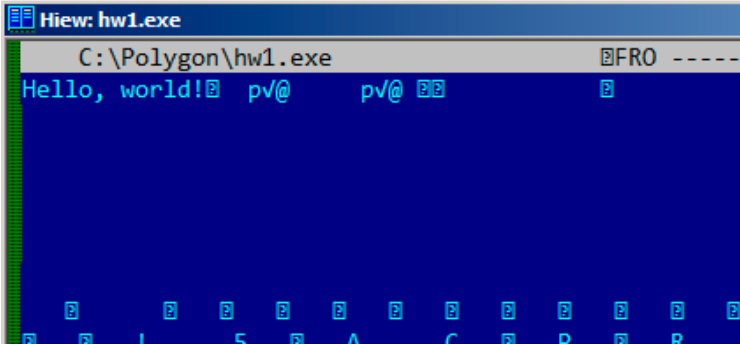


Figure 41.1: Hiew

### 41.1.2 Borland Delphi

The string in Pascal and Borland Delphi is preceded by 8-bit or 32-bit string length value.

For example:

Listing 41.1: 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
```

### 41.1.3 Unicode

Often, what is called by Unicode is a methods of strings encoding where each character occupies 2 bytes or 16 bits. This is common terminological mistake. Unicode is a standard assigning a number to each character of many writing systems of the world, but not describing encoding method.

Most popular encoding methods are: UTF-8 (often used in Internet and \*NIX systems) and UTF-16LE (used in Windows).

#### UTF-8

UTF-8 is one of the most successful methods of character encoding. All Latin symbols are encoded just like in an ASCII-encoding, and symbols beyond ASCII-table

are encoded by several bytes. 0 is encoded as it was before, so all standard C string functions works with UTF-8-strings just like any other string.

Let's see how symbols in various languages are encoded in UTF-8 and how it looks like in FAR in 437 codepage <sup>1</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) : أنا قادر على أكل الزجاج و هذا لا يؤلمني .
```

```
(Hebrew) : אני יכול לאכול זכוכית וזה לא מזיק לי .
```

```
(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.
```

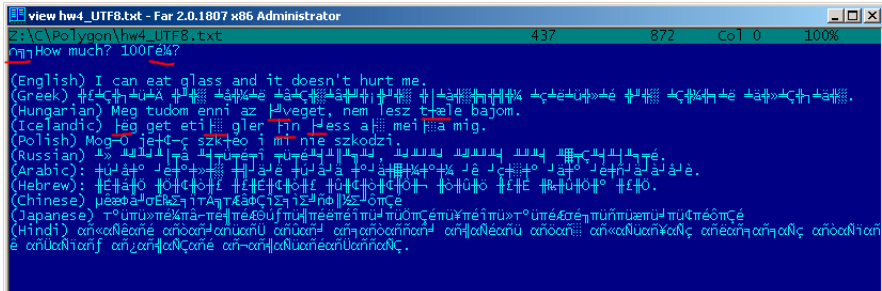


Figure 41.2: FAR: UTF-8

As it seems, English language string looks like as it is in ASCII-encoding. Hungarian language uses Latin symbols plus symbols with diacritic marks. These symbols are encoded by several bytes, I underscored them by red. The same story with Icelandic and Polish languages. I also used “Euro” currency symbol at the begin, which is encoded by 3 bytes. All the rest writing systems here have no connection with Latin. At least about Russian, Arabic, Hebrew and Hindi we could see recurring bytes, and that is not surprise: all symbols from the writing system is usually located in the same Unicode table, so their code begins with the same numbers.

<sup>1</sup>I've got example and translations from there: <http://www.columbia.edu/~fdc/utf8/>

At the very beginning, before “How much?” string we see 3 bytes, which is BOM<sup>2</sup> in fact. BOM defines encoding system to be used now.

## UTF-16LE

Many win32 functions in Windows has a suffix `-A` and `-W`. The first functions works with usual strings, the next with UTF-16LE-strings (*wide*). As in the second case, each symbol is usually stored in 16-bit value of *short* type.

Latin symbols in UTF-16 strings looks in Hiew or FAR as interleaved with zero byte:

```
int wmain()
{
    wprintf (L"Hello, world!\n");
};
```

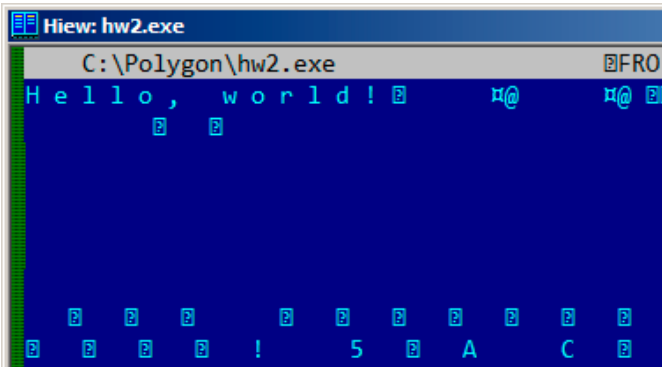


Figure 41.3: Hiew

We may often see this in [Windows NT](#) system files:

---

<sup>2</sup>Byte order mark

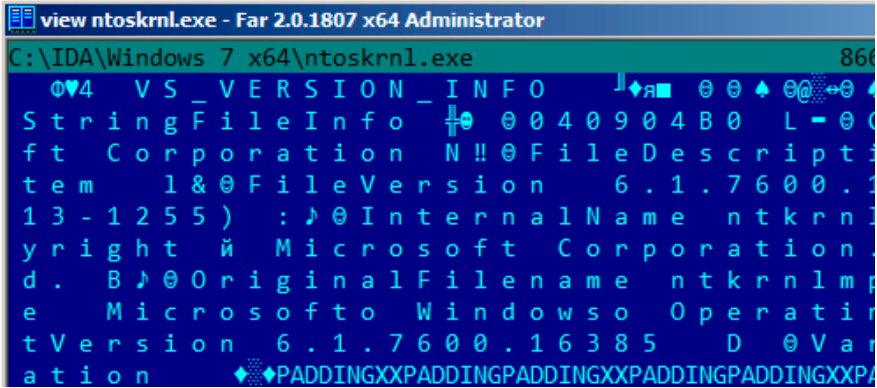


Figure 41.4: Hiew

String with characters occupying exactly 2 bytes are called by “Unicode” in IDA:

```
.data:0040E000 aHelloWorld:
.data:0040E000                unicode 0, <Hello, world!>
.data:0040E000                dw 0Ah, 0
```

Here is how Russian language string encoded in UTF-16LE may looks like:

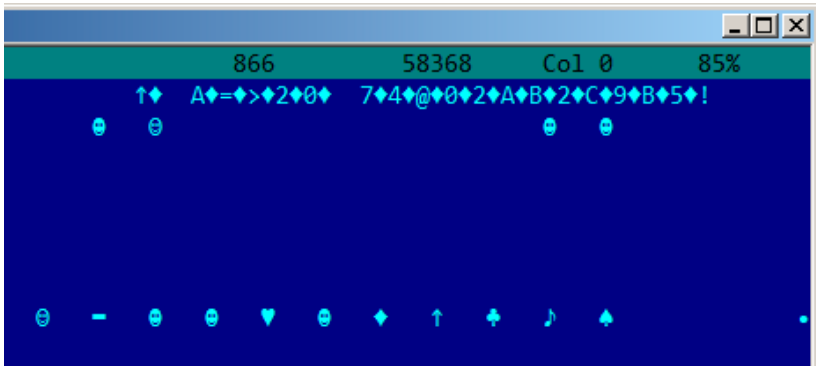


Figure 41.5: Hiew: UTF-16LE

What we can easily spot—is that symbols are interleaved by diamond character (which has code of 4). Indeed, Cyrillic symbols are located in the fourth Unicode plane<sup>3</sup>. Hence, all Cyrillic symbols in UTF-16LE are located in 0x400-0x4FF range.

Let’s back to the example with the string written in multiple languages. Here we can see it in UTF-16LE encoding.

<sup>3</sup>[https://en.wikipedia.org/wiki/Cyrillic\\_\(Unicode\\_block\)](https://en.wikipedia.org/wiki/Cyrillic_(Unicode_block))



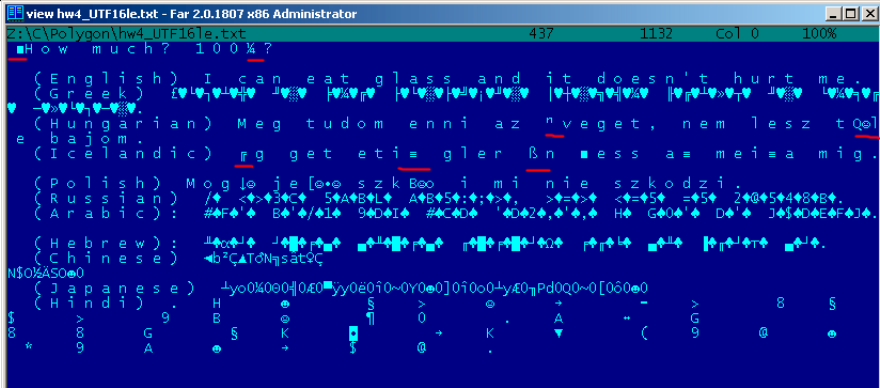


Figure 41.6: FAR: UTF-16LE

Here we can also see **BOM** in the very beginning. All Latin characters are interleaved with zero byte. I also underscored by red some characters with diacritic marks (Hungarian and Icelandic languages).

## 41.2 Error/debug messages

Debugging messages are often very helpful if present. In some sense, debugging messages are reporting about what's going on in program right now. Often these are `printf()`-like functions, which writes to log-files, and sometimes, not writing anything but calls are still present since this build is not a debug build but *release* one. If local or global variables are dumped in debugging messages, it might be helpful as well since it is possible to get variable names at least. For example, one of such functions in Oracle RDBMS is `ksdwrtr()`.

Meaningful text strings are often helpful. **IDA** disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen<sup>4</sup>.

Error messages may help us as well. In Oracle RDBMS, errors are reporting using group of functions.

More about it: <http://blog.yurichev.com/node/43>.

It is possible to find very quickly, which functions reporting about errors and in which conditions. By the way, it is often a reason why copy-protection systems has inarticulate cryptic error messages or just error numbers. No one happy when software cracker quickly understand why copy-protection is triggered just by error message.

One example of encrypted error messages is here: [63.2](#).

<sup>4</sup><http://blog.yurichev.com/node/32>

## Chapter 42

# Calls to assert()

Sometimes `assert()` macro presence is useful too: commonly this macro leaves source file name, line number and condition in code.

Most useful information is contained in `assert-condition`, we can deduce variable names, or structure field names from it. Another useful piece of information is file names – we can try to deduce what type of code is here. Also it is possible to recognize a well-known open-source libraries by file names.

Listing 42.1: 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 1Eh
.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 conditions and file names, that may lead us to open-source library. For example, if to “google” “sp->lzw\_nbits <= BITS\_MAX”, this predictably give us some open-source code, something related to LZW-compression.

# Chapter 43

## Constants

Humans, including programmers, often use round numbers like 10, 100, 1000, as well as in the code.

Practicing reverse engineer, usually know them well in hexadecimal representation: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Constants 0xAAAAAAAA (10101010101010101010101010101010) and 0x55555555 (01010101010101010101010101010101) are also popular—this is alternating bits. For example 0x55AA constant is used at least in boot-sector, [MBR](#)<sup>1</sup>, and in [ROM](#)<sup>2</sup> of IBM-compatibles extension cards.

Some algorithms, especially cryptographical, use distinct constants, which is easy to find in code using [IDA](#).

For example, MD5<sup>3</sup> algorithm initializes its own internal variables like:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

If you find these four constants usage in the code in a row –it is very high probability this function is related to MD5.

Another example is CRC16/CRC32 algorithms, often, calculation algorithms use precomputed tables like:

Listing 43.1: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
```

---

<sup>1</sup>Master Boot Record

<sup>2</sup>Read-only memory

<sup>3</sup><http://en.wikipedia.org/wiki/MD5>

```

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, ↵
    ↵ 0x0E40,
    ...

```

See also precomputed table for CRC32: [18.5](#).

## 43.1 Magic numbers

A lot of file formats defining a standard file header where *magic number*<sup>4</sup> is used.

For example, all Win32 and MS-DOS executables are started with two characters “MZ”<sup>5</sup>.

At the MIDI-file beginning “MThd” signature must be present. If we have a program which uses MIDI-files for something, very likely, it must check MIDI-files for validity by checking at least first 4 bytes.

This could be done like:

(*buf* pointing to the beginning of loaded file into memory)

```

cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file

```

...or by calling a function for comparing memory blocks `memcmp()` or any other equivalent code up to a `CMPSB` ([A.6.3](#)) instruction.

When you find such point you already may say where MIDI-file loading is starting, also, we could see a location of MIDI-file contents buffer and what is used from the buffer, and how.

### 43.1.1 DHCP

This applies to network protocols as well. For example, DHCP protocol network packets contains so-called *magic cookie*: 0x63538263. Any code generating DHCP protocol packets somewhere and somehow must embed this constant into packet. If we find it in the code we may find where it happen and not only this. Any program which can receive DHCP packet, must check *magic cookie*, comparing it with the constant.

For example, let’s take `dhcpcore.dll` file from Windows 7 x64 and search for the constant. And we found it, two times: it seems, the constant is used in two

<sup>4</sup>[http://en.wikipedia.org/wiki/Magic\\_number\\_\(programming\)](http://en.wikipedia.org/wiki/Magic_number_(programming))

<sup>5</sup>[http://en.wikipedia.org/wiki/DOS\\_MZ\\_executable](http://en.wikipedia.org/wiki/DOS_MZ_executable)

functions eloquently named as `DhcpExtractOptionsForValidation()` and `DhcpExtractFullOptions()`:

Listing 43.2: `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 the places where these constants accessed:

Listing 43.3: `dhcpcore.dll` (Windows 7 x64)

```
.text:000007FF6480875F mov     eax, [rsi]
.text:000007FF64808761 cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767 jnz    loc_7FF64817179
```

And:

Listing 43.4: `dhcpcore.dll` (Windows 7 x64)

```
.text:000007FF648082C7 mov     eax, [r12]
.text:000007FF648082CB cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1 jnz    loc_7FF648173AF
```

## 43.2 Constant searching

It is easy in [IDA](#): Alt-B or Alt-I. And for searching for constant in big pile of files, or for searching it in non-executable files, I wrote small utility *binary grep*<sup>6</sup>.

<sup>6</sup><https://github.com/yurichev/bgrep>

## Chapter 44

# Finding the right instructions

If the program is utilizing FPU instructions and there are very few of them in a code, one can try to check each one manually by debugger.

For example, we may be interesting, how Microsoft Excel calculating formulae entered by user. For example, division operation.

If to load excel.exe (from Office 2010) version 14.0.4756.1000 into [IDA](#), then make a full listing and to find each FDIV instructions (except ones which use constants as a second operand –obviously, it is not suits us):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we realizing they are just 144.

We can enter string like =(1/3) in Excel and check each instruction.

Checking each instruction in debugger or [tracer](#) (one may check 4 instruction at a time), it seems, we are lucky here and sought-for instruction is just 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) holding first argument (1) and second one is in [EBX].

Instruction next after FDIV (FSTP) writes result into memory:

```
.text:3011E91B DD 1E                                fstp     ↵
        ↵ qword ptr [esi]
```

If to set breakpoint on it, we may see result:

```
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-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 showing 666 in the cell what finally convincing us we find the right point.



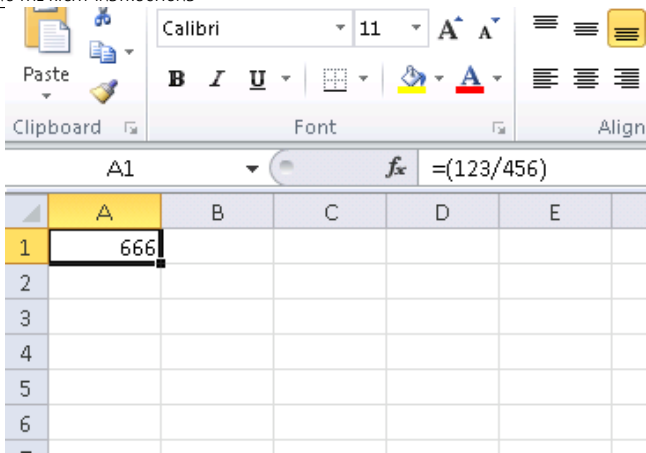


Figure 44.1: Practical joke worked

If to try the same Excel version, but x64, we will find only 12 FDIV instructions there, and the one we looking for –third.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0↵
↵ ,666)
```

It seems, a lot of division operations of *float* and *double* types, compiler replaced by SSE-instructions like DIVSD (DIVSD present here 268 in total).

# Chapter 45

## Suspicious code patterns

### 45.1 XOR instructions

instructions like `XOR op, op` (for example, `XOR EAX, EAX`) are usually used for setting register value to zero, but if operands are different, “exclusive or” operation is executed. This operation is rare in common programming, but used often in cryptography, including amateur one. Especially suspicious case if the second operand is big number. This may points to encrypting/decrypting, checksum computing, etc.

One exception to this observation worth to note is “canary” ([17.3](#)). Its generation and checking is often done using 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!=\r\n    ↵ $4) if($4!="esp") if ($4!="ebp") { print $1, $2, tmp, \r\n    ↵ ",", $4 } }' filename.lst
```

It is also worth to note that such script may also capture incorrectly disassembled code ([30](#)).

### 45.2 Hand-written assembly code

Modern compilers do not emit `LOOP` and `RCL` instructions. On the other hand, these instructions are well-known to coders who like to code in straight assembly language. If you spot these, it can be said, with a high probability, this fragment of code is hand-written. Such instructions are marked as (M) in the instructions list in appendix: [A.6](#).

Also function prologue/epilogue is not commonly present in hand-written assembly copy.

Commonly there is no fixed system in passing arguments into functions in the hand-written code.

Example from 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 into [WRK<sup>1</sup> v1.2](#) source code, this code can be found easily in the file `WRK-v1.2\base\ntos\ke\i386\cpu.asm`.

<sup>1</sup>Windows Research Kernel

## Chapter 46

# Using magic numbers while tracing

Often, main goal is to get to know, how a value was read from file, or received via network, being used. Often, manual tracing of a value is very labouring task. One of the simplest techniques (although not 100% reliable) is to use your own *magic number*.

This resembling X-ray computed tomography is some sense: radiocontrast agent is injected into patient's blood, which is used for improving visibility of internal structures in X-rays. For example, it is well known how blood of healthy human percolates in kidneys and if agent is in blood, it will 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 to write this, 4 byte holding number, to some point in file used by the program we investigate.

Then, while tracing this program, with `tracer` in the *code coverage* mode, and then, with the help of `grep` or just by searching in the text file (of tracing results), we can easily see, where the value was used and how.

Example of *grepable tracer* results in the *cc* mode:

```
0x150bf66 (_kziaia+0x14), e=      1 [MOV EBX, [EBP+8]] [EBP↵
  ↵ +8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=      1 [MOV EDX, [69AEB08h]] [69↵
  ↵ AEB08h]=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=0↵
  ↵ xf1ac360
```

This can be used for network packets as well. It is important to be unique for *magic number* and not to be present in the program's code.

Aside of `tracer`, DosBox (MS-DOS emulator) in heavydebug mode, is able to write information about all register's states for each executed instruction of program to plain text file<sup>1</sup>, so this technique may be useful for DOS programs as well.

---

<sup>1</sup>See also my blog post about this DosBox feature: <http://blog.yurichev.com/node/55>

## Chapter 47

# Other things

### 47.1 General idea

Reverse engineer should try to be in programmer's shoes as often as possible. To take his/her viewpoint and ask himself, how one solve some task here in this case.

### 47.2 C++

[RTTI \(31.1.5\)](#)-data may be also useful for C++ classes identification.

## Chapter 48

# Old-school techniques, nevertheless, interesting to know

### 48.1 Memory “snapshots” comparing

The technique of straightforward two memory snapshots comparing in order to see changes, was often used to hack 8-bit computer games and hacking “high score” files.

For example, if you got a loaded game on 8-bit computer (it is not much memory on these, but game is 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 somewhere, bullet count now 99, do second “snapshot” and then compare both: somewhere must be a byte which was 100 in the beginning and now it is 99. Considering a fact 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 holding bullets count. If to search all references to the address in disassembled game code, it is not very hard to find a piece of code [decrementing](#) bullets count, write [NOP](#) instruction there, or couple of [NOP](#)-s, we’ll have a game with e.g 100 bullets forever. Games on these 8-bit computers was commonly loaded on the same address, also, there were no much different versions of each game (commonly just one version was popular for a long span of time), enthusiastic gamers knew, which byte must be written (using BASIC instruction [POKE](#)) to which address in order to hack it. This led to “cheat” lists containing of [POKE](#) instructions published in magazines related to 8-bit games. See also: [http://en.wikipedia.org/wiki/PEEK\\_and\\_POKE](http://en.wikipedia.org/wiki/PEEK_and_POKE).

Likewise, it is easy to modify “high score” files, this may work not only with

8-bit games. Let's notice your score count and back the file up somewhere. When "high score" count will be different, just compare two files, it can be even done with DOS-utility FC<sup>1</sup> ("high score" files are often in binary form). There will be a point where couple of bytes will be different and it will be easy to see which ones are holding score number. However, game developers are aware of such tricks and may protect against it.

Somewhat similar example here in the book is: [69](#).

### 48.1.1 Windows registry

It is also possible to compare Windows registry before and after a program installation. It is very popular method of finding, which registry elements a program will use. Probably, this is a reason why "windows registry cleaner" shareware is so popular.

---

<sup>1</sup>MS-DOS utility for binary files comparing



**Part IV**

**OS-specific**

## Chapter 49

# Arguments passing methods (calling conventions)

### 49.1 cdecl

This is the most popular method for arguments passing to functions in C/C++ languages.

**Caller** pushing arguments to stack in reverse order: last argument, then penultimate element and finally –first argument. **Caller** also must return back value of the **stack pointer** (ESP) to its initial state after **callee** function exit.

Listing 49.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

### 49.2 stdcall

Almost the same thing as *cdecl*, with the exception the **callee** set ESP to initial state executing `RET x` instruction instead of `RET`, where `x = arguments number * sizeof(int)`<sup>1</sup>. **Caller** will not adjust **stack pointer** by `add esp, x` instruction.

Listing 49.2: stdcall

```
push arg3
```

<sup>1</sup>Size of *int* type variable is 4 in x86 systems and 8 in x64 systems

```

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 may take the function from 7.1 and change it slightly by adding `__stdcall` modifier:

```

int __stdcall f2 (int a, int b, int c)
{
    return a*b+c;
};

```

It will be compiled in almost the same way as 7.2, but you will see `RET 12` instead of `RET SP` is not aligned in caller.

As a consequence, number of function arguments can be easily deduced from `RETN n` instruction: just divide  $n$  by 4.

Listing 49.3: 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 ; ↗
↳ 0000000cH
_f2@12 ENDP

; ...
    push    3
    push    2
    push    1
    call   _f2@12

```

```

push    eax
push    OFFSET $SG81369
call    _printf
add     esp, 8

```

## 49.2.1 Variable arguments number functions

`printf()`-like functions are, probably, the only case of variable arguments functions in C/C++, but it is easy to illustrate an important difference between *cdecl* and *stdcall* with the help of it. Let's start with the idea the compiler knows argument count of each `printf()` function calling. However, called `printf()`, which is already compiled and located in `MSVCRT.DLL` (if to talk about Windows), has not any information about how much arguments were passed, however it can determine it from format string. Thus, if `printf()` would be *stdcall*-function and restored [stack pointer](#) to its initial state by counting number of arguments in format string, this could be dangerous situation, when one programmer's typo may provoke sudden program crash. Thus it is not suitable for such functions to use *stdcall*, *cdecl* is better.

## 49.3 fastcall

That's general naming for a method of passing some of arguments via registers and all others –via stack. It worked faster than *cdecl/stdcall* on older CPUs (because of smaller stack pressure). It will not help to gain performance on modern much more complex CPUs, however.

It is not a standardized way, so, various compilers may do it differently. Well known caveat: if you have two DLLs, one uses another, and they are built by different compilers with different *fastcall* calling conventions.

Both MSVC and GCC passing first and second argument via `ECX` and `EDX` and other arguments via stack.

[Stack pointer](#) must be restored to initial state by [callee](#) (like in *stdcall*).

Listing 49.4: fastcall

```

push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
.. do something ..
ret 4

```

For example, we may take the function from 7.1 and change it slightly by adding `__fastcall` modifier:

```
int __fastcall f3 (int a, int b, int c)
{
    return a*b+c;
};
```

Here is how it will be compiled:

Listing 49.5: 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 a **callee** returns **SP** by **RETN** instruction with operand. Which means, number of arguments can be deduced easily here as well.

### 49.3.1 GCC regparm

It is *fastcall* evolution<sup>2</sup> in some sense. With the `-mregparm` option it is possible to set, how many arguments will be passed via registers. 3 at maximum. Thus, EAX, EDX and ECX registers are to be used.

Of course, if number of arguments is less than 3, not all 3 registers are to be used.

**Caller** restores **stack pointer** to its initial state.

For the example, see (18.1.1).

<sup>2</sup><http://www.ohse.de/uwe/articles/gcc-attributes.html#func-regparm>

## 49.3.2 Watcom/OpenWatcom

It is called “register calling convention” here. First 4 arguments are passed via EAX, EDX, EBX and ECX registers. All the rest—via stack. Functions have underscore added to the function name in order to distinguish them from those having other calling convention.

## 49.4 thiscall

In C++, it is a *this* pointer to object passing into function-method.

In MSVC, *this* is usually passed in the ECX register.

In GCC, *this* pointer is passed as a first function-method argument. Thus it will be seen: internally, all function-methods has extra argument.

For the example, see (31.1.1).

## 49.5 x86-64

### 49.5.1 Windows x64

The method of arguments passing in Win64 is somewhat resembling to `fastcall`. First 4 arguments are passed via RCX, RDX, R8, R9, other —via stack. **Caller** also must prepare a space for 32 bytes or 4 64-bit values, so then **callee** can save there first 4 arguments. Short functions may use argument values just from registers, but larger may save its values for further use.

**Caller** also must return **stack pointer** into 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 %d\n", a, b, c, d, e, f, g);
};

int main()
{
    f1(1,2,3,4,5,6,7);
};
```

Listing 49.6: MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H
```

```

main    PROC
        sub     rsp, 72                ; ↵
        ↪ 00000048H

        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                ; ↵
        ↪ 00000048H
        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
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 72                ; ↵
        ↪ 00000048H

        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                                ; ↵
↳ 00000048H
        ret    0
f1      ENDP

```

Here we clearly see how 7 arguments are passed: 4 via registers and the rest 3 via stack. The code of `f1()` function's prologue saves the arguments in “scratch space”—a space in the stack intended exactly for the purpose. It is done because compiler may not be sure if it will be enough to use other registers without these 4, which will otherwise be occupied by arguments until function execution end. The “scratch space” allocation in the stack is the caller's duty.

Listing 49.7: Optimizing MSVC 2012 /Ob

```

$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                                ; ↵
↳ 00000048H

        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                                ; ↵
↳ 00000048H
        ret    0
f1      ENDP

```



```

main    PROC
        sub     rsp, 72                ; ↵
        ↪ 00000048H

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        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                ; ↵
        ↪ 00000048H
        ret     0
main    ENDP

```

If to compile the example with optimization switch, it is almost the same, but “scratch space” is not used, because no need to.

Also take a look on how MSVC 2012 optimizes primitive value loads into registers by using LEA (A.6.2). I’m not sure if it worth so, but maybe.

Another example of such thing is: [59.1](#).

### **this passing (C/C++)**

*this* pointer is passed in RCX, first method argument in RDX, etc. See also for an example: [31.1.1](#).

## **49.5.2 Linux x64**

The way arguments 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 are no “scratch space”, although [callee](#) may save register values in the stack, if it need-/wants to.

Listing 49.8: Optimizing GCC 4.7.3

```

.LC0:
        .string "%d %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 values are written into 32-bit parts of registers (e.g., EAX) but not to the whole 64-bit register (RAX). This is because each write to low 32-bit part of register automatically clears high 32 bits. Supposedly, it was done for x86-64 code porting simplification.

## 49.6 Returning values of *float* and *double* type

In all conventions except of Win64, values of type *float* or *double* are returning via the FPU register ST(0).

In Win64, values of *float* and *double* types are returned in the low 32 or 64 bits of XMM0 register.

## 49.7 Modifying arguments

Sometimes, C/C++ programmers (not limited to these [PL](#), though), may ask, what will happen if to modify arguments? The answer is simple: arguments are stored in the stack, that is where modification will occur. Calling functions are not use them after [callee](#) exit (I have not seen any opposite case in my practice).

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

```

void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
};

```

Listing 49.9: 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 may modify arguments easily. Of course, if it is not *references* in C++ (31.3), and if you not modify data a pointer pointing to ( then the effect will be propagated outside of current function).

## Chapter 50

# Thread Local Storage

It is a data area, specific to each thread. Every thread can store there what it needs. One famous example is C standard global variable *errno*. Multiple threads may simultaneously call a functions which returns error code in the *errno*, so global variable will not work correctly here, for multi-thread programs, *errno* must be stored in the [TLS](#).

In the C++11 standard, a new *thread\_local* modifier was added, showing that each thread will have its own version of the variable, it can be initialized, and it is located in the [TLS](#)<sup>1</sup>:

Listing 50.1: 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 to say about PE-files, in the resulting executable file, the *tmp* variable will be stored in the section devoted to [TLS](#).

---

<sup>1</sup> C11 also has thread support, optional though

# Chapter 51

## System calls (syscall-s)

As we know, all running processes inside OS are divided into two categories: those having all access to the hardware (“kernel space”) and those have not (“user space”).

There are OS kernel and usually drivers in the first category.

All applications are usually in the second category.

This separation is crucial for OS safety: it is very important not to give to any process possibility to screw up something in other processes or even in OS kernel.

On the other hand, failing driver or error inside OS kernel usually lead to kernel panic or BSOD<sup>1</sup>.

x86-processor protection 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) is a point where these two areas are connected. It can be said, this is the most principal API provided to application software.

As in Windows NT, syscalls table reside in SSDT<sup>2</sup>.

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, while it is easier to use syscalls, however, much more code should be written due to lower level of abstraction of the API. It is also worth noting that the syscall numbers may be different in various OS versions.

### 51.1 Linux

In Linux, syscall is usually called via `int 0x80`. Call number is passed in the EAX register, and any other parameters – in the other registers.

---

<sup>1</sup>Black Screen of Death

<sup>2</sup>System Service Dispatch Table

## Listing 51.1: Simple example of two syscalls usage

```

section .text
global _start

_start:
    mov     edx,len ; buf len
    mov     ecx,msg ; buf
    mov     ebx,1   ; file descriptor. stdout is 1
    mov     eax,4   ; syscall number. sys_write is 4
    int     0x80

    mov     eax,1   ; syscall number. sys_exit is 4
    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://syscalls.kernelgrok.com/>.  
For system calls intercepting and tracing in Linux, `strace(56)` can be used.

## 51.2 Windows

They are called by `int 0x2e` or using special x86 instruction `SYSENTER`.

The full list of syscalls in Windows: <http://j00ru.vexillium.org/ntapi/>.

Further reading:

“Windows Syscall Shellcode” by Piotr Bania:

<http://www.symantec.com/connect/articles/windows-syscall-shellcode>

# Chapter 52

## Linux

### 52.1 Position-independent code

While analyzing Linux shared (.so) libraries, one may frequently spot such code pattern:

Listing 52.1: 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
.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 a constant and by value in the EBX, which calculated at the beginning of each function. This is so-called **PIC**, it is intended to execute 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 crucial now in embedded systems without virtual memory support (where processes are all placed in single continuous memory block). It is also still used in \*NIX systems for shared libraries since shared libraries are shared across many processes while loaded in memory only once. But all these processes may map the same shared library on different addresses, so that is why shared library should be working correctly without fixing on any absolute address.

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 resulting .so file in [IDA](#):

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 52.2: 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:00000577      call   __x86_get_pc_thunk_bx
.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 - 2000↵
    ↵ h)[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: pointers to «*returning %d\n*» string and *global\_variable* are to be corrected at each function execution. The `__x86_get_pc_thunk_bx()` function return address of the point after call to itself (0x57C here) in the EBX. That's the simple way to get value of program counter (EIP) at some point. The 0x1A84 constant is related to the difference between this function begin and so-called *Global Offset Table Procedure Linkage Table* (GOT PLT), the section right after *Global Offset Table* (GOT), where pointer to *global\_variable* is. *IDA* shows these offset processed, so to understand them easily, 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]
```

So, EBX pointing to the GOT PLT section and to calculate pointer to *global\_variable* which 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 AMD64 instruction set supports RIP<sup>1</sup>-relative addressing, just to simplify PIC-code.

Let's compile the same C code in the same GCC version, but for x64.

IDA would simplify output code but suppressing RIP-relative addressing details, so I will run *objdump* instead to see the details:

```
0000000000000720 <f1>:
720:  48 8b 05 b9 08 20 00      mov     rax,QWORD PTR [rip+0x
    ↵ x2008b9]          # 200fe0 <_DYNAMIC+0x1d0>
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 address of 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.

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 aware of performance [Fog13a].

### 52.1.1 Windows

The PIC mechanism is not used in Windows DLLs. If Windows loader needs to load DLL on another base address, it “patches” DLL in memory (at the *FIXUP* places) in order to correct all addresses. This means, several Windows processes cannot share once loaded DLL on different addresses in different process’ memory blocks –since each loaded into memory DLL instance *fixed* to be work only at these addresses..

<sup>1</sup>program counter in AMD64

## 52.2 LD\_PRELOAD hack in Linux

This allows us to load our own dynamic libraries before others, even before system ones, like `libc.so.6`.

What, in turn, allows to “substitute” our written functions before original ones in system libraries. For example, it is easy to intercept all calls to the `time()`, `read()`, `write()`, etc.

Let’s see, if we are able to fool `uptime` utility. As we know, it tells how long the computer is working. With the help of `strace`(56), it is possible to see that this information the utility takes from 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, its contents is generated on fly in Linux kernel. There are just two numbers:

```
$ cat /proc/uptime
416690.91 415152.03
```

What we can learn from Wikipedia <sup>2</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 name of file to be opened with what we need and if it is so, it will write down the descriptor of the file opened. At second, `read()`, if it will be called for this file descriptor, will substitute output, and in other cases, will call original `read()` from `libc.so.6`. And also `close()`, will note, if the file we are currently follow is to be closed.

We will use the `dlopen()` and `dlsym()` functions to determine original addresses of functions in `libc.so.6`.

We need them because we must pass control to “real” functions.

<sup>2</sup><https://en.wikipedia.org/wiki/Uptime>

On the other hand, if we could intercept e.g. `strcmp()`, and follow each string comparisons in program, then `strcmp()` could be implemented easily on one's own, while not using original function <sup>3</sup>.

```
#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");
```

<sup>3</sup>For example, here is how simple `strcmp()` interception is works in article <sup>4</sup> written by Yong Huang

```

        if (read_ptr==NULL)
            die ("can't find read()\n");

        inited = 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);
};

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", 0 ↵
    ↵ 0xffffffff, 0xffffffff)+1;
    };
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};

```

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 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 will always points to filename and path of our library, it will be loaded for all starting programs.

More examples:

- Very simple interception of the `strcmp()` (Yong Huang) [http://yurichev.com/mirrors/LD\\_PRELOAD/Yong%20Huang%20LD\\_PRELOAD.txt](http://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt)
- Kevin Pulo – Fun with LD\_PRELOAD. A lot of examples and ideas. [http://yurichev.com/mirrors/LD\\_PRELOAD/lca2009.pdf](http://yurichev.com/mirrors/LD_PRELOAD/lca2009.pdf)
- File functions interception for compression/decompression files on fly (zlibc). <ftp://metalab.unc.edu/pub/Linux/libs/compression>

# Chapter 53

## Windows NT

### 53.1 CRT (win32)

Does program execution starts right at the `main()` function? No, it is not. If we would open any executable file in [IDA](#) or [HIEW](#), we will see [OEP](#)<sup>1</sup> pointing to another code. This is a code doing some maintenance and preparations before passing control flow to our code. It is called startup-code or CRT-code (C RunTime).

`main()` function takes an array of arguments passed in the command line, and also environment variables. But in fact, a generic string is passed to the program, CRT-code will find spaces in it and cut by parts. CRT-code is also prepares environment variables array `envp`. As of [GUI](#)<sup>2</sup> win32 applications, `WinMain` is used instead of `main()`, having their own arguments:

```
int CALLBACK WinMain(
    _In_ HINSTANCE hInstance,
    _In_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nCmdShow
);
```

CRT-code prepares them as well.

Also, the number returned by `main()` function is an exit code. It may be passed in CRT to the `ExitProcess()` function, taking exit code as argument.

Usually, each compiler has its own CRT-code.

Here is a typical CRT-code for MSVC 2008.

---

<sup>1</sup>Original Entry Point

<sup>2</sup>Graphical user interface

```

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
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
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], 0FFFFFFEh

```

```

140     mov     eax, [ebp+var_20]
141     call   __SEH_epilog4
142     retn

```

Here we may see calls to `GetCommandLineA()` (line 62), then to `setargv()` (line 66) and `setenvp()` (line 74), which are, apparently, fills global variables `argc`, `argv`, `envp`.

Finally, `main()` is called with these arguments (line 97).

There are also calls to the functions having self-describing names like `heap_init()` (line 35), `ioinit()` (line 54).

`Heap` is indeed initialized in [CRT](#). If you will try to use `malloc()` in the program without CRT, the program exiting abnormally with the error:

```

runtime error R6030
- CRT not initialized

```

Global object initializations in C++ is also occurred in the [CRT](#) before `main()` execution: [31.4.1](#).

A value `main()` returns is passed to `cexit()`, or to `$_LN32`, which in turn calling `doexit()`.

Is it possible to get rid of [CRT](#)? Yes, if you know what you do.

[MSVC](#) linker has `/ENTRY` option for setting 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 will get a runnable `.exe` with size 2560 bytes, there are PE-header inside, instructions calling `MessageBox`, two strings in the data segment, `MessageBox` function imported from `user32.dll` and nothing else.

This works, but you will not be able to write `WinMain` with its 4 arguments instead of `main()`. To be correct, you will be able to write so, but arguments will not be prepared at the moment of execution.

By the way, it is possible to make `.exe` even shorter by doing [PE<sup>3</sup>](#)-section aligning less than default 4096 bytes.

```

cl no_crt.c user32.lib /link /entry:main /align:16

```

Linker will say:

---

<sup>3</sup>Portable Executable: [53.2](#)

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image
↪ may not run
```

We getting .exe of 720 bytes size. It running in Windows 7 x86, but not in x64 (the error message will be showed when trying to execute). By applying even more efforts, it is possible to make executable even shorter, but as you can see, compatibility problems may arise quickly.

## 53.2 Win32 PE

PE is a executable file format used in Windows.

The difference between .exe, .dll and .sys is that .exe and .sys usually does not have exports, only imports.

A [DLL](#)<sup>4</sup>, just as any other PE-file, has entry point ([OEP](#)) (the function `DllMain()` is located at it) but usually this function does nothing.

.sys is usually device driver.

As of drivers, Windows require the checksum to be present in PE-file and must be correct<sup>5</sup>.

Starting at Windows Vista, driver files must also be signed by digital signature. It will fail to load without it.

Any PE-file begins with tiny DOS-program, printing a message like “This program cannot be run in DOS mode.” — if to run this program in DOS or Windows 3.1 (OS which are not aware about PE-format), this message will be printed.

### 53.2.1 Terminology

- Module — is a separate file, .exe or .dll.
- Process — a program loaded into memory and running. Commonly consisting of one .exe-file and bunch of .dll-files.
- Process memory — the memory a process works with. Each process has its own. There can usually be loaded modules, memory of the stack, [heap\(s\)](#), etc.
- [VA](#)<sup>6</sup> — is address which is used in program.
- Base address— is the address within a process memory at which a module will be loaded.

---

<sup>4</sup>Dynamic-link library

<sup>5</sup>For example, [Hiew\(58\)](#) can calculate it

<sup>6</sup>Virtual Address

- **RVA**<sup>7</sup> – is a **VA**-address minus base address. Many addresses in PE-file tables uses **RVA**-addresses.
- **IAT**<sup>8</sup> – an array of addresses of imported symbols<sup>9</sup>. Sometimes, a **IMAGE\_DIRECTORY** data directory points to the **IAT**. It is worth to note that **IDA** (as of 6.1) may allocate a pseudo-section named **.idata** for **IAT**, even if **IAT** is a part of another section!
- **INT**<sup>10</sup> – an array of names of symbols to be imported<sup>11</sup>.

## 53.2.2 Base address

The fact is that several module authors may prepare DLL-files for others and there is no possibility to reach agreement, which addresses will be assigned to whose modules.

So that is why if two necessary for process loading DLLs has the same base addresses, one of which will be loaded at this base address, and another – at the other spare space in process memory, and each virtual addresses in the second DLL will be corrected.

Often, **MSVC** linker generates an **.exe**-files with the base address **0x400000**, and with the code section started at **0x401000**. This mean **RVA** of code section start is **0x1000**. DLLs are often generated by **MSVC** linker with a base address of **0x10000000**<sup>12</sup>.

There is also another reason to load modules at various base addresses, rather at random ones.

It is **ASLR**<sup>13 14</sup>.

The fact is that a shellcode trying to be executed on a compromised system must call a system functions.

In older **OS** (in **Windows NT** line: before **Windows Vista**), system DLL (like **kernel32.dll**, **user32.dll**) were always loaded at the known addresses, and also if to recall that its versions were rarely changed, an addresses of functions were fixed and shellcode can call it directly.

In order to avoid this, **ASLR** method loads your program and all modules it needs at random base addresses, each time different.

**ASLR** support is denoted in PE-file by setting the flag **IMAGE\_DLL\_CHARACTERISTICS\_DYNAMIC\_BASE** [**RA09**].

---

<sup>7</sup>Relative Virtual Address

<sup>8</sup>Import Address Table

<sup>9</sup>[**Pie02**]

<sup>10</sup>Import Name Table

<sup>11</sup>[**Pie02**]

<sup>12</sup>This can be changed by **/BASE** linker option

<sup>13</sup>Address Space Layout Randomization

<sup>14</sup>[https://en.wikipedia.org/wiki/Address\\_space\\_layout\\_randomization](https://en.wikipedia.org/wiki/Address_space_layout_randomization)

### 53.2.3 Subsystem

There is also *subsystem* field, usually it is:

- native (.sys-driver),
- console (console application) or
- GUI (non-console).

### 53.2.4 OS version

A PE-file also specify minimal Windows version needed in order to load it. The table of version numbers stored in PE-file and corresponding Windows codenames is here<sup>15</sup>.

For example, **MSVC** 2005 compiles .exe-files running on Windows NT4 (version 4.00), but **MSVC** 2008 is not (files generated has version 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 compiler's options<sup>16</sup>, it is possible to force it to compile for Windows XP.

### 53.2.5 Sections

Division by sections, as it seems, is present in all executable file formats.

It is done in order to separate code from data, and data –from constant data.

- There will be flag *IMAGE\_SCN\_CNT\_CODE* or *IMAGE\_SCN\_MEM\_EXECUTE* on 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, in other words, protected from writing, there are may be flags *IMAGE\_SCN\_CNT\_INITIALIZED\_DATA* and *IMAGE\_SCN\_MEM\_READ* without *IMAGE\_SCN\_MEM\_WRITE*. A process will crash if it would try to write to this section.

Each section in PE-file may have a name, however, it is not very important. Often (but not always) code section have the name *.text*, data section — *.data*, constant data section — *.rdata* (*readable data*). Other popular section names are:

<sup>15</sup>[https://en.wikipedia.org/wiki/Windows\\_NT#Releases](https://en.wikipedia.org/wiki/Windows_NT#Releases)

<sup>16</sup><http://blogs.msdn.com/b/vcblog/archive/2012/10/08/10357555.aspx>

- `.idata`—imports section. `IDA` may create pseudo-section named like this: [53.2.1](#).
- `.edata`—exports section (rare)
- `.pdata`— section containing all information about exceptions in Windows NT for MIPS, IA64 and x64: [53.3.3](#)
- `.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 are often garble section names or replacing names to their own.

`MSVC` allows to declare data in arbitrarily named section <sup>17</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 modern versions of `MSVC` ( a separate `PDB`-files are used there for this purpose).

That is how 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;
```

18

A word about terminology: *PointerToRawData* it called “Offset” and *VirtualAddress* is called “RVA” in Hiew.

<sup>17</sup><http://msdn.microsoft.com/en-us/library/windows/desktop/cc307397.aspx>

<sup>18</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms680341\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms680341(v=vs.85).aspx)

## 53.2.6 Relocations (relocs)

AKA FIXUP-s (at least in Hiew).

This is also present in almost all executable file formats <sup>19</sup>.

Obviously, modules can be loaded on various base addresses, but how to deal with global variables for example? They must be accessed by an address. One solution is position-independent code (52.1). But it is not always suitable.

That is why relocations table is present. There are enumerated the addresses of points which needs to be corrected, in case of loading on another base address.

For example, there is a global variable at the address 0x410000 and this is how it is accessed:

A1 00 00 41 00	mov	eax, [000410000]
----------------	-----	------------------

Base address of module is 0x400000, RVA of global variable is 0x10000.

If the module is loading on the base address 0x500000, the factual address of the global variable must be 0x510000.

As we can see, address of variable is encoded in the instruction MOV, after the byte 0xA1.

That is why address of 4 bytes, after 0xA1, is written into relocs table.

If the module is loaded on different base address, OS-loader enumerates all addresses in table, finds each 32-bit word the address points on, subtracts real, original base address of it (we getting RVA here), and adds new base address to it.

If module is loading on original base address, nothing happens.

All global variables may 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.6.12.

OllyDbg underlines memory places to which relocs will be applied, for example: fig.12.11.

## 53.2.7 Exports and imports

As we all know, any executable program must use OS services and other DLL-libraries somehow.

It can be said, functions from one module (usually DLL) must be connected somehow to a points of their calls in other module (.exe-file or another DLL).

Each DLL has “exports” for this, this is table of functions plus its addresses in a module.

Each .exe-file or DLL has “imports”, this is a table of functions it needs for execution including list of DLL filenames.

<sup>19</sup>Even .exe-files in MS-DOS



After loading main .exe-file, OS-loader, processes imports table: it loads additional DLL-files, finds function names among DLL exports and writes their addresses down in an **IAT** of main .exe-module.

As we can notice, during loading, loader must compare a lot of function names, but strings comparison is not a very fast procedure, so, there is a support of “ordinals” or “hints”, that is a function numbers stored in the table instead of their names.

That is how they can be located faster in loading DLL. Ordinals are always present in “export” table.

For example, program using **MFC**<sup>20</sup> library usually loads mfc\*.dll by ordinals, and in such programs there are no **MFC** function names in **INT**.

While loading such program in **IDA**, it will ask for a path to mfs\*.dll files, in order to determine function names. If not to tell **IDA** path to this DLL, they will look like *mfc80\_123* instead of function names.

## Imports section

Often a separate section is allocated for imports table and everything related to it (with name like `.idata`), however, it is not a strict rule.

Imports is also confusing subject because of terminological mess. Let’s try to collect all information in one place.

---

<sup>20</sup>Microsoft Foundation Classes

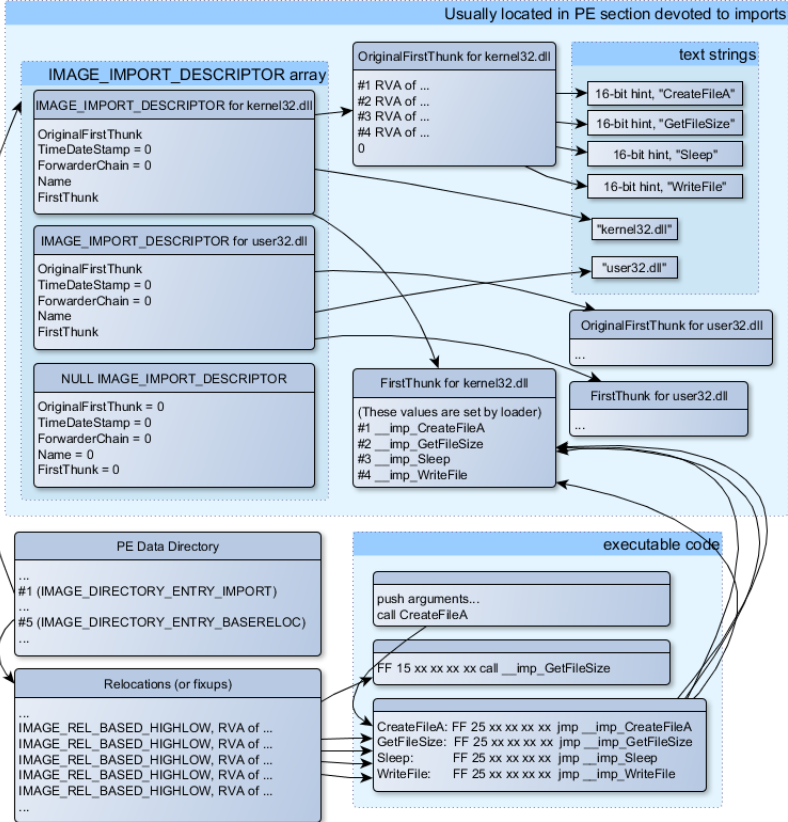


Figure 53.1: The scheme, uniting all PE-file structures related to imports

Main structure is the array of `IMAGE_IMPORT_DESCRIPTOR`. Each element for each DLL being imported.

Each element holds `RVA`-address of text string (DLL name) (`Name`).

`OriginalFirstThunk` is a `RVA`-address of `INT` table. This is array of `RVA`-addresses, each of which points to the text string with function name. Each string is prefixed by 16-bit integer (“hint”)—“ordinal” of function.

While loading, if it is possible to find function by ordinal, then strings comparison will not occur. Array is terminated by zero. There is also a pointer to the `IAT` table with a name `FirstThunk`, it is just `RVA`-address of the place where loader will write addresses of functions resolved.

The points where loader writes addresses, `IDA` marks like: `__imp_CreateFileA`, etc.

There are at least two ways to use addresses written by loader.

- The code will have instructions like `call __imp_CreateFileA`, and since the field

with the address of function imported is a global variable in some sense, the address of *call* instruction (plus 1 or 2) will be added to relocs table, for the case if module will be loaded on 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 slowing down the process of module loading.

- For each imported function, there is only one jump allocated, using `JMP` instruction plus reloc to this instruction. 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 has relative addresses, they are not to be corrected.

Both of these methods can be combined. Apparently, linker creates individual “thunk” if there are too many calls to the functions, but it’s not to be created by default.

By the way, an array of function addresses to which `FirstThunk` is pointing is not necessary to be located in `IAT` section. For example, I once wrote the `PE_add_import` utility for adding import 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 by call to another DLL, the following code my utility written:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

`FirstThunk` points to the first instruction. In other words, while loading your-dll.dll, loader writes address of the *function* right in the code.

It also worth noting a code section is usually write-protected, so my utility adds `IMAGE_SCN_MEM_WRITE` flag for code section. Otherwise, the program will crash while loading with the error code 5 (access denied).

One might ask: what if I supply a program with the DLL files set which are not supposed to change, is it possible to speed up loading process?

Yes, it is possible to write addresses of the functions to be imported into `FirstThunk` arrays in advance. The *Timestamp* field is present in the `IMAGE_IMPORT_DESCRIPTOR` structure. If a value is present there, then loader compare this value with date-time of the DLL file. If the values are equal to each other, then the loader is not do anything, and loading process will be faster. This is what called “old-style binding”<sup>21</sup>. There is the `BIND.EXE` utility in Windows SDK for this. For speeding up of loading of your program, Matt Pietrek in [Pie02], offers to do binding shortly after your program installation on the computer of the end user.

<sup>21</sup>[http://yurichev.com/PE\\_add\\_imports.html](http://yurichev.com/PE_add_imports.html)

<sup>22</sup><http://blogs.msdn.com/b/oldnewthing/archive/2010/03/18/9980802.aspx>.

There is also “new-style binding”, I will write about it in future

PE-files packers/encryptors may also compress/encrypt imports table. In this case, Windows loader, of course, will not load all necessary DLLs. Therefore, packer/encryptor do this on its own, with the help of *LoadLibrary()* and *GetProcAddress()* functions.

In the standard DLLs from Windows installation, *IAT* often is located right in the beginning of PE-file. Supposedly, it is done for optimization. While loading, .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 by parts as they are accessed. Probably, Microsoft developers decided it will be faster.

### 53.2.8 Resources

Resources in a PE-file is 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 *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, e.g. using ResHack editor(53.2.11).

### 53.2.9 .NET

.NET programs are compiled not into machine code but into special bytecode. Strictly speaking, there is bytecode instead of usual x86-code in the .exe-file, however, entry point (*OEP*) is pointing to the tiny fragment of x86-code:

```
jmp      mscoree.dll!_CorExeMain
```

.NET-loader is located in mscoree.dll, it will process the PE-file. It was so in pre-Windows XP *OS*. Starting from XP, *OS*-loader able to detect the .NET-file and run it without execution of that *JMP* instruction<sup>23</sup>.

### 53.2.10 TLS

This section holds initialized data for *TLS*(50) (if needed). When new thread starting, its *TLS*- data is initialized by the data from this section.

Aside from that, PE-file specification also provides initialization of *TLS*-section, so-called, *TLS* callbacks. If they are present, they will be called before control passing to the main entry point (*OEP*). This is used widely in the PE-file packers/encryptors.

<sup>23</sup>[http://msdn.microsoft.com/en-us/library/xh0859k0\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/xh0859k0(v=vs.110).aspx)

### 53.2.11 Tools

- objdump (present in cygwin) for dumping all PE-file structures.
- Hiew(58) as editor.
- pefile – Python-library for PE-file processing <sup>24</sup>.
- ResHack AKA Resource Hacker – resources editor <sup>25</sup>.
- PE\_add\_import<sup>26</sup> – simple tool for adding symbol(s) to PE executable import table.
- PE\_patcher<sup>27</sup> – simple tool for patching PE executables.
- PE\_search\_str\_refs<sup>28</sup> – simple tool for searching for a function in PE executables which use some text string.

### 53.2.12 Further reading

- Daniel Pistelli – The .NET File Format <sup>29</sup>

## 53.3 Windows SEH

### 53.3.1 Let's forget about MSVC

In Windows, SEH is intended for exceptions handling, nevertheless, it is language-agnostic, it is not related to the C++ or OOP in any way. Here we will take a look on SEH in isolated (from C++ and MSVC extensions) form.

Each running process has a chain of SEH-handlers, TIB has address of the last handler. When exception occurred (division by zero, incorrect address access, user exception triggered by calling to `RaiseException()` function), OS will find the last handler in TIB, and will call it with passing all information about CPU state (register values, etc) at the moment of exception. Exception handler will consider exception, was it made for it? If so, it will handle exception. If no, it will signal to OS that it cannot handle it and OS will call next handler in chain, until a handler which is able to handle the exception will be found.

At the very end of the chain, there a standard handler, showing well-known dialog box, informing a process crash, some technical information about CPU state

<sup>24</sup><https://code.google.com/p/pefile/>

<sup>25</sup><http://www.angusj.com/resourcehacker/>

<sup>26</sup>[http://yurichev.com/PE\\_add\\_imports.html](http://yurichev.com/PE_add_imports.html)

<sup>27</sup>[http://yurichev.com/PE\\_patcher.html](http://yurichev.com/PE_patcher.html)

<sup>28</sup>[http://yurichev.com/PE\\_search\\_str\\_refs.html](http://yurichev.com/PE_search_str_refs.html)

<sup>29</sup><http://www.codeproject.com/Articles/12585/The-NET-File-Format>

at the crash, and offering to collect all information and send it to developers in Microsoft.

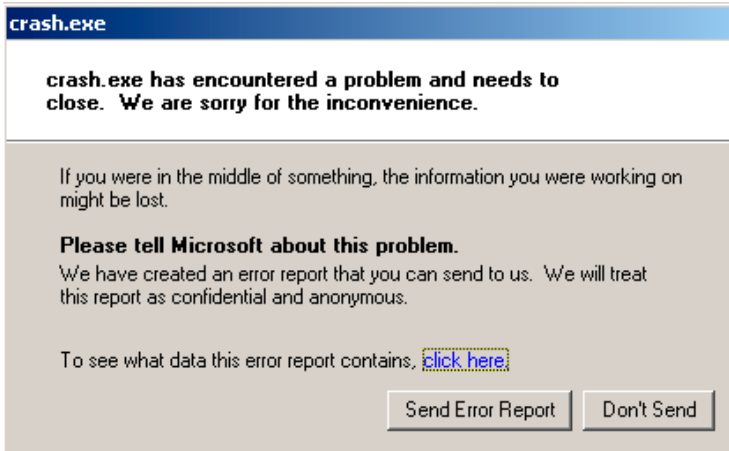


Figure 53.2: Windows XP

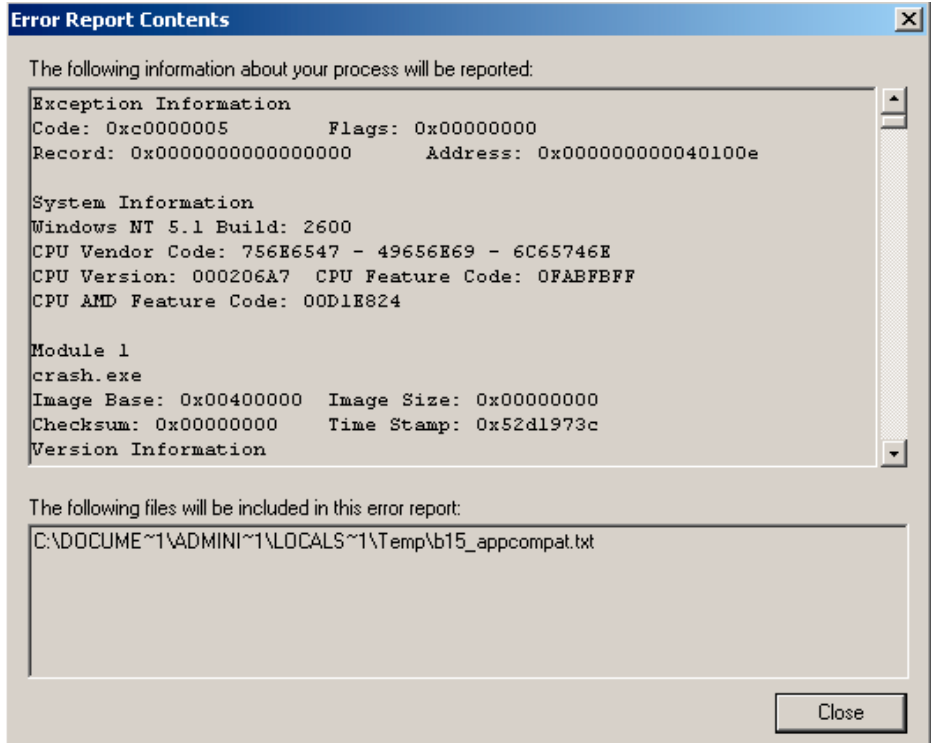


Figure 53.3: Windows XP

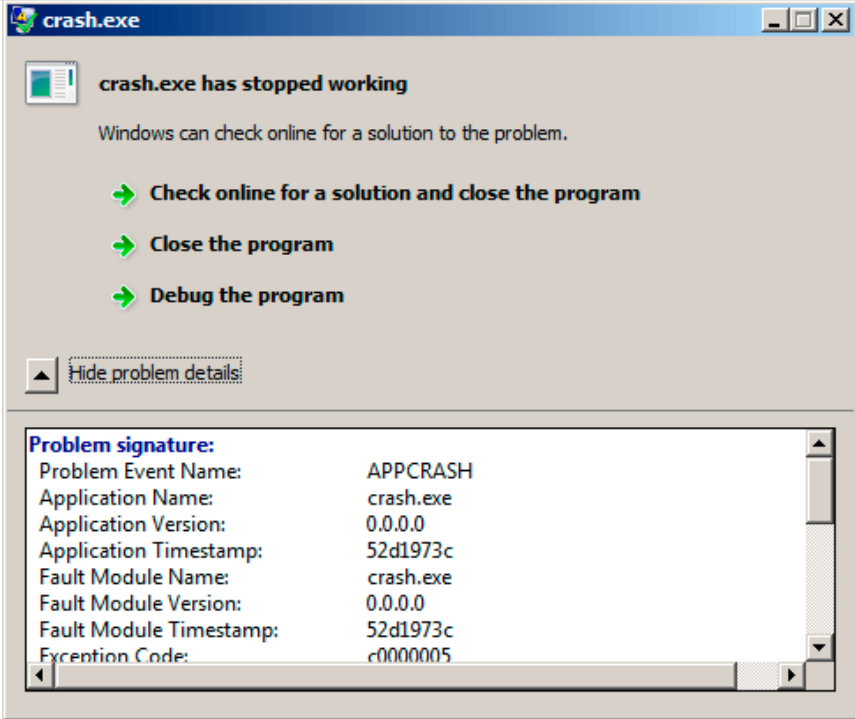


Figure 53.4: Windows 7

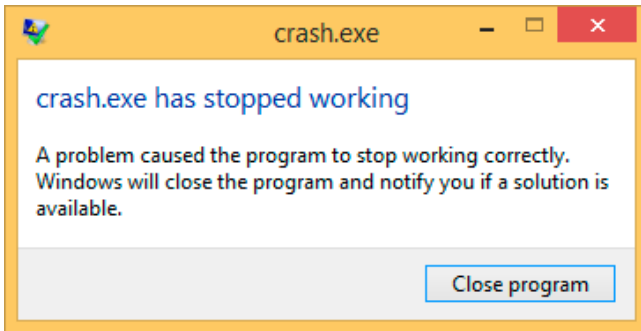


Figure 53.5: Windows 8.1

This handler was also called Dr. Watson earlier <sup>30</sup>.

By the way, some developers made their own handler, sending information

<sup>30</sup>[https://en.wikipedia.org/wiki/Dr.\\_Watson\\_\(debugger\)](https://en.wikipedia.org/wiki/Dr._Watson_(debugger))



about program crash to themselves. It is registered with the help of `SetUnhandledExceptionFilter` and will be called if OS do not have any other way to handle exception. Other example is Oracle RDBMS it saves huge dumps containing all possible information about CPU and memory state.

Let's write our own primitive exception handler <sup>31</sup>:

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

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    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
    }
}
```

<sup>31</sup>The example is based on the example from [Pie]

It is compiled with the SAFESEH option: `cl seh1.cpp /link /safeseh:no`  
More about SAFESEH here:

<http://msdn.microsoft.com/en-us/library/9a89h429.aspx>

```

        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
    {
        ↵ EXCEPTION_REGISTRATION record:           // make ↵
            push    handler                       // address of handler ↵
        ↵ function
            push    FS:[0]                       // address of previous ↵
        ↵ handler
            mov     FS:[0],ESP                   // add new ↵
        ↵ EXECEPTION_REGISTRATION
    }

    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
    {
        ↵ EXECEPTION_REGISTRATION record           // remove our ↵
            mov     eax,[ESP]                   // get pointer to ↵
        ↵ previous record
            mov     FS:[0], EAX                // install previous ↵
        ↵ record
            add     esp, 8                      // clean our ↵
        ↵ EXECEPTION_REGISTRATION off stack
    }
}

```

```

return 0;
}
    
```

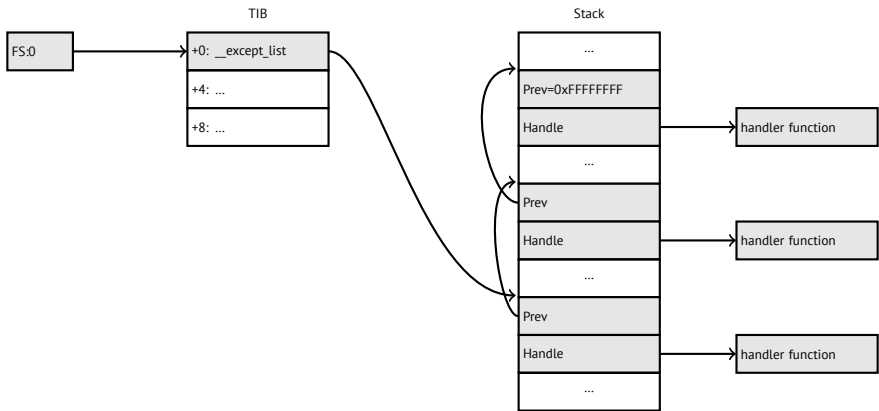
FS: segment register is pointing to the TIB in win32. The very first element in TIB is a pointer to the last handler in chain. We saving it in the stack and store an address of our handler there. The structure is named `_EXCEPTION_REGISTRATION`, it is a simplest singly-linked list and its elements are stored right in the stack.

Listing 53.1: MSVC/VC/crt/src/exsup.inc

```

_EXCEPTION\_REGISTRATION struc
    prev    dd    ?
    handler dd    ?
_EXCEPTION\_REGISTRATION ends
    
```

So each “handler” field points to handler and an each “prev” field points to previous record in the stack. The last record has `0xFFFFFFFF` (-1) in “prev” field.



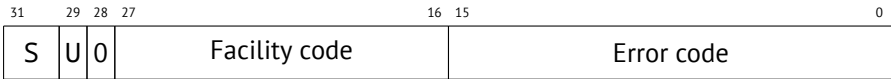
When our handler is installed, let's call `RaiseException()`<sup>32</sup>. This is user exception. Handler will check the code. If the code is `0xE1223344`, it will return `ExceptionContinueExecution`, which means that handler corrected CPU state (it is usually correction of EIP/ESP registers) and the OS can resume thread execution. If to alter the code slightly so the handler will return `ExceptionContinueSearch` then OS will call other handlers, and very unlikely the one who can handle it will be founded, since no one have information about it (rather about its code). You will see the standard Windows dialog about process crash.

What is the difference between system exceptions and user? Here is a system ones:

<sup>32</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms680552\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms680552(v=vs.85).aspx)

as defined in WinBase.h	as defined in ntstatus.h
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT

That is how the code is defined:



S is a basic status code: 11 – error; 10 – warning; 01 – informational; 00 – success. U – whether the code is user code.

That is why I chose 0xE1223344 – 0xE (1110b) mean this is 1) user exception; 2) error. But to be honest, this example works finely without these high bits.

Then we try to read a value from memory at the 0th address. Of course, there are nothing at this address in win32, so exception is raised. However, the very first handler will be called – yours, it will be notified first, checking the code on equality to the EXCEPTION\_ACCESS\_VIOLATION constant.

The code reading from memory at 0th address is looks like:

Listing 53.2: 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 error “on fly” and to continue program execution? Yes, our exception handler can fix EAX value and now let OS will execute this instruction once again. So that is what we do. `printf()` will print 1234, because, after execution of our handler, EAX will not be 0, it will contain address of global variable `new_value`. Execution will be resumed.

That is what is going on: memory manager in CPU signaling about error, the CPU suspends the thread, it finds exception handler in the Windows kernel, latter, in turn, is starting to call all handlers in SEH chain, one by one.

I use MSVC 2010 now, but of course, there are no any guarantee that EAX will be used for pointer.

This address replacement trick is looks showingly, and I offer it here for SEH internals illustration. Nevertheless, I cannot recall where it is used for “on-fly” error fixing in practice.

Why SEH-related records are stored right in the stack instead of some other place? Supposedly because then OS will not need to care about freeing this information, these records will be disposed when function finishing its execution. But I'm not 100%-sure and can be wrong. This is somewhat like `alloca()`: (4.2.4).

### 53.3.2 Now let's get back to MSVC

Supposedly, Microsoft programmers need exceptions in C, but not in C++, so they added a non-standard C extension to MSVC<sup>33</sup>. It is not related to C++ PL exceptions.

```

__try
{
    ...
}
__except(filter code)
{
    handler code
}

```

“Finally” block may be instead of handler code:

```

__try
{

```

<sup>33</sup><http://msdn.microsoft.com/en-us/library/swezy51.aspx>

```

    ...
}
__finally
{
    ...
}

```

The filter code is an expression, telling whether this handler code is corresponding to the exception raised. If your code is too big and cannot be fitted into one expression, a separate filter function can be defined.

There are a lot of such constructs in the Windows kernel. Here is couple of examples from there ([WRK](#)):

Listing 53.3: 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 53.4: 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 ) ) {

```

Here is also filter code example:

## Listing 53.5: WRK-v1.2/base/ntos/cache/copysup.c

```

LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
)

/*++

Routine Description:

    This routine serves as a 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.

    ExceptionCode - A pointer to an NTSTATUS that is to receive ↵
    ↵ the real
                       status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->↵
    ↵ ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
        (ExceptionPointer->ExceptionRecord->NumberParameters ↵
        ↵ >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->↵
        ↵ ExceptionRecord->ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

```

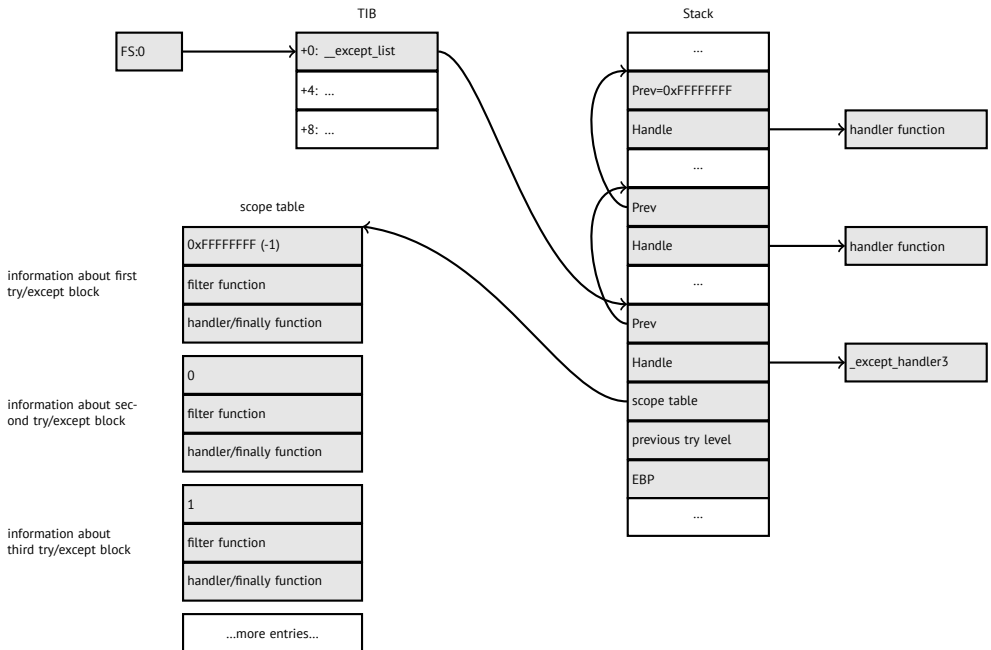
```
return EXCEPTION_EXECUTE_HANDLER;
}
```

Internally, SEH is an extension of 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 MSVC thing. Other win32-compilers may offer something completely different.

### SEH3

SEH3 has `_except_handler3` as handler functions, and extends `_EXCEPTION_REGIST` table, adding a pointer to the *scope table* and *previous try level* variable. SEH4 extends *scope table* by 4 values for buffer overflow protection.

*Scope table* is a table consisting of pointers to the filter and handler codes, for each level of *try/except* nestedness.



Again, it is very important to understand that OS take care only of *prev/handle* fields, and nothing more. It is job of `_except_handler3` function to read other fields, read *scope table*, and decide, which handler to execute and when.



The source code of `_except_handler3` function is closed. However, Sanos OS, which have win32 compatibility layer, has the same functions redeveloped, which are somewhat equivalent to those in Windows<sup>34</sup>. Another reimplementations are present in Wine<sup>35</sup> and ReactOS<sup>36</sup>.

If the *filter* pointer is zero, *handler* pointer is the pointer to a *finally* code.

During execution, *previous try level* value in the stack is changing, so the `_except_handler` will know about current state 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;
    __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 53.6: 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
```

<sup>34</sup><https://code.google.com/p/sanos/source/browse/src/win32/msvcrt/except.c>

<sup>35</sup>[https://github.com/mirrors/wine/blob/master/dlls/msvcrt/except\\_i386.c](https://github.com/mirrors/wine/blob/master/dlls/msvcrt/except_i386.c)

<sup>36</sup>[http://doxygen.reactos.org/d4/df2/lib\\_2sdk\\_2crt\\_2except\\_2except\\_8c\\_source.html](http://doxygen.reactos.org/d4/df2/lib_2sdk_2crt_2except_2except_8c_source.html)

```

; scope table

CONST      SEGMENT
$T74622    DD      0fffffffH      ; previous try level
           DD      FLAT:$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:$T74622 ; scope table
    push    OFFSET FLAT:__except_handler3 ; handler
    mov     eax, DWORD PTR fs:__except_list
    push    eax ; prev
    mov     DWORD PTR fs:__except_list, esp
    add     esp, -16
    push    ebx ; saved 3 registers
    push    esi ; saved 3 registers
    push    edi ; saved 3 registers
    mov     DWORD PTR __$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
    inc     eax
$L74619:
$L74626:
    ret     0

; handler code

$L74618:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74608 ; 'access violation, can't r
    ↪ 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 SEH frame is being constructed in the stack. *Scope table* is located in the CONST segment— indeed, these fields will not be changed. An interesting thing is how *previous try level* variable is changed. Initial value is 0xFFFFFFFF (−1). The moment when body of `try` statement is opened is marked as an instruction writing 0 to the variable. The moment when body of `try` statement is closed, −1 is returned back to it. We also see addresses of filter and handler code. Thus we can easily see the structure of *try/except* constructs in the function.

Since the SEH setup code in the function prologue may be shared between many of functions, sometimes compiler inserts a call to `SEH_prolog()` function in the prologue, which do that. SEH cleanup code may be in the `SEH_epilog()` function.

Let's try to run this example in [tracer](#):

```
tracer.exe -l:2.exe --dump-seh
```

Listing 53.7: 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=0↵
  ↳ x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (↵
  ↳ ntdll.dll!_FinalExceptionHandler@16)
```

We see that SEH chain is consisting of 4 handlers.

First two are located in our example. Two? But we made only one? Yes, another one is setting up in [CRT](#) function `_mainCRTStartup()`, and as it seems, it handles at least [FPU](#) exceptions. Its source code can be found in MSVS installation: `crt/src/winxfldr.c`.

Third is SEH4 frame in `ntdll.dll`, and the fourth handler is not MSVC-related located in `ntdll.dll`, and it has 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 have two entries, each entry for each block. *Previous try level* is changing as execution flow entering or exiting try block.

Listing 53.8: MSVC 2003

```

$SG74606 DB 'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB 'yes, that is our exception', 0aH, 00H
$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
$T74644  DD    0fffffffH    ; previous try level for outer ↵
    ↵ block
        DD    FLAT:$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:$SG74617 ; 'hello!'
    call    _printf
    add     esp, 4
    push    0
    push    0
    push    0
    push    1122867    ; 00112233H
    call    DWORD PTR __imp__RaiseException@16
    push    OFFSET FLAT:$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
$L74633:
    xor     eax, eax
    mov     ecx, DWORD PTR __$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 to set a breakpoint on `printf()` function which is called from the handler, we may also see how yet another SEH handler is added. Perhaps, yet another machinery inside of 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 53.9: tracer.exe output

```

(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x00008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll↵
  ↵ .dll!ExecuteHandler2@20+0x3a)

```

```

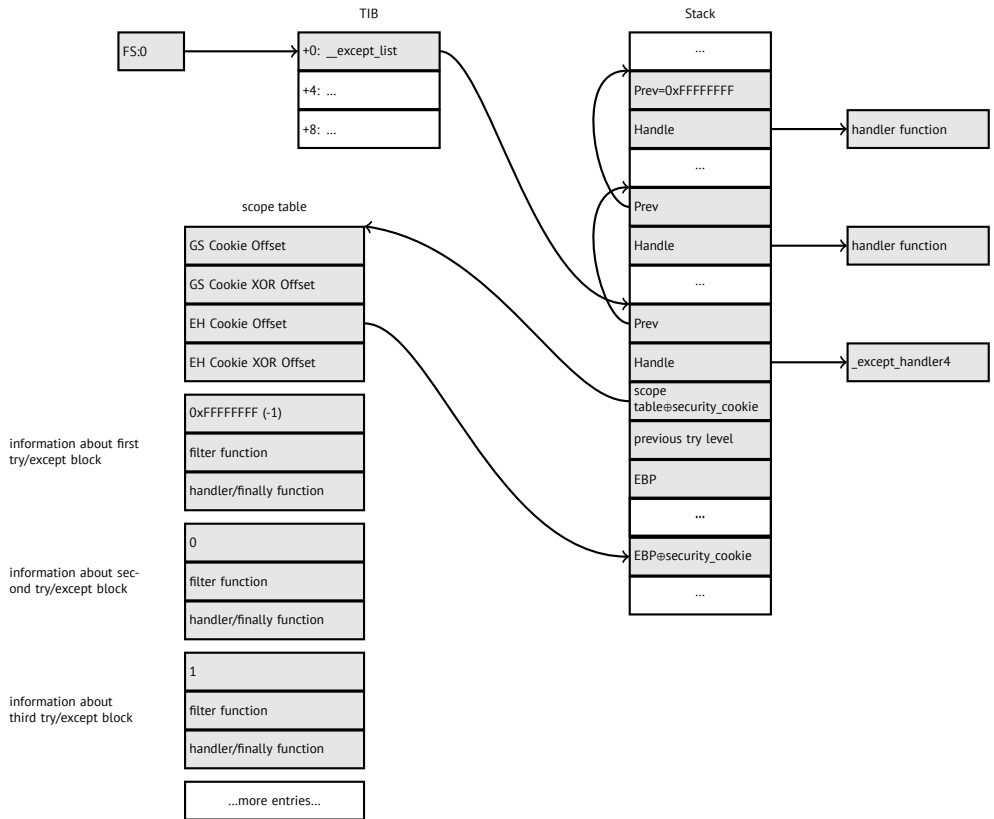
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!
  ↳ _except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.
  ↳ exe!main+0xb0) handler=0x40113b (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.
  ↳ 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:      GSCookieOffset=0xffffffff GSCookieXOROffset=0x0
                  EHCookieOffset=0xfffffccc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (
  ↳ ntdll.dll!__safe_se_handler_table+0x20) handler=0
  ↳ x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (
  ↳ ntdll.dll!_FinalExceptionHandler@16)

```

## SEH4

During buffer overflow (17.2) attack, address of the *scope table* can be rewritten, so starting at MSVC 2005, SEH3 was upgraded to SEH4 in order to have buffer overflow protection. The pointer to *scope table* is now **xored** with **security cookie**. *Scope table* extended to have a header, consisting of two pointers to *security cookies*. Each element have an offset inside of stack of another value: this is address of **stack frame** (EBP) **xored** with **security\_cookie** as well, placed in the stack. This value will be read during exception handling and checked, if it is correct. *Security cookie* in the stack is random each time, so remote attacker, hopefully, will not be able to predict it.

Initial *previous try level* is -2 in SEH4 instead of -1.



Here is both examples compiled in MSVC 2012 with SEH4:

Listing 53.10: 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$x SEGMENT
__sehable$_main DD 0fffffffEH ; GS Cookie Offset
                DD 00H ; GS Cookie XOR Offset
                DD 0fffffffccH ; EH Cookie Offset
                DD 00H ; EH Cookie XOR Offset
                DD 0fffffffEH ; previous try level
                DD FLAT:$LN12@main ; filter
                DD FLAT:$LN8@main ; handler
xdata$x ENDS
    
```

```

$T2 = -36          ; size = 4
_p$ = -32          ; size = 4
tv68 = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_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

```

Listing 53.11: 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$x   SEGMENT
__sehable$_main DD 0fffffffH ; GS Cookie Offset

```

```

        DD    00H                ; GS Cookie XOR Offset
        DD    0fffffff8H        ; EH Cookie Offset
        DD    00H                ; EH Cookie Offset
        DD    0fffffffEH        ; 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

```

```
$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
_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 a meaning of *cookies*: *Cookie Offset* is a difference between address of saved EBP value in stack and the  $EBP \oplus security\_cookie$  value in the stack. *Cookie XOR Offset* is additional difference between  $EBP \oplus security\_cookie$  value and what is stored in the stack. If this equation is not true, a process will be stopped due to stack corruption:

$$security\_cookie \oplus (CookieXOROffset + address\_of\_saved\_EBP) == stack[address\_of\_saved\_EBP + CookieOffset]$$

If *Cookie Offset* is  $-2$ , it is not present.

*Cookies* checking is also implemented in my [tracer](https://github.com/dennis714/tracer/blob/master/SEH.c), see <https://github.com/dennis714/tracer/blob/master/SEH.c> for details.

It is still possible to fall back to SEH3 in the compilers after (and including) MSVC 2005 by setting `/GS-` option, however, [CRT](#) code will use SEH4 anyway.

### 53.3.3 Windows x64

As you might think, it is not very fast thing to set up SEH frame at each function prologue. Another performance problem is to change *previous try level* value many times while function 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`, that is where OS exception handler takes all the information.

These are two examples from the previous section compiled for x64:

Listing 53.12: 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$filt$0 DD imagerel main$filt$0
        DD       imagerel main$filt$0+32
        DD       imagerel $unwind$main$filt$0
pdata  ENDS
xdata  SEGMENT
$unwind$main DD  020609H
        DD       030023206H
        DD       imagerel __C_specific_handler
        DD       01H
        DD       imagerel $LN9+8
        DD       imagerel $LN9+40
        DD       imagerel main$filt$0
        DD       imagerel $LN9+40
$unwind$main$filt$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$x  SEGMENT
main$filt$0 PROC
        push   rbp
        sub    rsp, 32
        mov    rbp, rdx
$LN5@main$filt$:
        mov    rax, QWORD PTR [rcx]
        xor    ecx, ecx
        cmp    DWORD PTR [rax], -1073741819; c0000005H
        sete   cl
        mov    eax, ecx
$LN7@main$filt$:
        add    rsp, 32
        pop    rbp
        ret    0
        int   3
main$filt$0 ENDP
text$x  ENDS

```

Listing 53.13: 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
$pdata$main DD imagerel $LN14
        DD    imagerel $LN14+95
        DD    imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT

```

```

$pdata$main$filt$0 DD imagerel main$filt$0
                   DD   imagerel main$filt$0+32
                   DD   imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
                   DD   imagerel main$filt$1+30
                   DD   imagerel $unwind$main$filt$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 $LN14+59
                   DD   imagerel main$filt$0
                   DD   imagerel $LN14+59
                   DD   imagerel $LN14+8
                   DD   imagerel $LN14+74
                   DD   imagerel main$filt$1
                   DD   imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
                   DD   050023206H
$unwind$main$filt$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$x  SEGMENT
main$filt$0 PROC
    push   rbp
    sub    rsp, 32
    mov    rbp, rdx
$LN10@main$filt$:
    mov    rax, QWORD PTR [rcx]
    xor    ecx, ecx
    cmp    DWORD PTR [rax], -1073741819; c0000005H
    sete   cl
    mov    eax, ecx
$LN12@main$filt$:
    add    rsp, 32
    pop    rbp
    ret    0
    int   3
main$filt$0 ENDP

main$filt$1 PROC
    push   rbp
    sub    rsp, 32
    mov    rbp, rdx
$LN6@main$filt$:
    mov    rax, QWORD PTR [rcx]
    mov    rdx, rcx
    mov    ecx, DWORD PTR [rax]
    call   filter_user_exceptions
    npad   1 ; align next label

```

```

$LN8@main$filt$:
    add     rsp, 32
    pop     rbp
    ret     0
    int     3
main$filt$1 ENDP
text$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=0x2
    ↪ %08X'
    call   printf
    cmp     ebx, 1122867; 00112233H
    jne    SHORT $LN2@filter_use
    lea    rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our 2
    ↪ exception'
    call   printf
    mov     eax, 1
    add     rsp, 32
    pop     rbp
    ret     0
$LN2@filter_use:
    lea    rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
    call   printf
    xor     eax, eax
    add     rsp, 32
    pop     rbp
    ret     0
filter_user_exceptions ENDP
_TEXT ENDS

```

Read [Sko12] for more detailed information about this.

Aside from exception information, `.pdata` is a section containing addresses of almost all function starts and ends, hence it may be useful for a tools targeting automated analysis.

### 53.3.4 Read more about SEH

[Pie], [Sko12].

## 53.4 Windows NT: Critical section

Critical sections in any OS are very important in multithreaded environment, mostly used for issuing a guarantee that only one thread will access some data, while blocking other threads and interrupts.

That is how CRITICAL\_SECTION structure is declared in [Windows NT](#) line OS:

Listing 53.14: (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 is how EnterCriticalSection() function works:

Listing 53.15: 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

```

The most important instruction in this code fragment is BTR (prefixed with LOCK): the zeroth bit is stored in CF flag and cleared in memory. This is [atomic operation](#), blocking all other CPUs to access this piece of memory (take a notice of LOCK prefix before BTR instruction). If the bit at LockCount was 1, fine, reset it and return from the function: we are in critical section. If not –critical section is already occupied by other thread, then wait. Wait is done there using WaitForSingleObject().

And here is how LeaveCriticalSection() function works:

Listing 53.16: 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, 0FFFFFFFFh
        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 summing LockCount value and 1 and stores result in EBX register, and at the same time 1 goes to 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 of memory.

LOCK prefix is very important: two threads, each of which working on separate CPUs or CPU cores may try to enter critical section and to modify the value in memory simultaneously, this will result in unpredictable behaviour.

**Part V**

**Tools**

# Chapter 54

# Disassembler

## 54.1 IDA

Older freeware version is available for downloading <sup>1</sup>.

Short hot-keys cheatsheet: [E.1](#)

---

<sup>1</sup><http://www.hex-rays.com/idapro/idadownfreeware.htm>

# Chapter 55

# Debugger

## 55.1 tracer

I use *tracer*<sup>1</sup> instead of debugger.

I stopped to use debugger eventually, since all I need from it is to spot a function's arguments while execution, or registers' state at some point. To load debugger each time is too much, so I wrote a small utility *tracer*. It has console-interface, working from command-line, enable us to intercept function execution, set breakpoints at arbitrary places, spot registers' state, modify it, etc.

However, as for learning purposes, it is highly advisable to trace code in debugger manually, watch how register's state changing (e.g. classic SoftICE, OllyDbg, WinDbg highlighting changed registers), flags, data, change them manually, watch reaction, etc.

## 55.2 OllyDbg

Very popular user-mode win32 debugger:

<http://www.ollydbg.de/>.

Short hot-keys cheatsheet: [E.2](#)

## 55.3 GDB

Not very popular debugger among reverse engineers, but very comfortable nevertheless. Some commands: [E.5](#).

---

<sup>1</sup><http://yurichev.com/tracer-en.html>

# Chapter 56

## System calls tracing

### 56.0.1 strace / dtruss

Will show which system calls (syscalls(51)) are called by process right now. For example:

```
# strace df -h
...
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such ↵
  ↵ file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF↵
  ↵ \1\1\1\0\0\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"...)
  ↵ 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|↵
  ↵ MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

Mac OS X has dtruss for the same aim.

The Cygwin also has strace, but if I understood correctly, it works only for .exe-files compiled for cygwin environment itself.

## Chapter 57

# Decompilers

There are only one known, publically available, high-quality decompiler to C code:

Hex-Rays:

<https://www.hex-rays.com/products/decompiler/>

## Chapter 58

# Other tools

- Microsoft Visual Studio Express<sup>1</sup>: Stripped-down free Visual Studio version, convenient for simple experiments. Some useful options: [E.3](#).
- Hiew<sup>2</sup> for small modifications of code in binary files.
- binary grep: the small utility for constants searching (or just any byte sequence) in a big pile of files, including non-executable: <https://github.com/yurichev/bgrep>.

---

<sup>1</sup><http://www.microsoft.com/express/Downloads/>

<sup>2</sup><http://www.hiew.ru/>

## **Part VI**

# **More examples**



## Chapter 59

# Task manager practical joke (Windows Vista)

I have only 4 CPU cores on my computer, so the Windows Task Manager shows only 4 CPU load graphs.

Let's see if it's possible to hack Task Manager slightly so it would detect more CPU cores on a computer.

Let us first think, how Task Manager would know number of cores? There are `GetSystemInfo()` win32 function present in win32 userspace which can tell us this. But it's not imported in `taskmgr.exe`. There are, however, another one in `NTAPI`, `NtQuerySystemInformation()`, which is used in `taskmgr.exe` in several places. To get number of cores, one should call this function with `SystemBasicInformation` constant in first argument (which is zero<sup>1</sup>).

Second argument should point to the buffer, which will receive all the information.

So we need 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 corresponding `PDB` file for exactly this executable and add all function names. It seems, Task Manager written in C++ and some of function names and classes are really speaking for themselves. There are classes `CAdapter`, `CNetPage`, `CPerfPage`, `CProcInfo`, `CProcPage`, `CSvcPage`, `CTaskPage`, `CUserPage`. Apparently, each class corresponding each tab in Task Manager.

I visited each call and I add comment with a value which is passed as the first function argument. There are "not zero" I wrote at some places, because, the value there was not clearly 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

<sup>1</sup>MSDN: [http://msdn.microsoft.com/en-us/library/windows/desktop/ms724509\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724509(v=vs.85).aspx)

all.

Dire...	T	Address	Text
wWinMain+50E	p		call cs:__imp_NtQuerySystemInformation; 0
wWinMain+542	p		call cs:__imp_NtQuerySystemInformation; 2
CPerfPage::TimerEvent(void)+200	p		call cs:__imp_NtQuerySystemInformation; not zero
InitPerfInfo(void)+2C	p		call cs:__imp_NtQuerySystemInformation; 0
InitPerfInfo(void)+F0	p		call cs:__imp_NtQuerySystemInformation; 8
CalcCpuTime(int)+5F	p		call cs:__imp_NtQuerySystemInformation; 8
CalcCpuTime(int)+248	p		call cs:__imp_NtQuerySystemInformation; 2
CPerfPage::CalcPhysicalMem(unsigned ...	p		call cs:__imp_NtQuerySystemInformation; not zero
CPerfPage::CalcPhysicalMem(unsigned ...	p		call cs:__imp_NtQuerySystemInformation; not zero
CProcPage::GetProcessInfo(void)+2B	p		call cs:__imp_NtQuerySystemInformation; 5
CProcPage::UpdateProcInfoArray(void)+...	p		call cs:__imp_NtQuerySystemInformation; 0
CProcPage::UpdateProcInfoArray(void)+...	p		call cs:__imp_NtQuerySystemInformation; 2
CProcPage::Initialize(HwND__)+201	p		call cs:__imp_NtQuerySystemInformation; 0
CProcPage::GetTaskListEx(void)+3C	p		call cs:__imp_NtQuerySystemInformation; 5

Figure 59.1: IDA: cross references to NtQuerySystemInformation()

Yes, the names are really speaking for themselves.

When I closely investigating each place where NtQuerySystemInformation(0, ?, ?, ?) is called, I quickly found what I need in the InitPerfInfo() function:

Listing 59.1: 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
.text:10000B4D0      xor     al, al
.text:10000B4D2      jmp    loc_10000B5EA
.text:10000B4D7 ; ↵
    ↵ -----
    ↵
.text:10000B4D7
.text:10000B4D7 loc_10000B4D7:                ; CODE ↵
    ↵ XREF: InitPerfInfo(void)+36
.text:10000B4D7      mov    eax, [rsp+0C78h+var_C50]
```

```

.text:1000B4DB      mov     esi, ebx
.text:1000B4DD      mov     r12d, 3E80h
.text:1000B4E3      mov     cs:?g_PageSize@@3KA, eax ; ↵
    ↳ ulong g_PageSize
.text:1000B4E9      shr     eax, 0Ah
.text:1000B4EC      lea    r13, __ImageBase
.text:1000B4F3      imul   eax, [rsp+0C78h+var_C4C]
.text:1000B4F8      cmp    [rsp+0C78h+var_C20], bp1
.text:1000B4FD      mov     cs:?g_MEMMax@@3_JA, rax ; ↵
    ↳ __int64 g_MEMMax
.text:1000B504      movzx  eax, [rsp+0C78h+var_C20] ; ↵
    ↳ number of CPUs
.text:1000B509      cmova  eax, ebp
.text:1000B50C      cmp    al, bl
.text:1000B50E      mov     cs:?g_cProcessors@@3EA, al ; ↵
    ↳ uchar g_cProcessors

```

`g_cProcessors` is a global variable, and this name was assigned by IDA according to [PDB](#) loaded from the Microsoft 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 returning structure format, which we can find in MSDN:

```

typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;

```

This is x64 system, so each `PVOID` takes 8 byte here. So all *reserved* fields in the structure takes  $24 + 4 * 8 = 56$ . Oh yes, this means, `var_C20` is the local stack is exactly `NumberOfProcessors` field of the `SYSTEM_BASIC_INFORMATION` structure.

Let's check if I'm right. Copy `taskmgr.exe` from `C:\Windows\System32` to some other PDB folder (so the *Windows Resource Protection* will not try to restore patched `taskmgr.exe`).

Let's open it in Hiew and find the place:

```

00 0000A8F8: 40386C2458      cmp     [rsp+0058], bp1
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: 488BF8        mov     rdi, rbx
00 0000A919: 498BD4        mov     rdx, r12
00 0000A91C: 8BCD        mov     ecx, ebp

```

Figure 59.2: Hiew: find the place to be patched

Let's replace MOVZX instruction by our. Let's pretend we've got 64 CPU cores. Add one additional NOP (because our instruction is shorter than original one):

```

00 0000A8F8: 40386C2458          cmp     [rsp+00000058],bp1
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,b1
00 0000A90E: 880574950100     mov     [000023E88],al
00 0000A914: 7645              jbe    0000A95B
00 0000A916: 488BFB           mov     rdi,rbx
00 0000A919: 498BD4           mov     rdx,r12
00 0000A91C: 8BCD             mov     ecx,ebp

```

Figure 59.3: Hiew: patch it

And it works! Of course, data in graphs is not correct. At times, Task Manager even shows overall CPU load more than 100%.

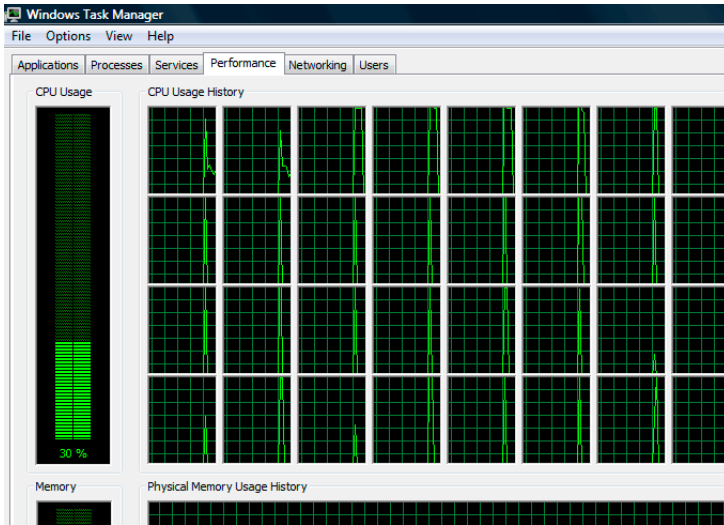


Figure 59.4: Fooled Windows Task Manager

I picked number of 64, because Task Manager is crashing if you try to set larger value. Apparently, Task Manager in Windows Vista was not tested on computer with larger count of cores. So there are probably some static data structures inside it limited to 64 cores.

## 59.1 Using LEA to load values

Sometimes, LEA is used in taskmgr.exe instead of MOV to set first argument of NtQuerySystemInformation():

## Listing 59.2: 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
        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

```

I honestly, don't know why, but that is what [MSVC](#) often does. Maybe this some kind of optimization and LEA works faster or better than load value using MOV?

Another example of such thing is: [49.5.1](#).

## Chapter 60

# Color Lines game practical joke

This is a very popular game with several implementations exist. I took one of them, called BallTriX, from 1997, available freely at <http://www.download-central.ws/Win32/Games/B/BallTriX/>. Here is how it looks: fig.60.1.

So let's see, will it be possible find random generator and do some trick with it. IDA quickly recognize 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                                ; ↗
    ↪ sub_402ACA+64 ...
.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
```

I'll call it "random". Let's not to dive into this function's code yet.

This function is referred from 3 places.

Here is 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:

```
.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 have only one argument. 10 is passed in first two cases and 5 in third. We may also notice that the board has size 10\*10 and there are 5 possible colors. This is it! The standard `rand()` function returns a number in 0..0x7FFF range and this is often inconvenient, so many programmers implement their own random functions which returns a random number in specified range. In our case, range is  $0..n - 1$  and  $n$  is passed as the sole argument to the function. We can quickly check this in any debugger.

So let's fix third function return at zero. I first replaced three instructions (PUSH/CALL/ADD) by **NOPs**. Then I add `XOR EAX, EAX` instruction, to clear EAX register.

```
.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 I did is replaced call to `random()` function by a code which always returns zero.

Let's run it now: [fig.60.2](#). Oh yes, it works<sup>1</sup>.

But why arguments to the `random()` functions are global variables? That's just because it's possible to change board size in game settings, so these values are not hardcoded. 10 and 5 values are just defaults.

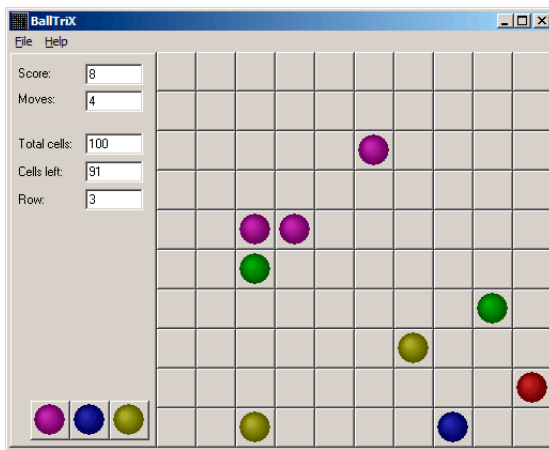


Figure 60.1: How this game looks usually

<sup>1</sup>I once did this as a joke for my coworkers with a hope they stop playing. They didn't.



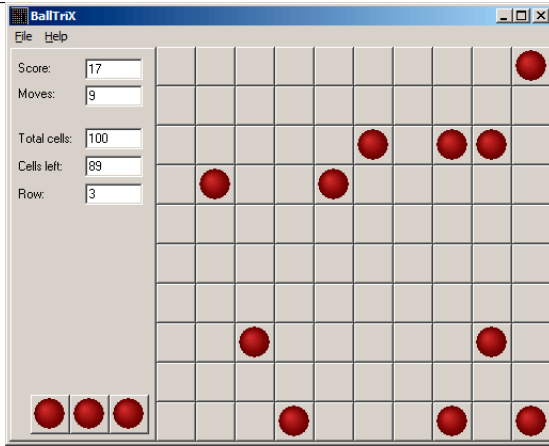


Figure 60.2: Practical joke works

## Chapter 61

# Minesweeper (Windows XP)

I'm not very good at playing Minesweeper, so I will try to reveal hidden mines in debugger.

As we know, Minesweeper places mines randomly, so there should be some kind of random numbers generator or call to the standard `rand()` C-function. What is really cool about reversing Microsoft products is that there are [PDB](#) file exist with symbols (function names, etc). When I load `winmine.exe` into [IDA](#), it downloads [PDB](#) file exactly for this executable and adds 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 developers. The function is very simple:

```
int Rnd(int limit)
{
    return rand() % limit;
};
```

(There are was no “limit” name in PDB-file; I named this argument so.)

So it returns a random value in range from 0 to specified limit.

Rnd() is called only from one place, this is function called StartGame(), and as it seems, this is exactly the code which place 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           ; ECX=↵
        ↳ ECX*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 to set board size, so the X (xBoxMac) and Y (yBoxMac) of board are global variables. They are passed to Rnd() and random coordinates are generated. Mine is placed by OR at 0x010036FA. And if it was placed before (it's possible if Rnd() pair will generate coordinates pair which already was generated), then TEST and JNZ at 0x010036E6 will jump to generation routine again.

cBombStart is the global variable containing total number of mines. So this is loop.

The width of array is 32 (we can conclude this by looking at SHL instruction, which multiplies one of the coordinates by 32).

The size of rgBlk global array can be easily determined by difference between rgBlk label in data segment and 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()+2
...
```

$864/32 = 27$ .

So the array size is  $27 * 32$ ? It is close to what we know: when I try to set board size to  $100 * 100$  in Minesweeper settings, it fallbacks to the board of size  $24 * 30$ .

So this is maximal board size here. And the array has fixed size for any board size.

So let's see all this in OllyDbg. I run Minesweeper, I attaching OllyDbg to it and I see memory dump at the address of `rgBlk` array(0x01005340) <sup>1</sup>.

So I got this memory dump of array:

Address	Hex dump
01005340	10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005360	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005370	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005380	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005390	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
010053A0	10 0F 0F 0F 0F 0F 0F 0F 8F 0F 10 0F 0F 0F 0F 0F
010053B0	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 10 0F 0F 0F 0F 0F
010053D0	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 10 0F 0F 0F 0F 0F
010053F0	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 10 0F 0F 0F 0F 0F
01005410	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 10 0F 0F 0F 0F 0F
01005430	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005440	10 8F 0F 0F 0F 0F 8F 0F 0F 8F 10 0F 0F 0F 0F 0F
01005450	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005460	10 0F 0F 0F 0F 8F 0F 0F 0F 8F 10 0F 0F 0F 0F 0F
01005470	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 0F 0F 0F 0F 0F
01005490	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
010054B0	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

OllyDbg, like any other hexadecimal editor, shows 16 bytes per line. So each 32-byte array row occupies here exactly 2 lines.

This is beginner level (9\*9 board).

There are some square structure can be seen visually (0x10 bytes).

I click "Run" in OllyDbg to unfreeze Minesweeper process, then I clicked randomly at Minesweeper window and trapped into mine, but now I see all mines: fig.61.1.

By comparing mine places and dump, we can conclude that 0x10 mean border, 0x0F—empty block, 0x8F—mine.

Now I added commentaries and also enclosed all 0x8F bytes into square brackets:

border :

<sup>1</sup>All addresses here are for Minesweeper for Windows XP SP3 english. They may differ for other service packs.

```

01005340 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #1:
01005360 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005370 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #2:
01005380 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005390 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #3:
010053A0 10 0F 0F 0F 0F 0F 0F 0F 0F[8F]0F 10 0F 0F 0F 0F 0F
010053B0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #4:
010053C0 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053D0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #5:
010053E0 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053F0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #6:
01005400 10 0F 0F[8F]0F 0F[8F]0F 0F 0F 10 0F 0F 0F 0F 0F 0F
01005410 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #7:
01005420 10[8F]0F 0F[8F]0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F 0F
01005430 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #8:
01005440 10[8F]0F 0F 0F 0F[8F]0F 0F[8F]10 0F 0F 0F 0F 0F 0F
01005450 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #9:
01005460 10 0F 0F 0F 0F[8F]0F 0F 0F[8F]10 0F 0F 0F 0F 0F 0F
01005470 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
border:
01005480 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005490 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F

```

Now I removed 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 0F 0F 0F[8F]0F
0F 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 screenshot.

What is interesting is that I can modify array right in OllyDbg. I removed all mines by changing all 0x8F bytes by 0x0F, and then what I got in Minesweeper:

## fig.61.2.

I also removed all them and add them at the first line: fig.61.3.

Well, debugger is not very convenient for eavesdropping (which was my goal anyway), so I wrote small utility to dump board contents:

```
// 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];

    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 `PID`<sup>2</sup> <sup>3</sup> and address of array (0x01005340 for Windows XP SP3 english) and it will dump it <sup>4</sup>.

It attaches to win32 process by `PID` and just read process memory by address.



Figure 61.1: Mines

<sup>2</sup>Program/process ID

<sup>3</sup>PID can be shown in Task Manager (enable it in "View → Select Columns")

<sup>4</sup>Compiled executable is here: [http://beginners.re/examples/minesweeper\\_WinXP/minesweeper\\_cheater.exe](http://beginners.re/examples/minesweeper_WinXP/minesweeper_cheater.exe)

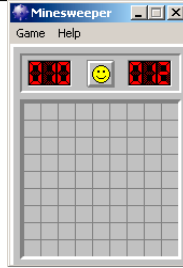


Figure 61.2: I removed all mines in debugger

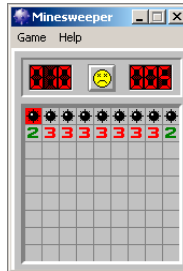


Figure 61.3: Mines I set in debugger

## 61.1 Exercises

- Why border bytes (0x10) are exist in array? What they are for if they are not visible in Minesweeper interface? How to do without them?
- As it turns out, there are more values possible (for open blocks, for flagged by user, etc). Try to find meaning of each.
- Modify my utility so it will remove all mines or set them by fixed pattern you want in the Minesweeper process currently running.
- Modify my utility so it can work without array address specified and without PDB file. Yes, it's possible to find board information in data segment of Minesweeper running process automatically. Hint: [F.5.1](#).



## Chapter 62

# Hand decompiling + Z3 SMT solver

Amateur cryptography is usually (unintentionally) very weak and can be broken easily—for cryptographers, of course.

But let's pretend we are not among these crypto-professionals.

I once found this one-way hash function, converting 64-bit value to another one and we need to try to reverse its flow back.

But what is hash-function? Simplest example is CRC32, an algorithm providing “stronger” checksum for integrity checking purposes. It is impossible to restore original text from the hash value, it just has much less information: there can be long text, but CRC32 result is always limited to 32 bits. But CRC32 is not cryptographically secure: it is known how to alter a text in that way so the resulting CRC32 hash value will be one we need. Cryptographical hash functions are protected from this. They are widely used to hash user passwords in order to store them in the database, like MD5, SHA1, etc. Indeed: an internet forum database may not contain user passwords (stolen database will compromise all user's passwords) but only hashes (a cracker will not be able to reveal passwords). Besides, an internet forum engine is not aware of your password, it should only check if its hash is the same as in the database, then it will give you access in this case. One of the simplest passwords cracking methods is just to brute-force all passwords in order to wait when resulting value will be the same as we need. Other methods are much more complex.

## 62.1 Hand decompiling

Here its assembly language listing in [IDA](#):

```

sub_401510      proc near
                ; ECX = input
                mov     rdx, 5D7E0D1F2E0F1F84h
                mov     rax, rcx           ; input
                imul   rax, rdx
                mov     rdx, 388D76AEE8CB1500h
                mov     ecx, eax
                and     ecx, 0Fh
                ror    rax, cl
                xor    rax, rdx
                mov     rdx, 0D2E9EE7E83C4285Bh
                mov     ecx, eax
                and     ecx, 0Fh
                rol    rax, cl
                lea    r8, [rax+rdx]
                mov     rdx, 8888888888888889h
                mov     rax, r8
                mul    rdx
                shr    rdx, 5
                mov     rax, rdx
                lea    rcx, [r8+rdx*4]
                shl    rax, 6
                sub    rcx, rax
                mov     rax, r8
                rol    rax, cl
                ; EAX = output
                retn
sub_401510      endp

```

The example was compiled by GCC, so the first argument is passed in ECX.

If Hex-Rays is not in list of our possessions, or we distrust to it, we may try to reverse this code manually. One method is to represent [CPU](#) registers as local C variables and replace each instruction by one-line equivalent expression, like:

```

uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
    rax=rcx;
    rax*=rdx;
    rdx=0x388D76AEE8CB1500;

```

```

    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=rdx;
    rdx=0xD2E9EE7E83C4285B;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+rdx;
    rdx=0x8888888888888889;
    rax=r8;
    rax*=rdx;
    rdx=rdx>>5;
    rax=rdx;
    rcx=r8+rdx*4;
    rax=rax<<6;
    rcx=rcx-rax;
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
};

```

If to be careful enough, this code can be compiled and will even work in the same way as original one.

Then, we will rewrite it gradually, keeping in mind all registers usage. Attention and focusing is very important here—any tiny typo may ruin all your work!

Here is a first step:

```

uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
    rax=rcx;
    rax*=rdx;
    rdx=0x388D76AEE8CB1500;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=rdx;
    rdx=0xD2E9EE7E83C4285B;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+rdx;

    rdx=0x8888888888888889;
    rax=r8;
    rax*=rdx;
    // RDX here is a high part of multiplication result
    rdx=rdx>>5;
    // RDX here is division result!
    rax=rdx;

```

```

    rcx=r8+rdx*4;
    rax=rax<<6;
    rcx=rcx-rax;
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
};

```

Next step:

```

uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
    rax=rcx;
    rax*=rdx;
    rdx=0x388D76AEE8CB1500;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=rdx;
    rdx=0xD2E9EE7E83C4285B;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+rdx;

    rdx=0x8888888888888889;
    rax=r8;
    rax*=rdx;
    // RDX here is a high part of multiplication result
    rdx=rdx>>5;
    // RDX here is division result!
    rax=rdx;

    rcx=(r8+rdx*4)-(rax<<6);
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
};

```

We may spot division using multiplication (15.3). Indeed, let's calculate divider in Wolfram Mathematica:

#### Listing 62.1: Wolfram Mathematica

```

In[1]:=N[2^(64 + 5)/16^8888888888888889]
Out[1]:=60.

```

We get this:

```

uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
    rax=rcx;
    rax*=rdx;
    rdx=0x388D76AEE8CB1500;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=rdx;
    rdx=0xD2E9EE7E83C4285B;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+rdx;

    rax=rdx=r8/60;

    rcx=(r8+rax*4)-(rax*64);
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
};

```

Another step:

```

uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    rax=input;
    rax*=0x5D7E0D1F2E0F1F84;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=0x388D76AEE8CB1500;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+0xD2E9EE7E83C4285B;

    rcx=r8-(r8/60)*60;
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
};

```

By simple reducing, we finally see that it's not [quotient](#) calculated, but division remainder:

```

uint64_t f(uint64_t input)
{

```

```

uint64_t rax, rbx, rcx, rdx, r8;

rax=input;
rax*=0x5D7E0D1F2E0F1F84;
rax=_lrotr(rax, rax&0xF); // rotate right
rax^=0x388D76AEE8CB1500;
rax=_lrotl(rax, rax&0xF); // rotate left
r8=rax+0xD2E9EE7E83C4285B;

return _lrotl (r8, r8 % 60); // rotate left
};

```

We end up on something fancy formatted source-code:

```

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>

#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B

uint64_t hash(uint64_t v)
{
    v*=C1;
    v=_lrotr(v, v&0xF); // rotate right
    v^=C2;
    v=_lrotl(v, v&0xF); // rotate left
    v+=C3;
    v=_lrotl(v, v % 60); // rotate left
    return v;
};

int main()
{
    printf ("%llu\n", hash(...));
};

```

Since we are not cryptanalysts we can't find an easy way to generate input value for some specific output value. Rotate instruction coefficients are look frightening—it's a warranty that the function is not bijective, it has collisions, or, speaking more simply, many inputs may be possible for one output.

Brute-force is not solution because values are 64-bit ones, that's beyond reality.

## 62.2 Now let's use Z3 SMT solver

Still, without any special cryptographical knowledge, we may try to break this algorithm using excellent SMT solver from Microsoft Research named Z3<sup>1</sup>. It is in fact theorem prover, but we will use it as SMT solver. In terms of simplicity, we may think about it as a system capable of solving huge equation systems.

Here is a Python source code:

```

1 from z3 import *
2
3 C1=0x5D7E0D1F2E0F1F84
4 C2=0x388D76AEE8CB1500
5 C3=0xD2E9EE7E83C4285B
6
7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5
  ↵ i6 outp', 64)
8
9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
18
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print (" outp=0x%X" % m[outp].as_long())

```

This will be our first solver.

We see variable definitions on line 7. These are just 64-bit variables.  $i1..i6$  are intermediate variables, representing values in registers between instruction executions.

Then we add so called constraints on lines 10..15. The very last constraint at 17 is most important: we will try to find input value for which our algorithm will produce 10816636949158156260.

Essentially, SMT-solver searches for (any) values that satisfy all constraints.

RotateRight, RotateLeft, URem – are functions from Z3 Python [API](#), they are not related to Python [PL](#).

Then we run it:

<sup>1</sup><http://z3.codeplex.com/>

```

...>python.exe 1.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 1364123924608584563,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
inp=0x12EE577B63E80B73
outp=0x961C69FF0AEFD7E4

```

“sat” mean “satisfiable”, i.e., solver was able to found at least one solution. The solution is printed inside square brackets. Two last lines are input/output pair in hexadecimal form. Yes, indeed, if we run our function with `0x12EE577B63E80B73` on input, the algorithm will produce the value we were looking for.

But, as we are noticed before, the function we work with is not bijective, so there are may be other correct input values. Z3 SMT solver is not capable of producing more than one result, but let's hack our example slightly, by adding line 19, meaning, look for any other results than this:

```

1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
18
19 s.add(inp!=0x12EE577B63E80B73)
20
21 print s.check()
22 m=s.model()
23 print m
24 print (" inp=0x%X" % m[inp].as_long())
25 print (" outp=0x%X" % m[outp].as_long())

```



Indeed, it found other correct result:

```
...>python.exe 2.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 10587495961463360371,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
inp=0x92EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

This can be automated. Each found result may be added as constraint and the next result will be searched for. Here is slightly sophisticated example:

```
1 from z3 import *
2
3 C1=0x5D7E0D1F2E0F1F84
4 C2=0x388D76AEE8CB1500
5 C3=0xD2E9EE7E83C4285B
6
7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i
   ↳ i6 outp', 64)
8
9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
18
19 # copped from http://stackoverflow.com/questions/11867611/
   ↳ z3py-checking-all-solutions-for-equation
20 result=[]
21 while True:
22     if s.check() == sat:
23         m = s.model()
24         print m[inp]
25         result.append(m)
26         # Create a new constraint the blocks the current model
27         block = []
28         for d in m:
```

```

29         # d is a declaration
30         if d.arity() > 0:
31             raise Z3Exception("uninterpreted functions are ↵
↳ not supported")
32         # create a constant from declaration
33         c=d()
34         if is_array(c) or c.sort().kind() == ↵
↳ Z3_UNINTERPRETED_SORT:
35             raise Z3Exception("arrays and uninterpreted ↵
↳ sorts are not supported")
36             block.append(c != m[d])
37         s.add(Or(block))
38     else:
39         print "results total=",len(result)
40         break

```

We got:

```

1364123924608584563
1234567890
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
1906652839273745429
11130024876128521237
15741710894555909141
6518338857701133333
5975809943035972467
15199181979890748275
10587495961463360371
results total= 16

```

So there are 16 correct input values are possible for 0x92EE577B63E80B73 as a result.

The second is 1234567890— it is indeed a value I used originally while preparing this example.

Let's also try to research our algorithm more. By some sadistic purposes, let's find, are there any possible input/output pair in which lower 32-bit parts are equal to each other?

Let's remove *outp* constraint and add another, at line 17:

```

1 from z3 import *
2
3 C1=0x5D7E0D1F2E0F1F84

```

```

4 C2=0x388D76AEE8CB1500
5 C3=0xD2E9EE7E83C4285B
6
7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5
  ↪ i6 outp', 64)
8
9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print ("outp=0x%X" % m[outp].as_long())

```

It is indeed so:

```

sat
[i1 = 14869545517796235860,
 i3 = 8388171335828825253,
 i5 = 6918262285561543945,
 inp = 1370377541658871093,
 outp = 14543180351754208565,
 i4 = 10167065714588685486,
 i2 = 5541032613289652645]
inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535

```

Let's be more sadistic and add another constraint: last 16-bit should be 0x1234:

```

1 from z3 import *
2
3 C1=0x5D7E0D1F2E0F1F84
4 C2=0x388D76AEE8CB1500
5 C3=0xD2E9EE7E83C4285B
6
7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5
  ↪ i6 outp', 64)
8
9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))

```

```

12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18 s.add(outp & 0xFFFF == 0x1234)
19
20 print s.check()
21 m=s.model()
22 print m
23 print (" inp=0x%X" % m[inp].as_long())
24 print ("outp=0x%X" % m[outp].as_long())

```

Oh yes, this possible as well:

```

sat
[i1 = 2834222860503985872,
 i3 = 2294680776671411152,
 i5 = 17492621421353821227,
 inp = 461881484695179828,
 outp = 419247225543463476,
 i4 = 2294680776671411152,
 i2 = 2834222860503985872]
inp=0x668EEC35F961234
outp=0x5D177215F961234

```

Z3 works very fast and it means that algorithm is weak, it is not cryptographical at all (like the most of amateur cryptography).

Will it be possible to tackle real cryptography by these methods? Real algorithms like AES, RSA, etc, can also be represented as huge system of equations, but these are that huge that are impossible to work with on computers, now or in near future. Of course, cryptographers are aware of this.

Another article I wrote about Z3 is [[Yur12](#)].

# Chapter 63

## Dongles

Occasionally I do software copy-protection [dongle](#) replacements, or “dongle emulators” and here are couple examples of my work <sup>1</sup>.

About one of not described cases you may also read here: [\[Yur12\]](#).

### 63.1 Example #1: MacOS Classic and PowerPC

I’ve got a program for MacOS Classic <sup>2</sup>, for PowerPC. The company who developed the software product was disappeared long time ago, so the (legal) customer was afraid of physical dongle damage.

While running without dongle connected, a message box with a text “Invalid Security Device” appeared. Luckily, this text string can be found easily in the executable binary file.

I was not very familiar both with Mac OS Classic and PowerPC, but I tried anyway.

[IDA](#) opens 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 error message, I’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          clrliw. %r0, %r3, 2  
↳ 24
```

<sup>1</sup>Read more about it: <http://yurichev.com/dongles.html>

<sup>2</sup>pre-UNIX MacOS

```

seg000:000C880C 40 82 00 40          bne      OK
seg000:000C8810 80 62 9F D8          lwz      %r3, ↵
    ↵ TC_aInvalidSecurityDevice
...

```

Yes, this is PowerPC code. The CPU is very typical 32-bit RISC of 1990s era. Each instruction occupies 4 bytes (just as in MIPS and ARM) and its names are somewhat resembling MIPS instruction names.

check1() is a function name I gave to it lately. BL is *Branch Link* instruction, e.g., intended for subroutines calling. The crucial point is BNE instruction jumping if dongle protection check is passed or not jumping if error is occurred: then the address of the text string being loaded into r3 register for the subsequent passage into message box routine.

From the [SK95] I've got to know the r3 register is used for values returning (and r4, in case of 64-bit values).

Another yet unknown instruction is CLRLWI. From [IBM00] I've got to know that this instruction do both clearing and loading. In our case, it clears 24 high bits from the value in r3 and put it to r0, so it is analogical to MOVZX in x86 (14.1.1), but it also sets the flags, so the BNE can check them after.

Let's take a look into check1() function:

```

seg000:00101B40          check1: # CODE XREF: seg000:00063↵
    ↵ E7Cp
seg000:00101B40          # sub_64070+160p ...
seg000:00101B40
seg000:00101B40          .set arg_8, 8
seg000:00101B40          mflr   %r0
seg000:00101B44          stw    %r0, arg_8(%sp)
seg000:00101B48          stwu   %sp, -0x40(%sp)
seg000:00101B4C          bl     check2
seg000:00101B50          nop
seg000:00101B54          lwz    %r0, 0x40+arg_8(%sp↵
    ↵ )
seg000:00101B58          addi   %sp, %sp, 0x40
seg000:00101B5C          mtlr   %r0
seg000:00101B60          blr
seg000:00101B60          # End of function check1

```

As I can see in IDA, that function is called from many places in program, but only r3 register value is checked right after each call. All this function does is calling other function, so it is **thunk function**: there is function prologue and epilogue, but r3 register is not touched, so check1() returns what check2() returns.

BLR<sup>3</sup> is seems return from function, but since IDA does functions layout, we

<sup>3</sup>(PowerPC) Branch to Link Register

probably do not need to be interesting in this. It seems, since it is a typical RISC, subroutines are called using [link register](#), just like in ARM.

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)
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    cmlwi  %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

```

I'm lucky again: some function names are left in the executable (debug symbols section? I'm not sure, since I'm not very familiar with the file format, maybe it is some kind of PE exports? (53.2.7)), like `.RBEFINDNEXT()` and `.RBEFINDFIRST()`. Eventually these functions are calling other functions with names like `.GetNextDeviceV`, `.USBSendPKT()`, so these are clearly dealing with USB device.

There are even a function named `.GetNextEve3Device()`—sounds familiar, there was Sentinel Eve3 dongle for ADB port (present on Macs) in 1990s.

Let's first take a look on how `r3` register is set before return simultaneously ignoring all we see. We know that "good" `r3` value should be non-zero, zero `r3` will lead execution flow to the message box with an error message.

There are two instructions `li %r3, 1` present in the function and one `li %r3, 0` (*Load Immediate*, i.e., loading value into register). The very first instruction at `0x001186B0`—frankly speaking, I don't know what it mean, I need some more time to learn PowerPC assembly language.

What we see next is, however, easier to understand: `.RBEFINDFIRST()` is called: in case of its failure, 0 is written into `r3` and we jump to `exit`, otherwise another function is called (`check3()`)—if it is failing too, the `.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 16-bit value.

`B` (meaning *branch*) is unconditional jump.

`BEQ` is 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                                .using dword_24B704, %r30
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  clr1wi. %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
seg000:001187CC 38 60 00 00  li   %r3, 0
seg000:001187D0 48 00 02 A0  b    exit
seg000:001187D4
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  clr1wi %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
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  clr1wi. %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
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, 0xFEAO
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
seg000:00118870          loc_118870: # CODE XREF: check3+128j
    ↪ j
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+13j
    ↪ Cj
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
    ↪ j
seg000:00118898 80 DE 00 00    lwz    %r6, dword_24B704
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     .RBEREAD
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
    ↪ j

```

```

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+18↵
↳ Cj
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 .RBEREAD
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+1↵
↳ C4j
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+1↵
↳ D8j
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+1↵
↳ ECj
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      .RBEREAD
seg000:00118948 60 00 00 00    nop
seg000:0011894C 54 60 04 3F    clrldwi.%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+214↵
    ↵ j
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+228↵
    ↵ j
seg000:00118970 28 1F 00 03    cmplwi %r31, 3
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
                                # 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    clrldwi.%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+264↵
    ↵ j
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+278↵
    ↵ j
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+280↵
    ↵ j
seg000:001189D0                # check3+288j
seg000:001189D0 57 A0 06 3E    clrldwi %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    clrldwi %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+2↵
    ↵ A4j
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+29↵
    ↵ Cj
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    clrldwi %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+2↵
    ↵ D8j
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+2↵
    ↵ ECj
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
seg000:00118A44
seg000:00118A44                good1:    # CODE XREF: check3+2↵
    ↵ F4j
seg000:00118A44                # check3+2FCj
seg000:00118A44 57 A0 06 3E    clrldwi %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    clrldwi %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+318↵
    ↵ j
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+19↵
    ↵ Cj
seg000:00118A6C                # check3+238j ...
seg000:00118A6C 38 60 00 00    li      %r3, 0
seg000:00118A70
seg000:00118A70                exit:   # CODE XREF: check3+44j
    # 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()`. The function is probably return some values from the dongle, so they are compared here with hard-coded variables using `CMPLWI`.

We also see that `r3` register is also filled before each call to `.RBEREAD()` by one of these values: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Probably memory address or something like that?

Yes, indeed, by googling these function names it is easy to find Sentinel Eve3 dongle manual!

I probably even do not need to learn other PowerPC instructions: all this function does is just calls `.RBEREAD()`, compare its results with constants and returns 1 if comparisons are fine or 0 otherwise.

OK, all we've got is that `check1()` should return always 1 or any other non-

zero value. But since I'm not very confident in PowerPC instructions, I will be careful: I will patch jumps in `check2()` at `0x001186FC` and `0x00118718`.

At `0x001186FC` I wrote bytes `0x48` and `0` thus converting `BEQ` instruction into `B` (unconditional jump): I spot its opcode in the code without even referring to [\[IBM00\]](#).

At `0x00118718` I wrote `0x60` and 3 zero bytes thus converting it to `NOP` instruction: I spot its opcode in the code too.

And all it works now without dongle connected.

Summarizing, such small modifications can be done with [IDA](#) and minimal assembly language knowledge.

## 63.2 Example #2: SCO OpenServer

An ancient software for SCO OpenServer from 1997 developed by a company disappeared long time ago.

There is a special dongle driver to be installed in the system, containing text strings: "Copyright 1989, Rainbow Technologies, Inc., Irvine, CA" and "Sentinel Integrated Driver Ver. 3.0".

After driver installation in SCO OpenServer, these device files are appeared in `/dev` filesystem:

```
/dev/rbsl8
/dev/rbsl9
/dev/rbsl10
```

The program without dongle connected reports error, but the error string cannot be found in the executables.

Thanks to [IDA](#), it does its job perfectly working out COFF executable used in SCO OpenServer.

I've tried to find "rbsl" 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/s1%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, 0FFFFFFFh
.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
.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
.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%s"
.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
.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 should communicate with driver somehow and that is how it is.

The only place `SSQC()` function 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() is called at least from 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 I 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+FDr
.data:004016D2                db 0
```

```

...
.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
.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
.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
.text:0000D6D1  loc_D6D1: ; CODE XREF: sys_info+98j
      ; 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
.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” are sounds familiar: there was a Sentinel Pro dongle by Rainbow with no memory, providing only one crypto-hashing secret function.

A short description about what hash function is, read here: [62](#).

But let’s back to the program. So the program can only check the presence or absence dongle connected. No other information can be written to such dongle with no memory. Two-character codes are commands (we can see how commands are handled in `SSQC()` function) and all other strings are hashed inside the dongle transforming into 16-bit number. The algorithm was secret, so it was not possible to write driver replacement or to remake dongle hardware emulating it perfectly. However, it was always possible to intercept all accesses to it and to find what constants the hash function results compared to. Needless to say it is possible to build a robust software copy protection scheme based on secret cryptographic hash-function: let it to encrypt/decrypt data files your software dealing with.

But let’s back to the code.

Codes 51/52/53 are used for LPT printer port selection. 3x/4x is for “family” selection (that’s how Sentinel Pro dongles are differentiated from each other: more than one dongle can be connected to 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 global variable `ctl_model`.

Another text string to be passed is “PRESS ANY KEY TO CONTINUE:”, but the result is not checked. I don’t know why, probably by mistake <sup>4</sup>.

<sup>4</sup>What a strange feeling: to reveal bugs in such ancient software.

Let's see where the value from the global variable `ctl_mode` is used.

One of such places 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\\bOKG]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 into decryption routine and printed.

Error strings decryption routine is seems simple [xoring](#):

```
.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 I was unable to find error messages in the executable files, because they are encrypted, this is popular practice.

Another call to SSQ() hashing function passes "offln" string to it and comparing result with 0xFE81 and 0x12A9. If it not so, it deals with some timer() function (maybe waiting for poorly connected dongle to be reconnected and check again?) and then decrypt another error message to dump.

```

.text:0000DA55      loc_DA55:                                ; CODE ↗
        ↳ XREF: sync_sys+24Cj
.text:0000DA55      push   offset a0ffln ; "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, 0FFFFFF8EFh
.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 I 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
.text:0000D9DA      mov     [ebp+var_18], 1
.text:0000D9E1      add    esp, 18h
.text:0000D9E4      call   pcw_kbhit
.text:0000D9E9      test   eax, eax
.text:0000D9EB      jz     short loc_D9FB

...

; this name I 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, 1 ↵
    ↳ Fh, 7, 1Eh, 1Ah

```

Dongle bypassing is pretty straightforward: just patch all jumps after CMP the relevant instructions.

Another option is to write our own SCO OpenServer driver.

### 63.2.1 Decrypting error messages

By the way, we can also try to decrypt all error messages. The algorithm, locating in `err_warn()` function is very simple, indeed:

Listing 63.1: Decrypting 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 supplied to the decrypting 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 I 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
```

The algorithm is simple [xoring](#): each byte is xored with a key, but key is increased by 3 after processing of each byte.

I wrote a simple Python script to check my insights:

Listing 63.2: 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, 0x
  ↵ x76,
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 decrypted message.

There are also other encrypted messages with corresponding keys. But needless to say that it is possible to decrypt them without keys. First, we may observe that key is byte in fact. It is because core decrypting instruction (XOR) works on byte level. Key is located in ESI register, but only byte part of ESI is used. Hence, key may be greater than 255, but its value will always be rounded.

As a consequence, we can just try brute-force, trying all possible keys in 0..255 range. We will also skip messages containing unprintable characters.

Listing 63.3: Python 3.x

```

#!/usr/bin/python
import sys, curses.ascii

msgs=[
[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x
  ↵ x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x
  ↵ x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x
  ↵ x5C,
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, 0x
  ↵ x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x
  ↵ x2B,
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, 0x
  ↵ x23,

```

```

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, 0 ↵
↵ x36,
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

```

And we got:

#### Listing 63.4: 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#ejwjbbl`nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cduhld#ghtme?!#!'!#!
message #3
key= 7 value= Bk<waoqNUpu$`yreao\wmpusj,bkIjh
key= 8 value= Mj?vfnrOjqv%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`dbefe
message #5
key= 17 value= check security device station
key= 18 value= `ijbh!td`tmhwx'efwfbf!tubuVnm!`!
```

There are some garbage, but we can quickly find English-language messages!

By the way, since algorithm is simple xoring encryption, the very same function can be used for encrypting messages. If we need, we can encrypt our own messages, and patch the program by inserting them.

### 63.3 Example #3: MS-DOS

Another very old software for MS-DOS from 1995 also developed by a company disappeared long time ago.

In the pre-DOS extenders era, all the software for MS-DOS were mostly rely on 16-bit 8086 or 80286 CPUs, so en masse code was 16-bit. 16-bit code is mostly same as you already saw in this book, but all registers are 16-bit and there are less number of instructions available.

MS-DOS environment has no any system drivers, any program may deal with bare hardware via ports, so here you may see OUT/IN instructions, which are mostly present in drivers in our times (it is impossible to access ports directly in [user mode](#) in all modern OS).

Given that, the MS-DOS program working with a dongle should access LPT printer port directly. So we can just search for such instructions. And yes, here it is:

```
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:

```

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
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
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

```

It is also Sentinel Pro “hashing” dongle as in the previous example. I figured out its type by noticing that a text strings are also passed here and 16 bit values are also returned and compared with others.

So that is how Sentinel Pro is accessed via ports. Output port address is usually 0x378, i.e., printer port, the data to the old printers in pre-USB era were passed to it. The port is one-directional, because when it was developed, no one can imagined someone will need to transfer information from the printer<sup>5</sup>. The only way to get information from the printer, is a status register on port 0x379, it contain such bits as “paper out”, “ack”, “busy” –thus printer may signal to the host computer that it is ready or not and if a paper present in it. So the dongle return information from one of these bits, by one bit at each iteration.

`_in_port_2` has address of status word (0x379) and `_in_port_1` has control register address (0x37A).

It seems, the dongle return information via “busy” flag at `seg030:00B9`: each bit is stored in the DI register, later returned at the function end.

What all these bytes sent to output port mean? I don’t know. Probably 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 a dongle checking routine:

```

00000000 struct_0      struc ; (sizeof=0x1B)
00000000 field_0      db 25 dup(?)          ; string(C)
00000019 _A          dw ?
0000001B struct_0      ends

dseg:3CBC 61 63 72 75+_Q  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', 0FFAEh>
dseg:3EBD 63 6F 70 79+   struct_0 <'copy', 0F557h>

```

<sup>5</sup>If to consider Centronics only. Following IEEE 1284 standard allows to transfer information from the printer.

```

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
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 _Q
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+1↵
    ↵ Fj
seg030:01AA          ; check_dongle+5Ej
seg030:01AA 33 C0          xor     ax, ax
seg030:01AC
seg030:01AC          loc_35B0C: ; CODE XREF: check_dongle+63↵
    ↵ j
seg030:01AC 5E          pop     si
seg030:01AD C9          leave
seg030:01AE CB          retf
seg030:01AE          check_dongle endp

```

Since the routine may be called too frequently, e.g., before each important software feature executing, and the dongle accessing process is generally slow (because of slow printer port and also slow MCU<sup>6</sup> in the dongle), so they probably added a way to skip dongle checking too often, using checking current time in `biostime()` function.

`get_rand()` function uses 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 dongle, and then the result of hashing is compared with correct value.

Text strings are seems to be chosen randomly as well.

And that 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

```

<sup>6</sup>Microcontroller unit

```

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

```

Dongle bypassing 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

```

Observant reader might recall that `strcpy()` C function usually requires two pointers in arguments, but we saw how 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. Read more about it here: [78](#).

So as you may see, `strcpy()` and any other function taking pointer(s) in arguments, works with 16-bit pairs.

Let's back to our example. `DS` is currently set to data segment located in the executable, that is where the text string is stored.

In the `sent_pro()` function, each byte of string is loaded at `seg030:00EF`: the `LES` instruction loads `ES:BX` pair simultaneously from the passed argument. The `MOV` at `seg030:00F5` loads the byte from the memory to which `ES:BX` pair points.

At `seg030:00F2` only 16-bit word is **incremented**, not segment value. This means, the string passed to the function cannot be located on two data segments boundaries.

## Chapter 64

# "QR9": Rubik's cube inspired amateur crypto-algorithm

Sometimes amateur cryptosystems appear to be pretty bizarre.

I was asked to reverse engineer an amateur cryptoalgorithm of some data crypting utility, source code of which was lost<sup>1</sup>.

Here is also [IDA](#) exported listing from original crypting utility:

```
.text:00541000 set_bit      proc near                ; CODE ↗
    ↘ XREF: rotate1+42
.text:00541000                                ; ↗
    ↘ rotate2+42 ...
.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
.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
```

<sup>1</sup>I also got permit from customer to publish the algorithm details



```

.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
.text:00541081 first_loop1_begin:      ; CODE ↵
    ↵ XREF: rotate1+2E
.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          jnl   short first_loop2_begin
.text:00541097          inc    edi
.text:00541098          add    ebx, 8
.text:0054109B          cmp    edi, 8
.text:0054109E          jnl   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      jnl   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:005410D1      align 10h
.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      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:005410F4                push   edi                      ; loop1
.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                jnl   short loc_5410F3
.text:00541107                inc    edi                      ; ↵
    ↵ increment loop2 counter
.text:00541108                add    ebx, 8
.text:0054110B                cmp    edi, 8
.text:0054110E                jnl   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
.text:00541119 loc_541119:                                ; CODE ↵
    ↵ XREF: rotate2+57
.text:00541119                xor    esi, esi                ; loop2 ↵
    ↵ counter
.text:0054111B
.text:0054111B loc_54111B:                                ; CODE ↵
    ↵ XREF: rotate2+4E
.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      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
.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      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
.text:005411AE      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 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
.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
.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
.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
.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
.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
.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
.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   __strdup
.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
.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          jnl    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
.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 ↵
    ↳ crypted 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:005413FE
.text:005413FF          align 10h
.text:00541400
.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)
.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
.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 crypted!\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]
.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 are given by me while analysis.

I started from top. Here is a function taking 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 file and report error in case of error:

```
.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
.text:00541348 loc_541348:
```

Get 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
.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 file size aligned on a 64-byte boundary. This is because this cryptoalgorithm works with only 64-byte blocks. Its operation is pretty straightforward: divide file size by 64, forget about remainder and add 1, then multiple by 64. The following code removes remainder as if value was already divided by 64 and adds 64. It is almost the same.

```
.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
```

Allocate buffer with aligned size:

```
.text:00541373                push   esi                ; Size
.text:00541374                call   _malloc
```

Call `memset()`, e.g., clears allocated buffer<sup>2</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 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 buffer, buffer size (aligned) and password string.

<sup>2</sup>`malloc() + memset()` could be replaced by `calloc()`

```
.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 output file. By the way, developer forgot to check if it was created correctly! 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
```

Newly created file handle is in the EDI register now. Write signature "QR9".

```
.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
```

Write 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 crypted 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 file and free allocated buffer:

```

.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

```

Here is 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=ftell (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);
};

```

Decrypting 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
.text:00541473      repe   cmpsb
.text:00541475      jz     short loc_541489

```

Report an error if signature is absent:

```

.text:00541477      push    offset aFileIsNotCrypt ; ↵
        ↳ "File is not crypted!\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 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=ftell (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 crypted!\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
.text:0054126A loc_54126A:

```

This fragment of code copies part of input buffer to internal array I named later "cube64". The size is in the ECX register. MOVSD means *move 32-bit dword*, so, 16 of 32-bit dwords 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 crypted 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 contain ↗
    ↘ amount of crypted data.
.text:0054129F      rep movsd
```

If EBP is not bigger that input argument size, then continue to 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 into function `rotate_all_with_password()`. It takes two arguments: password string and number. In `crypt()`, number 1 is used, and in the `decrypt()` function (where `rotate_all_with_password()` function is called too), 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 for character in 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()`, 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 password contains non-alphabetical latin character, it is skipped! Indeed, if we run crypting utility and try non-alphabetical latin characters in 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 "a" value (97) from 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 means, 26 Latin alphabet symbols will become values in range 0..23, (24 in total).

```
.text:005411EA
.text:005411EA skip_subtracting:                ; CODE ↵
    ↵ XREF: rotate_all_with_password+35
```

This is actually division via multiplication. Read more about it in the “Division by 9” section (15.3).

The code actually divides password character 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 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 ; .. it 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 remainder is 2, call rotate3(). The EDI is a second argument of the rotate\_all\_with\_password() function. As I already wrote, 1 is for crypting operations and 3 is for decrypting. So, here is a loop. When crypting, rotate1/2/3 will 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
.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 next character from password string.

```

.text:00541249 next_character_in_password:
.text:00541249      mov   al, [ebp+1]

```

**Increment** character pointer within 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 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 rotate1/2/3 functions. Each function calls two another functions. I eventually gave them names 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 array cube64:  $arg\_4 + arg\_0 * 8$ . Then shift a byte from an array by  $arg\_8$  bits right. Isolate 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

```

Value in the DL is 1 here. Shift left it by  $arg\_8$ . For example, if  $arg\_8$  is 4, value in the DL register became  $0x10$  or  $1000b$  in binary form.

```

.text:00541017             shl     dl, cl
.text:00541019             mov     cl, cube64[eax+esi*8]

```

Get bit from array and explicitly set one.

```

.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 state after shift was 0x10 or 1000b in binary form, there will be 0xEF after NOT instruction or 11101111b in binary form.

```
.text:00541034          not     dl
```

This instruction clears bit, in other words, it saves all bits in CL which are also set in DL except those in DL which are cleared. This means that if DL is e.g. 11101111b in binary form, all bits will be saved except 5th (counting from lowest bit).

```
.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 specific bit in array, or sets it otherwise.

We also know the array size is 64. First two arguments both in the `set_bit()` and `get_bit()` functions could be seen as 2D coordinates. Then array will be 8\*8 matrix.

Here is C representation of what we already know:

```
#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)     ((var) |= (bit))
#define REMOVE_BIT(var, bit)  ((var) &= ~(bit))

char 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);
};

int 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 rotate1/2/3 functions.

```
.text:00541070 rotate1      proc near
.text:00541070
```

Internal array allocation in local stack, its size 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 internal array:

```
.text:0054107D         lea   ebx, [esp+50h+↵
    ↵ internal_array_64]
.text:00541081
```

Two nested loops are here:

```
.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
.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     jnl  short first_loop2_begin
```



```
.text:00541097    inc     edi                ; increment loop 2 ↵
    ↵ counter
.text:00541098    add     ebx, 8             ; increment internal ↵
    ↵ array pointer by 8 at each loop 1 iteration
.text:0054109B    cmp     edi, 8
.text:0054109E    jnl    short first_loop1_begin
```

...we see that both loop counters are in range 0..7. Also they are used as the first and the second arguments of the `get_bit()` function. Third argument of the `get_bit()` is the only argument of `rotate1()`. What `get_bit()` returns, is being placed into internal array.

Prepare pointer to 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    second_loop1_begin:
.text:005410A9    xor     esi, esi          ; ESI is loop 2 ↵
    ↵ counter
.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    jnl    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, 0FFFFFFFFh
.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 contents from internal array to cube global array via `set_bit()` function, *but*, in different order! Now loop 1 counter is in range 7 to 0, **decrementing** at each iteration!

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]);
};
```

Not very understandable, but if we will 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, 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 endp

```

It is *almost* the same, except of different order of arguments of the `get_bit()` and `set_bit()`. Let's rewrite it in C-like code:

```

void rotate2 (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 (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 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 3D cube 8\*8\*8, where each element is bit, get\_bit() and set\_bit() take just coordinates of bit on input.

rotate1/2/3 functions are in fact rotating all bits in specific plane. Three functions are each for each cube side and v argument is setting plane in range 0..7.

Maybe, algorithm's author was thinking of 8\*8\*8 Rubik's cube<sup>3</sup>!

Yes, indeed.

Let's get closer into decrypt() function, I rewrote it here:

```

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;
    }
};

```

<sup>3</sup>[http://en.wikipedia.org/wiki/Rubik's\\_Cube](http://en.wikipedia.org/wiki/Rubik's_Cube)

```

    }
    while (i<sz);

    free (p);
};

```

It is almost the same except of `crypt()`, but password string is reversed by `strrev()`<sup>4</sup> standard C function and `rotate_all()` is called with argument 3.

This means, in case of decryption, each corresponding `rotate1/2/3` call will 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 need to undo effect of rotating one place in clockwise direction, rotate it thrice in counter-clockwise direction.

`rotate1()` is apparently for rotating "front" plane. `rotate2()` is apparently for rotating "top" plane. `rotate3()` is apparently for rotating "left" plane.

Let's get back to the core of `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 side (one of three) and plane (one of 8).  $3 \cdot 8 = 24$ , that is why two last characters of Latin alphabet are remapped to fit an alphabet of exactly 24 elements.

The algorithm is clearly weak: in case of short passwords, one can see, that in crypted file there are an original bytes of the original file in binary files editor.

Here is reconstructed whole source code:

```

#include <windows.h>

#include <stdio.h>
#include <assert.h>

```

<sup>4</sup>[http://msdn.microsoft.com/en-us/library/9hby7w40\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/9hby7w40(VS.80).aspx)

```

#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 (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++) 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++;
    };
};

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)
{
    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=ftell (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=ftell (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 crypted!\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;

```

};

# Chapter 65

# SAP

## 65.1 About SAP client network traffic compression

(Tracing connection between TDW\_NOCOMPRESS SAPGUI<sup>1</sup> environment variable to the pesky nagging pop-up window and actual data compression routine.)

It is known that network traffic between SAPGUI and SAP is not encrypted by default, it is rather compressed (read here<sup>2</sup> and here<sup>3</sup>).

It is also known that by setting environment variable *TDW\_NOCOMPRESS* to 1, it is possible to turn network packets compression off.

But you will see a nagging pop-up window cannot be closed:

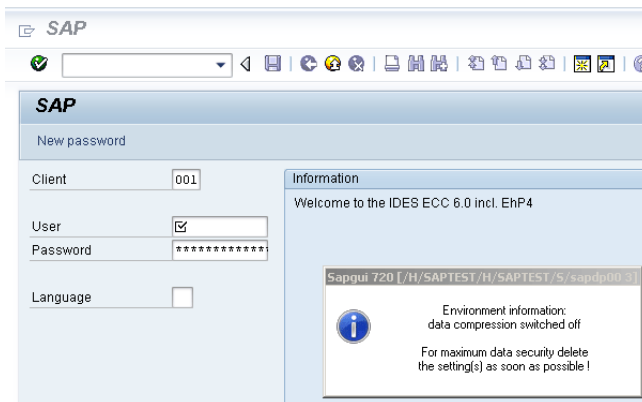


Figure 65.1: Screenshot

<sup>1</sup>SAP GUI client

<sup>2</sup><http://blog.yurichev.com/node/44>

<sup>3</sup><http://blog.yurichev.com/node/47>

Let's see, if we can remove the window somehow.

But before this, let's see what we already know. First: we know the environment variable `TDW_NOCOMPRESS` is checked somewhere inside of SAPGUI client. Second: string like "data compression switched off" must be present somewhere too. With the help of FAR file manager I found that both of these strings are stored in the `SAPguilib.dll` file.

So let's open `SAPguilib.dll` in `IDA` and search for "`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::CStringT::operator ↵
    ↵ PCXSTR
.text:6440D544      call   ds:mfc90_910
.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
```

String returned by `chk_env()` via second argument is then handled by MFC string functions and then `atoi()`<sup>4</sup> is called. After that, numerical value is stored to `edi+15h`.

Also take a look onto `chk_env()` function (I gave a name to it):

<sup>4</sup>standard C library function, converting number in string into number

```

.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      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      loc_64413F72:
.text:64413F72      mov     ecx, [ebp+DstSize]
.text:64413F75      push   ecx

```

```

.text:64413F76          mov     ecx, [ebp+arg_4]

; demangled name: ATL::CStringT<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
.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   0FFFFFFFh
.text:64413FA0          mov     ecx, [ebp+arg_4]

; demangled name: ATL::CStringT::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          loc_64413FB3:
.text:64413FB3          mov     ecx, [ebp+arg_4]

; demangled name: const char* ATL::CStringT::operator ↵
↳ PCXSTR
.text:64413FB6          call   ds:mfc90_910
.text:64413FBC          loc_64413FBC:
.text:64413FBC
.text:64413FBC          mov     esp, ebp
.text:64413FBE          pop    ebp
.text:64413FBF          retn
.text:64413FBF          chk_env          endp

```

Yes. `getenv_s()`<sup>5</sup> function is Microsoft security-enhanced version of `getenv()`<sup>6</sup>.

<sup>5</sup>[http://msdn.microsoft.com/en-us/library/tb2sfw2z\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/tb2sfw2z(VS.80).aspx)

<sup>6</sup>Standard C library returning environment variable

There is also a MFC string manipulations.

Lots of other environment variables are checked as well. Here is a list of all variables being checked and what SAPGUI could write to 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 ..."
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_FULLMENU	"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 ..."

Settings for each variable are written to the array via pointer in the EDI register. EDI is being 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
.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 "data record mode switched on" string? Yes, and here is the only reference in function



CDwsGui::PrepareInfoWindow(). How do I know class/method names?  
 There is a lot of special debugging calls writing to 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 contents of the pesky nagging pop-up window 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]
.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::↵
    ↵ CStringT<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::↵
    ↵ CStringT<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
.text:64404FD3  loc_64404FD3:
.text:64404FD3          xor     ebx, ebx
.text:64404FD5          inc     ebx
.text:64404FD6
.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
.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
.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"
.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
.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"
.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 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:644050BB      lea   ecx, [esi+2854h]
.text:644050BE      mov    [ebp+rc.right], ebx
.text:644050C1      mov    [ebp+rc.bottom], 0B4h

; demangled name: ATL::CStringT::GetLength(void)
.text:644050C8      call   ds:mfc90_3178
.text:644050CE      push   eax             ; ↵
    ↵ cchText
.text:644050CF      lea   ecx, [esi+2854h]

; demangled name: const char* ATL::CStringT::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, 0FFFFFFD4h
.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 ; ↵
    ↵ "\nCwsgui::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_epilog3
.text:6440519A      retn

```

```
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

ECX at function start gets pointer to object (since it is thiscall (31.1.1)-type of function). In our case, the object obviously has class type *CDwsGui*. Depends of option turned on in the object, specific message part will be concatenated to resulting message.

If value at `this+0x3D` address is not zero, compression is off:

```
.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"
.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
.text:6440501D bypass:
```

It is interesting, that finally, *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
.text:64405072  loc_64405072:

start drawing:

.text:64405072      push   edi                ; hWndd
.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 get SAPGUI working without the pesky nagging pop-up window appearing!

Now let's dig deeper and find connection between 0x15 offset in the `load_command_1` (I gave the name to the function) function and `this+0x3D` variable in the `CDws-Gui::PrepareInfoWindow`. Are we sure the value is the same?

I'm starting to search for all occurrences of 0x15 value in code. For a small programs like SAPGUI, it sometimes works. Here is the first occurrence I got:

```
.text:64404C19  sub_64404C19  proc near
.text:64404C19
.text:64404C19  arg_0        = dword ptr 4
.text:64404C19
.text:64404C19      push   ebx
.text:64404C1A      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 was called from the function named *CDwsGui::CopyOptions!* And thanks again for debugging information.

But the real answer in the function *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: array filled in the *load\_command\_line()* function is actually placed in the *CDwsGui* class but on *this+0x28* address. *0x15 + 0x28* is exactly *0x3D*. OK, we found the point where the value is copied to.

Let's also find other places where *0x3D* offset is used. Here is one of them in the *CDwsGui::SapguiRun* function (again, thanks to 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::CStringT::operator
  ↳ PCWSTR
.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::CStringT::operator ↵
  ↳ PCWSTR
.text:64409D75          call     ds:mfc90_910
.text:64409D75          ; no ↵
  ↳ arguments
.text:64409D7B          push    eax
.text:64409D7C          push    esi
.text:64409D7D          lea    ecx, [esi+8]
.text:64409D80          call   ↵
  ↳ CConnectionContext__CreateNetwork

```

Let's check our findings. Replace the `setz al` here to the `xor eax, eax / nop` instructions, clear `TDW_NOCOMPRESS` environment variable and run `SAPGUI`. Wow! There is no more pesky nagging window (just as expected, because variable is not set) but in Wireshark we can see the network packets are not compressed anymore! Obviously, this is the point where compression flag is to be set in the `CConnectionContext` object.

So, compression flag is passed in the 5th argument of function `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

```

Compression flag is passing here in the 5th argument to the `CNetwork::CNetwork` constructor.

And here is how `CNetwork` constructor sets a flag in the `CNetwork` object according to the 5th argument *and* an another variable which probably could affect network packets compression too.

```

.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
.text:64411E09          xor     eax, eax          ; EAX -> 0
    ↪ 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 *this+0x3A4* address.

Now let's dig across *SAPguilib.dll* for *0x3A4* value. And here is the second occurrence in the *CDwsGui::OnClientMessageWrite* (endless thanks for 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
    ↪ 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
.text:64406FAC          push   ecx                ; int
.text:64406FAD          push   eax                ; Src
.text:64406FAE          push   dword ptr [edi+28C0h] ; int
    ↪ int
.text:64406FB4          call  sub_644055C5        ; actual compression routine
    ↪ 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 into *sub\_644055C5*. In it we can only see 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 aErrorCsrcompre ;↵
    ↵ "\nERROR: CsrCompress: invalid handle"
.text:64417481      call   eax ; dword_644F94C8
.text:64417483      add   esp, 4
```

Voilà! We've found the function which actually compresses data. As I revealed in past <sup>7</sup>, this function is used in SAP and also open-source MaxDB project. So it is available in sources.

Doing last check here:

```
.text:64406F79      cmp     dword ptr [ecx+3A4h], 1
.text:64406F80      jnz    compression_flag_is_zero
```

Replace JNZ here for unconditional JMP. Remove environment variable TDW\_NOCOMPRESS. Voilà! In Wireshark we see the client messages are not compressed. Server responses, however, are compressed.

So we found exact connection between environment variable and the point where data compression routine may be called or may be bypassed.

## 65.2 SAP 6.0 password checking functions

While returning again to my SAP 6.0 IDE installed in VMware box, I figured out I forgot the password for SAP\* account, then it back to my memory, but now I got error message «*Password logon no longer possible - too many failed attempts*», since I've spent all these attempts in trying to recall it.

First extremely good news is the full *disp+work.pdb* PDB-file is supplied with SAP, it contain almost everything: function names, structures, types, local variable and argument names, etc. What a lavish gift!

<sup>7</sup>[http://conus.info/utils/SAP\\_pkt\\_decompr.txt](http://conus.info/utils/SAP_pkt_decompr.txt)

I got TYPEINFODUMP<sup>8</sup> utility for converting PDB files into something readable and greppable.

Here is an example of function information + its arguments + its local variables:

```

FUNCTION ThVmcSysEvent
  Address:      10143190  Size:      675 bytes  Index:      ↗
    ↳ 60483  TypeIndex:  60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, ↗
    ↳ unsigned short*)
Flags: 0
PARAMETER events
  Address: Reg335+288  Size:      4 bytes  Index:      60488  ↗
    ↳ TypeIndex:      60489
  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

```

<sup>8</sup><http://www.debuginfo.com/tools/typeinfodump.html>

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 is: *debugging* calls (there are plenty of them) are very useful. Here you can also notice *ct\_level* global variable<sup>9</sup>, reflecting current trace level. There is a lot of such debugging inclusions 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:hdl       ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock

```

If current trace level is bigger or equal to threshold defined in the code here, debugging message will be written to log files like *dev\_w0*, *dev\_disp*, and other *dev\** files.

Let's do grepping on file I got with the help of TYPEINFODUMP utility:

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

I got:

```

FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt

```

<sup>9</sup>More about trace level: [http://help.sap.com/saphelp\\_nwpi71/helpdata/en/46/962416a5a613e8e1000000a155369/content.htm](http://help.sap.com/saphelp_nwpi71/helpdata/en/46/962416a5a613e8e1000000a155369/content.htm)

```

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::GetTypeNames
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 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 string this function I found may write to 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, I quickly noticed the problem is exactly in it. It is called from *chckpass()* function –one of the password checking functions.

First, I would like to be sure I'm at the correct point:

Run my [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
```

Call path is: *syssigni()* -> *DylSigni()* -> *dychkurs()* -> *usrexist()* -> *chckpass()*.

Number 0x35 is an error returning in *chckpass()* at that point:

```
.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! I can successfully login now.

By the way, if I try to pretend I forgot the password, fixing *chckpass()* function return value at 0 is enough to bypass 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 *password\_attempt\_limit\_exceeded()* function is that at the very beginning of it, this call might be seen:

```
lea    rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call   sapparam
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 *sapgpparam()* used to query value of some configuration parameter. This function can be called from 1768 different places. It seems, with the help of this information, we can easily find places in code, control flow of which can be affected by specific configuration parameters.

It is really sweet. Function names are very clear, much clearer than in the Oracle RDBMS. It seems, *disp+work* process written in C++. It was apparently rewritten some time ago?

# Chapter 66

## Oracle RDBMS

### 66.1 V\$VERSION table in the Oracle RDBMS

Oracle RDBMS 11.2 is a huge program, main module `oracle.exe` contain approx. 124,000 functions. For comparison, Windows 7 x86 kernel (`ntoskrnl.exe`) – approx. 11,000 functions and Linux 3.9.8 kernel (with default drivers compiled) – 31,000 functions.

Let's start with an easy question. Where Oracle RDBMS get all this information, when we execute such simple statement in SQL\*Plus:

```
SQL> select * from V$VERSION;
```

And we've got:

```
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 we may find a string `V$VERSION`?

As of `win32-version`, `oracle.exe` file contain the string, which can be investigated easily. But we can also use object (`.o`) files from Linux version of Oracle RDBMS since, unlike `win32` version `oracle.exe`, function names (and global variables as well) are preserved there.

So, `kqf.o` file contain `V$VERSION` string. The object file is in the main Oracle-library `libserver11.a`.

A reference to this text string we may find in the `kqfviv` table stored in the same file, `kqf.o`:

Listing 66.1: `kqf.o`

```
.rodata:0800C4A0 kqfviv          dd 0Bh                ; DATA↵
    ↪ XREF: kqfchk:loc_8003A6D
.rodata:0800C4A0                ; ↵
    ↪ kqfgbn+34
.rodata:0800C4A4                dd offset _2__STRING_10102_0 ; ↵
    ↪ "GV$WAITSTAT"
.rodata:0800C4A8                dd 4
.rodata:0800C4AC                dd offset _2__STRING_10103_0 ; ↵
    ↪ "NULL"
.rodata:0800C4B0                dd 3
.rodata:0800C4B4                dd 0
.rodata:0800C4B8                dd 195h
.rodata:0800C4BC                dd 4
.rodata:0800C4C0                dd 0
.rodata:0800C4C4                dd 0FFFFFFC1CBh
.rodata:0800C4C8                dd 3
.rodata:0800C4CC                dd 0
.rodata:0800C4D0                dd 0Ah
.rodata:0800C4D4                dd offset _2__STRING_10104_0 ; ↵
    ↪ "V$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 0FFFFFFC003h
.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
.rodata:0800C51C                dd 15h
.rodata:0800C520                dd 0
.rodata:0800C524                dd 0FFFFFFC1EDh
```

```

.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 0FFFFFFC1EEh
.rodata:0800C558      dd 5
.rodata:0800C55C      dd 0

```

By the way, often, while analysing Oracle RDBMS internals, you may ask yourself, why functions and global variable names are so weird. Supposedly, since Oracle RDBMS is very old product and was developed in C in 1980-s. And that was a time when C standard guaranteed function names/variables support only up to 6 characters inclusive: «6 significant initial characters in an external identifier»<sup>1</sup>

Probably, the table `kqfvw` contain most (maybe even all) views prefixed with `V$`, these are *fixed views*, present all the time. Superficially, by noticing cyclic recurrence of data, we can easily see that each `kqfvw` 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., there are described 1023 of all possible *fixed views*. We will return to this number later.

As we can see, there is not much information in these numbers in fields. The very first number is always equals to name of view (without terminating zero. This is correct for each element. But this information is not very useful.

We also know that information about all fixed views can be retrieved from *fixed view* named `V$FIXED_VIEW_DEFINITION` (by the way, the information for this view is also taken from `kqfvw` and `kqfvip` tables.) By the way, there are 1023 elements too.

```

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='↵
    ↵ V$VERSION';

VIEW_NAME
-----
VIEW_DEFINITION
-----
    ↵

```

<sup>1</sup>Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988)

```
V$VERSION
select BANNER from GV$VERSION where inst_id = USERENV('
↳ Instance')
```

So, V\$VERSION is some kind of *think 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
```

Tables prefixed as X\$ in the Oracle RDBMS– is service tables too, undocumented, cannot be changed by user and they are refreshed dynamically.

Let's also try to search the text `select BANNER from GV$VERSION where inst_id = USERENV('Instance')` in the `kqf.o` file and we find it in the `kqfvip` table:

Listing 66.2: `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(indx,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(bitand(cfflg,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 too. The second field pointing to another table, containing table fields for this *fixed view*. As of V\$VERSION, this table contain only two elements, first is 6 and second is BANNER string (the number (6) is this string length) and after, *terminating* element contain 0 and *null C-string*:

Listing 66.3: 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 `kqfviv` and `kqfvip` tables, we may get SQL-statements which are executed when user wants to query information from specific *fixed view*.

So I wrote an oracle tables<sup>2</sup> program, so to gather all this information from Oracle RDBMS for Linux object files. For V\$VERSION, we may find this:

Listing 66.4: Result of oracle tables

```
kqfviv_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 ↵
    ↵ 0x4
kqfvip_element.statement: [select BANNER from GV$VERSION where ↵
    ↵ inst_id = USERENV('Instance')]
kqfvip_element.params:
[BANNER]
```

and:

Listing 66.5: Result of oracle tables

```
kqfviv_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0 ↵
    ↵ xxxffc192 0x1
kqfvip_element.statement: [select inst_id, banner from ↵
    ↵ x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

GV\$VERSION *fixed view* is distinct from V\$VERSION in only that way that it contains one more field with *instance* identifier. Anyway, we stuck at the table X\$VERSION. Just like any other X\$-tables, it is undocumented, however, we can query it:

<sup>2</sup>[http://yurichev.com/oracle\\_tables.html](http://yurichev.com/oracle_tables.html)

```
SQL> select * from x$version;
```

```
ADDR          INDX    INST_ID
```

```
-----  
BANNER  
-----
```

```
↵
```

```
ODBAF574          0          1  
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - ↵  
  ↵ Production
```

```
...
```

This table has additional fields like ADDR and INDX.

While scrolling `kqf.o` in [IDA](#) we may spot another table containing pointer to the X\$VERSION string, this is `kqftab`:

Listing 66.6: `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 X\$-table names, apparently, to all Oracle RDBMS 11.2 X\$-tables. But again, we have not enough information. I have no idea, what `kqvt` string means. `kq` prefix may means *kernel* and *query*. `v`, apparently, means *version* and `t` – *type*? Frankly speaking, I do not know.

The table named similarly can be found in `kqf.o`:

Listing 66.7: `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"
.rodata:0808C3CC          kqftap_param <6, offset ↵
↳ _2__STRING_5017_0, 601h, 0, 0, 0, 50h, 0, 0> ; "BANNER"
.rodata:0808C3F0          kqftap_param <0, offset ↵
↳ _2__STRING_0_0, 0, 0, 0, 0, 0, 0, 0>
```

It contain information about all fields in the X\$VERSION table. The only reference to this table present in the `kqftap` table:

Listing 66.8: `kqf.o`

```
.rodata:08042680          kqftap_element <0, offset ↵
↳ kqvt_c_0, offset kqvrow, 0> ; element 0x1f6
```

It is interesting that this element here is `0x1f6`th (502nd), just as a pointer to the X\$VERSION string in the `kqftab` table. Probably, `kqftap` and `kqftab` tables are complement each other, just like `kqfvip` and `kqfviv`. We also see a pointer to the `kqvrow()` function. Finally, we got something useful!

So I added these tables to my oracle tables<sup>3</sup> utility too. For X\$VERSION I've got:

Listing 66.9: Result of oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0↵
↳ xffffc075 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
```

<sup>3</sup>[http://yurichev.com/oracle\\_tables.html](http://yurichev.com/oracle_tables.html)

```
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 called 6 times in row (from the `qerfxFetch()` function) while querying `X$VERSION` table.

Let's run [tracer](#) in the `cc` mode (it will comment 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
```

```
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 ; "xkqvvsn 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
    ↪ xc98cd58
    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.%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 %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
    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]=0↵
↳ xacce190
    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

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]=0↵
↳ xc98cd58
    call    _kghfrf            ; 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                                ; EAX=0
_kqvrow_  endp

```

Now it is easy to see that row number is passed from outside of function. The function returns the string constructing it as follows:

String 1	Using vsnstr, vsnnum, vsnban global variables. Calling sprintf().
String 2	Calling kkvxsn().
String 3	Calling lmxver().
String 4	Calling npinli(), nrtnsvrs().
String 5	Calling lxvers().

That's how corresponding functions are called for determining each module's version.

## 66.2 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, this table's contents is cleared each time table querying. Are we able to find why? Let's back to tables we already know: `kqftab` and `kqftap` which were generated with `oracle tables4` help, containing all information about X\$-tables, now we can see here, the `ksmlrs()` function is called to prepare this table's elements:

Listing 66.10: Result of oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0x
  ↳ xffffc0bb 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=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x
  ↳ x0
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 the `tracer` help it is easy to see this function is called each time we query the X\$KSMLRU table.

<sup>4</sup>[http://yurichev.com/oracle\\_tables.html](http://yurichev.com/oracle_tables.html)



Here we see a references to the `ksmsplu_sp()` and `ksmsplu_jp()` functions, each of them call the `ksmsplu()` finally. At the end of the `ksmsplu()` function we see a call to the `memset()`:

Listing 66.11: `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
.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 would take a risk, block `memset()` call and see what will happen?

Let's run `tracer` with the following options: set breakpoint at `0x434C7A` (the point where `memset()` arguments are to be passed), thus, that `tracer` set program counter EIP at this point to the point where passed to the `memset()` arguments are to be cleared (at `0x434C8A`) It can be said, we just simulate an unconditional jump from the address `0x434C7A` to `0x434C8A`.

```

tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0↗
    ↪ x00434C8A)

```

(Important: all these addresses are valid only for win32-version of Oracle RDBMS 11.2)

Indeed, now we can query X\$KSMLRU table as many times as we want and it is not clearing 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 behaviour, but as an experiment of locating piece of code we need, that is perfectly suit our needs!

## 66.3 V\$TIMER table in Oracle RDBMS

V\$TIMER is another *fixed view*, reflecting 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>5</sup>)

It is interesting the periods are different for Oracle for win32 and for Linux. Will we able to find a function generating this value?

As we can see, this information is finally taken from 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
-----
VIEW_DEFINITION
```

<sup>5</sup>[http://docs.oracle.com/cd/B28359\\_01/server.111/b28320/dynviews\\_3104.htm](http://docs.oracle.com/cd/B28359_01/server.111/b28320/dynviews_3104.htm)

```

GV$TIMER
select inst_id,ksutmtim from x$ksutm

```

Now we stuck in a small problem, there are no references to value generating function(s) in the tables kqftab/kqftap:

Listing 66.12: Result of oracle tables

```

kqftab_element.name: [X$KSUTM] ? : [ksutm] 0x1 0x4 0x4 0x0 0x
↳ xffffc09b 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

```

Let's try to find a string KSUTMTIM, and we find 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 function kqfd\_DRN\_ksutm\_c() is mentioned in 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 are a function `ksugtm()` referenced here. Let's see what's in it (Linux x86):

Listing 66.13: `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]
            mov    [edx], eax
            mov    eax, 4
            mov    esp, ebp
            pop    ebp
            retn
ksugtm      endp
```

Almost the same code in win32-version.

Is this the function we are looking for? Let's see:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0 ↵
↳ x4
```

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 66.14: 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 returning via 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
mov     eax, [ebp+arg_0]
call   $+5
pop     ebx
nop
; PIC mode
mov     ebx, offset _GLOBAL_OFFSET_TABLE_
mov     dword ptr [eax], 0
call   sltrgtime64 ; 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 `sltrgtime64()` and division of its result by 10 (15.3))  
 And win32-version:

```

_slgcs  proc near ; CODE XREF: ↗
↳ _dbgefghTtElResetCount+15 ; _dbgerRunActions+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, 0CCCCCDh
        mul   edx
        shr   edx, 3
        mov    eax, edx
        mov    esp, ebp
        pop   ebp
        retn
_slgcs  endp

```

It is just result of `GetTickCount()`<sup>6</sup> divided by 10 (15.3).

<sup>6</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408(v=vs.85).aspx)

Voilà! That's why win32-version and Linux x86 version show different results, just because they are generated by different OS functions.

*Drain* apparently means *connecting* specific table column to specific function.

I added the table `kqfd_tab_registry_0` to oracle tables<sup>7</sup>, now we can see, how table column's variables are *connected* to specific functions:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tab1_fetch] [NULL] [NULL] [↗
↳ kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tab1_fetch] [NULL] [NULL] ↗
↳ [kqfd_DRN_ksusg_c]
```

*OPN*, apparently, *open*, and *DRN*, apparently, meaning *drain*.

<sup>7</sup>[http://yurichev.com/oracle\\_tables.html](http://yurichev.com/oracle_tables.html)

## Chapter 67

# Handwritten assembly code

### 67.1 EICAR test file

This .COM-file is intended for antivirus testing, it is possible to run in in MS-DOS and it will print string: “EICAR-STANDARD-ANTIVIRUS-TEST-FILE!”<sup>1</sup>.

Its important property is that it’s entirely consisting of printable ASCII-symbols, which, in turn, makes possible to create it in any text editor:

```
X50!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE! \r
↳ $H+H*
```

Let’s decompile it:

```
; initial conditions: SP=0FFFEh, 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 = FFFEh and SS:[FFFE] = 214Fh
0105 25 40 41    and      ax, 4140h
; AX = 140h, SP = FFFEh 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 = FFFEh and SS:[FFFE] = 214Fh
010A 34 5C       xor      al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010C 50          push     ax
010D 5A          pop      dx
```

<sup>1</sup>[https://en.wikipedia.org/wiki/EICAR\\_test\\_file](https://en.wikipedia.org/wiki/EICAR_test_file)



```

; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh 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

```

I added comments about 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, address of which is passed in DS:DX. By the way, the string should be terminated with '\$' 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 register (AH and DX) values we need;
- preparing INT 21 and INT 20 opcodes in memory;
- executing INT 21 and INT 20.

By the way, this technique is widely used in shellcode constructing, when one need to pass x86-code in the string form.

Here is also a list of all x86 instructions which has printable opcodes: [A.6.6](#).

# Chapter 68

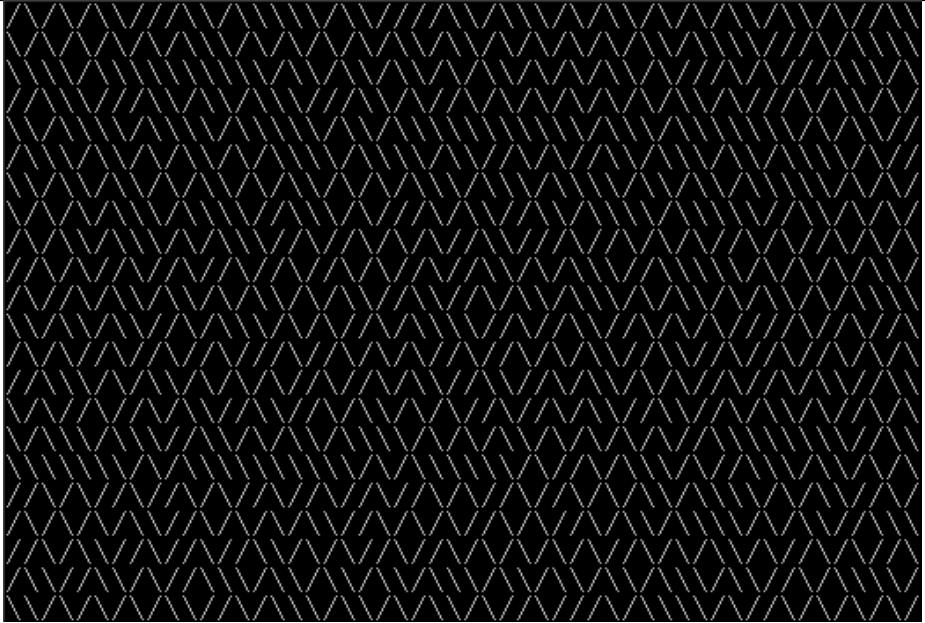
## Demos

Demos (or demomaking) was an excellent exercise in mathematics, computer graphics programming and very tight x86 hand coding.

### 68.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10

All examples here are MS-DOS .COM files.

In [\[al12\]](#) we can read about one of the most simplest possible random maze generators. It just prints slash or backslash character randomly and endlessly, resulting something like:



There are some known implementations for 16-bit x86.

### 68.1.1 Trixter's 42 byte version

The listing taken from his website<sup>1</sup>, but comments are mine.

```

00000000: B001      mov     al,1      ; set 40x25 ↵
    ↵ videomode
00000002: CD10      int     010
00000004: 30FF      xor     bh,bh     ; set videopage ↵
    ↵ 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

```

<sup>1</sup><http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/>

```

00000013: E440      in      al,040
00000015: 9D          popf                    ; enable ↵
    ↵ interrupts by restoring IF flag
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

```

Pseudo-random value here is in fact the time passed from the system boot, taken from 8253 time chip, the value increases by one 18.2 times per second.

By writing zero to port 43h, we mean the command is "select counter 0", "counter latch", "binary counter" (not BCD<sup>2</sup> value).

Interrupts enabled back with POPF instruction, which restores IF flag as well.

It is not possible to use IN instruction with other registers instead of AL, hence that shuffling.

### 68.1.2 My attempt to reduce Trixter's version: 27 bytes

We can say that since we use timer not to get precise time value, but pseudo-random one, so we may not spent time (and code) to disable interrupts. Another thing we might say that we need only bit from a low 8-bit part, so let's read only it.

I reduced the code slightly and I'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 '/'

```

<sup>2</sup>Binary-coded decimal

```

; output character to screen
00000013: B40E      mov     ah,00E
00000015: CD10      int     010
00000017: E2EA      loop   00000003
; exit to DOS
00000019: CD20      int     020

```

### 68.1.3 Take a random memory garbage as a source of randomness

Since it is MS-DOS, there are no memory protection at all, we can read from whatever address. Even more than that: simple LODSB instruction will read byte from DS:SI address, but it's not a problem if register values are not setted up, let it read 1) random bytes; 2) from random memory place!

So it is suggested in Trixter webpage<sup>3</sup> to use LODSB without any setup.

It is also suggested that SCASB instruction can be used instead, because it sets flag according to the byte it read.

Another idea to minimize code is to use INT 29h DOS syscall, which just prints character stored in AL register.

That is what Peter Ferrie and Andrey “herm1t” Baranovich did (11 and 10 bytes)<sup>4</sup>:

Listing 68.1: 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 use value in AL register, it subtract random memory byte value from 5Ch value in AL. JP is rare instruction, here it used for checking 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 random memory byte, but by sum of bits, this (hoperfully) makes result more distributed.

It is possible to make this even shorter by using undocumented x86 instruction SALC (AKA SETALC) (“Set AL CF”). It was introduced in NEC V20 CPU and sets AL to 0xFF if CF is 1 or to 0 if otherwise. So this code will not run on 8086/8088.

<sup>3</sup><http://trixter.oldschool.org/2012/12/17/maze-generation-in-thirteen-bytes/>

<sup>4</sup><http://pferrie.host22.com/misc/10print.htm>

## Listing 68.2: Peter Ferrie: 10 bytes

```

; AL is random at this point
00000000: AE          scasb
; CF is set accoring 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](#)<sup>5</sup> code of backslash (“\”) is 0x5C and 0x2F for slash (“/”). So we need to convert one (pseudo-random) bit in CF flag to 0x5C or 0x2F value.

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 just optput it to screen.

### 68.1.4 Conclusion

It is also worth adding that result may be different in DOSBox, [Windows NT](#) and even MS-DOS, due to different conditions: timer chip may be emulated differently, initial register contents may be different as well.

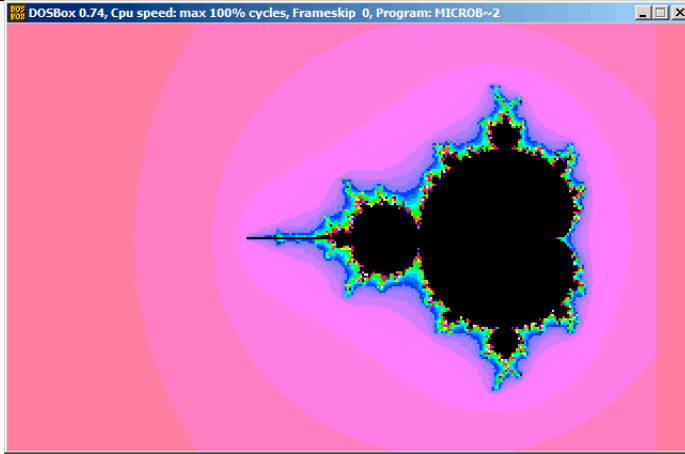
## 68.2 Mandelbrot set

Just found a demo<sup>6</sup> written by “Sir\_Lagsalot” in 2009, drawing Mandelbrot set which is just a x86 program with executable file size only 64 bytes. There are only 30 16-bit x86 instructions.

Here it is what it draws:

<sup>5</sup>American Standard Code for Information Interchange

<sup>6</sup>Download it [here](#).



Let's try to understand, how it works.

## 68.2.1 Theory

### A word about complex numbers

Complex number is a number consisting of two (real (Re) and imaginary (Im) parts).

Complex plane is a two-dimensional plane where any complex number can be placed: real part is one coordinate and imaginary part is another.

Some basic rules we need to know:

- Addition:  $(a + bi) + (c + di) = (a + c) + (b + d)i$

In other words:

$$\text{Re}(\text{sum}) = \text{Re}(a) + \text{Re}(b)$$

$$\text{Im}(\text{sum}) = \text{Im}(a) + \text{Im}(b)$$

- Multiplication:  $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$

In other words:

$$\text{Re}(\text{product}) = \text{Re}(a) \cdot \text{Re}(c) - \text{Re}(b) \cdot \text{Re}(d)$$

$$\text{Im}(\text{product}) = \text{Im}(b) \cdot \text{Im}(c) + \text{Im}(a) \cdot \text{Im}(d)$$

- Square:  $(a + bi)^2 = (a + bi)(a + bi) = (a^2 - b^2) + (2ab)i$

In other words:

$$\text{Re}(\text{square}) = \text{Re}(a)^2 - \text{Im}(a)^2$$

$$\text{Im}(\text{square}) = 2 \cdot \text{Re}(a) \cdot \text{Im}(a)$$

## How to draw Mandelbrot set

Mandelbrot set is a set of points for which  $z_{n+1} = z_n^2 + c$  (where  $z$  and  $c$  are complex numbers and  $c$  is starting value) recursive sequence is not approach infinity.

In plain English language:

- Enumerate all points on screen.
- Check, if specific point is in Mandelbrot set.
- Here is how to check it:
  - Represent point as complex number.
  - Get square of it.
  - Add starting value of point to it.
  - Goes off limits? Break, if yes.
  - Move point to the new place at coordinates we just calculated.
  - Repeat all this for some reasonable number of iterations.
- Moving point was still in limits? Draw point then.
- Moving point eventually gone off limits?
  - (For black-white image) do not draw anything.
  - (For colored image) transform iterations number to some color. So the color will shows the speed at which point gone off limits.

Here is Pythonesque algorithms I wrote for both complex and integer number representations:

Listing 68.3: 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

# colored
for each point on screen P:
    iterations = if check_if_is_in_set (P)
    map iterations to color
    draw color point

```

Integer version is where operations on complex numbers are replaced to integer operations according to rules I described above.

Listing 68.4: 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 C# source I get from Wikipedia article<sup>7</sup>, but I modified it so it prints iteration numbers instead of some symbol<sup>8</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))
                    {
                        realTemp2 = (realTemp * realTemp) - (↵
↳ imagTemp * imagTemp) - realCoord;
                        imagTemp = (2 * realTemp * imagTemp) - ↵
↳ imagCoord;
                        realTemp = realTemp2;
                        arg = (realTemp * realTemp) + (imagTemp↵
↳ * imagTemp);
                        iterations += 1;
                    }
                    Console.Write("{0,2:D} ", iterations);
                }
                Console.Write("\n");
            }
            Console.ReadKey();
        }
    }
}
```

<sup>7</sup><http://goo.gl/KJ9g>

<sup>8</sup>Here is also executable file: [http://beginners.re/examples/mandelbrot/dump\\_iterations.exe](http://beginners.re/examples/mandelbrot/dump_iterations.exe)

```
}  
}  
}
```

Here is resulting file, which is too wide to include it here:

<http://beginners.re/examples/mandelbrot/result.txt>.

Maximal iteration number is 40, so when you see 40 in this dump, this mean this point was wandering 40 iterations but never gone off limits. Number  $n$  less then 40 mean that point remaining inside bounds only for  $n$  iterations, then it gone outside it.

There is a cool demo available at <http://demonstrations.wolfram.com/MandelbrotSetDoodle/>, it shows visually how the point is moving on plane on each iteration at some specific point. I made two screenshots.

First, I clicked inside yellow area and we see that trajectory (green lines) is eventually swirled at some point inside: fig.68.1. This mean, the point I clicked belongs to Mandelbrot set.

Then I clicked outside yellow area and we see much more chaotic point movement, which is quickly goes off bounds: fig.68.2. This mean the point not belongs to Mandelbrot set.

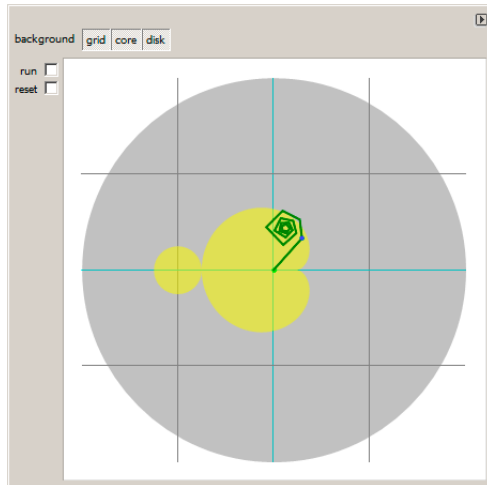


Figure 68.1: I clicked inside yellow area

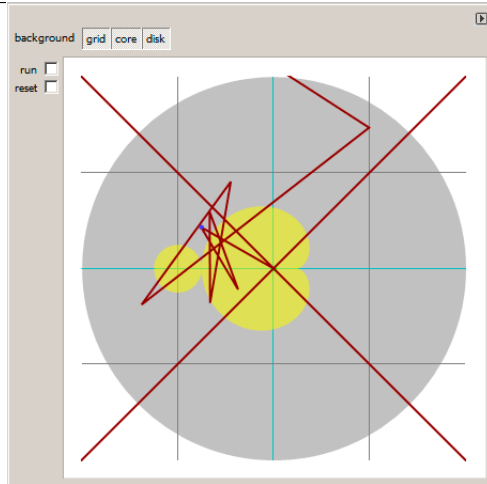


Figure 68.2: I clicked outside yellow area

Another good demo there is: <http://demonstrations.wolfram.com/IteratesForTheMandelbrotSet/>.

## 68.2.2 Let's back to the demo

The demo, although very tiny (just 64 bytes or 30 instructions), implements the common algorithm I described here, but using some coding tricks.

Source code is easily downloadable, so I got it, but I also added my comments:

Listing 68.5: 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 0xFFFF
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 then 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 rage -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 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
62
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 encoded by one byte, so the buffer size is 0xFA00 bytes. It is addressed as ES:DI registers pairs.

ES should be 0xA000 here, because this is segment address of VGA video buffer, but storing 0xA000 to ES requires at least 4 bytes (PUSH 0A000h / POP ES). Read more about 16-bit MS-DOS memory model: [78](#).

Assuming, BX is zero here, and Program Segment Prefix is at zeroth address, 2-byte LES AX, [BX] instruction will store 0x20CD to AX and 0x9FFF to ES. So, the program will start to draw 16 pixels or bytes before actual video buffer. But this is MS-DOS, there are no memory protection, so nothing will crash. That's why you see red strip of 16 pixels width at right. Whole picture is shifted left by 16 pixels. This is the price of 2 bytes saving.

- Eternal loop processing each pixel. Probably, most common way to enumerate all pixels on screen is two loops: one for X-coordinate, another for Y-coordinate. But then you'll need to multiplicates coordinates to find a byte in VGA video buffer. Author of this demo decide to do it otherwise: enumerate all bytes in video buffer by one single loop instead of two, and get coordinates of current point using division. Resulting coordinates are: X in range of  $-100..99$  and Y in range of  $-256..63$ . You may see on screenshot that picture is somewhat shifted to the right part of screen. That's because the

biggest heart-shaped black hole is usually drawn on coordinates 0,0 and these are shifted here to right. Could author just subtract 160 from value to get X in range of  $-160..159$ ? Yes, but instruction `SUB DX, 160` takes 4 bytes, while `DEC DH-2` bytes (which subtracts `0x100` (256) from `DX`). So the whole picture shifted is the cost of another 2 bytes of saved space.

- Check, if the current point is inside Mandelbrot set. The algorithm is the same I described.
  - The loop is organized used `LOOP` instructions which use `CX` register as counter. Author could set iteration number to some specific number, but he didn't: 320 is already in `CX` (was set at line 35), and this is good maximal iteration number anyway. We save here some space by not reloading `CX` register with other value.
  - `IMUL` is used here instead of `MUL`, because we work with signed values: remember that 0,0 coordinates should be somewhere near screen center. The same thing about `SAR` (arithmetic shift for signed values): it's used instead of `SHR`.
  - Another idea is to simplify bounds check. We would need to check coordinate pair, i.e., two variables. What author does is just checks thrice for overflow: two square operations and one addition. Indeed, we use 16-bit registers, which holds signed values in range of  $-32768..32767$ , so if any of coordinate is greater than 32767 during signed multiplication, this point is definitely out of bounds: we jump to `MandelBreak` label.
  - There are also division by 64 (`SAR` instruction). 64 sets scale. Try to increase value and you will get closer look, or to decrease to more distant look.
- We are at `MandelBreak` label, there are two ways of getting here: loop ended with `CX=0` (point is inside Mandelbrot set); or because overflow was happened (`CX` still holds some value). Now we write low 8-bit part of `CX` (`CL`) to the video buffer. Default palette is rough, nevertheless, 0 is black: hence we see black holes in places where points are in Mandelbrot set. Palette can be initialized at program start, but remember, that's only 64 bytes program!
  - Program is running in eternal loop, because additional check where to stop, or user interface is additional instructions.

Some other optimization tricks:

- 1-byte `CWD` is used here for clearing `DX` instead of 2-byte `XOR DX, DX` or even 3-byte `MOV DX, 0`.
- 1-byte `XCHG AX, CX` is used instead of 2-byte `MOV AX, CX`. Current `AX` value is not needed here anyway.

- DI (position in video buffer) is not initialized, and it is 0xFFFFE at start<sup>9</sup>. That's OK, because the program works for all DI in range of 0..0xFFFF eternally, and user will not notice it was started off the screen (last pixel of 320\*200 video buffer is at the address 0xF9FF). So some work is actually done off the limits of screen. Otherwise, you'll need additional instructions to set DI to 0; check for video buffer end.

### 68.2.3 My “fixed” version

Listing 68.6: 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
31
32 xor bx,bx
33 xor si,si

```

<sup>9</sup>More information about initial register values: <https://code.google.com/p/corkami/wiki/InitialValues#DOS>



```

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

```

I made attempt to fix all these oddities: now palette is smooth grayscale, video buffer is at correct place (lines 19..20), picture is drawing on center of screen (line 30), program eventually ends and waiting for user keypress (lines 58..68). But now it's much bigger: 105 bytes (or 54 instructions)<sup>10</sup>.

<sup>10</sup>You can experiment by yourself: get DosBox and NASM and compile it as: `nasm fiole.asm -fbin -o file.com`

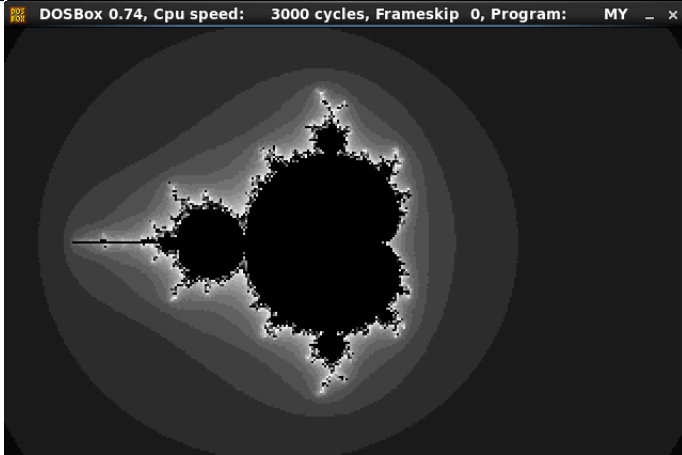


Figure 68.3: My “fixed” version

## **Part VII**

# **Examples of reversing proprietary file formats**

## Chapter 69

# Millenium game save file

The “Millenium Return to Earth” is an ancient DOS game (1991), allowing to mine resources, build ships, equip them to other planets, and so on<sup>1</sup>.

Like many other games, it allows to save all game state into file.

Let’s see, if we can find something in it.

So there is a mine in the game. Mines at some planets work faster, or slower on another. Resource set is also different.

Here I see what resources are mined at the time: fig.69.1. I save a game state. This is a file of size 9538 bytes.

I wait some “days” here in game, and now we’ve got more resources at the mine: fig.69.2. I saved game state again.

Now let’s try just to do binary comparison of the save files using 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
```

<sup>1</sup>It can be downloaded for free [here](#)

```

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 unfull here, there are more differences, but I cut result to the most interesting.

At first state, I have 14 “units” of hydrogen and 102 “units” of oxygen. I have 22 and 155 “units” respectively at the second state. If these values are saved into save-file, we should see this in difference. And indeed so. There are 0x0E (14) at 0xBDE position and this value is 0x16 (22) in new version of file. This could be for hydrogen. There is 0x66 (102) at position 0xBDC in old version and 0x9B (155) in new version of file. This could be for oxygen.

There both of files I put on my website for those who wants to inspect them (or experiment) more: [http://beginners.re/examples/millennium\\_DOS\\_game/](http://beginners.re/examples/millennium_DOS_game/).

Here is a new version of file opened in Hiew, I marked values related to resources mined in the game: fig.69.3. I checked each, and these are. These are clearly 16-bit values: not a strange thing for DOS 16-bit software where *int* has 16-bit width.

Let’s check our assumptions. I’m writing 1234 (0x4D2) value at first position (this should be hydrogen): fig.69.4.

Then I loaded changed file into game and taking a look on mine statistics: fig.69.5. So yes, this is it.

Now let’s try to finish the game as soon as possible, set maximal values everywhere: fig.69.6.

0xFFFF is 65535, so yes, we now have a lot of resources: fig.69.7.

I skipped some “days” in game and oops! I have lower amount of some resources: fig.69.8. That’s just overflow. Game developer probably didn’t thought about such high amounts of resources, so there are probably no overflow check, but mine is “working” in the games, resources are added, hence overflow. I shouldn’t be that greedy, I suppose.

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 are often can be easily patched like that.



Figure 69.1: Mine: state 1



Figure 69.2: Mine: state 2

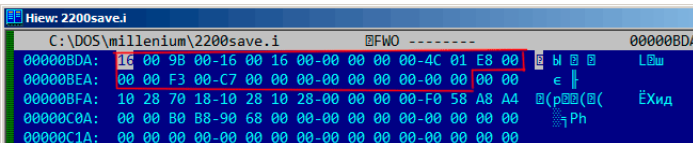


Figure 69.3: Hiew: state 1

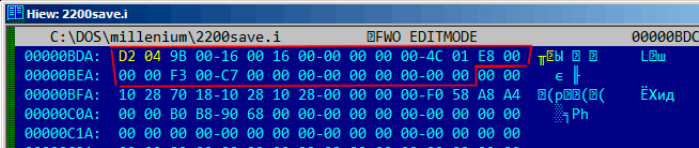


Figure 69.4: Hiew: let's write 1234 (0x4D2) there



Figure 69.5: Let's check for hydrogen value

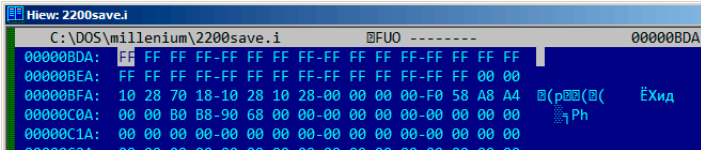


Figure 69.6: Hiew: let's set maximal values

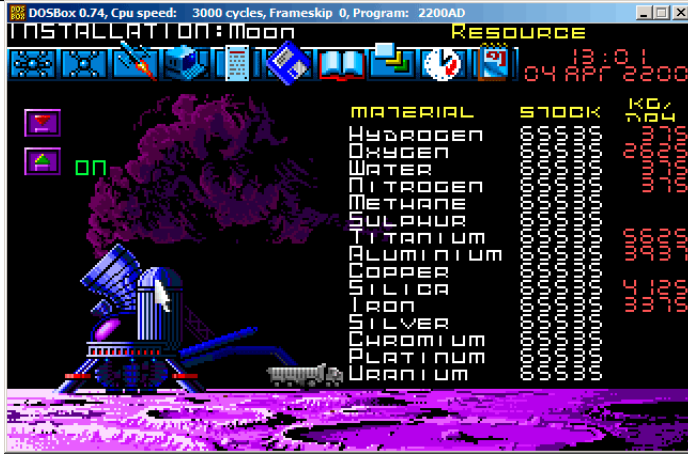


Figure 69.7: All resources are 65535 (0xFFFF) indeed



Figure 69.8: Resource variables overflow



## Chapter 70

# Oracle RDBMS: .SYM-files

When Oracle RDBMS process experiencing some kind of crash, it writes a lot of information into log files, including stack trace, like:

```

----- 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
                                EB1C8A8 1
_kpoal8()+2832      CALLrel  _opifch2()    89 5 EB1CC74
_opiodr()+1248      CALLreg  00000000      5E 1C ↵
  ↵ EB1F0A0
_ttcpip()+1051      CALLreg  00000000      5E 1C ↵
  ↵ EB1F0A0 0
_opitsk()+1404      CALL???  00000000      C96C040 5E ↵
  ↵ EB1F0A0 0 EB1ED30
                                EB1F1CC 53↵
  ↵ E52E 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 7↵

```

```

↪ C88A7F4 EB1FC34 0
4()+830 EB1FD04
77E6481C CALLreg 00000000 E41FF9C 0 0 ↪
↪ E41FF9C 0 EB1FFC4
00000000 CALL??? 00000000

```

But of course, Oracle RDBMS executables must have some kind of debug information or map files with symbol information included or something like that.

Windows NT Oracle RDBMS have 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. I chose shortest `orawtc8.sym` file, coming with `orawtc8.dll` file in Oracle 8.1.7<sup>1</sup>.

Here is the file opened in Hiew: [fig.70.1](#).

By comparing the file with other .SYM files, we can quickly see that OSYM is always header (and footer), so this is maybe file signature.

We also see that basically, file format is: OSYM + some binary data + zero delimited text strings + OSYM. Strings are, obviously, function and global variable names. I marked OSYM signatures and strings here: [fig.70.2](#).

Well, let's see. In Hiew, I marked the whole strings block (except trailing OSYM signatures) and put it into separate file. Then I run UNIX `strings` and `wc` utilities to count strings in there:

```

strings strings_block | wc -l
66

```

So there are 66 text strings. Please note that number.

We can say, in general, as a rule, number of *anything* is often stored separately in binary files. It's indeed so, we can find 66 value (0x42) at the file begin, right after 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.

<sup>1</sup>I chose ancient Oracle RDBMS version intentionally due to smaller size of its modules

Why I think it's 32-bit? Because, Oracle RDBMS symbol files may be pretty big. The oracle.sym for the main oracle.exe (version 10.2.0.4) executable contain 0x3A38E (238478) symbols. 16-bit value isn't enough here.

I checked other .SYM files like this and it proves my guess: the value after 32-bit OSYM signature is always reflects number of text strings in the file.

It's a general feature of almost all binary files: header with signature plus some other information about file.

Now let's investigate closer what this binary block is. Using Hiew again, I put the block starting at address 8 (i.e., after 32-bit *count* value) till strings block into separate binary file.

Let's see the binary block in Hiew: fig.70.3.

There is some clear pattern in it. I added red lines to divide the block: fig.70.4.

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, so started by 0x10 and 0 bytes. Other values (starting at 0x108) are in 0x0000xxxx form, so always started by two zero bytes.

Let's dump the block as 32-bit values array:

Listing 70.1: first column is address

```
$ od -v -t x4 binary_block
0000000 10001000 10001080 100010f0 10001150
0000020 10001160 100011c0 100011d0 10001370
0000040 10001540 10001550 10001560 10001580
0000060 100015a0 100015a6 100015ac 100015b2
0000100 100015b8 100015be 100015c4 100015ca
0000120 100015d0 100015e0 100016b0 10001760
0000140 10001766 1000176c 10001780 100017b0
0000160 100017d0 100017e0 10001810 10001816
0000200 10002000 10002004 10002008 1000200c
0000220 10002010 10002014 10002018 1000201c
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 started with  $0x1000$  may be addresses. This is .SYM file for DLL after all, and, default base address of win32 DLLs is  $0x10000000$ , and the code is usually started at  $0x10001000$ .

When I open `orawtc8.dll` file in IDA, 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, 0FFFFFFFFh
.text:6035100B          jnz    short loc_6035104F
.text:6035100D          mov     ecx, hModule
.text:60351013          xor     eax, eax
.text:60351015          cmp     ecx, 0FFFFFFFFh
.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          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 it seems "ax\_reg".

The second function is:

```
.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
...
```

"ax\_unreg" string is also the second string in strings block! Starting address of the second function is 0x60351080, and the second value in the binary block is 10001080. So this is address, but for the DLL with default base address.

I can quickly check and be sure that first 66 values in array (i.e., first half of array) are just function addresses in DLL, including some labels, etc. Well, what's then other part of array? Other 66 values starting at 0x0000? These are seems to be in range [0...0x3F8]. And they are not looks like bitfields: sequence of numbers is growing. The last hexadecimal digit is seems to be random, so, it's unlikely an address of something (it would be divisible by 4 or maybe 8 or 0x10 otherwise).

Let's ask ourselves: what else Oracle RDBMS developers would save here, in this file? Quick wild guess: it could be an address of the text string (function name). It can be quickly checked, and yes, each number is just position of the first character in the strings block.

This is it! All done.

I wrote an utility to convert these .SYM files into [IDA](#) script, so I can load .idc script and it will set 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 (argv[1]); // file name
    assert (argv[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 timedatestamp (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, "_wtcsrln0");
    MakeName(0x60351160, "_wtcsrln");
    MakeName(0x603511C0, "_wtcsrfre");
    MakeName(0x603511D0, "_wtclkm");
    MakeName(0x60351370, "_wtcstu");
    ...
}

```

The files I used for example are here: <http://beginners.re/examples/oracle/SYM/>.

Oh, let's also try Oracle RDBMS for win64. There should be 64-bit addresses instead, right?

The 8-byte pattern is visible even easier here: fig.70.5.

So yes, all tables now has 64-bit elements, even string offsets! The signature is now OSYMMAM64, to distinguish target platform, I suppose.

This is it! Here is also my library in which I have function to access Oracle RDBMS.SYM-files: [https://github.com/dennis714/porg/blob/master/oracle\\_sym.c](https://github.com/dennis714/porg/blob/master/oracle_sym.c).

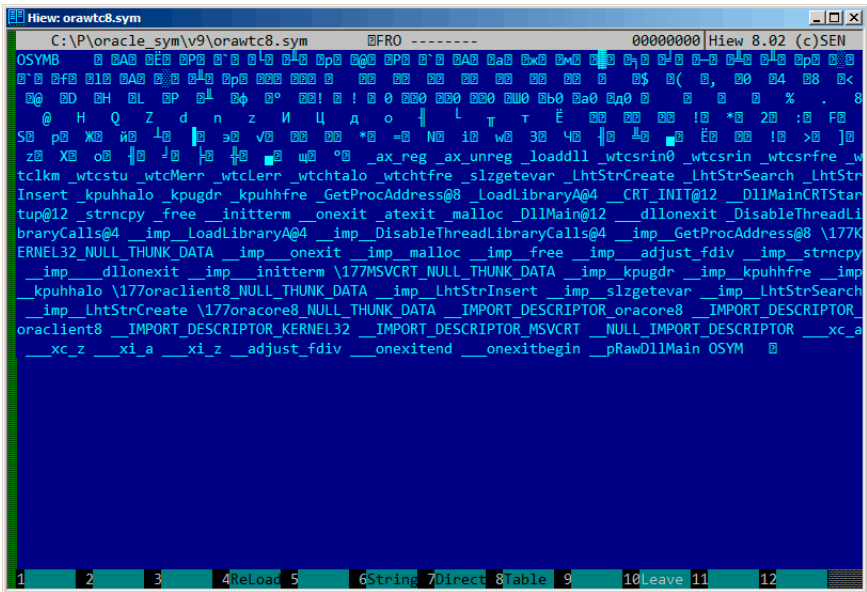


Figure 70.1: The whole file in Hiew



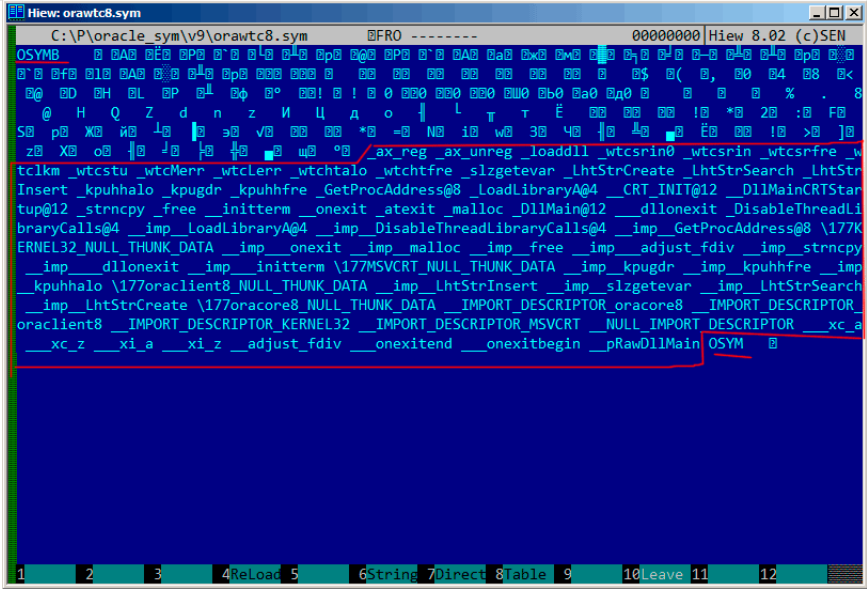


Figure 70.2: OSYM signature and text strings

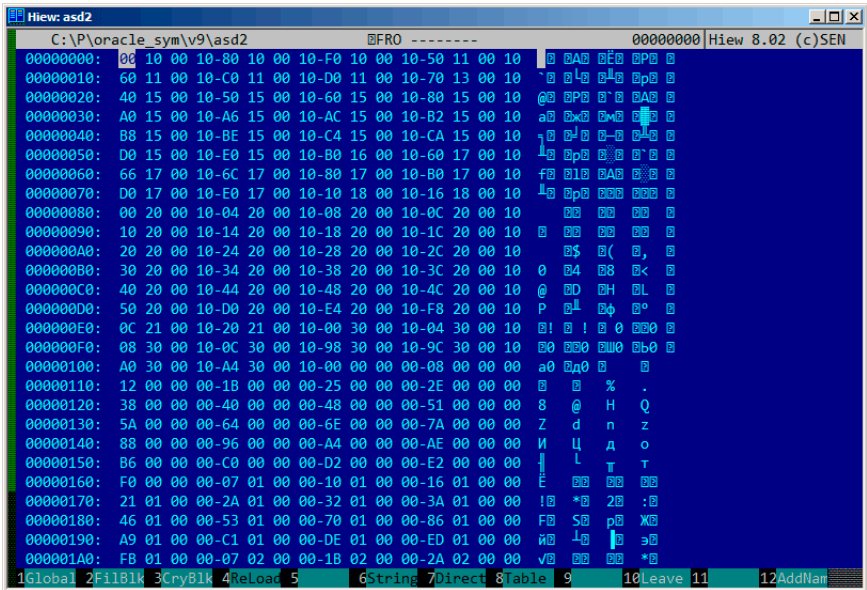


Figure 70.3: Binary block

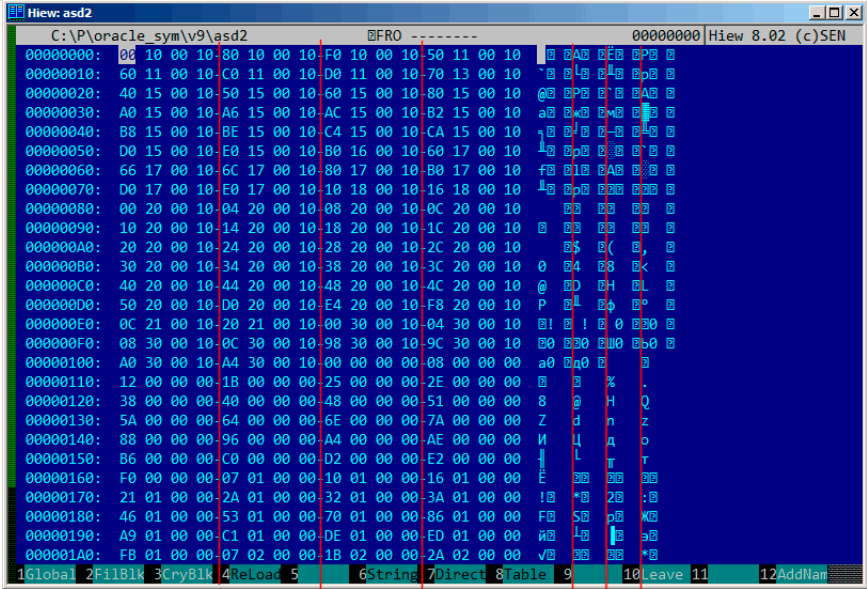


Figure 70.4: Binary block patterns

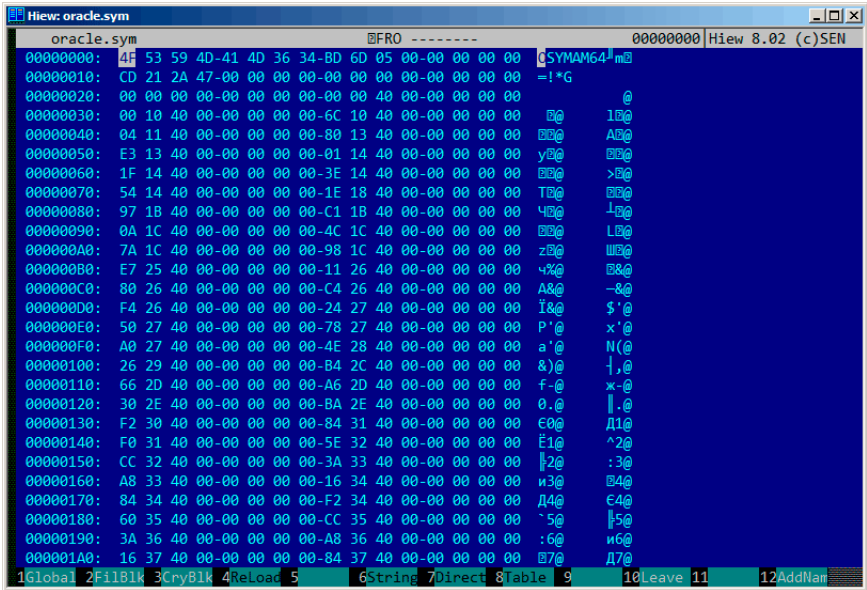


Figure 70.5: .SYM-file example from Oracle RDBMS for win64

# Chapter 71

## 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 containing error messages with corresponding numbers. Let's try to understand its format and find a way to unpack it.

There are Oracle RDBMS error message files in text form, so we can compare text and packed binary files <sup>1</sup>.

This is the beginning of ORAUS.MSG text files with irrelevant comments stripped:

Listing 71.1: 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"
```

---

<sup>1</sup>Text files are exist in Oracle RDBMS not for every .MSB file, so that's why I worked on its file format

```

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 error code. The second is maybe some additional flags, but I'm not sure. Now let's open ORAUS.MSB binary file and find these text strings. And there are: [fig.71.1](#).

We see the text strings (including those with which ORAUS.MSG file started) interleaved with some binary values. By quick investigation, we can see that main part of binary file is divided by blocks of size 0x200 (512) bytes.

Let's see contents of the first block: [fig.71.2](#).

Here we see texts of first messages errors. What we also see is that there are no zero bytes between error messages. This mean, these are not zero-terminated C-strings. As a consequence, a length of each error message must be coded somehow. Let's also try to find error numbers. The ORAUS.MSG files started with these: 0, 1, 17 (0x11), 18 (0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)... I found these numbers in the beginning of block and marked them with red lines. The period between error codes is 6 bytes. This mean, there are probably 6 bytes of information allocated for each error message.

The first 16-bit value (0xA here or 10), mean number of messages in each block: I checked this by investigating other blocks. Indeed: error messages has 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 I already noted, since this is not zero-terminating C-strings, a string size should be encoded somewhere. The size of the first string "normal, successful completion" is 29 (0x1D) bytes. The size of the second string "%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 should somehow determine position in block of the string it needs to load, right? The first string "normal, successful completion" is started at the position of 0x1444 (if to count starting at the binary file) or at 0x44 (starting at the block begin). The second string "unique constraint (%s.%s) violated" is started at the position of 0x1461 (from the file start) or at 0x61 (starting at the block begin). These (0x44 and 0x61) are familiar numbers! We can clearly see them at the block start.

So, each 6-byte block is:

- 16-bit error number;
- 16-bit zero (may be additional flags);
- 16-bit starting position of the text string within the current block.

I can quickly check other values and be sure I'm right. And there are also the last "dummy" 6-byte block with zero error number and starting position beyond the last error message last character. Probably that's how text message length is determined? We just enumerate 6-byte blocks to find error number we need, then we get text string position, then we get position of the text string by looking onto next 6-byte block! Thus we determine string boundaries! This method allows to save a space by not saving text string size in the file! I cannot say it saves a lot, but it's a clever trick.

Let's back to the header of .MSB-file: fig.71.3. I quickly found number of blocks in file (marked by red). I checked other .MSB-files and that's true for any. There are a lot of other values, but I didn't investigate them, since by job (unpacking utility) was done. If I would write .MSB-file packer, I would probably need to understand other value meanings.

There is also a table came after header which probably contain 16-bit values: fig.71.4. Their size can be determined visually (I draw red lines). When I'm dumping these values, I found that each 16-bit number is a last error code for each block.

So that's how Oracle RDBMS quickly finds error message:

- load a table I called last\_errnos (containing last error number for each block);
- find a block containing error code we need, assuming all error codes increasing across each block and the file as well;
- load specific block;
- enumerate 6-byte structures until specific error number is found;
- get a position of the first character from current 6-byte block;
- get a position of the last character from the next 6-byte block;
- load all characters from message in this range.

This is C-program I wrote which unpacks .MSB-files: [http://beginners.re/examples/oracle/MSB/oracle\\_msb.c](http://beginners.re/examples/oracle/MSB/oracle_msb.c).

There are also two files I used in the example (Oracle RDBMS 11.1.0.6): <http://beginners.re/examples/oracle/MSB/oraus.msb>, <http://beginners.re/examples/oracle/MSB/oraus.msg>.

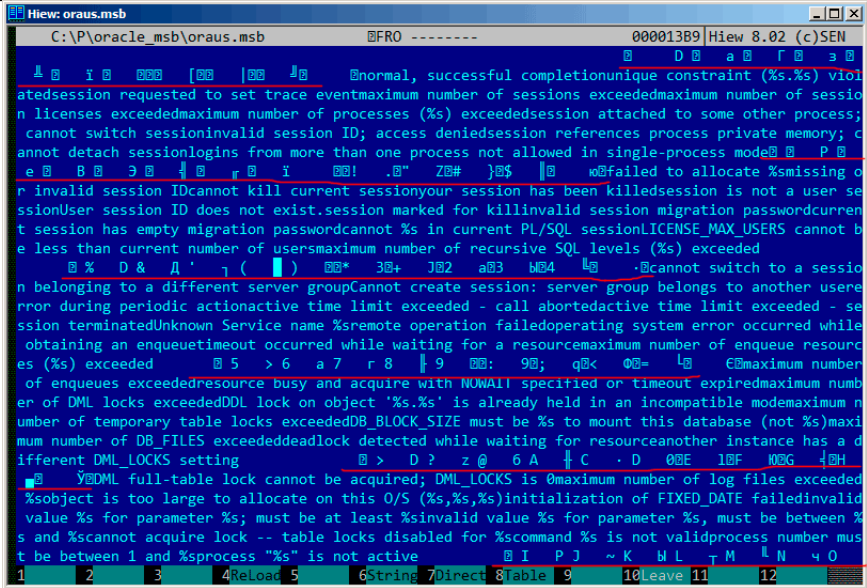


Figure 71.1: Hiew: first block

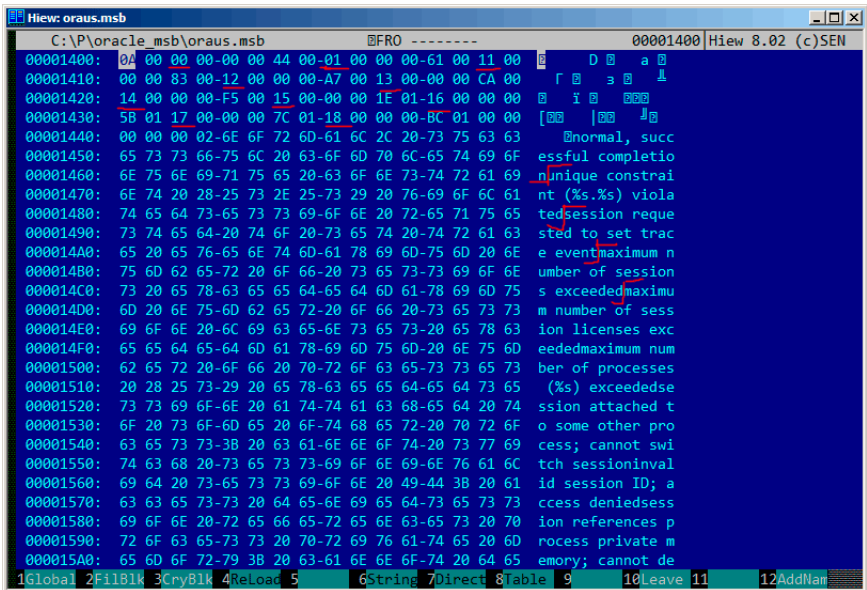


Figure 71.2: Hiew: first block

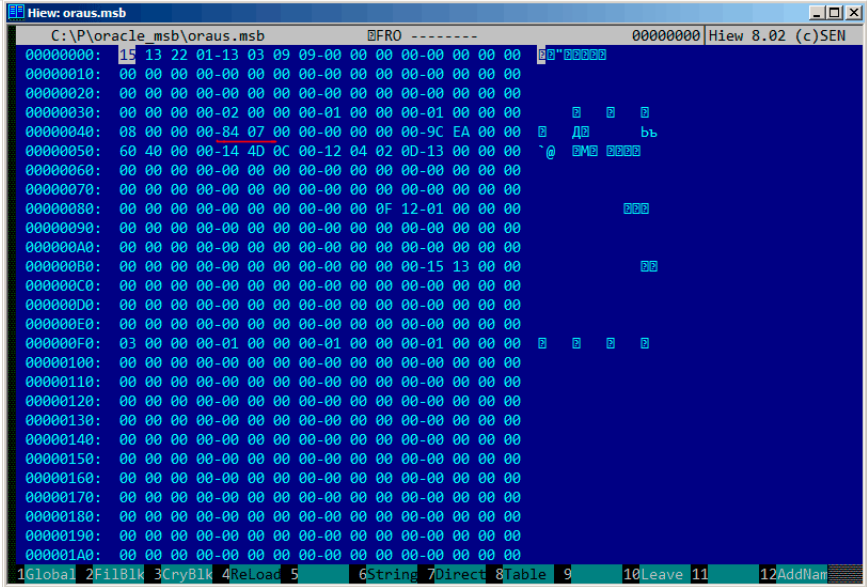


Figure 71.3: Hiew: file header

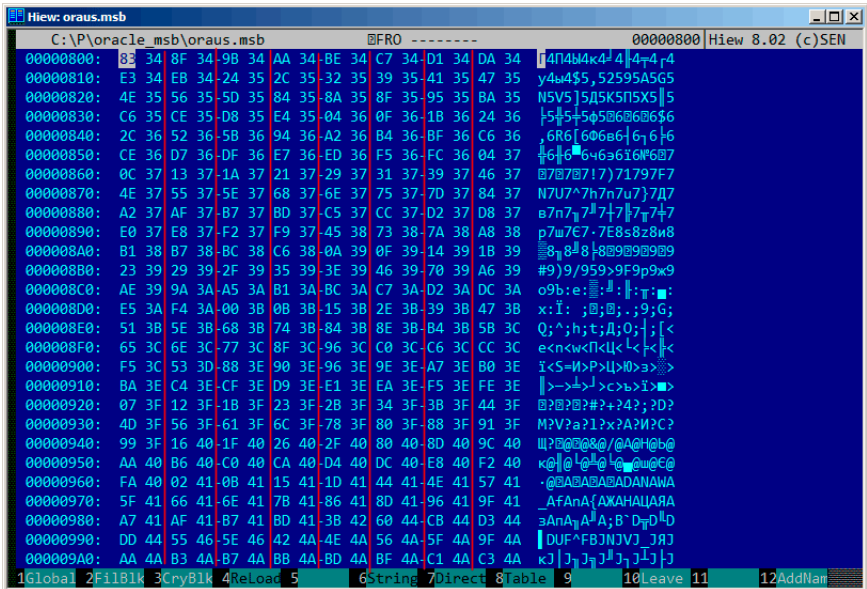


Figure 71.4: Hiew: last\_errms table

## 71.1 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 was an interesting and yet easy task to understand proprietary file format without looking into Oracle RDBMS code.



## **Part VIII**

# **Other things**

# Chapter 72

## npad

It is an assembly language macro for label aligning by a specific border.

That's often need for the busy labels to where control flow is often passed, e.g., loop body begin. So the CPU will effectively load data or code from the memory, through memory bus, cache lines, etc.

Taken from `listing.inc` (MSVC):

By the way, it is curious example of different NOP variations. All these instructions has no effects whatsoever, but has different size.

The goal to have single idle instruction instead of couple of NOP-s, it is accepted that is 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
                ; lea ebx, [ebx+00000000]
                DB 8DH, 9BH, 00H, 00H, 00H, 00H
            else
                if size eq 7
                    ; lea esp, [esp+00000000]
                    DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
                else
                    if size eq 8
                        ; jmp .+8; .npad 6
                        DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
                    else
                        if size eq 9
                            ; jmp .+9; .npad 7
                            DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
                        else
                            if size eq 10
                                ; jmp .+A; .npad 7; .npad 1
                                DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
                            else
                                if size eq 11
                                    ; jmp .+B; .npad 7; .npad 2
                                    DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8↵
                                ↵ BH, 0FFH
                            else
                                if size eq 12
                                    ; jmp .+C; .npad 7; .npad 3
                                    DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8↵
                                ↵ DH, 49H, 00H
                            else
                                if size eq 13
                                    ; jmp .+D; .npad 7; .npad 4
                                    DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, ↵
                                ↵ 8DH, 64H, 24H, 00H
                            else
                                if size eq 14
                                    ; jmp .+E; .npad 7; .npad 5
                                    DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, ↵

```



# Chapter 73

## Executable files patching

### 73.1 Text strings

C strings are most easily patched (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. New string should not be bigger than old, because it's a risk to overwrite some other value or code there. Using this method, a lot of software was *localized* in MS-DOS era, at least in ex-USSR countries in 80's and 90's. It was a reason why so weird abbreviations were present in *localized* software: it was no room for longer strings.

As of Delphi strings, a string size should also be corrected, if needed.

### 73.2 x86 code

Frequent patching tasks are:

- One of the most frequently jobs is to disable some instruction. It is often done by filling it by byte 0x90 (**NOP**).
- Conditional jumps, which have opcode like `74 xx (JZ)`, may also be filled by two **NOPs**. It is also possible to disable conditional jump by writing 0 at the second byte (*jump offset*).
- Another frequent job is to make conditional jump to trigger always: this can be done by writing 0xEB instead of opcode, it means **JMP**.
- A function execution can be disabled by writing `RETN (0xC3)` at its beginning. This is true for all functions excluding `stdcall` (49.2). While patching

`stdcall` functions, one should determine number of arguments (for example, by finding `RETN` in this function), and use `RETN` with 16-bit argument (`0xC2`).

- Sometimes, a disabled functions should return 0 or 1. This can be done by `MOV EAX, 0` or `MOV EAX, 1`, but it's slightly verbose. Better way is `XOR EAX, EAX` (2 bytes `0x31 0xC0`) or `XOR EAX, EAX / INC EAX` (3 bytes `0x31 0xC0 0x40`).

A software may be protected against modifications. This protection is often done by reading executable code and doing some checksumming. Therefore, the code should be read before protection will be triggered. This can be determined by setting breakpoint on reading memory.

[tracer](#) has `BPM` option for this.

PE executable file relocs ([53.2.6](#)) should not be touched while patching, because Windows loader will overwrite a new code. (They are grayed in Hiew, for example: [fig.6.12](#)). As a last resort, it is possible to write jumps circumventing relocs, or one will need to edit relocs table.

# Chapter 74

## Compiler intrinsic

A function specific to a compiler which is not usual library function. Compiler generate a specific machine code instead of 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 present in most CPUs. For programmer's convenience, at least MSVC has pseudofunctions `_rotl()` and `_rotr()`<sup>1</sup> which are translated by compiler directly to the ROL/ROR x86 instructions.

Another example are functions enabling to generate SSE-instructions right in the code.

Full list of MSVC intrinsics: <http://msdn.microsoft.com/en-us/library/26td21ds.aspx>.

---

<sup>1</sup><http://msdn.microsoft.com/en-us/library/5cc576c4.aspx>

## Chapter 75

# Compiler's anomalies

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. Second JZ is thus senseless.

Listing 75.1: kdli.o from libserver11.a

```
.text:08114CF1                                loc_8114CF1: ; CODE XREF: ↗
    ↘ __PGOSF539_kdlimemSer+89A
.text:08114CF1                                ; ↗
    ↘ __PGOSF539_kdlimemSer+3994
.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 75.2: from the same code

```
.text:0811A2A5                                loc_811A2A5: ; CODE XREF: ↗
    ↘ kdliSerLengths+11C
```



```

.text:0811A2A5                ; kdliSerLengths↵
    ↵ +1C1
.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 probably code generator bug was not found by tests, because, resulting code is working correctly anyway.

Another compiler anomalies here in this book: [18.2.4](#), [13.0.3](#), [14.2.7](#), [17.7](#), [18.4.2](#).

I demonstrate such cases here, so to understand that such compilers errors are possible and sometimes one should not to rack one's brain and think why compiler generated such strange code.

# Chapter 76

## OpenMP

OpenMP is one of the simplest ways to parallelize simple algorithm.

As an example, let's try to build a program to compute cryptographic *nonce*. In my simplistic example, *nonce* is a number added to the plain unencrypted text in order to produce hash with some specific feature. For example, at some step, Bitcoin protocol require to find a such *nonce* so resulting hash will contain specific number of running zeroes. This is also called “proof of work”<sup>1</sup> ( i.e., system prove 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 a numbers to the “hello, world!” string in order to find such number when “hello, world!\_<number>” will contain at least 3 zero bytes after hashing this string by SHA512 algorithm.

Let's limit our brute-force to the interval in 0..INT32\_MAX-1 (i.e., 0x7FFFFFFE 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__
```

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Proof-of-work\\_system](https://en.wikipedia.org/wiki/Proof-of-work_system)

```

#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
    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);
};

```

check\_nonce() function is just add a number to the string, hashes it by SHA512 algorithm and checks for 3 zero bytes in the result.

Very important part of the code is:

```

#pragma omp parallel for
for (i=0; i<INT32_MAX; i++)
    check_nonce (i);

```

Yes, that simple, without #pragma we just call check\_nonce() for each number from 0 to INT32\_MAX (0x7fffffff or 2147483647). With #pragma, a compiler adds a special code which will slice the loop interval to smaller intervals, to run them by all CPU cores available<sup>2</sup>.

The example may be compiled<sup>3</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

```

<sup>2</sup>N.B.: I intentionally demonstrate here simplest possible example, but in practice, usage of OpenMP may be harder and more complex

<sup>3</sup>sha512.clh and u64.h files can be taken from the OpenSSL library: <http://www.openssl.org/source/>

## 76.1 MSVC

Now that's how MSVC 2012 generates main loop:

Listing 76.1: MSVC 2012

```

push    OFFSET _main$omp$1
push    0
push    1
call    __vcomp_fork
add     esp, 16                ; ↗
↳ 00000010H

```

All functions prefixed by `vcomp` are OpenMP-related and stored in the `vcomp*.dll` file. So here is a group of threads are started.

Let's take a look on `_main$omp$1`:

Listing 76.2: MSVC 2012

```

$T1 = -8                ; size ↗
↳ = 4
$T2 = -4                ; size ↗
↳ = 4
_main$omp$1 PROC        ; ↗
↳ COMDAT
    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    ; 7↗
↳ ffffffffH
    push    0
    call    __vcomp_for_static_simple_init
    mov     esi, DWORD PTR $T1[ebp]
    add     esp, 24        ; ↗
↳ 00000018H
    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

```

This function will be started  $n$  times in parallel, where  $n$  is number of CPU cores. `vcomp_for_static_simple_init()` is calculating interval for the `for()` construct for the current thread, depending on the current thread number. Loop begin and end values are stored in `$T1` and `$T2` local variables. You may also notice `7fffffffh` (or `2147483646`) as an argument to the `vcomp_for_static_simple_init` function—this is a number of iterations of the whole loop to be divided evenly.

Then we see a new loop with a call to `check_nonce()` function which do all work.

I also added some code in the beginning of `check_nonce()` function to gather statistics, with which arguments the function was called.

This is what we see while 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]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7fffffff

```

Yes, result is correct, first 3 bytes are zeroes:

```

C:\...\sha512sum test
000000 ↵
  ↵ f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403 ↵
  ↵
df6e3fe6019f5764fc9975e505a7395fed780fee50eb38dd4c0279cb114672e2 ↵
  ↵ *test

```

Running time is  $\approx 2.3$  seconds on my 4-core Intel Xeon E3-1220 3.10 GHz. In the task manager I see 5 threads: 1 main thread + 4 more started. I did not any further optimizations to keep my 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 loop was finely sliced by 4 even parts. Oh well, almost even, if not to 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 76.3: 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                ; ↵
↳ 0000000cH
; Line 56
    mov     ecx, DWORD PTR _checked
    mov     eax, ecx
    cdq
    mov     esi, 100000           ; ↵
↳ 000186a0H
    idiv    esi
    test    edx, edx
    jne     SHORT $LN1@check_nonc
; Line 57
    push    ecx
    push    OFFSET ??_C@_0M@NPNHLI00@checked?$DN?$CFd?6? ↵
↳ $AA@
    call    _printf
    pop     ecx
    pop     ecx
$LN1@check_nonc:
    push    DWORD PTR _$vcomp$critsect$
    call    __vcomp_leave_critsect
    pop     ecx
```

As it turns out, `vcomp_atomic_add_i4()` function in the `vcomp*.dll` is just a tiny function having `LOCK XADD` instruction<sup>4</sup>.

<sup>4</sup>Read more about `LOCK` prefix: [A.6.1](#)

5. `vcomp_enter_critsect()` eventually calling win32 API function `EnterCriticalSection`.

## 76.2 GCC

GCC 4.8.1 produces the program which shows exactly the same statistics table, so, GCC implementation divides the loop by parts in the same fashion.

Listing 76.4: 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 implementation, what GCC code is doing is starting 3 threads, but also runs fourth in the current thread. So there will be 4 threads instead of 5 as in MSVC.

Here is a `main._omp_fn.0` function:

Listing 76.5: 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

```

<sup>5</sup>Read more about critical sections here: [53.4](#)



```

    add    eax, edx
    lea   ebx, [rax+rcx]
    cmp   eax, ebx
    jge   .L14
    mov   DWORD PTR [rbp-20], eax
.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 that division clearly: by calling to `omp_get_num_threads()` and `omp_get_thread_num()` we got number of threads running, and also current thread number, and then determine loop interval. Then run `check_nonce()`.

GCC also inserted `LOCK ADD` instruction right in the code, where MSVC generated call to separate DLL function:

Listing 76.6: 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
```

Functions prefixed with GOMP are from GNU OpenMP library. Unlike `vcomp*.dll`, its sources are freely available: <https://github.com/mirrors/gcc/tree/master/libgomp>.

## Chapter 77

# Itanium

Although almost failed, another very interesting architecture is Intel Itanium (IA64). While OOE CPUs decides how to rearrange instructions and execute them in parallel, EPIC<sup>1</sup> was an attempt to shift these decisions to the compiler: to let it group instructions at the compile stage.

This result in notoriously complex compilers.

Here is one sample of IA64 code: simple cryptoalgorithm from Linux kernel:

Listing 77.1: 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;
```

<sup>1</sup>Explicitly parallel instruction computing

```

    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:

Listing 77.2: 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 7F 20 ld4 r14 = [r34] ↵
↵                                     // load y
00CC|92 F3 CE 6B                       movl r17 = 0↵
↵                                     xFFFFFFFF9E3779B9;; // 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 size of 16 bytes and consists of template code + 3 instructions. IDA shows bundles into 6+6+4 bytes – you may easily spot the pattern.

All 3 instructions from each bundle usually executes simultaneously, unless one

of instructions have “stop bit”.

Supposedly, Intel and HP engineers gathered statistics of most occurred instruction patterns and decided to bring bundle types (AKA “templates”): a bundle code defines instruction types in the bundle. There are 12 of them. For example, zeroth bundle type is MII, meaning: first instruction is Memory (load or store), second and third are I (integer instructions). Another example is bundle type 0x1d: MFB: first instruction is Memory (load or store), second is Float (FPU instruction), third is Branch (branch instruction).

If compiler cannot pick suitable instruction to relevant bundle slot, it may insert **NOP**: you may see here `nop . i` instructions (**NOP** at the place where integer instruction might be) or `nop . m` (a memory instruction might be at this slot). **NOP**s are inserted automatically when one use assembly language manually.

And that is not all. Bundles are also grouped. Each bundle may have “stop bit”, so all the consecutive bundles with terminating bundle which have “stop bit” may be executed simultaneously. In practice, Itanium 2 may execute 2 bundles at once, resulting execution of 6 instructions at once.

So all instructions inside bundle and bundle group cannot interfere with each other (i.e., should not have data hazards). If they do, results will be undefined.

Each stop bit is marked in assembly language as two semicolons (; ;) after instruction. So, instructions at [180-19c] may be executed simultaneously: they do not interfere. Next group is [1a0-1bc].

We also see a stop bit at 22c. The next instruction at 230 have stop bit too. This mean, this instruction is to be executed as isolated from all others (as in **CISC**). Indeed: the next instruction at 236 use result from it (value in register r10), so they cannot be executed at the same time. Apparently, compiler was not able to find a better way to parallelize instructions, which is, in other words, to load **CPU** as much as possible, hence too much stop bits and **NOP**s. Manual assembly programming is tedious job as well: programmer should group instructions manually.

Programmer is still able to add stop-bits to each instructions, but this will degrade all the performance Itanium was made for.

Interesting examples of manual **IA64** assembly code can be found in Linux kernel sources:

<http://lxr.free-electrons.com/source/arch/ia64/lib/>.

Another introductory Itanium assembly paper: [Bur].

Another very interesting Itanium feature is *speculative execution* and NaT (“not a thing”) bit, somewhat resembling **NaN** numbers:

<http://blogs.msdn.com/b/oldnewthing/archive/2004/01/19/60162.aspx>.

# Chapter 78

## 8086 memory model

Dealing with 16-bit programs for MS-DOS or Win16 (63.3 or 34.5), we can see that pointer consisting of two 16-bit values. What it means? Oh yes, that is another MS-DOS and 8086 weird artefact.

8086/8088 was a 16-bit CPU, but was able to address 20-bit address RAM (thus resulting 1MB external memory). 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 inheritor of 8-bit 8080 CPU. The 8080 has 16-bit memory spaces, i.e., it was able to address only 64KB. And probably of old software porting reason<sup>1</sup>, 8086 can support 64KB windows, many of them placed simultaneously within 1MB address space. This is some kind of toy-level virtualization. All 8086 registers are 16-bit, so to address more, a special segment registers (CS, DS, ES, SS) were introduced. Each 20-bit pointer is calculated using values from a segment register and an address register pair (e.g. DS:BX) as follows:

$$real\_address = (segment\_register \ll 4) + address\_register$$

For example, graphics (EGA<sup>2</sup>, VGA<sup>3</sup>) video RAM window on old IBM PC-compatibles has size of 64KB. For accessing it, a 0xA000 value should be stored in one of segment registers, e.g. into DS. Then DS:0 will address the very first byte of video RAM and DS:0xFFFF is the very last byte of RAM. The real address on 20-bit address bus, however, will range from 0xA0000 to 0xAFFFF.

The program may contain hardcoded addresses like 0x1234, but OS may need to load program on arbitrary addresses, so it recalculates segment register values in such a way, so the program will not care about where in the RAM it is placed.

---

<sup>1</sup>I'm not 100% sure here

<sup>2</sup>Enhanced Graphics Adapter

<sup>3</sup>Video Graphics Array

So, any pointer in the old MS-DOS environment was in fact consisted of segment address and the address inside segment, i.e., two 16-bit values. 20-bit was enough for that, though, but one will need to recalculate the addresses very often: passing more information on stack seems better space/convenience balance.

By the way, because of all this, it was not possible to allocate the memory block larger than 64KB.

Segment registers were reused at 80286 as selectors, serving different function.

When 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 “serious” OS, switching CPU into protected mode and providing much better memory APIs for the programs which still needs to be runned from MS-DOS. Widely popular examples include DOS/4GW (DOOM video game was compiled for it), Phar Lap, PMODE.

By the way, the same way of addressing memory was in 16-bit line of Windows 3.x, before Win32.



# Chapter 79

## Basic blocks reordering

### 79.1 Profile-guided optimization

This optimization method may move some [basic blocks](#) to another section of the executable binary file.

Obviously, there are parts in function which are executed most often (e.g., loop bodies) and less often (e.g., error reporting code, exception handlers).

The compiler adding instrumentation code into the executable, then developer run it with a lot of tests for statistics collecting. Then the compiler, with the help of statistics gathered, prepares final 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 memory.

Example from Oracle RDBMS code, which was compiled by Intel C++:

Listing 79.1: orageneric11.dll (win32)

```

public _skgfsync
_skgfsync    proc near
; address 0x6030D86A

        db        66h
        nop
        push     ebp
        mov     ebp, esp
        mov     edx, [ebp+0Ch]
        test    edx, edx
        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, 0FFFFFFCh
    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   ecx
    push   edx
    push   ecx
    push   offset ... ; "skgfsync(se=0x%x, ctx=0x%x)
    ↪ x, iov=0x%x)\n"
    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 of these two code fragments is almost 9 MB.

All infrequently executed code was placed at the end of the code section of DLL file, among all function parts. This part of function was marked by Intel C++ compiler with `VInfr eq` prefix. Here we see that a part of function which writes to log-file (presumably in case of error or warning or something like that) which was probably not executed very often when Oracle developers gathered statistics (if was executed at all). The writing to log basic block is eventually return control flow into the “hot” part of the function.

Another “infrequent” part is a [basic block](#) returning error code 27050.

In Linux ELF files, all infrequently executed code is moved by Intel C++ into 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 to its core and error handling parts.

## **Part IX**

# **Books/blogs worth reading**

# Chapter 80

## Books

### 80.1 Windows

[RA09].

### 80.2 C/C++

[ISO13].

### 80.3 x86 / x86-64

[Int13], [AMD13a]

### 80.4 ARM

ARM manuals: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html>

# Chapter 81

## Blogs

### 81.1 Windows

- [Microsoft: Raymond Chen](#)
- <http://www.nynaeve.net/>

# Chapter 82

## Other

There are two excellent RE<sup>1</sup>-related subreddits on reddit.com: [ReverseEngineering](#) and [REMath](#) ( for the topics on the intersection of [RE](#) and mathematics).

There are also [RE](#) part of Stack Exchange website:

<http://reverseengineering.stackexchange.com/>.

On IRC there are [##re](#) channel on FreeNode<sup>2</sup>.

---

<sup>1</sup>Reverse Engineering

<sup>2</sup><https://freenode.net/>

**Part X**

**Exercises**



---

There are two questions almost for every exercise, if otherwise is not specified:

1) What this function does? Answer in one-sentence form.

2) Rewrite this function into C/C++.

It is allowed to use Google to search for any leads. However, if you like to make your task harder, you may try to solve it without Google.

Hints and solutions are in the appendix of this book.

# Chapter 83

## Level 1

Level 1 exercises are ones you may try to solve in mind.

### 83.1 Exercise 1.4

This program requires password. Find it.

As an additional exercise, try to change the password by patching executable file. It may also has a different length. What shortest password is possible here?

Try also to crash the program using only string input.

- win32: <http://beginners.re/exercises/1/4/password1.exe>
- Linux x86: [http://beginners.re/exercises/1/4/password1\\_Linux\\_x86.tar](http://beginners.re/exercises/1/4/password1_Linux_x86.tar)
- Mac OS X: [http://beginners.re/exercises/1/4/password1\\_MacOSX64.tar](http://beginners.re/exercises/1/4/password1_MacOSX64.tar)

# Chapter 84

## Level 2

For solving exercises of level 2, you probably will need text editor or paper and pencil.

### 84.1 Exercise 2.4

This is standard C library function. Source code taken from MSVC 2010.

#### 84.1.1 Optimizing MSVC 2010

```
PUBLIC    _f
_TEXT   SEGMENT
_arg1$ = 8           ; size = 4
_arg2$ = 12         ; size = 4
_f      PROC
    push    esi
    mov     esi, DWORD PTR _arg1$[esp]
    push    edi
    mov     edi, DWORD PTR _arg2$[esp+4]
    cmp     BYTE PTR [edi], 0
    mov     eax, esi
    je      SHORT $LN7@f
    mov     dl, BYTE PTR [esi]
    push    ebx
    test    dl, dl
    je      SHORT $LN4@f
    sub     esi, edi
    npad    6 ; align next label
$LL5@f:
    mov     ecx, edi
```

```

    test    dl, dl
    je     SHORT $LN2@f
$LL3@f:
    mov     dl, BYTE PTR [ecx]
    test   dl, dl
    je     SHORT $LN14@f
    movsx  ebx, BYTE PTR [esi+ecx]
    movsx  edx, dl
    sub    ebx, edx
    jne    SHORT $LN2@f
    inc    ecx
    cmp    BYTE PTR [esi+ecx], bl
    jne    SHORT $LL3@f
$LN2@f:
    cmp    BYTE PTR [ecx], 0
    je     SHORT $LN14@f
    mov    dl, BYTE PTR [eax+1]
    inc    eax
    inc    esi
    test   dl, dl
    jne    SHORT $LL5@f
    xor    eax, eax
    pop    ebx
    pop    edi
    pop    esi
    ret    0
_f      ENDP
_TEXT   ENDS
END

```

## 84.1.2 GCC 4.4.1

```

f      public f
      proc near

var_C  = dword ptr -0Ch
var_8  = dword ptr -8
var_4  = dword ptr -4
arg_0  = dword ptr 8
arg_4  = dword ptr 0Ch

      push    ebp
      mov     ebp, esp
      sub    esp, 10h
      mov     eax, [ebp+arg_0]
      mov     [ebp+var_4], eax

```

```
    mov     eax, [ebp+arg_4]
    movzx  eax, byte ptr [eax]
    test   al, al
    jnz    short loc_8048443
    mov     eax, [ebp+arg_0]
    jmp    short locret_8048453

loc_80483F4:
    mov     eax, [ebp+var_4]
    mov     [ebp+var_8], eax
    mov     eax, [ebp+arg_4]
    mov     [ebp+var_C], eax
    jmp    short loc_804840A

loc_8048402:
    add     [ebp+var_8], 1
    add     [ebp+var_C], 1

loc_804840A:
    mov     eax, [ebp+var_8]
    movzx  eax, byte ptr [eax]
    test   al, al
    jz     short loc_804842E
    mov     eax, [ebp+var_C]
    movzx  eax, byte ptr [eax]
    test   al, al
    jz     short loc_804842E
    mov     eax, [ebp+var_8]
    movzx  edx, byte ptr [eax]
    mov     eax, [ebp+var_C]
    movzx  eax, byte ptr [eax]
    cmp    dl, al
    jz     short loc_8048402

loc_804842E:
    mov     eax, [ebp+var_C]
    movzx  eax, byte ptr [eax]
    test   al, al
    jnz    short loc_804843D
    mov     eax, [ebp+var_4]
    jmp    short locret_8048453

loc_804843D:
    add     [ebp+var_4], 1
    jmp    short loc_8048444
```

```

loc_8048443:
                nop

loc_8048444:
                mov     eax, [ebp+var_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_80483F4
                mov     eax, 0

locret_8048453:
                leave
                retn

f               endp

```

### 84.1.3 Optimizing Keil (ARM mode)

```

                PUSH     {r4,lr}
                LDRB     r2,[r1,#0]
                CMP     r2,#0
                POPEQ   {r4,pc}
                B       |L0.80|

|L0.20|
                LDRB     r12,[r3,#0]
                CMP     r12,#0
                BEQ     |L0.64|
                LDRB     r4,[r2,#0]
                CMP     r4,#0
                POPEQ   {r4,pc}
                CMP     r12,r4
                ADDEQ   r3,r3,#1
                ADDEQ   r2,r2,#1
                BEQ     |L0.20|
                B       |L0.76|

|L0.64|
                LDRB     r2,[r2,#0]
                CMP     r2,#0
                POPEQ   {r4,pc}

|L0.76|
                ADD     r0,r0,#1

|L0.80|
                LDRB     r2,[r0,#0]
                CMP     r2,#0
                MOVNE   r3,r0
                MOVNE   r2,r1
                MOVEQ   r0,#0

```

BNE	L0.20
POP	{r4,pc}

### 84.1.4 Optimizing Keil (thumb mode)

	PUSH	{r4,r5,lr}
	LDRB	r2,[r1,#0]
	CMP	r2,#0
	BEQ	L0.54
	B	L0.46
L0.10		
	MOVS	r3,r0
	MOVS	r2,r1
	B	L0.20
L0.16		
	ADDS	r3,r3,#1
	ADDS	r2,r2,#1
L0.20		
	LDRB	r4,[r3,#0]
	CMP	r4,#0
	BEQ	L0.38
	LDRB	r5,[r2,#0]
	CMP	r5,#0
	BEQ	L0.54
	CMP	r4,r5
	BEQ	L0.16
	B	L0.44
L0.38		
	LDRB	r2,[r2,#0]
	CMP	r2,#0
	BEQ	L0.54
L0.44		
	ADDS	r0,r0,#1
L0.46		
	LDRB	r2,[r0,#0]
	CMP	r2,#0
	BNE	L0.10
	MOVS	r0,#0
L0.54		
	POP	{r4,r5,pc}

## 84.2 Exercise 2.5

This exercise is rather on knowledge than on reading code.

The function is taken from OpenWatcom.

## 84.2.1 Optimizing MSVC 2010

```

_DATA      SEGMENT
COMM      __v:DWORD
_DATA      ENDS
PUBLIC    __real@3e45798ee2308c3a
PUBLIC    __real@4147ffff80000000
PUBLIC    __real@4150017ec0000000
PUBLIC    _f
EXTRN    __fltused:DWORD
CONST     SEGMENT
__real@3e45798ee2308c3a DQ 03e45798ee2308c3ar      ; 1e-008
__real@4147ffff80000000 DQ 04147ffff80000000r      ; 3.14573e+006
__real@4150017ec0000000 DQ 04150017ec0000000r      ; 4.19584e+006
CONST     ENDS
_TEXT     SEGMENT
_v1$ = -16          ; size = 8
_v2$ = -8           ; size = 8
_f       PROC
    sub     esp, 16          ; 00000010H
    fld     QWORD PTR __real@4150017ec0000000
    fstp    QWORD PTR _v1$[esp+16]
    fld     QWORD PTR __real@4147ffff80000000
    fstp    QWORD PTR _v2$[esp+16]
    fld     QWORD PTR _v1$[esp+16]
    fld     QWORD PTR _v1$[esp+16]
    fdiv    QWORD PTR _v2$[esp+16]
    fmul    QWORD PTR _v2$[esp+16]
    fsubp   ST(1), ST(0)
    fcomp   QWORD PTR __real@3e45798ee2308c3a
    fnstsw ax
    test    ah, 65          ; 00000041H
    jne     SHORT $LN1@f
    or      DWORD PTR __v, 1
$LN1@f:
    add     esp, 16          ; 00000010H
    ret     0
_f       ENDP
_TEXT     ENDS

```



## 84.3 Exercise 2.6

### 84.3.1 Optimizing MSVC 2010

```

PUBLIC    _f
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_k0$ = -12        ; size = 4
_k3$ = -8        ; size = 4
_k2$ = -4        ; size = 4
_v$ = 8         ; size = 4
_k1$ = 12       ; size = 4
_k$ = 12        ; size = 4
_f      PROC
    sub     esp, 12        ; 0000000cH
    mov     ecx, DWORD PTR _v$[esp+8]
    mov     eax, DWORD PTR [ecx]
    mov     ecx, DWORD PTR [ecx+4]
    push    ebx
    push    esi
    mov     esi, DWORD PTR _k$[esp+16]
    push    edi
    mov     edi, DWORD PTR [esi]
    mov     DWORD PTR _k0$[esp+24], edi
    mov     edi, DWORD PTR [esi+4]
    mov     DWORD PTR _k1$[esp+20], edi
    mov     edi, DWORD PTR [esi+8]
    mov     esi, DWORD PTR [esi+12]
    xor     edx, edx
    mov     DWORD PTR _k2$[esp+24], edi
    mov     DWORD PTR _k3$[esp+24], esi
    lea    edi, DWORD PTR [edx+32]
$LL8@f:
    mov     esi, ecx
    shr     esi, 5
    add     esi, DWORD PTR _k1$[esp+20]
    mov     ebx, ecx
    shl     ebx, 4
    add     ebx, DWORD PTR _k0$[esp+24]
    sub     edx, 1640531527    ; 61c88647H
    xor     esi, ebx
    lea    ebx, DWORD PTR [edx+ecx]
    xor     esi, ebx
    add     eax, esi
    mov     esi, eax
    shr     esi, 5
    add     esi, DWORD PTR _k3$[esp+24]

```

```

mov    ebx, eax
shl    ebx, 4
add    ebx, DWORD PTR _k2$[esp+24]
xor    esi, ebx
lea    ebx, DWORD PTR [edx+eax]
xor    esi, ebx
add    ecx, esi
dec    edi
jne    SHORT $LL8@f
mov    edx, DWORD PTR _v$[esp+20]
pop    edi
pop    esi
mov    DWORD PTR [edx], eax
mov    DWORD PTR [edx+4], ecx
pop    ebx
add    esp, 12                ; 0000000cH
ret    0
_f    ENDP

```

### 84.3.2 Optimizing Keil (ARM mode)

```

PUSH    {r4-r10,lr}
ADD     r5,r1,#8
LDM     r5,{r5,r7}
LDR     r2,[r0,#4]
LDR     r3,[r0,#0]
LDR     r4,|L0.116|
LDR     r6,[r1,#4]
LDR     r8,[r1,#0]
MOV     r12,#0
MOV     r1,r12
|L0.40|
ADD     r12,r12,r4
ADD     r9,r8,r2,LSL #4
ADD     r10,r2,r12
EOR     r9,r9,r10
ADD     r10,r6,r2,LSR #5
EOR     r9,r9,r10
ADD     r3,r3,r9
ADD     r9,r5,r3,LSL #4
ADD     r10,r3,r12
EOR     r9,r9,r10
ADD     r10,r7,r3,LSR #5
EOR     r9,r9,r10
ADD     r1,r1,#1
CMP     r1,#0x20

```

```

ADD    r2,r2,r9
STRCS  r2,[r0,#4]
STRCS  r3,[r0,#0]
BCC    |L0.40|
POP    {r4-r10,pc}

```

|L0.116|

```

DCD    0x9e3779b9

```

### 84.3.3 Optimizing Keil (thumb mode)

```

PUSH   {r1-r7,lr}
LDR    r5,|L0.84|
LDR    r3,[r0,#0]
LDR    r2,[r0,#4]
STR    r5,[sp,#8]
MOVS   r6,r1
LDM    r6,{r6,r7}
LDR    r5,[r1,#8]
STR    r6,[sp,#4]
LDR    r6,[r1,#0xc]
MOVS   r4,#0
MOVS   r1,r4
MOV    lr,r5
MOV    r12,r6
STR    r7,[sp,#0]

```

|L0.30|

```

LDR    r5,[sp,#8]
LSLS   r6,r2,#4
ADDS   r4,r4,r5
LDR    r5,[sp,#4]
LSRS   r7,r2,#5
ADDS   r5,r6,r5
ADDS   r6,r2,r4
EORS   r5,r5,r6
LDR    r6,[sp,#0]
ADDS   r1,r1,#1
ADDS   r6,r7,r6
EORS   r5,r5,r6
ADDS   r3,r5,r3
LSLS   r5,r3,#4
ADDS   r6,r3,r4
ADD    r5,r5,lr
EORS   r5,r5,r6
LSRS   r6,r3,#5
ADD    r6,r6,r12

```

EORS	r5,r5,r6
ADDS	r2,r5,r2
CMP	r1,#0x20
BCC	L0.30
STR	r3,[r0,#0]
STR	r2,[r0,#4]
POP	{r1-r7,pc}
L0.84	
DCD	0x9e3779b9

## 84.4 Exercise 2.13

This is a well-known cryptoalgorithm of the past. How it's called?

### 84.4.1 Optimizing MSVC 2012

```

_in$ = 8 ; size ↗
  ↘ = 2
_f PROC
  movzx ecx, WORD PTR _in$[esp-4]
  lea eax, DWORD PTR [ecx*4]
  xor eax, ecx
  add eax, eax
  xor eax, ecx
  shl eax, 2
  xor eax, ecx
  and eax, 32 ; ↗
  ↘ 00000020H
  shl eax, 10 ; ↗
  ↘ 0000000aH
  shr ecx, 1
  or eax, ecx
  ret 0
_f ENDP

```

### 84.4.2 Keil (ARM mode)

f PROC	
EOR	r1,r0,r0,LSR #2
EOR	r1,r1,r0,LSR #3
EOR	r1,r1,r0,LSR #5

```

AND    r1,r1,#1
LSR    r0,r0,#1
ORR    r0,r0,r1,LSL #15
BX     lr
ENDP

```

### 84.4.3 Keil (thumb mode)

```

f PROC
LSRS   r1,r0,#2
EORS   r1,r1,r0
LSRS   r2,r0,#3
EORS   r1,r1,r2
LSRS   r2,r0,#5
EORS   r1,r1,r2
LSLS   r1,r1,#31
LSRS   r0,r0,#1
LSRS   r1,r1,#16
ORRS   r0,r0,r1
BX     lr
ENDP

```

## 84.5 Exercise 2.14

Another well-known algorithm. The function takes two variables and returning one.

### 84.5.1 MSVC 2012

```

_rt$1 = -4 ; size ↙
  ↙ = 4
_rt$2 = 8 ; size ↙
  ↙ = 4
_x$ = 8 ; size ↙
  ↙ = 4
_y$ = 12 ; size ↙
  ↙ = 4
?f@@YAIIII@Z PROC ; f
    push    ecx
    push    esi
    mov     esi, DWORD PTR _x$[esp+4]
    test   esi, esi

```

```

    jne     SHORT $LN7@f
    mov     eax, DWORD PTR _y$[esp+4]
    pop     esi
    pop     ecx
    ret     0
$LN7@f:
    mov     edx, DWORD PTR _y$[esp+4]
    mov     eax, esi
    test    edx, edx
    je      SHORT $LN8@f
    or      eax, edx
    push    edi
    bsf     edi, eax
    bsf     eax, esi
    mov     ecx, eax
    mov     DWORD PTR _rt$1[esp+12], eax
    bsf     eax, edx
    shr     esi, cl
    mov     ecx, eax
    shr     edx, cl
    mov     DWORD PTR _rt$2[esp+8], eax
    cmp     esi, edx
    je      SHORT $LN22@f
$LN23@f:
    jbe     SHORT $LN2@f
    xor     esi, edx
    xor     edx, esi
    xor     esi, edx
$LN2@f:
    cmp     esi, 1
    je      SHORT $LN22@f
    sub     edx, esi
    bsf     eax, edx
    mov     ecx, eax
    shr     edx, cl
    mov     DWORD PTR _rt$2[esp+8], eax
    cmp     esi, edx
    jne     SHORT $LN23@f
$LN22@f:
    mov     ecx, edi
    shl     esi, cl
    pop     edi
    mov     eax, esi
$LN8@f:
    pop     esi
    pop     ecx
    ret     0

```

```
?f@YAIIII@Z ENDP
```

## 84.5.2 Keil (ARM mode)

```

||f1|| PROC
    CMP        r0,#0
    RSB        r1,r0,#0
    AND        r0,r0,r1
    CLZ        r0,r0
    RSBNE     r0,r0,#0x1f
    BX        lr
    ENDP

f PROC
    MOVS      r2,r0
    MOV       r3,r1
    MOVEQ     r0,r1
    CMPNE     r3,#0
    PUSH      {lr}
    POPEQ     {pc}
    ORR       r0,r2,r3
    BL        ||f1||
    MOV       r12,r0
    MOV       r0,r2
    BL        ||f1||
    LSR       r2,r2,r0

|L0.196|
    MOV       r0,r3
    BL        ||f1||
    LSR       r0,r3,r0
    CMP       r2,r0
    EORHI     r1,r2,r0
    EORHI     r0,r0,r1
    EORHI     r2,r1,r0
    BEQ       |L0.240|
    CMP       r2,#1
    SUBNE     r3,r0,r2
    BNE       |L0.196|

|L0.240|
    LSL       r0,r2,r12
    POP       {pc}
    ENDP

```

## 84.5.3 GCC 4.6.3 for Raspberry Pi (ARM mode)

```

f:
    subs    r3, r0, #0
    beq     .L162
    cmp     r1, #0
    moveq   r1, r3
    beq     .L162
    orr     r2, r1, r3
    rsb     ip, r2, #0
    and     ip, ip, r2
    cmp     r2, #0
    rsb     r2, r3, #0
    and     r2, r2, r3
    clz     r2, r2
    rsb     r2, r2, #31
    clz     ip, ip
    rsbne   ip, ip, #31
    mov     r3, r3, lsr r2
    b       .L169

.L171:
    eorhi   r1, r1, r2
    eorhi   r3, r1, r2
    cmp     r3, #1
    rsb     r1, r3, r1
    beq     .L167

.L169:
    rsb     r0, r1, #0
    and     r0, r0, r1
    cmp     r1, #0
    clz     r0, r0
    mov     r2, r0
    rsbne   r2, r0, #31
    mov     r1, r1, lsr r2
    cmp     r3, r1
    eor     r2, r1, r3
    bne     .L171

.L167:
    mov     r1, r3, asl ip

.L162:
    mov     r0, r1
    bx     lr

```

## 84.6 Exercise 2.15

Well-known algorithm again. What it does?



Take also notice that the code for x86 uses FPU, but SIMD-instructions are used instead in x64 code. That's OK: 25.

## 84.6.1 Optimizing MSVC 2012 x64

```

__real@412e848000000000 DQ 0412e848000000000r ; 1e+006
__real@4010000000000000 DQ 04010000000000000r ; 4
__real@4008000000000000 DQ 04008000000000000r ; 3
__real@3f800000 DD 03f800000r ; 1

tmp$1 = 8
tmp$2 = 8
f PROC
    movsdx xmm3, QWORD PTR __real@4008000000000000
    movss xmm4, DWORD PTR __real@3f800000
    mov edx, DWORD PTR ?RNG_state@?1??get_rand@@@9@9
    xor ecx, ecx
    mov r8d, 200000 ; 00030↵
    ↪ d40H
    npad 2 ; align next label
$LL4@f:
    imul edx, 1664525 ; ↵
    ↪ 0019660dH
    add edx, 1013904223 ; 3↵
    ↪ c6ef35fH
    mov eax, edx
    and eax, 8388607 ; 007↵
    ↪ fffffH
    imul edx, 1664525 ; ↵
    ↪ 0019660dH
    bts eax, 30
    add edx, 1013904223 ; 3↵
    ↪ c6ef35fH
    mov DWORD PTR tmp$2[rsp], eax
    mov eax, edx
    and eax, 8388607 ; 007↵
    ↪ fffffH
    bts eax, 30
    movss xmm0, DWORD PTR tmp$2[rsp]
    mov DWORD PTR tmp$1[rsp], eax
    cvtps2pd xmm0, xmm0
    subss xmm0, xmm3
    cvtpd2ps xmm2, xmm0
    movss xmm0, DWORD PTR tmp$1[rsp]
    cvtps2pd xmm0, xmm0
    mulss xmm2, xmm2

```

```

    subsd    xmm0, xmm3
    cvtpd2ps xmm1, xmm0
    mulss    xmm1, xmm1
    addss    xmm1, xmm2
    comiss   xmm4, xmm1
    jbe     SHORT $LN3@f
    inc     ecx
$LN3@f:
    imul    edx, 1664525           ; ↵
↳ 0019660dH
    add     edx, 1013904223       ; 3↵
↳ c6ef35fH
    mov     eax, edx
    and    eax, 8388607           ; 007↵
↳ ffffffH
    imul    edx, 1664525           ; ↵
↳ 0019660dH
    bts     eax, 30
    add     edx, 1013904223       ; 3↵
↳ c6ef35fH
    mov     DWORD PTR tmp$2[rs], eax
    mov     eax, edx
    and    eax, 8388607           ; 007↵
↳ ffffffH
    bts     eax, 30
    movss   xmm0, DWORD PTR tmp$2[rs]
    mov     DWORD PTR tmp$1[rs], eax
    cvtps2pd xmm0, xmm0
    subsd   xmm0, xmm3
    cvtpd2ps xmm2, xmm0
    movss   xmm0, DWORD PTR tmp$1[rs]
    cvtps2pd xmm0, xmm0
    mulss   xmm2, xmm2
    subsd   xmm0, xmm3
    cvtpd2ps xmm1, xmm0
    mulss   xmm1, xmm1
    addss   xmm1, xmm2
    comiss   xmm4, xmm1
    jbe     SHORT $LN15@f
    inc     ecx
$LN15@f:
    imul    edx, 1664525           ; ↵
↳ 0019660dH
    add     edx, 1013904223       ; 3↵
↳ c6ef35fH
    mov     eax, edx

```

```

    and     eax, 8388607                ; 007↵
↳ ffffffH
    imul   edx, 1664525                ; ↵
↳ 0019660dH
    bts    eax, 30
    add    edx, 1013904223            ; 3↵
↳ c6ef35fH
    mov    DWORD PTR tmp$2[rsp], eax
    mov    eax, edx
    and    eax, 8388607                ; 007↵
↳ ffffffH
    bts    eax, 30
    movss  xmm0, DWORD PTR tmp$2[rsp]
    mov    DWORD PTR tmp$1[rsp], eax
    cvtpps2pd xmm0, xmm0
    subss  xmm0, xmm3
    cvtpps2pd xmm2, xmm0
    movss  xmm0, DWORD PTR tmp$1[rsp]
    cvtpps2pd xmm0, xmm0
    mulss  xmm2, xmm2
    subss  xmm0, xmm3
    cvtpps2pd xmm1, xmm0
    mulss  xmm1, xmm1
    addss  xmm1, xmm2
    comiss xmm4, xmm1
    jbe    SHORT $LN16@f
    inc    ecx
$LN16@f:
    imul   edx, 1664525                ; ↵
↳ 0019660dH
    add    edx, 1013904223            ; 3↵
↳ c6ef35fH
    mov    eax, edx
    and    eax, 8388607                ; 007↵
↳ ffffffH
    imul   edx, 1664525                ; ↵
↳ 0019660dH
    bts    eax, 30
    add    edx, 1013904223            ; 3↵
↳ c6ef35fH
    mov    DWORD PTR tmp$2[rsp], eax
    mov    eax, edx
    and    eax, 8388607                ; 007↵
↳ ffffffH
    bts    eax, 30
    movss  xmm0, DWORD PTR tmp$2[rsp]
    mov    DWORD PTR tmp$1[rsp], eax

```

```

    cvtpps2pd xmm0, xmm0
    subsd    xmm0, xmm3
    cvtprd2ps xmm2, xmm0
    movss   xmm0, DWORD PTR tmp$1[rsip]
    cvtpps2pd xmm0, xmm0
    mulss   xmm2, xmm2
    subsd   xmm0, xmm3
    cvtprd2ps xmm1, xmm0
    mulss   xmm1, xmm1
    addss   xmm1, xmm2
    comiss  xmm4, xmm1
    jbe     SHORT $LN17@f
    inc     ecx
$LN17@f:
    imul    edx, 1664525                ; 2
    ↪ 0019660dH
    add     edx, 1013904223            ; 32
    ↪ c6ef35fH
    mov     eax, edx
    and     eax, 8388607                ; 0072
    ↪ fffffH
    imul    edx, 1664525                ; 2
    ↪ 0019660dH
    bts    eax, 30
    add     edx, 1013904223            ; 32
    ↪ c6ef35fH
    mov     DWORD PTR tmp$2[rsip], eax
    mov     eax, edx
    and     eax, 8388607                ; 0072
    ↪ fffffH
    bts    eax, 30
    movss  xmm0, DWORD PTR tmp$2[rsip]
    mov    DWORD PTR tmp$1[rsip], eax
    cvtpps2pd xmm0, xmm0
    subsd   xmm0, xmm3
    cvtprd2ps xmm2, xmm0
    movss  xmm0, DWORD PTR tmp$1[rsip]
    cvtpps2pd xmm0, xmm0
    mulss  xmm2, xmm2
    subsd  xmm0, xmm3
    cvtprd2ps xmm1, xmm0
    mulss  xmm1, xmm1
    addss  xmm1, xmm2
    comiss xmm4, xmm1
    jbe    SHORT $LN18@f
    inc    ecx
$LN18@f:

```

```

    dec     r8
    jne     $LL4@f
    movd    xmm0, ecx
    mov     DWORD PTR ?RNG_state@?1??get_rand@@@9@9, edx
    cvtdq2ps xmm0, xmm0
    cvtps2pd xmm1, xmm0
    mulsd   xmm1, QWORD PTR __real@4010000000000000
    divsd   xmm1, QWORD PTR __real@412e848000000000
    cvtpd2ps xmm0, xmm1
    ret     0
f:      ENDP

```

## 84.6.2 Optimizing GCC 4.4.6 x64

```

f1:
    mov     eax, DWORD PTR v1.2084[rip]
    imul   eax, eax, 1664525
    add    eax, 1013904223
    mov    DWORD PTR v1.2084[rip], eax
    and    eax, 8388607
    or     eax, 1073741824
    mov    DWORD PTR [rsp-4], eax
    movss  xmm0, DWORD PTR [rsp-4]
    subss  xmm0, DWORD PTR .LC0[rip]
    ret

f:
    push   rbp
    xor    ebp, ebp
    push   rbx
    xor    ebx, ebx
    sub    rsp, 16

.LC0:
    xor    eax, eax
    call   f1
    xor    eax, eax
    movss  DWORD PTR [rsp], xmm0
    call   f1
    movss  xmm1, DWORD PTR [rsp]
    mulss  xmm0, xmm0
    mulss  xmm1, xmm1
    lea   eax, [rbx+1]
    addss  xmm1, xmm0
    movss  xmm0, DWORD PTR .LC1[rip]
    ucomiss xmm0, xmm1
    cmova  ebx, eax
    add    ebp, 1

```

```

    cmp     ebp, 1000000
    jne     .L6
    cvtsi2ss      xmm0, ebx
    unpcklps     xmm0, xmm0
    cvtps2pd     xmm0, xmm0
    mulsd      xmm0, QWORD PTR .LC2[rip]
    divsd      xmm0, QWORD PTR .LC3[rip]
    add      rsp, 16
    pop      rbx
    pop      rbp
    unpcklpd     xmm0, xmm0
    cvtpd2ps     xmm0, xmm0
    ret
v1.2084:
    .long    305419896
.LC0:
    .long    1077936128
.LC1:
    .long    1065353216
.LC2:
    .long    0
    .long    1074790400
.LC3:
    .long    0
    .long    1093567616

```

### 84.6.3 Optimizing GCC 4.8.1 x86

```

f1:
    sub     esp, 4
    imul   eax, DWORD PTR v1.2023, 1664525
    add    eax, 1013904223
    mov    DWORD PTR v1.2023, eax
    and    eax, 8388607
    or     eax, 1073741824
    mov    DWORD PTR [esp], eax
    fld   DWORD PTR [esp]
    fsub   DWORD PTR .LC0
    add    esp, 4
    ret

f:
    push   esi
    mov    esi, 1000000
    push   ebx
    xor    ebx, ebx
    sub    esp, 16

```

```

.L7:
    call    f1
    fstp   DWORD PTR [esp]
    call    f1
    lea    eax, [ebx+1]
    fld    DWORD PTR [esp]
    fmul   st, st(0)
    fxch   st(1)
    fmul   st, st(0)
    faddp  st(1), st
    fld1
    fucomip st, st(1)
    fstp   st(0)
    cmova  ebx, eax
    sub    esi, 1
    jne    .L7
    mov    DWORD PTR [esp+4], ebx
    fld    DWORD PTR [esp+4]
    fmul   DWORD PTR .LC3
    fdiv   DWORD PTR .LC4
    fstp   DWORD PTR [esp+8]
    fld    DWORD PTR [esp+8]
    add    esp, 16
    pop    ebx
    pop    esi
    ret

v1.2023:
    .long  305419896

.LC0:
    .long  1077936128

.LC3:
    .long  1082130432

.LC4:
    .long  1232348160

```

#### 84.6.4 Keil (ARM mode): Cortex-R4F CPU as target

```

f1 PROC
    LDR    r1, |L0.184|
    LDR    r0, [r1, #0] ; v1
    LDR    r2, |L0.188|
    VMOV.F32 s1, #3.00000000
    MUL    r0, r0, r2
    LDR    r2, |L0.192|
    ADD    r0, r0, r2

```

```

    STR    r0,[r1,#0] ; v1
    BFC    r0,#23,#9
    ORR    r0,r0,#0x40000000
    VMOV   s0,r0
    VSUB.F32 s0,s0,s1
    BX     lr
    ENDP

f      PROC
    PUSH   {r4,r5,lr}
    MOV    r4,#0
    LDR    r5,|L0.196|
    MOV    r3,r4
|L0.68|
    BL     f1
    VMOV.F32 s2,s0
    BL     f1
    VMOV.F32 s1,s2
    ADD    r3,r3,#1
    VMUL.F32 s1,s1,s1
    VMLA.F32 s1,s0,s0
    VMOV   r0,s1
    CMP    r0,#0x3f800000
    ADDLT  r4,r4,#1
    CMP    r3,r5
    BLT    |L0.68|
    VMOV   s0,r4
    VMOV.F64 d1,#4.00000000
    VCVT.F32.S32 s0,s0
    VCVT.F64.F32 d0,s0
    VMUL.F64 d0,d0,d1
    VLDR   d1,|L0.200|
    VDIV.F64 d2,d0,d1
    VCVT.F32.F64 s0,d2
    POP    {r4,r5,pc}
    ENDP

|L0.184|
    DCD    ||.data||
|L0.188|
    DCD    0x0019660d
|L0.192|
    DCD    0x3c6ef35f
|L0.196|
    DCD    0x000f4240
|L0.200|
    DCFD   0x412e848000000000 ; 1000000

```



```

          DCD      0x00000000
          AREA    ||.data||, DATA, ALIGN=2
v1
          DCD      0x12345678

```

## 84.7 Exercise 2.16

Well-known function. What it computes? Why stack overflows if 4 and 2 are supplied at input? Are there any error?

### 84.7.1 Optimizing MSVC 2012 x64

```

m$ = 48
n$ = 56
f   PROC
$LN14:
    push    rbx
    sub     rsp, 32
    mov     eax, edx
    mov     ebx, ecx
    test    ecx, ecx
    je     SHORT $LN11@f
$LL5@f:
    test    eax, eax
    jne    SHORT $LN1@f
    mov     eax, 1
    jmp    SHORT $LN12@f
$LN1@f:
    lea    edx, DWORD PTR [rax-1]
    mov     ecx, ebx
    call   f
$LN12@f:
    dec     ebx
    test    ebx, ebx
    jne    SHORT $LL5@f
$LN11@f:
    inc     eax
    add     rsp, 32
    pop    rbx
    ret    0
f       ENDP

```

## 84.7.2 Optimizing Keil (ARM mode)

```
f PROC
    PUSH    {r4,lr}
    MOVS   r4,r0
    ADDEQ  r0,r1,#1
    POPEQ  {r4,pc}
    CMP    r1,#0
    MOVEQ  r1,#1
    SUBEQ  r0,r0,#1
    BEQ    |L0.48|
    SUB    r1,r1,#1
    BL     f
    MOV    r1,r0
    SUB    r0,r4,#1
|L0.48|
    POP    {r4,lr}
    B     f
    ENDP
```

## 84.7.3 Optimizing Keil (thumb mode)

```
f PROC
    PUSH    {r4,lr}
    MOVS   r4,r0
    BEQ    |L0.26|
    CMP    r1,#0
    BEQ    |L0.30|
    SUBS   r1,r1,#1
    BL     f
    MOVS   r1,r0
|L0.18|
    SUBS   r0,r4,#1
    BL     f
    POP    {r4,pc}
|L0.26|
    ADDS   r0,r1,#1
    POP    {r4,pc}
|L0.30|
    MOVS   r1,#1
    B     |L0.18|
    ENDP
```

## 84.8 Exercise 2.17

This program prints some information to `stdout`, each time different. What is it?  
Compiled binaries:

- Linux x64: [http://beginners.re/exercises/2/17/17\\_Linux\\_x64.tar](http://beginners.re/exercises/2/17/17_Linux_x64.tar)
- Mac OS X: [http://beginners.re/exercises/2/17/17\\_MacOSX\\_x64.tar](http://beginners.re/exercises/2/17/17_MacOSX_x64.tar)
- Win32: [http://beginners.re/exercises/2/17/17\\_win32.exe](http://beginners.re/exercises/2/17/17_win32.exe)
- Win64: [http://beginners.re/exercises/2/17/17\\_win64.exe](http://beginners.re/exercises/2/17/17_win64.exe)

As of Windows versions, you may need to install [MSVC 2012](#) redistributable.

## 84.9 Exercise 2.18

This program requires password. Find it.

By the way, multiple passwords may work. Try to find more.

As an additional exercise, try to change the password by patching executable file.

- Win32: <http://beginners.re/exercises/2/18/password2.exe>
- Linux x86: [http://beginners.re/exercises/2/18/password2\\_Linux\\_x86.tar](http://beginners.re/exercises/2/18/password2_Linux_x86.tar)
- Mac OS X: [http://beginners.re/exercises/2/18/password2\\_MacOSX64.tar](http://beginners.re/exercises/2/18/password2_MacOSX64.tar)

## 84.10 Exercise 2.19

The same as in exercise 2.18.

- Win32: <http://beginners.re/exercises/2/19/password3.exe>
- Linux x86: [http://beginners.re/exercises/2/19/password3\\_Linux\\_x86.tar](http://beginners.re/exercises/2/19/password3_Linux_x86.tar)
- Mac OS X: [http://beginners.re/exercises/2/19/password3\\_MacOSX64.tar](http://beginners.re/exercises/2/19/password3_MacOSX64.tar)

# Chapter 85

## Level 3

For solving level 3 tasks, you'll probably need considerable amount of time, maybe up to one day.

### 85.1 Exercise 3.2

There is a small executable file with a well-known cryptosystem inside. Try to identify it.

- Windows x86: [http://beginners.re/exercises/3/2/unknown\\_cryptosystem.exe](http://beginners.re/exercises/3/2/unknown_cryptosystem.exe)
- Linux x86: [http://beginners.re/exercises/3/2/unknown\\_encryption\\_linux86.tar](http://beginners.re/exercises/3/2/unknown_encryption_linux86.tar)
- Mac OS X (x64): [http://beginners.re/exercises/3/2/unknown\\_encryption\\_MacOSX.tar](http://beginners.re/exercises/3/2/unknown_encryption_MacOSX.tar)

### 85.2 Exercise 3.3

There is a small executable file, some utility. It opens another file, reads it, calculate something and prints a float number. Try to understand what it do.

- Windows x86: [http://beginners.re/exercises/3/3/unknown\\_utility\\_2\\_3.exe](http://beginners.re/exercises/3/3/unknown_utility_2_3.exe)
- Linux x86: [http://beginners.re/exercises/3/3/unknown\\_utility\\_2\\_3\\_Linux86.tar](http://beginners.re/exercises/3/3/unknown_utility_2_3_Linux86.tar)
- Mac OS X (x64): [http://beginners.re/exercises/3/3/unknown\\_utility\\_2\\_3\\_MacOSX.tar](http://beginners.re/exercises/3/3/unknown_utility_2_3_MacOSX.tar)

## 85.3 Exercise 3.4

There is an utility which encrypts/decrypts files, by password. There is an encrypted text file, password is unknown. Encrypted file is a text in English language. The utility uses relatively strong cryptosystem, nevertheless, it was implemented with a serious blunder. Since the mistake present, it is possible to decrypt the file with a little effort.

Try to find the mistake and decrypt the file.

- Windows x86: [http://beginners.re/exercises/3/4/amateur\\_cryptor.exe](http://beginners.re/exercises/3/4/amateur_cryptor.exe)
- Text file: [http://beginners.re/exercises/3/4/text\\_encrypted](http://beginners.re/exercises/3/4/text_encrypted)

## 85.4 Exercise 3.5

This is software copy protection imitation, which uses key file. The key file contain user (or customer) name and serial number.

There are two tasks:

- (Easy) with the help of [tracer](#) or any other debugger, force the program to accept changed key file.
- (Medium) your goal is to modify user name to another, however, it is not allowed to patch the program.
- Windows x86: [http://beginners.re/exercises/3/5/super\\_mega\\_protection.exe](http://beginners.re/exercises/3/5/super_mega_protection.exe)
- Linux x86: [http://beginners.re/exercises/3/5/super\\_mega\\_protect.tar](http://beginners.re/exercises/3/5/super_mega_protect.tar)
- Mac OS X (x64) [http://beginners.re/exercises/3/5/super\\_mega\\_protection\\_MacOSX.tar](http://beginners.re/exercises/3/5/super_mega_protection_MacOSX.tar)
- Key file: <http://beginners.re/exercises/3/5/sample.key>

## 85.5 Exercise 3.6

Here is a very primitive toy web-server, supporting only static files, without CGI<sup>1</sup>, etc. At least 4 vulnerabilities are left here intentionally. Try to find them all and exploit them in order for breaking into a remote host.

---

<sup>1</sup>Common Gateway Interface

- Windows x86: [http://beginners.re/exercises/3/6/webserv\\_win32.rar](http://beginners.re/exercises/3/6/webserv_win32.rar)
- Linux x86: [http://beginners.re/exercises/3/6/webserv\\_Linux\\_x86.tar](http://beginners.re/exercises/3/6/webserv_Linux_x86.tar)
- Mac OS X (x64): [http://beginners.re/exercises/3/6/webserv\\_MacOSX\\_x64.tar](http://beginners.re/exercises/3/6/webserv_MacOSX_x64.tar)

## 85.6 Exercise 3.8

It's a well known data compression algorithm. However, due to mistake (or typo), it decompress incorrectly. Here we can see this bug in these examples.

This is a text used as a source: <http://beginners.re/exercises/3/8/test.txt>

This is a text compressed correctly: <http://beginners.re/exercises/3/8/test.compressed>

This is incorrectly uncompressed text: [http://beginners.re/exercises/3/8/test.uncompressed\\_incorrectly](http://beginners.re/exercises/3/8/test.uncompressed_incorrectly).

Try to find and fix bug. With some effort, it can be done even by patching.

- Windows x86: [http://beginners.re/exercises/3/8/compressor\\_win32.exe](http://beginners.re/exercises/3/8/compressor_win32.exe)
- Linux x86: [http://beginners.re/exercises/3/8/compressor\\_linux86.tar](http://beginners.re/exercises/3/8/compressor_linux86.tar)
- Mac OS X (x64): [http://beginners.re/exercises/3/8/compressor\\_MacOSX64.tar](http://beginners.re/exercises/3/8/compressor_MacOSX64.tar)

## Chapter 86

# crackme / keygenme

Couple of my [keygenmes](#):

<http://crackmes.de/users/yonkie/>

# Afterword



## Chapter 87

# Questions?

Do not hesitate to mail any questions to the author: <dennis(a)yurichev.com>

There is also supporting forum, you may ask any questions there :

<http://forum.yurichev.com/>

Please, also do not hesitate to send me any corrections (including grammar ones (you see how horrible my English is?)), etc.

# Appendix

# Appendix A

## x86

### A.1 Terminology

Common for 16-bit (8086/80286), 32-bit (80386, etc), 64-bit.

**byte** 8-bit. DB assembly directive is used for defining variables and array of bytes. Bytes are passed in 8-bit part of registers: AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL.

**word** 16-bit. DW assembly directive –”. Words are passed in 16-bit part of 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 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](#).

### A.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 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 low byte part of 16-bit register, high byte part

or a 16-bit register as a whole. Probably, 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 x86-84, 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 byte parts of these registers are prefixed by *L*, e.g.: *R8L*, but *IDA* names these registers by adding *B* suffix, e.g.: *R8B*.

### A.2.1 RAX/EAX/AX/AL

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

**AKA** accumulator. The result of function if usually returned via this register.

### A.2.2 RBX/EBX/BX/BL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RBX <sup>x64</sup>							
EBX						BX	
						BH	BL

### A.2.3 RCX/ECX/CX/CL

7th (byte number)	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).

### A.2.4 RDX/EDX/DX/DL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RDX <sup>x64</sup>							
EDX						DX	
						DH	DL

**A.2.5 RSI/ESI/SI/SIL**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RSI <sup>x64</sup>							
				ESI			
						SI	
							SIL <sup>x64</sup>

**AKA** “source”. Used as source in the instructions REP MOV<sub>S</sub>x, REP CMPS<sub>x</sub>.

**A.2.6 RDI/EDI/DI/DIL**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RDI <sup>x64</sup>							
				EDI			
						DI	
							DIL <sup>x64</sup>

**AKA** “destination”. Used as a pointer to destination place in the instructions REP MOV<sub>S</sub>x, REP STOS<sub>x</sub>.

**A.2.7 R8/R8D/R8W/R8L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R8							
				R8D			
						R8W	
							R8L

**A.2.8 R9/R9D/R9W/R9L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R9							
				R9D			
						R9W	
							R9L

**A.2.9 R10/R10D/R10W/R10L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R10							
				R10D			
						R10W	
							R10L

**A.2.10 R11/R11D/R11W/R11L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R11							
				R11D			
						R11W	
							R11L

**A.2.11 R12/R12D/R12W/R12L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R12							
				R12D			
						R12W	
							R12L

**A.2.12 R13/R13D/R13W/R13L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R13							
				R13D			
						R13W	
							R13L

**A.2.13 R14/R14D/R14W/R14L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R14							
				R14D			
						R14W	
							R14L

**A.2.14 R15/R15D/R15W/R15L**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R15							
				R15D			
						R15W	
							R15L

**A.2.15 RSP/ESP/SP/SPL**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RSP							
				ESP			
						SP	
							SPL

**AKA stack pointer.** Usually points to the current stack except those cases when it is not yet initialized.

**A.2.16 RBP/EBP/BP/BPL**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RBP							
				EBP			
						BP	
							BPL

**AKA frame pointer.** Usually used for local variables and arguments of function accessing. More about it: (6.2.1).

**A.2.17 RIP/EIP/IP**

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RIP <sup>x64</sup>							
				EIP			
						IP	

**AKA “instruction pointer”** <sup>1</sup>. Usually always points to the current instruction. Cannot be modified, however, it is possible to do (which is equivalent to):

```
MOV EAX, ...
JMP EAX
```

Or:

```
PUSH value
RET
```

<sup>1</sup>Sometimes also called “program counter”

## A.2.18 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 done for faster access to the [TLS](#) and other structures like [TIB](#).

In the past, these registers were used as segment registers ([78](#)).

## A.2.19 Flags register

AKA EFLAGS.

Bit (mask)	Abbreviation (meaning)	Description
0 (1)	CF (Carry)	The CLC/STC/CMC instructions for setting/resetting/toggling t
2 (4)	PF (Parity)	<a href="#">(16.3.1)</a> .
4 (0x10)	AF (Adjust)	
6 (0x40)	ZF (Zero)	Setting to 0 if the last operation's result wa
7 (0x80)	SF (Sign)	
8 (0x100)	TF (Trap)	Used for debugging. If turned on, an exception will l generated after each instruction
9 (0x200)	IF (Interrupt enable)	Are interrupts enabled. The CLI/STI instructions are us for the flag setting/resetting
10 (0x400)	DF (Direction)	A directions is set for the REP MOVSt, REP CMPSx, REP L The CLD/STD instructions are u for the flag setting/resetting
11 (0x800)	OF (Overflow)	
12, 13 (0x3000)	IOPL (I/O privilege level) <sup>80286</sup>	
14 (0x4000)	NT (Nested task) <sup>80286</sup>	
16 (0x10000)	RF (Resume) <sup>80386</sup>	Used for debugging. CPU will ignore hardware break if the flag is set.
17 (0x20000)	VM (Virtual 8086 mode) <sup>80386</sup>	
18 (0x40000)	AC (Alignment check) <sup>80486</sup>	
19 (0x80000)	VIF (Virtual interrupt) <sup>Pentium</sup>	
20 (0x100000)	VIP (Virtual interrupt pending) <sup>Pentium</sup>	
21 (0x200000)	ID (Identification) <sup>Pentium</sup>	

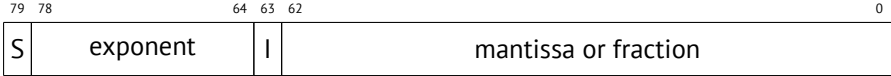


All the rest flags are reserved.

## A.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:



( S—sign, I—integer part )

### A.3.1 Control Word

Register controlling behaviour of the FPU.

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 unsi 1 – respect both $+\infty$ and $-\infty$

The PM, UM, OM, ZM, DM, IM flags are defining if to generate exception in case of corresponding errors.

### A.3.2 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 are signaling about exceptions.

About the C3, C2, C1, C0 read more: [\(16.3.1\)](#).

N.B.: When ST( $x$ ) is used, FPU adds  $x$  to TOP (by modulo 8) and that is how it gets internal register's number.

### A.3.3 Tag Word

The register has current information about number's registers usage.

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)

For each tag:

- 00 – The register contains a non-zero value
- 01 – The register contains 0

- 10 – The register contains a special value (NAN<sup>2</sup>, ∞, or denormal)
- 11 – The register is empty

## A.4 SIMD-registers

### A.4.1 MMX-registers

8 64-bit registers: MM0..MM7.

### A.4.2 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.

## A.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

### A.5.1 DR6

Bit (mask)	Description
0 (1)	B0 – breakpoint #1 was triggered
1 (2)	B1 – breakpoint #2 was triggered
2 (4)	B2 – breakpoint #3 was triggered
3 (8)	B3 – breakpoint #4 was triggered
13 (0x2000)	BD – modification attempt of one of DRx registers. may be raised if GD is enabled
14 (0x4000)	BS – single step breakpoint (TF flag was set in EFLAGS). Highest priority. Other bits may also be set.
15 (0x8000)	BT (task switch flag)

<sup>2</sup>Not a Number

N.B. Single step breakpoint is a breakpoint occurring after each instruction. It can be enabled by setting TF in EFLAGS (A.2.19).

## A.5.2 DR7

Breakpoint types are set here.

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 will be raised if any MOV instruction tries to modify one of DRx registers
16,17 (0x30000)	breakpoint #1: R/W – type
18,19 (0xC0000)	breakpoint #1: LEN – length
20,21 (0x3000000)	breakpoint #2: R/W – type
22,23 (0xC000000)	breakpoint #2: LEN – length
24,25 (0x30000000)	breakpoint #3: R/W – type
26,27 (0xC0000000)	breakpoint #3: LEN – length
28,29 (0x300000000)	breakpoint #4: R/W – type
30,31 (0xC00000000)	breakpoint #4: LEN – length

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

## A.6 Instructions

Instructions marked as (M) are not usually generated by compiler: if you see it, it is probably hand-written piece of assembly code, or this is compiler intrinsic (74).

Only most frequently used instructions are listed here. Read [Int13] or [AMD13a] for a full documentation.

Should one memorize instruction opcodes? No, only those which are used for code patching (73.2). All the rest opcodes are not needed to be memorized.

### A.6.1 Prefixes

**LOCK** force CPU to make exclusive access to the RAM in multiprocessor environment. For the sake of simplification, it can be said that when instruction with this prefix is executed, all other CPUs in multiprocessor system is stopped. Most often it is used for critical sections, semaphores, mutexes. Commonly used with ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. Read more about critical sections (53.4).

**REP** used with MOV<sub>Sx</sub> and STOS<sub>x</sub>: execute the instruction in loop, counter is located in the CX/ECX/RCX register. For detailed description, read more about MOV<sub>Sx</sub> (A.6.2) and STOS<sub>x</sub> (A.6.2) instructions.

Instructions prefixed by REP are sensitive to DF flag, which is used to set direction.

**REPE/REPNE** (AKA REPZ/REPNZ) used with CMPS<sub>x</sub> and SCAS<sub>x</sub>: execute the last instruction in loop, count is set in the CX/ECX/RCX register. It will terminate prematurely if ZF is 0 (REPE) or if ZF is 1 (REPNE).

For detailed description, read more about CMPS<sub>x</sub> (A.6.3) and SCAS<sub>x</sub> (A.6.2) instructions.

Instructions prefixed by REPE/REPNE are sensitive to DF flag, which is used to set direction.

### A.6.2 Most frequently used instructions

These can be memorized in the first place.

**ADC** (*add with carry*) add values, **increment** result if CF flag is set. often used for addition of large values, for example, to add two 64-bit values in 32-bit environment using two ADD and ADC instructions, for example:

```

; work with 64-bit values: add val1 to val2.
; .lo mean 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: [22](#).

**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 no results writing

**DEC** [decrement](#). CF flag is not touched.

**IMUL** signed multiply

**INC** [increment](#). CF flag is not touched.

**JCXZ, JECXZ, JRCXZ** (M) jump if CX/ECX/RCX=0

**JMP** jump to another address. Opcode has [jump offset](#).

**Jcc** (where cc—condition code)

A lot of instructions has synonyms (denoted with AKA), this was done for convenience. Synonymous instructions are translating into the same opcode. Opcode has [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.
- 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:



**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 address

This instruction was intended not for values summing and multiplication but for address forming, e.g., for forming address of array element by adding array address, element index, with multiplication of element size<sup>3</sup>.

So, the difference between MOV and LEA is that MOV forms memory address and loads value from memory or stores it there, but LEA just forms an address.

<sup>3</sup>See also: [http://en.wikipedia.org/wiki/Addressing\\_mode](http://en.wikipedia.org/wiki/Addressing_mode)

But nevertheless, it can be used for any other calculations.

LEA is convenient because the computations performing by it is not alter CPU flags. This may be very important for OOE processors (to make less count of data dependencies).

```
int f(int a, int b)
{
    return a*8+b;
};
```

Listing A.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;
};
```

Listing A.2: Intel C++ 2011

```
_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 instead of one IMUL will perform faster.

**MOVS<sub>B</sub>/MOVSW/MOVSD/MOVSQ** copy byte/ 16-bit word/ 32-bit word/ 64-bit word address of which is in the SI/ESI/RSI into the place address of which is in the DI/EDI/RDI.

Together with REP prefix, it will repeated in loop, count is stored in the CX/ECX/RCX register: it works like memcpy() in C. If block size is known



to compiler on compile stage, `memcpy()` is often inlined into short code fragment using `REP MOVSD`, sometimes even as several instructions.

`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 will work faster then copying 15 bytes using just one `REP MOVSB`).

**MOVSB** load with sign extension see also: (14.1.1)

**MOVSD** load and clear all the rest bits see also: (14.1.1)

**MOV** load value. this instruction was named awry resulting confusion (data are not moved but rather copied), in other architectures the same instructions is usually named "LOAD" or something like that.

One important thing: if to set low 16-bit part of 32-bit register in 32-bit mode, high 16 bits will remain as they were. But if to modify low 32-bit of register in 64-bit mode, high 32 bits of registers will be cleared.

Supposedly, it was done for x86-64 code porting simplification.

**MUL** unsigned multiply

**NEG** negation:  $op = -op$

**NOP** `NOP`. Opcode is 0x90, so it is in fact mean `XCHG EAX, EAX` idle instruction. This means, x86 do not have dedicated `NOP` instruction (as in many `RISC`). More examples of such operations: (72).

`NOP` may be generated by compiler for aligning labels on 16-byte boundary. Another very popular usage of `NOP` is to replace manually (patch) some instruction like conditional jump to `NOP` in order to disable its execution.

**NOT**  $op1: op1 = \neg op1$ . logical inversion

**OR** logical "or"

**POP** get value from the stack:  $value = SS:[ESP]$ ;  $ESP = ESP + 4$  (or 8)

**PUSH** push value to 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 is translating into `RETN` (“return near”) or, in MS-DOS times, where memory was addressed differently (78), into `RETF` (“return far”).

`RET` may have operand. Its algorithm then will be: `POP tmp; ADD ESP op1; JMP tmp`. `RET` with operand usually ends functions with `stdcall` calling convention, see also: 49.2.

**SAHF** copy bits from AH to CPU flags:



**SBB** (*subtraction with borrow*) subtract values, **decrement** result if `CF` flag is set. 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 mean 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: 22.

**SCASB/SCASW/SCASD/SCASQ** (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word stored in the `AX/EAX/RAX` with a variable address of which is in the `DI/EDI/RDI`. Set flags as `CMP` does.

This instruction is often used with `REPNE` prefix: continue to scan a buffer until a special value stored in `AX/EAX/RAX` is found. Hence “NE” in `REPNE`: continue to scan if compared values are not equal and stop when equal.

It is often used as `strlen()` C standard function, to determine **ASCIIZ** string length:

Example:

```

lea    edi, string
mov    ecx, 0FFFFFFFFh ; scan 2^32-1 bytes, i.e., almost ↙
        ↘ "infinitely"
xor    eax, eax        ; 0 is the terminator
repne scasb
add    edi, 0FFFFFFFFh ; correct it

```

```

; now EDI points to the last character of the ASCIIZ ↵
↵ string.

; let's determine string length
; current ECX = -1-strlen

not    ecx
dec    ecx

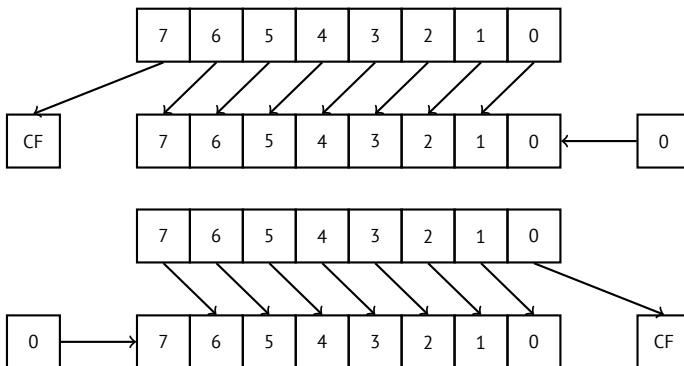
; now ECX contain string length

```

If to use different AX/EAX/RAX value, the function will act as `memchr()` standard C function, i.e., it will find specific byte.

**SHL** shift value left

**SHR** shift value right:



This instruction is frequently used for multiplication and division by  $2^n$ . Another very frequent application is bit fields processing: [18](#).

**SHRD** op1, op2, op3: shift value in op2 right by op3 bits, taking bits from op1.

Example: [22](#).

**STOSB/STOSW/STOSD/STOSQ** store byte/ 16-bit word/ 32-bit word/ 64-bit word from AX/EAX/RAX into the place address of which is in the DI/EDI/RDI.

Together with REP prefix, it will repeated in loop, count is stored in the CX/ECX/RCX register: it works like `memset()` in C. If block size is known to compiler on compile stage, `memset()` is often inlined into short code fragment using REP MOVSB, 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 will work faster then storing 15 bytes using just one REP STOSB).

**SUB** subtract values. frequently occurred pattern `SUB reg, reg` meaning write 0 to reg.

**TEST** same as AND but without results saving, see also: [18](#)

**XCHG** exchange values in operands

**XOR** `op1, op2: XOR4 values.`  $op1 = op1 \oplus op2$ . Frequently occurred pattern `XOR reg, reg` meaning write 0 to register.

### A.6.3 Less frequently used instructions

**BSF** *bit scan forward*, see also: [23.2](#)

**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

---

<sup>4</sup>eXclusive OR

These instructions consider value's sign, extending it to high part of newly constructed value. See also: [22.4](#).

**CLD** clear DF flag.

**CLI** (M) clear IF flag

**CMC** (M) toggle CF flag

**CMOVcc** conditional MOV: load if condition is true The condition codes are the same as in Jcc instructions ([A.6.2](#)).

**CMPSB/CMPSW/CMPSD/CMPSQ** (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word from the place address of which is in the SI/ESI/RSI with a variable address of which is in the DI/EDI/RDI. Set flags as CMP does.

Together with REP prefix, it will repeated in loop, count is stored in the CX/ECX/RCX register, the process will be running until ZF flag is zero (e.g., until compared values are equal to each other, hence "E" in REPE).

It works like memcmp() in C.

Example from Windows NT kernel ([WRK v1.2](#)):

Listing A.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 dwords, try ↗
; ↘ bytes
        repe     cmpsd              ; compare dwords
        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 32-bit words comparison (CMPSD) if block size is multiple of 4, or per-byte comparison (CMPSB) otherwise.

**CPUID** get information about [CPU](#) features. see also: [\(19.6.1\)](#).

**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 syscalls. Registers `AX/BX/CX/DX/SI/DI` were filled by arguments and jump to the address in the Interrupt Vector Table (located at the address space beginning) will be occurred. It was popular because `INT` has short opcode (2 bytes) and the program which needs some MS-DOS services is not bothering by determining service's entry point address. Interrupt handler return control flow to called using `IRET` instruction.

Most busy MS-DOS interrupt number was `0x21`, serving a huge amount of its [API](#). See also: [\[Bro\]](#) for the most comprehensive interrupt lists and other MS-DOS information.

In post-MS-DOS era, this instruction was still used as syscall both in Linux and Windows [\(51\)](#), but later replaced by `SYSENTER` or `SYSCALL` instruction.

**INT 3 (M)**: this instruction is somewhat standing aside of `INT`, it has its own 1-byte opcode (`0xCC`), and actively used while debugging. Often, debuggers just write `0xCC` byte at the address of breakpoint to be set, and when exception is raised, original byte will be restored and original instruction at this address will be re-executed.

As of [Windows NT](#), an `EXCEPTION_BREAKPOINT` exception will be raised when [CPU](#) executes this instruction. This debugging event may be intercepted and handled by a host debugger, if loaded. If it is not loaded, Windows will offer to run one of the registered in the system debuggers. If [MSVS<sup>5</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 are checking integrity of the code loaded.

[MSVC](#) has [compiler intrinsic](#) for the instruction: `__debugbreak()`<sup>6</sup>.

<sup>5</sup>Microsoft Visual Studio

<sup>6</sup><http://msdn.microsoft.com/en-us/library/f408b4et.aspx>



There are also a win32 function in kernel32.dll named `DebugBreak()`<sup>7</sup>, which also executes `INT 3`.

**IN** (M) input data from port. The instruction is usually can be seen in OS drivers or in old MS-DOS code, for example (63.3).

**IRET** : was used in MS-DOS environment for returning from interrupt handler after it was called by `INT` instruction. Equivalent to `POP tmp; POPF; JMP tmp`.

**LOOP** (M) **decrement** CX/ECX/RCX, jump if it is still not zero.

**OUT** (M) output data to port. The instruction is usually can be seen in OS drivers or in old MS-DOS code, for example (63.3).

**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 stack.

**POPCNT** population count. counts number of 1 bits in value. **AKA** “hamming weight”. **AKA** “NSA instruction” due of 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.

[Sch94]

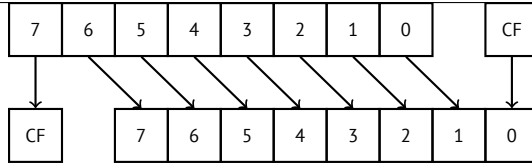
**POPF** restore flags from stack (**AKA** EFLAGS register)

**PUSHA** (M) pushes values of (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.

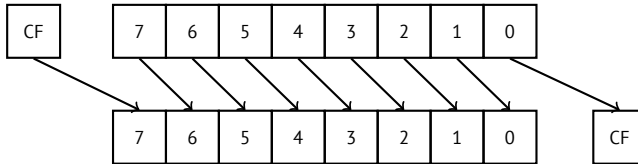
**PUSHF** push flags (**AKA** EFLAGS register)

**RCL** (M) rotate left via CF flag:

<sup>7</sup>[http://msdn.microsoft.com/en-us/library/windows/desktop/ms679297\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms679297(v=vs.85).aspx)

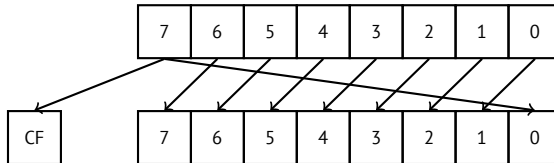


**RCR** (M) rotate right via CF flag:

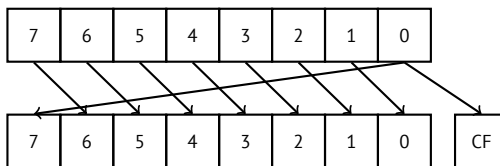


**ROL/ROR** (M) cyclic shift

ROL: rotate left:



ROR: rotate right:

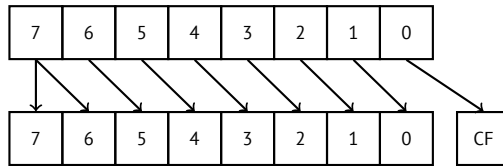


Despite the fact that almost all CPUs has these instructions, there are no corresponding operations in the C/C++, so the compilers of these PLs are usually not generating these instructions.

For programmer's convenience, at least MSVC has pseudofunctions (compiler intrinsics) `_rotl()` and `_rotr()`<sup>8</sup>, which are translated by compiler directly to these instructions.

**SAL** Arithmetic shift left, synonymous to SHL

<sup>8</sup><http://msdn.microsoft.com/en-us/library/5cc576c4.aspx>

**SAR** Arithmetic shift right

Hence, sign bit is always stayed at the place of **MSB**.

**SETcc** op: load 1 to op (byte only) if condition is true or zero otherwise. The condition codes are the same as in Jcc instructions ([A.6.2](#)).

**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 `ntoskrnl.exe` Windows kernel file only in hand-written memory copy routines.

**STI** (M) set IF flag

**SYSCALL** (AMD) call syscall ([51](#))

**SYSENTER** (Intel) call syscall ([51](#))

**UD2** (M) undefined instruction, raises exception. used for testing.

**A.6.4 FPU instructions**

-R in mnemonic usually means that operands are reversed, -P means that one element is popped from the stack after instruction execution, -PP means that two elements are popped.

-P instructions are often useful when we do not need a 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., summed values in the stack are replaced by sum

**FCHS**  $ST(0)=-ST(0)$

**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., 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 (A.3) 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 (A.3) into 16-bit op after checking for pending exceptions.

**FNSTCW** op: store FPU control word (A.3) into 16-bit op.

**FSTSW** op: store FPU status word (A.3.2) into 16-bit op after checking for pending exceptions.

**FNSTSW** op: store FPU status word (A.3.2) 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., summed values in the stack are replaced by 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., summed values in the stack are replaced by 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 performs just like FCOM, but exception is raised only if one of 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)

## A.6.5 SIMD instructions

## A.6.6 Instructions having printable ASCII opcode

(In 32-bit mode).

It can be suitable for shellcode constructing. See also: [67.1](#).

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
⌵	60	PUSHA
a	61	POPA
f	66	(in 32-bit mode) switch to 16-bit operand size
g	67	(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

Summarizing: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSHA, XOR.

# Appendix B

## ARM

### B.1 Terminology

ARM was initially developed as 32-bit [CPU](#), so that's why *word* here, unlike x86, is 32-bit.

**byte** 8-bit. `DB` assembly directive is used for defining variables and array of bytes.

**halfword** 16-bit. `DCW` assembly directive —”—.

**word** 32-bit. `DCD` assembly directive —”—.

**doubleword** 64-bit.

**quadword** 128-bit.

### B.2 Versions

- ARMv4: thumb mode appeared.
- ARMv6: used in iPhone 1st gen., iPhone 3G (Samsung 32-bit RISC ARM 1176JZ(F)-S supporting 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 are no thumb mode in 64-bit mode, only ARM (4-byte instructions).



## B.3 32-bit ARM (AArch32)

### B.3.1 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”: function arguments are usually passed in them, and values in them are not necessary to restore upon function exit.

### B.3.2 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

### B.3.3 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

S-registers are 32-bit ones, used for single precision numbers storage.  
D-registers are 64-bit ones, used for double precision numbers storage.

D- and S-registers share the same physical space in CPU—it is possible to access D-register via S-registers (it is senseless though).

Likewise, [NEON](#) Q-registers are 128-bit ones and share the same physical space in CPU with other floating point registers.

In VFP 32 S-registers are present: S0..S31.

In VFPv2 there are 16 D-registers added, which are, in fact, occupy the same space as S0..S31.

In VFPv3 ([NEON](#) or “Advanced SIMD”) there are 16 more D-registers added, resulting D0..D31, but D16..D31 registers are not sharing a space with other S-registers.

In [NEON](#) or “Advanced SIMD” there are also 16 128-bit Q-registers added, which share the same space as D0..D31.

## B.4 64-bit ARM (AArch64)

### B.4.1 General purpose registers

Register count 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, callee function may use it and not restore.
- X16
- X17
- X18
- X19...X29—callee function may use, but should restore them upon exit.
- X29—used as [FP](#) (at least GCC)
- X30—“Procedure Link Register” [AKA LR \(link register\)](#).
- X31—register always containing zero [AKA XZR](#) or “Zero Register”. It’s 32-bit part called WZR.
- [SP](#), not general register anymore.

See also: [[ARM13c](#)].

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

## B.5 Instructions

There is -S suffix for some instructions in ARM, indicating the instruction will set the flags according to the result, and without it –the flags will not be touched. For example ADD unlike ADDS will add two numbers, but flags will not be touched. Such instructions are convenient to use between CMP where flags are set and, e.g. conditional jumps, where flags are used.

## Appendix C

# Some GCC library functions

name	meaning
<code>__divdi3</code>	signed division
<code>__moddi3</code>	getting remainder (modulo) of signed division
<code>__udivdi3</code>	unsigned division
<code>__umoddi3</code>	getting remainder (modulo) of unsigned division

## Appendix D

# Some MSVC library functions

ll in function name mean “long long”, e.g., 64-bit data type.

name	meaning
<code>__alldiv</code>	signed division
<code>__allmul</code>	multiplication
<code>__allrem</code>	remainder of signed division
<code>__allshl</code>	shift left
<code>__allshr</code>	signed shift right
<code>__aulldiv</code>	unsigned division
<code>__aullrem</code>	remainder of unsigned division
<code>__aullshr</code>	unsigned shift right

Multiplication and shift left procedures are the same for both signed and unsigned numbers, hence only one function for each operation here.

The source code of these function can be founded in the installed [MSVS](#), in `VC/crt/src/`

# Appendix E

# Cheatsheets

## E.1 IDA

Short 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)
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>1</sup> for hiding some often used patterns of inline code.

## E.2 OllyDbg

Short hot-keys cheatsheet:

<sup>1</sup>[https://github.com/yurichev/IDA\\_scripts](https://github.com/yurichev/IDA_scripts)

hot-key	meaning
F7	trace into
F8	step over
F9	run
Ctrl-F2	restart

## E.3 MSVC

Some useful options I used through this book.

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 $n$ -byte boundary
/MD	produced executable will use MSVCR* .DLL

Some information about MSVC versions: [39.1](#).

## E.4 GCC

Some useful options I used through this book.

option	meaning
-Os	code size optimization
-O3	maximum optimization
-regparm= $n$	how many arguments will 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

## E.5 GDB

Some of commands I used in this book:



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 at address stored in RDI
x/10w ...	dump 10 memory words
x/s ...	dump memory as string
x/i ...	dump memory as code
x/10c ...	dump 10 characters
x/b ...	dump bytes
x/h ...	dump 16-bit halfwords
x/g ...	dump giant (64-bit) words
finish	execute till the end of function
next	next instruction (don't dive into functions)
step	next instruction (dive into functions)
frame n	switch stack frame
info break	list of breakpoints
del n	delete breakpoint

# Appendix F

## Exercise solutions

### F.1 Per chapter

#### F.1.1 “Stack” chapter

##### Exercise #1

Exercise: [4.5.1](#).

Non-optimizing MSVC, these numbers are: saved EBP value, [RA](#) and `argc`. It's easy to be assured in that by running the example with different number of arguments in command-line.

Optimizing MSVC, these numbers are: [RA](#), `argc` and a pointer to `argv[]` array.

GCC 4.8.x allocates 16-byte space in `main()` function prologue, hence different output numbers.

##### Exercise #2

Exercise: [4.5.2](#).

This code prints UNIX time.

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

int main()
{
    printf ("%d\n", time(NULL));
};
```

## F.1.2 “switch()/case/default” chapter

### F.1.3 Exercise #1

Exercise: [12.5.1](#).

Hint: `printf()` may be called only from the one single place.

### F.1.4 “Loops” chapter

#### F.1.5 Exercise #3

Exercise: [13.2.3](#).

```
#include <stdio.h>

int main()
{
    int i;
    for (i=100; i>0; i--)
        printf ("%d\n", i);
};
```

#### F.1.6 Exercise #4

Exercise: [13.2.4](#).

```
#include <stdio.h>

int main()
{
    int i;
    for (i=1; i<100; i=i+3)
        printf ("%d\n", i);
};
```

### F.1.7 “Simple C-strings processings” chapter

#### Exercise #1

Exercise: [14.3.1](#).

This is a function counting spaces in the input C-string.

```
int f(char *s)
{
    int rt=0;
    for (;*s;s++)
```

```
{
    if (*s==' ')
        rt++;
};
return rt;
};
```

## F.1.8 “Replacing arithmetic instructions to other ones” chapter

### Exercise #1

Exercise: [15.4.1](#).

```
int f(int a)
{
    return a/661;
};
```

### Exercise #2

Exercise: [15.4.2](#).

```
int f(int a)
{
    return a*7;
};
```

## F.1.9 “Floating-point unit” chapter

### Exercise #1

Exercise: [16.5.2](#).

Calculating arithmetic mean for 5 *double* values.

```
double f(double a1, double a2, double a3, double a4, double a5)
{
    return (a1+a2+a3+a4+a5) / 5;
};
```

## F.1.10 “Arrays” chapter

### Exercise #1

Exercise: [17.10.1](#).

Solution: two 100\*200 matrices of *double* type addition.

C/C++ source code:

```
#define M    100
#define N    200

void s(double *a, double *b, double *c)
{
    for(int i=0;i<N;i++)
        for(int j=0;j<M;j++)
            *(c+i*M+j)=*(a+i*M+j) + *(b+i*M+j);
};
```

## Exercise #2

Exercise: [17.10.2](#).

Solution: two matrices (one is 100\*200, second is 100\*300) of *double* type multiplication, result: 100\*300 matrix.

C/C++ source code:

```
#define M    100
#define N    200
#define P    300

void m(double *a, double *b, double *c)
{
    for(int i=0;i<M;i++)
        for(int j=0;j<P;j++)
        {
            *(c+i*M+j)=0;
            for (int k=0;k<N;k++) *(c+i*M+j)+=*(a+i*M+j) * *(b+i*M+j);
        }
};
```

## Exercise #3

Exercise: [17.10.3](#).

```
double f(double array[50][120], int x, int y)
{
    return array[x][y];
};
```

**Exercise #4**Exercise: [17.10.4](#).

```
int f(int array[50][60][80], int x, int y, int z)
{
    return array[x][y][z];
};
```

**Exercise #5**Exercise: [17.10.5](#).

This code just calculates multiplication table.

```
int tbl[10][10];

int main()
{
    int x, y;
    for (x=0; x<10; x++)
        for (y=0; y<10; y++)
            tbl[x][y]=x*y;
};
```

**F.1.11 “Working with specific bits” chapter****Exercise #1**Exercise: [18.8.1](#). This is a function which changes [endianness](#) in 32-bit value.

```
unsigned int f(unsigned int a)
{
    return ((a>>24)&0xff) | ((a<<8)&0xff0000) | ((a>>8)&0x
    ↵ 0xff00) | ((a<<24)&0xff000000);
};
```

Additional question: x86 instruction can do this. Which one?

**Exercise #2**Exercise: [18.8.2](#).This function converts [BCD](#)-packed 32-bit value into usual one.

```
#include <stdio.h>

unsigned int f(unsigned int a)
```

```

{
    int i=0;
    int j=1;
    unsigned int rt=0;
    for (;i<=28; i+=4, j*=10)
        rt+=((a>>i)&0xF) * j;
    return rt;
};

int main()
{
    // test
    printf ("%d\n", f(0x12345678));
    printf ("%d\n", f(0x1234567));
    printf ("%d\n", f(0x123456));
    printf ("%d\n", f(0x12345));
    printf ("%d\n", f(0x1234));
    printf ("%d\n", f(0x123));
    printf ("%d\n", f(0x12));
    printf ("%d\n", f(0x1));
};

```

### Exercise #3

Exercise: [18.8.3](#).

```

#include <windows.h>

int main()
{
    MessageBox(NULL, "hello, world!", "caption",
        MB_TOPMOST | MB_ICONINFORMATION | MB_HELP | ↵
        ↵ MB_YESNOCANCEL);
};

```

### Exercise #4

Exercise: [18.8.4](#).

This function just multiplies two 32-bit numbers, returning 64-bit [product](#). Well, this is a case when simple observing input/outputs may solve problem faster.

```

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

// source code taken from

```

```
// http://www4.wittenberg.edu/academics/mathcomp/shelburne/ ↵
↳ comp255/notes/binarymultiplication.pdf

uint64_t mult (uint32_t m, uint32_t n)
{
    uint64_t p = 0; // initialize product p to 0
    while (n != 0) // while multiplier n is not 0
    {
        if (n & 1) // test LSB of multiplier
            p = p + m; // if 1 then add multiplicand m
        m = m << 1; // left shift multiplicand
        n = n >> 1; // right shift multiplier
    }
    return p;
}

int main()
{
    printf ("%d\n", mult (2, 7));
    printf ("%d\n", mult (3, 11));
    printf ("%d\n", mult (4, 111));
};
```

## F.1.12 “Structures” chapter

### Exercise #1

Exercise: [19.7.1](#).

This program shows user ID of file owner.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    struct stat sb;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
        return 0;
    }

    if (stat(argv[1], &sb) == -1)
```



```

    {
        // error
        return 0;
    }

    printf("%ld\n", (long) sb.st_uid);
}

```

## Exercise #1

Exercise: [19.7.2](#).

Hint: you may get some information on how values are treated with JCC, MOVSX and MOVZX instructions.

```

#include <stdio.h>

struct some_struct
{
    int a;
    unsigned int b;
    float f;
    double d;
    char c;
    unsigned char uc;
};

void f(struct some_struct *s)
{
    if (s->a > 1000)
    {
        if (s->b > 10)
        {
            printf ("%f\n", s->f * 444 + s->d * ↵
↳ 123);
            printf ("%c, %d\n", s->c, s->uc);
        }
        else
        {
            printf ("error #2\n");
        }
    }
    else
    {
        printf ("error #1\n");
    }
};
};

```

## F.1.13 “Obfuscation” chapter

### Exercise #1

Exercise: 32.5.1.

Source code: [http://beginners.re/exercise-solutions/per\\_chapter/obfuscation.c](http://beginners.re/exercise-solutions/per_chapter/obfuscation.c).

## F.2 Level 1

### F.2.1 Exercise 1.1

That was a function returning maximal value from two.

### F.2.2 Exercise 1.4

Source code: <http://beginners.re/exercise-solutions/1/4/password1.c>

## F.3 Level 2

### F.3.1 Exercise 2.4

Solution: strstr().

C source code:

```
char * strstr (
    const char * str1,
    const char * str2
)
{
    char *cp = (char *) str1;
    char *s1, *s2;

    if ( !*str2 )
        return((char *)str1);

    while (*cp)
    {
        s1 = cp;
        s2 = (char *) str2;

        while ( *s1 && *s2 && !(*s1-*s2) )
            s1++, s2++;
```

```

        if (!*s2)
            return(cp);

        cp++;
    }

    return(NULL);
}

```

### F.3.2 Exercise 2.5

Hint #1: Keep in mind that `__v`—global variable.

Hint #2: The function is called in CRT startup code, before `main()` execution.

Solution: early Pentium CPU FDIV bug checking<sup>1</sup>.

C source code:

```

unsigned __v; // __v

enum e {
    PROB_P5_DIV = 0x0001
};

void f( void ) // __verify_pentium_fdiv_bug
{
    /*
     * Verify we have got the Pentium FDIV problem.
     * The volatiles are to scare the optimizer away.
     */
    volatile double    v1    = 4195835;
    volatile double    v2    = 3145727;

    if( (v1 - (v1/v2)*v2) > 1.0e-8 ) {
        __v |= PROB_P5_DIV;
    }
}

```

### F.3.3 Exercise 2.6

Hint: it might be helpful to google a constant used here.

Solution: TEA<sup>2</sup> encryption algorithm.

C source code (taken from [http://en.wikipedia.org/wiki/Tiny\\_Encryption\\_Algorithm](http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm)):

<sup>1</sup>[http://en.wikipedia.org/wiki/Pentium\\_FDIV\\_bug](http://en.wikipedia.org/wiki/Pentium_FDIV_bug)

<sup>2</sup>Tiny Encryption Algorithm

```

void f (unsigned int* v, unsigned int* k) {
    unsigned int v0=v[0], v1=v[1], sum=0, i;           /* set ↵
    ↵ up */
    unsigned int delta=0x9e3779b9;                   /* a key↵
    ↵ schedule constant */
    unsigned int k0=k[0], k1=k[1], k2=k[2], k3=k[3]; /* cache↵
    ↵ key */
    for (i=0; i < 32; i++) {                          /* basic ↵
    ↵ cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }  /* end cycle↵
    ↵ */
    v[0]=v0; v[1]=v1;
}

```

### F.3.4 Exercise 2.13

The cryptoalgorithm is linear feedback shift register<sup>3</sup>.

Source code: <http://beginners.re/exercise-solutions/2/13/LFSR.c>

### F.3.5 Exercise 2.14

This is algorithm of finding greater common divisor (GCD).

Source code: <http://beginners.re/exercise-solutions/2/14/GCD.c>

### F.3.6 Exercise 2.15

Pi value calculation using Monte-Carlo method.

Source code: <http://beginners.re/exercise-solutions/2/15/monte.c>

### F.3.7 Exercise 2.16

It is Ackermann function<sup>4</sup>.

```

int ack (int m, int n)
{

```

<sup>3</sup>[https://en.wikipedia.org/wiki/Linear\\_feedback\\_shift\\_register](https://en.wikipedia.org/wiki/Linear_feedback_shift_register)

<sup>4</sup>[https://en.wikipedia.org/wiki/Ackermann\\_function](https://en.wikipedia.org/wiki/Ackermann_function)

```
    if (m==0)
        return n+1;
    if (n==0)
        return ack (m-1, 1);
    return ack(m-1, ack (m, n-1));
};
```

### F.3.8 Exercise 2.17

This is 1D cellular automation working by *Rule 110*:

[https://en.wikipedia.org/wiki/Rule\\_110](https://en.wikipedia.org/wiki/Rule_110).

Source code: <http://beginners.re/exercise-solutions/2/17/CA.c>

### F.3.9 Exercise 2.18

Source code: <http://beginners.re/exercise-solutions/2/18/>

### F.3.10 Exercise 2.19

Source code: <http://beginners.re/exercise-solutions/2/19/>

## F.4 Level 3

### F.4.1 Exercise 3.2

Hint: easiest way is to find by values in the tables.

Commented C source code:

<http://beginners.re/exercise-solutions/3/2/gost.c>

### F.4.2 Exercise 3.3

Commented C source code:

<http://beginners.re/exercise-solutions/3/3/entropy.c>

### F.4.3 Exercise 3.4

Commented C source code, and also decrypted file: <http://beginners.re/exercise-solutions/3/4/>

## F.4.4 Exercise 3.5

Hint: as we can see, the string with user name occupies not the whole file.

Bytes after terminated zero till offset 0x7F are ignored by program.

Commented C source code:

[http://beginners.re/exercise-solutions/3/5/crc16\\_keyfile\\_check.c](http://beginners.re/exercise-solutions/3/5/crc16_keyfile_check.c)

## F.4.5 Exercise 3.6

Commented C source code:

<http://beginners.re/exercise-solutions/3/6/>

As another exercise, now you may try to fix all vulnerabilities you found in this web-server.

## F.4.6 Exercise 3.8

Commented C source code:

<http://beginners.re/exercise-solutions/3/8/>

# F.5 Other

## F.5.1 “Minesweeper (Windows XP)” example

Example: [61](#).

Hint: think about border bytes (0x10) pattern.

# **Acronyms used**

<b>OS</b> Operating System .....	7
<b>OOP</b> Object-Oriented Programming .....	526
<b>PL</b> Programming language .....	3
<b>PRNG</b> Pseudorandom number generator .....	421
<b>ROM</b> Read-only memory .....	655
<b>RA</b> Return Address .....	19
<b>PE</b> Portable Executable: <a href="#">53.2</a> .....	694
<b>SP</b> <a href="#">stack pointer</a> . SP/ESP/RSP in x86/x64. SP in ARM. ....	15
<b>DLL</b> Dynamic-link library .....	695
<b>PC</b> Program Counter. IP/EIP/RIP in x86/64. PC in ARM. ....	15
<b>LR</b> Link Register .....	15
<b>IDA</b> Interactive Disassembler .....	7
<b>IAT</b> Import Address Table .....	696
<b>INT</b> Import Name Table .....	696
<b>RVA</b> Relative Virtual Address .....	696
<b>VA</b> Virtual Address .....	695
<b>OEP</b> Original Entry Point .....	690



---

<b>MSVC</b> Microsoft Visual C++	
<b>MSVS</b> Microsoft Visual Studio	1027
<b>ASLR</b> Address Space Layout Randomization	696
<b>MFC</b> Microsoft Foundation Classes	700
<b>TLS</b> Thread Local Storage	v
<b>AKA</b> Also Known As	
<b>CRT</b> C runtime library: sec:CRT	7
<b>CPU</b> Central processing unit	v
<b>FPU</b> Floating-point unit	226
<b>CISC</b> Complex instruction set computing	16
<b>RISC</b> Reduced instruction set computing	4
<b>GUI</b> Graphical user interface	690
<b>RTTI</b> Run-time type information	547
<b>BSS</b> Block Started by Symbol	106
<b>SIMD</b> Single instruction, multiple data	446
<b>BSOD</b> Black Screen of Death	680
<b>DBMS</b> Database management systems	v

<b>ISA</b> Instruction Set Architecture .....	<b>3</b>
<b>CGI</b> Common Gateway Interface .....	<b>1000</b>
<b>HPC</b> High-Performance Computing .....	<b>504</b>
<b>SEH</b> Structured Exception Handling: <a href="#">53.3</a> .....	<b>33</b>
<b>ELF</b> Executable file format widely used in *NIX system including Linux .....	<b>v</b>
<b>TIB</b> Thread Information Block .....	<b>289</b>
<b>TEA</b> Tiny Encryption Algorithm .....	<b>1054</b>
<b>PIC</b> Position Independent Code: <a href="#">52.1</a> .....	<b>v</b>
<b>NAN</b> Not a Number .....	<b>1014</b>
<b>NOP</b> No Operation .....	<b>84</b>
<b>BEQ</b> (PowerPC, ARM) Branch if Equal .....	<b>87</b>
<b>BNE</b> (PowerPC, ARM) Branch if Not Equal .....	<b>194</b>
<b>BLR</b> (PowerPC) Branch to Link Register .....	<b>777</b>
<b>XOR</b> eXclusive OR .....	<b>1023</b>
<b>MCU</b> Microcontroller unit .....	<b>808</b>
<b>RAM</b> Random-access memory .....	<b>77</b>
<b>EGA</b> Enhanced Graphics Adapter .....	<b>962</b>

<b>VGA</b> Video Graphics Array .....	962
<b>API</b> Application programming interface .....	643
<b>ASCII</b> American Standard Code for Information Interchange .....	905
<b>ASCIIZ</b> ASCII Zero (null-terminated ASCII string) .....	83
<b>IA64</b> Intel Architecture 64 (Itanium): <a href="#">77</a> .....	633
<b>EPIC</b> Explicitly parallel instruction computing .....	958
<b>OOE</b> Out-of-order execution .....	635
<b>MSDN</b> Microsoft Developer Network .....	374
<b>MSB</b> Most significant bit/byte .....	360
<b>LSB</b> Least significant bit/byte	
<b>STL</b> (C++) Standard Template Library: <a href="#">31.4</a> .....	557
<b>PODT</b> (C++) Plain Old Data Type .....	575
<b>HDD</b> Hard disk drive .....	592
<b>VM</b> Virtual Memory	
<b>WRK</b> Windows Research Kernel .....	662
<b>GPR</b> General Purpose Registers .....	3
<b>SSDT</b> System Service Dispatch Table .....	680

<b>RE</b> Reverse Engineering .....	970
<b>BCD</b> Binary-coded decimal .....	903
<b>BOM</b> Byte order mark .....	650
<b>GDB</b> GNU debugger .....	45
<b>FP</b> Frame Pointer .....	21
<b>MBR</b> Master Boot Record .....	655
<b>JPE</b> Jump Parity Even (x86 instruction) .....	244
<b>CIDR</b> Classless Inter-Domain Routing .....	360
<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) .....	25
<b>LDMED</b> Load Multiple Empty Descending (ARM instruction) .....	25
<b>STMFA</b> Store Multiple Full Ascending (ARM instruction) .....	25
<b>LDMFA</b> Load Multiple Full Ascending (ARM instruction) .....	25
<b>STMEA</b> Store Multiple Empty Ascending (ARM instruction) .....	25
<b>LDMEA</b> Load Multiple Empty Ascending (ARM instruction) .....	25
<b>APSR</b> (ARM) Application Program Status Register .....	265

---

<b>FPSCR</b> (ARM) Floating-Point Status and Control Register .....	265
<b>PID</b> Program/process ID.....	762
<b>LF</b> Line feed (10 or '\n' in C/C++).....	197
<b>CR</b> Carriage return (13 or '\r' in C/C++) .....	197

# Glossary

- decrement** Decrease by 1. [15](#), [163](#), [189](#), [607](#), [666](#), [845](#), [1017](#), [1021](#), [1028](#)
- increment** Increase by 1. [16](#), [163](#), [167](#), [189](#), [193](#), [345](#), [607](#), [810](#), [839](#), [1016](#), [1017](#)
- integral data type** usual numbers, but not floating point ones. [240](#)
- product** Multiplication result. [89](#), [221](#), [229](#), [234](#), [447](#), [477](#), [1050](#)
- stack pointer** A register pointing to the place in the stack. [6](#), [8](#), [16](#), [25](#), [29](#), [40](#), [54](#), [56](#), [70](#), [91](#), [528](#), [614](#), [669](#), [671–673](#), [1010](#), [1018](#), [1036](#), [1059](#)
- tail call** It is when compiler (or interpreter) transforms recursion (with which it is possible: *tail recursion*) into iteration for efficiency: [http://en.wikipedia.org/wiki/Tail\\_call](http://en.wikipedia.org/wiki/Tail_call). [24](#)
- quotient** Division result. [217](#), [218](#), [228](#), [229](#), [234](#), [476](#), [768](#)
- anti-pattern** Generally considered as bad practice. [28](#), [71](#), [634](#)
- atomic operation** “ατομος” mean “indivisible” in Greek, so atomic operation is what guaranteed not to be broke up during operation by other threads. [739](#), [954](#)
- basic block** a group of instructions not having jump/branch instructions, and also not having jumps inside block from the outside. In IDA it looks just like as a list of instructions without breaking empty lines . [964](#), [966](#)
- callee** A function being called by another. [23](#), [28](#), [29](#), [42](#), [63](#), [78](#), [88](#), [91](#), [94](#), [138](#), [462](#), [528](#), [614](#), [669](#), [671–673](#), [676](#), [677](#)
- caller** A function calling another. [7](#), [42](#), [78](#), [88](#), [89](#), [93](#), [102](#), [138](#), [462](#), [485](#), [528](#), [669](#), [670](#), [672](#), [673](#)
- compiler intrinsic** A function specific to a compiler which is not usual library function. Compiler generate a specific machine code instead of call to it. It is often a pseudofunction for specific CPU instruction. Read more: ([74](#)). [1027](#)

**CP/M** Control Program for Microcomputers: a very basic disk OS used before MS-DOS. [900](#)

**dongle** Dongle is a small piece of hardware connected to LPT printer port (in past) or to USB. Its function was akin to security token, it has some memory and, sometimes, secret (crypto-)hashing algorithm. [776](#)

**endianness** Byte order: [36](#), [19](#), [73](#), [379](#), [1023](#), [1049](#)

**GiB** Gibibyte:  $2^{30}$  or 1024 mebibytes or 1073741824 bytes. [12](#)

**heap** usually, a big chunk of memory provided by OS so that applications can divide it by themselves as they wish. malloc()/free() works with heap. [26](#), [29](#), [382](#), [552](#), [555](#), [556](#), [575](#), [577](#), [694](#), [695](#)

**jump offset** a part of JMP or Jcc instruction opcode, it just to be added to the address of the next instruction, and thus is how new PC is calculated. May be negative as well. [84](#), [121](#), [1017](#)

**kernel mode** A restrictions-free CPU mode in which it executes OS kernel and drivers. cf. [user mode](#). [1067](#)

**keygenme** A program which imitates fictional software protection, for which one needs to make a keys/licenses generator. [1002](#)

**leaf function** A function which is not calling any other function. [28](#)

**link register** (RISC) A register where return address is usually stored. This makes calling leaf functions without stack usage, i.e., faster. [28](#), [778](#), [1036](#), [1037](#)

**loop unwinding** It is when a compiler instead of generation loop code of  $n$  iteration, generates just  $n$  copies of the loop body, in order to get rid of loop maintenance instructions. [166](#)

**name mangling** used at least in C++, where compiler need to encode name of class, method and argument types in the one string, which will become internal name of the function. read more here: [31.1.1](#). [526](#), [638](#), [639](#)

**NaN** not a number: special cases of floating point numbers, usually signaling about errors . [243](#), [257](#), [961](#)

**NEON** AKA “Advanced SIMD”—SIMD from ARM. [1037](#)

**NOP** “no operation”, idle instruction. [666](#)

- NTAPI** API available only in Windows NT line. Largely, not documented by Microsoft. [748](#)
- PDB** (Win32) Debugging information file, usually just function names, but sometimes also function arguments and local variables names. [637](#), [698](#), [748](#), [750](#), [757](#), [758](#), [763](#), [871](#), [872](#)
- POKE** BASIC language instruction writing byte on specific address. [666](#)
- register allocator** Compiler's function assigning local variables to CPU registers. [188](#), [331](#), [462](#)
- reverse engineering** act of understanding, how the thing works, sometimes, in order to clone it. [v](#), [1027](#)
- security cookie** A random value, different at each execution. Read more about it: [17.3](#). [725](#)
- stack frame** Part of stack containing information specific to the current functions: local variables, function arguments, [RA](#), etc. [65](#), [89](#), [90](#), [495](#), [496](#), [725](#)
- stdout** standard output. [18](#), [30](#), [138](#), [998](#)
- thunk function** Tiny function with a single role: call another function. [20](#), [432](#), [777](#), [790](#)
- tracer** My own simple debugging tool. Read more about it: [55.1](#). [168–170](#), [645](#), [658](#), [663](#), [664](#), [719](#), [732](#), [875](#), [885](#), [891](#), [892](#), [896](#), [945](#), [1000](#)
- user mode** A restricted CPU mode in which it executes all applied software code. cf. [kernel mode](#). [802](#), [1066](#)
- Windows NT** Windows NT, 2000, XP, Vista, 7, 8. [300](#), [457](#), [612](#), [650](#), [680](#), [696](#), [738](#), [905](#), [1027](#)
- word** data type fitting in [GPR](#). In the computers older than personal, memory size was often measured in words rather than bytes. [560](#)
- xoring** often used in English language, meaning applying [XOR](#) operation. [725](#), [794](#), [799](#)



# Index

- .NET, 703
- AT&T syntax, 10, 32
- Buffer Overflow, 278, 725
- C language elements
  - Pointers, 63, 70, 105, 424, 461
  - Post-decrement, 607
  - Post-increment, 607
  - Pre-decrement, 607
  - Pre-increment, 607
- C99, 103
  - bool, 328
  - restrict, 501
  - variable length arrays, 292
- const, 6, 77
- for, 163, 358
- if, 116, 137
- return, 7, 79, 102
- switch, 135, 137, 142
- while, 186
- C standard library
  - alloca(), 30, 292, 634, 712
  - assert(), 298, 653
  - atexit(), 559
  - atoi(), 856
  - calloc(), 829
  - close(), 686
  - exit(), 485
  - free(), 634
  - fseek(), 828
  - ftell(), 828
  - getenv(), 858
  - localtime(), 626
  - localtime\_r(), 389
  - longjmp(), 138
  - malloc(), 383, 634
  - memchr(), 1021
  - memcmp(), 514, 656, 1024
  - memcpy(), 10, 64, 510, 1019
  - memset(), 891, 1022
  - open(), 686
  - pow(), 237
  - puts(), 18
  - qsort(), 424
  - rand(), 643, 753, 757, 808
  - read(), 686
  - realloc(), 634
  - scanf(), 63
  - strcmp(), 506, 686
  - strcpy(), 10, 510, 809
  - strlen(), 186, 199, 455, 509, 1021
  - strstr(), 1053
  - time(), 626
  - tolower(), 837
- Compiler's anomalies, 176, 207, 312, 339, 352, 947
- C++, 877
  - exceptions, 712
  - C++11, 575, 679
  - ostream, 548
  - References, 549
  - RTTI, 547
  - STL, 637
    - std::forward\_list, 574
    - std::list, 560
    - std::map, 585
    - std::set, 585

- std::string, 550
- std::vector, 575
- grep usage, 170, 267, 637, 658, 663, 873
- Intel syntax, 10, 14
- Mac OS X, 744
- position-independent code, 15, 682
- RAM, 76
- ROM, 76, 77
- Base address, 695
- Global variables, 71
- Binary tree, 585
- Dynamically loaded libraries, 19
- Linker, 77, 526
- RISC pipeline, 126
- Non-a-numbers (NaNs), 257
- OOP
  - Polymorphism, 526
- Buffer overflow, 283
- Hash functions, 764
- Recursion, 23, 27, 500
  - Tail recursion, 23
- Stack, 25, 88, 138
  - Stack overflow, 27
  - Stack frame, 65
- Syntactic Sugar, 137, 391
- thumb-2 mode, 19
- iPod/iPhone/iPad, 14
- OllyDbg, 40, 66, 73, 89, 106, 119, 148, 167, 189, 229, 243, 251, 274, 279, 282, 301, 302, 345, 379, 401, 402, 406, 410, 428, 699, 743, 1042
- Oracle RDBMS, 7, 447, 652, 708, 878, 890, 893, 924, 934, 947, 964
- Angry Birds, 266, 267
- ARM, 193, 520, 777, 1035
  - ARM mode, 4
  - Instructions
    - ADD, 18, 99, 125, 171, 220, 353, 1038
    - ADDAL, 125
    - ADDCC, 153
    - ADDS, 97, 1038
    - ADR, 15, 125
    - ADRcc, 125, 126, 142, 635
    - ADRP/ADD pair, 21, 55, 297, 313, 610
    - ASR, 368
    - ASRS, 220, 339
    - B, 54, 125, 127
    - BCS, 127, 268
    - BEQ, 87, 142
    - BGE, 127
    - BIC, 339, 340, 371
    - BL, 15, 17, 19–21, 125
    - BLcc, 125, 126
    - BLE, 127
    - BLS, 127
    - BLT, 171
    - BLX, 19
    - BNE, 127
    - BX, 96, 155
    - CLZ, 984, 986
    - CMP, 87, 125, 142, 153, 171, 353, 1038
    - CSEL, 132, 354
    - FCMPE, 269
    - FCSEL, 269
    - FMOV, 609
    - IDIV, 218
    - IT, 134, 265, 291
    - LDMccFD, 126
    - LDMEA, 25
    - LDMED, 25
    - LDMFA, 25
    - LDMFD, 16, 25, 126
    - LDP, 22
    - LDR, 57, 70, 77, 277, 295, 607
    - LDR.W, 309
    - LDRB, 403
    - LDRB.W, 194
    - LDRSB, 193
    - LSL, 353, 368
    - LSL.W, 353
    - LSLS, 278
    - LSR, 368

- MADD, 97
- MLA, 96, 97
- MOV, 16, 18, 220, 353
- MOVK, 609
- MOVT, 18, 220
- MOVT.W, 19
- MOVW, 19
- MUL, 99
- MULS, 97
- MVNS, 194
- ORR, 339
- POP, 15–17, 25, 28
- PUSH, 17, 25, 28
- RET, 22
- RSB, 309, 353
- SMMUL, 220
- STMEA, 25
- STMED, 25
- STMFA, 25, 59
- STMFD, 15, 25
- STMIA, 57
- STMIB, 59
- STP, 20, 55
- STR, 56, 277
- SUB, 56, 309, 353
- SUBEQ, 195
- SXTB, 404
- SXTW, 313
- TEST, 188
- TST, 333, 353
- VADD, 234
- VDIV, 234
- VLDR, 234
- VMOV, 234, 265
- VMOVGT, 265
- VMRS, 265
- VMUL, 234
- Pipeline, 153
- Mode switching, 96, 155
- Addressing modes, 607
- mode switching, 19
- Registers
  - APSR, 265
  - FPSCR, 265
  - Link Register, 15, 16, 28, 54, 155, 1036
  - R0, 100, 1036
  - scratch registers, 194, 1036
  - X0, 1037
  - Z, 87, 1036
- thumb mode, 4, 127, 155
- thumb-2 mode, 4, 155, 265, 267
- armel, 235
- armhf, 235
- Condition codes, 125
- D-registers, 234, 1036
- Data processing instructions, 220
- DCB, 16
- hard float, 235
- if-then block, 265
- Leaf function, 28
- Optional operators
  - ASR, 220, 353
  - LSL, 277, 309, 353, 609
  - LSR, 220, 353
  - ROR, 353
  - RRX, 353
- S-registers, 234, 1036
- soft float, 235
- ASLR, 696
- AWK, 661
- bash, 101
- BASIC
  - POKE, 666
- binary grep, 657, 746
- BIND.EXE, 702
- Bitcoin, 949
- Borland C++Builder, 639
- Borland Delphi, 639, 648, 944
- BSoD, 680
- BSS, 697
- C11, 679
- Callbacks, 424
- Canary, 286
- cdecl, 40, 669

- COFF, [787](#)
- column-major order, [301](#)
- Compiler intrinsic, [31](#), [946](#)
- CRC32, [355](#), [764](#)
- CRT, [690](#), [719](#)
- Cygnwin, [639](#), [645](#), [704](#), [744](#)
  
- DES, [446](#), [462](#)
- dlopen(), [686](#)
- dlsym(), [686](#)
- DOSBox, [905](#)
- DosBox, [664](#)
- double, [227](#), [677](#)
- dtruss, [744](#)
  
- EICAR, [899](#)
- ELF, [74](#)
- Error messages, [652](#)
  
- fastcall, [11](#), [62](#), [330](#), [671](#)
- float, [227](#), [677](#)
- FORTRAN, [301](#), [313](#), [501](#), [639](#)
- Function epilogue, [23](#), [54](#), [57](#), [126](#), [403](#), [661](#)
- Function prologue, [8](#), [23](#), [28](#), [56](#), [286](#), [661](#)
- Fused multiply-add, [96](#), [97](#)
  
- GCC, [638](#), [1039](#), [1043](#)
- GDB, [45](#), [50](#), [285](#), [433](#), [434](#), [743](#), [1043](#)
- Glibc, [433](#)
  
- Hex-Rays, [765](#)
- Hiew, [83](#), [121](#), [647](#), [698](#), [699](#), [704](#), [945](#)
  
- IDA, [79](#), [515](#), [651](#), [929](#), [1041](#)
  - var\_?, [57](#), [70](#)
- IEEE 754, [226](#), [421](#), [472](#), [1006](#)
- Inline code, [172](#), [339](#), [505](#), [534](#), [580](#)
- Intel
  - 8080, [193](#)
  - 8086, [193](#), [338](#), [802](#)
    - Memory model, [623](#), [962](#)
  - 8253, [903](#)
  - 80286, [802](#), [963](#)
  - 80386, [338](#), [963](#)
  - 80486, [226](#)
  - FPU, [226](#)
  - Intel C++, [7](#), [447](#), [947](#), [964](#), [1019](#)
  - Itanium, [958](#)
  
  - jumpable, [147](#), [155](#)
  
  - Keil, [14](#)
  - kernel panic, [680](#)
  - kernel space, [680](#)
  
  - LD\_PRELOAD, [686](#)
  - Linux, [332](#), [682](#), [878](#)
    - libc.so.6, [330](#), [432](#)
  - LLVM, [14](#)
  - long double, [227](#)
  - Loop unwinding, [166](#)
  
  - Mac OS Classic, [776](#)
  - MD5, [655](#), [764](#)
  - MFC, [700](#), [858](#)
  - MIDI, [656](#)
  - MinGW, [639](#)
  - MIPS, [522](#), [697](#), [777](#)
  - MS-DOS, [288](#), [619](#), [656](#), [664](#), [666](#), [695](#), [802](#), [899](#), [901](#), [919](#), [944](#), [962](#), [1006](#), [1020](#), [1027](#), [1028](#)
    - DOS extenders, [963](#)
  - MSVC, [1040](#), [1043](#)
  
  - Name mangling, [526](#)
  - NEC V20, [904](#)
  
  - objdump, [685](#), [704](#)
  - OEP, [695](#), [703](#)
  - opaque predicate, [604](#)
  - OpenMP, [642](#), [949](#)
  - OpenWatcom, [639](#), [673](#), [978](#)
  
  - Page (memory), [457](#)
  - Pascal, [648](#)
  - PDP-11, [607](#)
  - PowerPC, [776](#)
  - puts() instead of printf(), [18](#), [68](#), [101](#), [123](#)

- Raspberry Pi, 14
- ReactOS, 716
- Register allocation, 462
- Relocation, 19
- row-major order, 301
- RVA, 696
  
- SAP, 637, 871
- SCO OpenServer, 787
- Scratch space, 675
- Security cookie, 286, 725
- SHA1, 764
- SHA512, 949
- Shadow space, 93, 94, 474
- Shellcode, 603, 680, 696, 900, 1032
- Signed numbers, 118, 630
- SIMD, 472, 514
- SSE, 472
- SSE2, 472
- stdcall, 669, 944
- strace, 686, 744
- syscall, 330, 680, 744
  
- TCP/IP, 633
- thiscall, 525, 528, 673
- thunk-functions, 20, 702, 777, 790
- TLS, 289, 679, 697, 703, 1011
  - Callbacks, 703
- tracer, 168, 429, 430, 645, 658, 663, 719, 732, 743, 875, 885, 891, 892, 895, 945, 1000
  
- Unicode, 648
- Unrolled loop, 172, 291, 511
- uptime, 686
- USB, 779
- user space, 680
- UTF-16LE, 648, 650
- UTF-8, 648
  
- VA, 695
  
- Watcom, 639
- Windows, 738
  - API, 1006
  - IAT, 696
  - INT, 696
  - KERNEL32.DLL, 329
  - MSVCR80.DLL, 427
  - NTAPI, 748
  - ntoskrnl.exe, 878
  - PDB, 637, 698, 748, 757, 871
  - Structured Exception Handling, 33, 704
  - TIB, 289, 704, 1011
  - Win32, 328, 650, 685, 695, 963
    - GetProcAddress, 703
    - LoadLibrary, 703
    - Ordinal, 700
    - RaiseException(), 704
    - SetUnhandledExceptionFilter(), 708
  - Windows 2000, 697
  - Windows 3.x, 612, 963
  - Windows NT4, 697
  - Windows Vista, 695, 748
  - Windows XP, 697, 703, 757
- Wine, 716
- Wolfram Mathematica, 223, 224, 767
  
- x86
  - Flags
    - CF, 1016, 1017, 1021, 1024, 1028, 1030
    - DF, 1024, 1030
    - IF, 1024, 1030
  - Instructions
    - AAA, 1034
    - AAS, 1034
    - ADC, 439, 619, 1016
    - ADD, 6, 40, 89, 619, 1017
    - ADDSD, 473
    - ADDSS, 483
    - ADRCc, 131
    - AND, 8, 329, 335, 344, 369, 410, 1017, 1023
    - BSF, 459, 986, 1023
    - BSR, 1023
    - BSWAP, 633, 1023
    - BT, 1023

BTC, 1023  
BTR, 739, 1023  
BTS, 1023  
CALL, 6, 26, 517, 701, 1017  
CBW, 631, 1023  
CDQ, 444, 631, 1023  
CDQE, 631, 1023  
CLD, 1024  
CLI, 1024  
CMC, 1024  
CMOVcc, 126, 131, 133, 635, 1024  
CMP, 78, 79, 1017, 1034  
CMPSB, 656, 1024  
CMPSD, 1024  
CMPSQ, 1024  
CMPSW, 1024  
COMISD, 481  
COMISS, 483  
CPLD, 407, 1027  
CWD, 619, 631, 914, 1023  
CWDE, 631, 1023  
DEC, 189, 1017, 1034  
DIV, 631, 1027  
DIVSD, 473, 660  
FABS, 1030  
FADD, 1030  
FADDP, 229, 233, 1030  
FATRET, 350, 351  
FCHS, 1030  
FCMOVcc, 260  
FCOM, 250, 257, 1030  
FCOMP, 242, 1030  
FCOMPP, 1030  
FDIV, 228, 658, 659, 1031, 1054  
FDIVP, 228, 1031  
FDIVR, 233, 1031  
FDIVRP, 1031  
FILD, 1031  
FIST, 1031  
FISTP, 1031  
FLD, 238, 242, 1031  
FLD1, 1031  
FLDCW, 1031  
FLDZ, 1031  
FMUL, 229, 1031  
FMULP, 1031  
FNSTCW, 1031  
FNSTSW, 242, 258, 1031  
FSINCOS, 1031  
FSQRT, 1031  
FST, 1032  
FSTCW, 1031  
FSTP, 238, 1032  
FSTSW, 1031  
FSUB, 1032  
FSUBP, 1032  
FSUBR, 1032  
FSUBRP, 1032  
FUCOM, 257, 1032  
FUCOMI, 260  
FUCOMP, 1032  
FUCOMPP, 257, 1032  
FWAIT, 226  
FXCH, 1032  
IDIV, 631, 1027  
IMUL, 89, 311, 631, 1017, 1034  
IN, 517, 802, 903, 1028  
INC, 189, 945, 1017, 1034  
INT, 900, 1027  
INT3, 645  
IRET, 1027, 1028  
JA, 118, 259, 630, 1017, 1034  
JAE, 118, 1017, 1034  
JB, 118, 630, 1017, 1034  
JBE, 118, 1017, 1034  
JC, 1017  
JCXZ, 1017  
JE, 137, 1017, 1034  
JECXZ, 1017  
JG, 118, 630, 1017  
JGE, 117, 1017  
JL, 118, 630, 1017  
JLE, 117, 1017  
JMP, 26, 54, 702, 944, 1017  
JNA, 1017  
JNAE, 1017

JNB, 1017  
 JNBE, 258, 1017  
 JNC, 1017  
 JNE, 78, 79, 117, 1017, 1034  
 JNG, 1017  
 JNGE, 1017  
 JNL, 1017  
 JNLE, 1017  
 JNO, 1017, 1034  
 JNS, 1017, 1034  
 JNZ, 1017  
 JO, 1017, 1034  
 JP, 243, 904, 1017, 1034  
 JPO, 1017  
 JRCXZ, 1017  
 JS, 1017, 1034  
 JZ, 87, 137, 947, 1017  
 LAHF, 1018  
 LEA, 65, 92, 360, 386, 487, 676,  
     752, 1018  
 LEAVE, 8, 1018  
 LES, 809, 913  
 LOCK, 738  
 LODSB, 904  
 LOOP, 163, 181, 182, 661, 914,  
     1028  
 MAXSD, 482  
 MOV, 7, 10, 510, 517, 699, 945,  
     1020  
 MOVDQA, 451  
 MOVDQU, 451  
 MOVSB, 1019  
 MOVSD, 480, 512, 835, 1019  
 MOVSDX, 480  
 MOVSQ, 1019  
 MOVSS, 483  
 MOVSW, 1019  
 MOVSX, 187, 193, 402–404, 631,  
     1020  
 MOVSDX, 293  
 MOVZX, 188, 383, 777, 1020  
 MUL, 631, 1020  
 MULSD, 473  
 NEG, 1020  
 NOP, 360, 941, 944, 1020  
 NOT, 192, 194, 842, 1020  
 OR, 199, 335, 1020  
 OUT, 517, 802, 1028  
 PADDD, 451  
 PCMPEQB, 458  
 PLMULHW, 447  
 PLMULLD, 447  
 PMOVMASKB, 459  
 POP, 7, 25, 27, 1020, 1034  
 POPA, 1028, 1034  
 POPCNT, 1028  
 POPF, 903, 1028  
 PUSH, 6, 8, 25, 26, 65, 517, 1020,  
     1034  
 PUSHA, 1028, 1034  
 PUSHF, 1028  
 PXOR, 458  
 RCL, 661, 1028  
 RCR, 1028  
 RET, 7, 27, 286, 528, 614, 944,  
     1020  
 ROL, 351, 946, 1029  
 ROR, 946, 1029  
 SAHF, 257, 1021  
 SAL, 1029  
 SALC, 904  
 SAR, 368, 631, 914, 1029  
 SBB, 439, 1021  
 SCASB, 904, 1021  
 SCASD, 1021  
 SCASQ, 1021  
 SCASW, 1021  
 SETALC, 904  
 SETcc, 189, 258, 1030  
 SHL, 211, 274, 368, 1022  
 SHR, 216, 368, 410, 1022  
 SHRD, 443, 1022  
 STC, 1030  
 STD, 1030  
 STI, 1030  
 STOSB, 1022

STOSD, [1022](#)  
STOSQ, [1022](#)  
STOSW, [1022](#)  
SUB, [7](#), [8](#), [79](#), [137](#), [1017](#), [1023](#)  
SYSCALL, [1027](#), [1030](#)  
SYSENTER, [681](#), [1027](#), [1030](#)  
TEST, [187](#), [329](#), [333](#), [369](#), [1023](#)  
UD2, [1030](#)  
XADD, [740](#)  
XCHG, [1020](#), [1023](#)  
XOR, [7](#), [79](#), [192](#), [661](#), [794](#), [945](#),  
[1023](#), [1034](#)

Prefixes

LOCK, [739](#), [1016](#)  
REP, [1016](#), [1019](#), [1022](#)  
REPE/REPNE, [1016](#)  
REPNE, [1021](#)

Registers

Flags, [79](#), [119](#), [1011](#)  
AH, [1018](#), [1021](#)  
CS, [962](#)  
DR6, [1014](#)  
DR7, [1015](#)  
DS, [962](#)  
EAX, [78](#), [100](#)  
EBP, [65](#), [89](#)  
ECX, [525](#)  
ES, [913](#), [962](#)  
ESP, [40](#), [65](#)  
JMP, [151](#)  
RIP, [685](#)  
SS, [962](#)  
ZF, [79](#), [329](#)

AVX, [446](#)  
FPU, [1012](#)  
MMX, [446](#)  
SSE, [446](#)  
SSE2, [446](#)

x86-64, [11](#), [48](#), [63](#), [68](#), [85](#), [91](#), [461](#), [472](#),  
[519](#), [673](#), [685](#), [1007](#), [1014](#)

Xcode, [14](#)

Z3, [764](#), [770](#)



# Bibliography

- [al12] Nick Montfort et al. *10 PRINT CHR\$(205.5+RND(1)); : GOTO 10*. Also available as [http://trope-tank.mit.edu/10\\_PRINT\\_121114.pdf](http://trope-tank.mit.edu/10_PRINT_121114.pdf). The MIT Press, 2012.
- [AMD13a] AMD. *AMD64 Architecture Programmer's Manual*. Also available as <http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/>. 2013.
- [AMD13b] AMD. *Software Optimization Guide for AMD Family 16h Processors*. Also available as [http://yurichev.com/mirrors/AMD/SOG\\_16h\\_52128\\_PUB\\_Rev1\\_1.pdf](http://yurichev.com/mirrors/AMD/SOG_16h_52128_PUB_Rev1_1.pdf). 2013.
- [App10] Apple. *iOSABI Function Call Guide*. Also available as <http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf>. 2010.
- [ARM12] ARM. *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. 2012.
- [ARM13a] ARM. *ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*. 2013.
- [ARM13b] ARM. *ELF for the ARM 64-bit Architecture (AArch64)*. Also available as [http://infocenter.arm.com/help/topic/com.arm.doc.ihl0056b/IHI0056B\\_aaelf64.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ihl0056b/IHI0056B_aaelf64.pdf). 2013.
- [ARM13c] ARM. *Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*. Also available as [http://infocenter.arm.com/help/topic/com.arm.doc.ihl0055b/IHI0055B\\_aapcs64.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ihl0055b/IHI0055B_aapcs64.pdf). 2013.
- [ble02] blexim. "Basic Integer Overflows". In: *Phrack* (2002). Also available as <http://yurichev.com/mirrors/phrack/p60-0x0a.txt>.
- [Bro] Ralf Brown. *The x86 Interrupt List*. Also available as <http://www.cs.cmu.edu/~ralf/files.html>.
- [Bur] Mike Burrell. "Writing Efficient Itanium 2 Assembly Code". In: (). Also available as <http://yurichev.com/mirrors/RE/itanium.pdf>.

- [Cli] Marshall Cline. C++ FAQ. Also available as <http://www.parashift.com/c++-faq-lite/index.html>.
- [Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.
- [Dij68] Edsger W. Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Commun. ACM* 11.3 (Mar. 1968), pp. 147–148. ISSN: 0001-0782. DOI: [10.1145/362929.362947](https://doi.org/10.1145/362929.362947). URL: <http://doi.acm.org/10.1145/362929.362947>.
- [Dol13] Stephen Dolan. "mov is Turing-complete". In: (2013). Also available as <http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf>.
- [Fog13a] Agner Fog. *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms*. [http://agner.org/optimize/optimizing\\_cpp.pdf](http://agner.org/optimize/optimizing_cpp.pdf). 2013.
- [Fog13b] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs / An optimization guide for assembly programmers and compiler makers*. <http://agner.org/optimize/microarchitecture.pdf>. 2013.
- [Fog14] Agner Fog. *Calling conventions*. [http://www.agner.org/optimize/calling\\_conventions.pdf](http://www.agner.org/optimize/calling_conventions.pdf). 2014.
- [IBM00] IBM. *PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*. Also available as [http://yurichev.com/mirrors/PowerPC/6xx\\_pem.pdf](http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf). 2000.
- [Int13] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C*. Also available as <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>. 2013.
- [ISO07] ISO. *ISO/IEC 9899:TC3 (C C99 standard)*. Also available as <http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf>. 2007.
- [ISO13] ISO. *ISO/IEC 14882:2011 (C++ 11 standard)*. Also available as <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>. 2013.
- [Ker88] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.
- [Knu74] Donald E. Knuth. "Structured Programming with go to Statements". In: *ACM Comput. Surv.* 6.4 (Dec. 1974). Also available as <http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf>, pp. 261–301. ISSN: 0360-0300. DOI: [10.1145/356635.356640](https://doi.org/10.1145/356635.356640). URL: <http://doi.acm.org/10.1145/356635.356640>.

- [Knu98] Donald E. Knuth. *The Art of Computer Programming Volumes 1-3 Boxed Set*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201485419.
- [Loh10] Eugene Loh. "The Ideal HPC Programming Language". In: *Queue* 8.6 (June 2010), 30:30–30:38. ISSN: 1542-7730. DOI: [10.1145/1810226.1820518](https://doi.acm.org/10.1145/1810226.1820518). URL: <http://doi.acm.org/10.1145/1810226.1820518>.
- [Ltd94] Advanced RISC Machines Ltd. *The ARM Cookbook*. Also available as [http://yurichev.com/ref/ARM%20Cookbook%20\(1994\).1994](http://yurichev.com/ref/ARM%20Cookbook%20(1994).1994).
- [Mit13] Michael Matz / Jan Hubicka / Andreas Jaeger / Mark Mitchell. *System V Application Binary Interface. AMD64 Architecture Processor Supplement*. Also available as <http://x86-64.org/documentation/abi.pdf>. 2013.
- [One96] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack* (1996). Also available as <http://yurichev.com/mirrors/phrack/p49-0x0e.txt>.
- [Pie] Matt Pietrek. "A Crash Course on the Depths of Win32™ Structured Exception Handling". In: *MSDN magazine* (). URL: <http://www.microsoft.com/msj/0197/Exception/Exception.aspx>.
- [Pie02] Matt Pietrek. "An In-Depth Look into the Win32 Portable Executable File Format". In: *MSDN magazine* (2002). URL: <http://msdn.microsoft.com/en-us/magazine/bb985992.aspx>.
- [RA09] Mark E. Russinovich and David A. Solomon with Alex Ionescu. *Windows® Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. 2009.
- [Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Also available as <http://catb.org/esr/writings/taoup/html/>. Pearson Education, 2003. ISBN: 0131429019.
- [Rit79] Dennis M. Ritchie. "The Evolution of the Unix Time-sharing System". In: (1979).
- [Rit86] Dennis M. Ritchie. *Where did ++ come from? (net.lang.c)*. [http://yurichev.com/mirrors/C/c\\_dmr\\_postincrement.txt](http://yurichev.com/mirrors/C/c_dmr_postincrement.txt). [Online; accessed 2013]. 1986.
- [Rit93] Dennis M. Ritchie. "The development of the C language". In: *SIGPLAN Not.* 28.3 (Mar. 1993). Also available as <http://yurichev.com/mirrors/C/dmr-The%20Development%20of%20the%20C%20Language-1993.pdf>, pp. 201–208. ISSN: 0362-1340. DOI: [10.1145/155360.155580](https://doi.acm.org/10.1145/155360.155580). URL: <http://doi.acm.org/10.1145/155360.155580>.

- [RT74] D. M. Ritchie and K. Thompson. "The UNIX Time Sharing System". In: (1974). Also available as <http://dl.acm.org/citation.cfm?id=361061>.
- [Sch94] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 1994.
- [SK95] SunSoft Steve Zucker and IBM Kari Karhi. *SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement*. Also available as [http://yurichev.com/mirrors/PowerPC/elfspec\\_ppc.pdf](http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf). 1995.
- [Sko12] Igor Skochinsky. *Compiler Internals: Exceptions and RTTI*. Also available as <http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf>. 2012.
- [Str13] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. 2013.
- [War02] Henry S. Warren. *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201914654.
- [Yur12] Dennis Yurichev. "Finding unknown algorithm using only input/output pairs and Z3 SMT solver". In: (2012). Also available as [http://yurichev.com/writings/z3\\_rockey.pdf](http://yurichev.com/writings/z3_rockey.pdf).
- [Yur13] Dennis Yurichev. *C/C++ programming language notes*. Also available as <http://yurichev.com/writings/C-notes-en.pdf>. 2013.