Igor’s tip of the week

season three

from 20/08/2021 to 26/08/2022

hex-rays

Igor’s Tip Season 3 has arrived! Fuelled by the triumph of the previous seasons, we embark on a mission to showcase the full extent of IDA’s capabilities. In keeping with tradition, Igor presents a blend of fundamental and advanced IDA features, catering to novices and seasoned experts alike. This season, we venture deep into the realm of working with data types, unveiling less-known operations, and unleashing the full potential of the Decompiler. In the concluding sections, Igor discloses strategies for automating repetitive tasks and personalizing IDA’s User Interface to harmonize with your distinct workflow.

We cordially invite you to join us for this promising Season 3, and keep following Igor’s Tip every Friday!

Usage: basic and advanced usage of IDA features

Types: working with types

Hidden: hidden gems, not widely known but useful functionality

Decompiler: related to the Hex-Rays decompiler

Automation: automating repetitive tasks

Customization: customizing IDA UI to better suit your workflow

a blog series on
HEX-RAYS BLOG

CHECK ALL ARTICLES : **WWW.HEX-RAYS.COM/BLOG/**

#105: Offsets with custom base

09 Sep 2022

https://hex-rays.com/blog/igors-tip-of-the-week-105-offsets-with-custom-base/

We've already covered simple offsets[1], where an operand value or a data value matches an address in the program and so can be directly converted to an offset. However, programs may also employ more complex, or indirect ways of referring to a location. One common approach is using a small offset from some predefined base address.

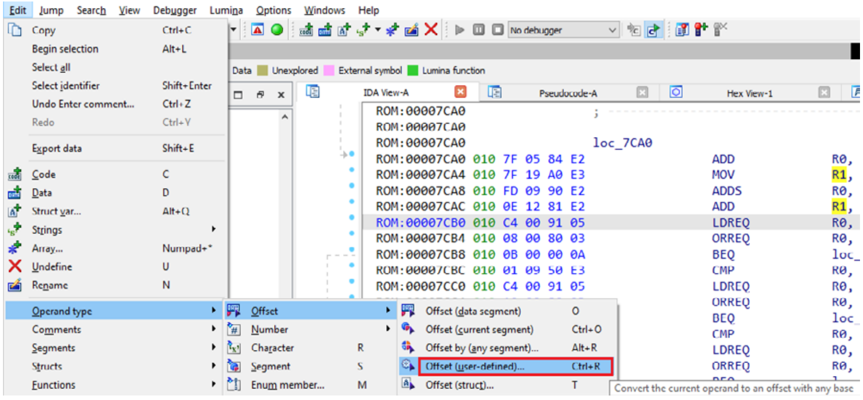
Offset (displacement) from a register

Many processors support instructions with addressing modes called "register with displacement", "register with offset" or similar. Operands in such mode may use syntax similar to following:

1. reg(offset)
2. offset(reg)
3. reg[offset]
4. [reg, offset]
5. [reg+offset]
6. etc.

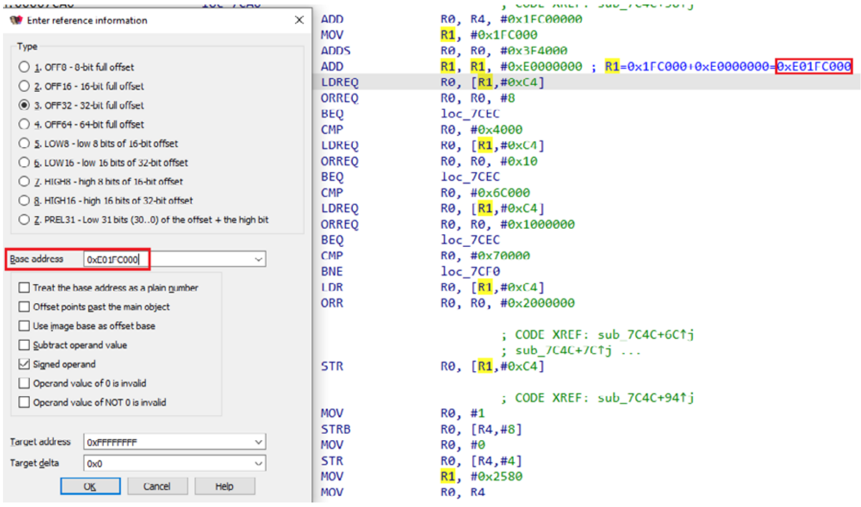
The basic logic is the same in all cases: offset is added to the value of the register and then used as a number or (more commonly) as an address. In the latter case it may be useful to have IDA calculate the final address for you and add the cross-reference to it. If you know the value of the register at the time this instruction is executed (e.g. it is set in the preceding instructions), it is very simple to do:

1. With the cursor on the operand, Invoke Edit > Operand type > Offset > Offset (user-defined), or press `Ctrl–R`;



```
ROM:00007CA0                         ;
ROM:00007CA0
ROM:00007CA0                         loc_7CA0
ROM:00007CA0 010 7F 05 84 E2                          ADD              R0,
ROM:00007CA4 010 7F 19 A0 E3                          MOV              R1,
ROM:00007CA8 010 FD 09 90 E2                          ADDS             R0,
ROM:00007CAC 010 0E 12 81 E2                          ADD              R1,
ROM:00007CB0 010 C4 00 91 05                          LDREQ            R0,
ROM:00007CB4 010 08 00 80 03                          ORREQ            R0,
ROM:00007CB8 010 0B 00 00 0A                          BEQ              loc_
ROM:00007CBC 010 01 09 50 E3                          CMP              R0,
ROM:00007CC0 010 C4 00 91 05                          LDREQ            R0,
                                                      ORREQ            R0,
                                                      BEQ              loc_
                                                      CMP              R0,
                                                      LDREQ            R0,
                                                      ORREQ            R0,
```

2. Enter the register value in the Base address field;



○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
● 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset
○ Z. PREL31 - Low 31 bits (30..0) of the offset + the high bit

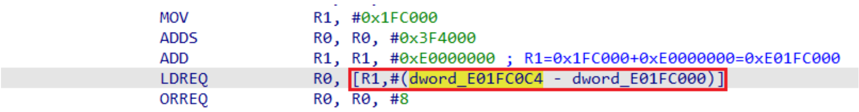
Base address: 0xE01FC000

□ Treat the base address as a plain number
□ Offset points past the main object
□ Use image base as offset base
□ Subtract operand value
☑ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address: 0xFFFFFFFF
Target delta: 0x0

```
ADD             R0, R4, #0x1FC00000
MOV             R1, #0x1FC000
ADDS            R0, R0, #0x3F4000
ADD             R1, R1, #0xE0000000 ; R1=0x1FC000+0xE0000000=0xE01FC000
LDREQ           R0, [R1,#0xC4]
ORREQ           R0, R0, #8
BEQ             loc_7CEC
CMP             R0, #0x4000
LDREQ           R0, [R1,#0xC4]
ORREQ           R0, R0, #0x10
BEQ             loc_7CEC
CMP             R0, #0x6C000
LDREQ           R0, [R1,#0xC4]
ORREQ           R0, R0, #0x1000000
BEQ             loc_7CEC
CMP             R0, #0x70000
BNE             loc_7CF0
LDR             R0, [R1,#0xC4]
ORR             R0, R0, #0x2000000

                            ; CODE XREF: sub_7C4C+6C↑j
                            ; sub_7C4C+7C↑j ...
STR             R0, [R1,#0xC4]

                            ; CODE XREF: sub_7C4C+94↑j
MOV             R0, #1
STRB            R0, [R4,#8]
MOV             R0, #0
STR             R0, [R4,#4]
MOV             R1, #0x2580
MOV             R0, R4
```

3. Click OK;



```
MOV             R1, #0x1FC000
ADDS            R0, R0, #0x3F4000
ADD             R1, R1, #0xE0000000 ; R1=0x1FC000+0xE0000000=0xE01FC000
LDREQ           R0, [R1,#(dword_E01FC0C4 - dword_E01FC000)]
ORREQ           R0, R0, #8
```

4. IDA will calculate the final address, replace the offset value by an equivalent expression, and add a cross-reference to destination:

Now it is obvious that the location being referenced is `dword_E01FC0C4`.

See also:
IDA Help: Convert operand to offset (user-defined base)[2]
IDA Help: Complex Offset Expression[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-95-offsets/
[2] https://www.hex-rays.com/products/ida/support/idadoc/470.shtml
[3] https://www.hex-rays.com/products/ida/support/idadoc/471.shtml

#106: Outlined functions

16 Sep 2022

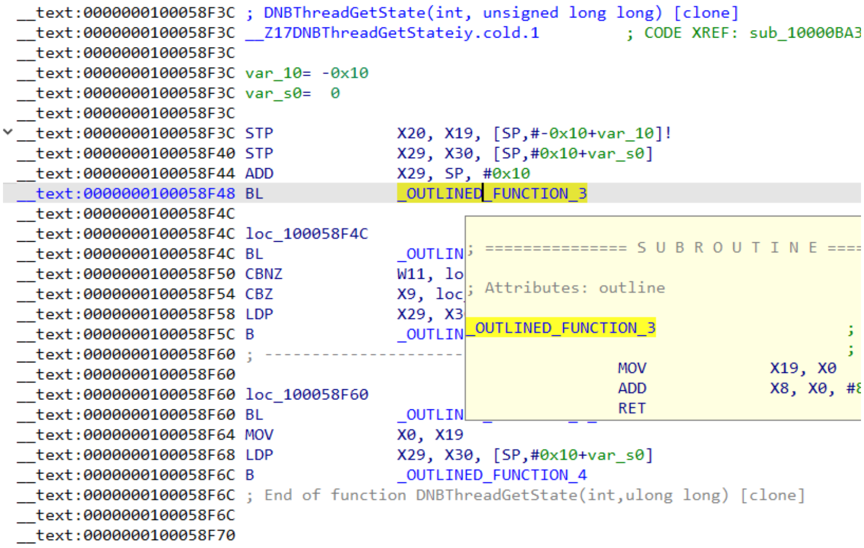
https://hex-rays.com/blog/igors-tip-of-the-week-106-outlined-functions/

The release notes for IDA 8.0[1] mention outlined functions. What are those and how to deal with them in IDA?

Function outlining is an optimization that saves code size by identifying recurring sequences of machine code and replacing each instance of the sequence with a call to a new function that contains the identified sequence of operations. It can be considered an extension of the shared function tail[2] optimization by sharing not only tails but arbitrary common parts of functions.

Function outlining example

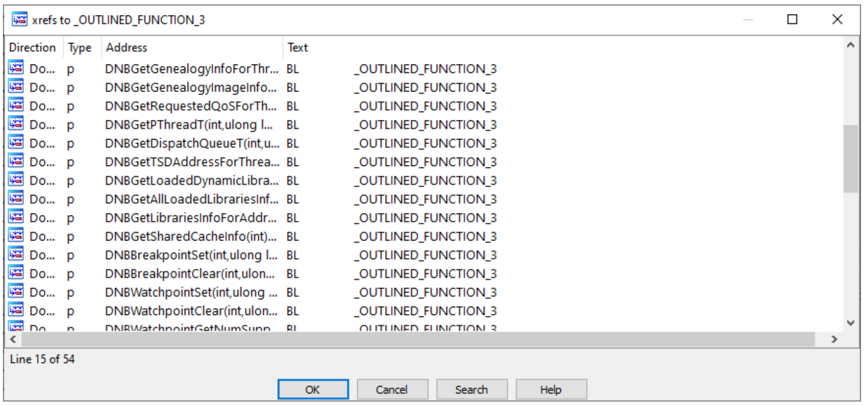
For example, here's a function from iOS's **debugserver** with some calls to outlined fragments:



```
  __text:0000000100058F3C ; DNBThreadGetState(int, unsigned long long) [clone]
  __text:0000000100058F3C __Z17DNBThreadGetStateiy.cold.1         ; CODE XREF: sub_10000BA3
  __text:0000000100058F3C
  __text:0000000100058F3C var_10= -0x10
  __text:0000000100058F3C var_s0=  0
  __text:0000000100058F3C
˅ __text:0000000100058F3C STP             X20, X19, [SP,#-0x10+var_10]!
  __text:0000000100058F40 STP             X29, X30, [SP,#0x10+var_s0]
  __text:0000000100058F44 ADD             X29, SP, #0x10
  __text:0000000100058F48 BL              _OUTLINED_FUNCTION_3
  __text:0000000100058F4C
  __text:0000000100058F4C loc_100058F4C
  __text:0000000100058F4C BL              _OUTLIN
  __text:0000000100058F50 CBNZ            W11, lo
  __text:0000000100058F54 CBZ             X9, loc
  __text:0000000100058F58 LDP             X29, X3
  __text:0000000100058F5C B               _OUTLIN
  __text:0000000100058F60 ; --------------------
  __text:0000000100058F60
  __text:0000000100058F60 loc_100058F60
  __text:0000000100058F60 BL              _OUTLIN
  __text:0000000100058F64 MOV             X0, X19
  __text:0000000100058F68 LDP             X29, X30, [SP,#0x10+var_s0]
  __text:0000000100058F6C B               _OUTLINED_FUNCTION_4
  __text:0000000100058F6C ; End of function DNBThreadGetState(int,ulong long) [clone]
  __text:0000000100058F6C
  __text:0000000100058F70
```

```
; =============== S U B R O U T I N E ====

; Attributes: outline

_OUTLINED_FUNCTION_3                                   ;
                                                       ;
               MOV             X19, X0
               ADD             X8, X0, #8
               RET
```

The first fragment contains only two instructions besides the return instruction so it may not sound like we're saving much, but by looking at the cross-references you'll see that it is used in many places:



xrefs to _OUTLINED_FUNCTION_3

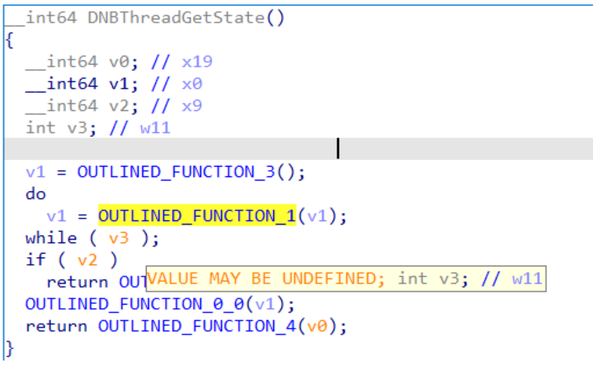
Direction	Type	Address	Text
Do...	p	DNBGetGenealogyInfoForThr...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetGenealogyImageInfo...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetRequestedQoSForTh...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetPThreadT(int,ulong l...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetDispatchQueueT(int,u...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetTSDAddressForThrea...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetLoadedDynamicLibra...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetAllLoadedLibrariesInf...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetLibrariesInfoForAddr...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBGetSharedCacheInfo(int)...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBBreakpointSet(int,ulong l...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBBreakpointClear(int,ulon...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBWatchpointSet(int,ulong ...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBWatchpointClear(int,ulon...	BL _OUTLINED_FUNCTION_3
Do...	p	DNBWatchpointGetNumSupp...	BL _OUTLINED_FUNCTION_3

Line 15 of 54

So the savings accumulated across the whole program can be quite substantial.

Handling outlined functions in decompiler

If we decompile the function, the calls to outlined fragments are shown as is, and the registers used or set by them show up as potentially undefined (orange color):

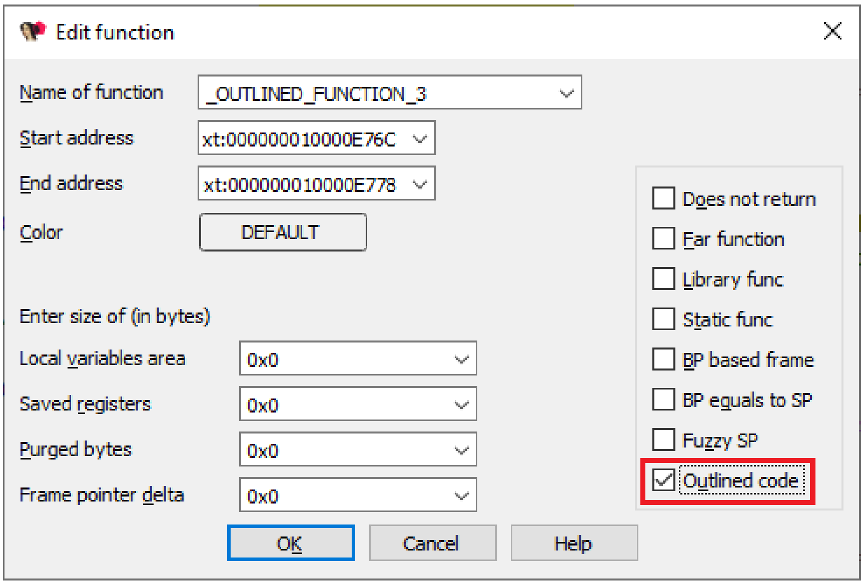


```
__int64 DNBThreadGetState()
{
  __int64 v0; // x19
  __int64 v1; // x0
  __int64 v2; // x9
  int v3; // w11

  v1 = OUTLINED_FUNCTION_3();
  do
    v1 = OUTLINED_FUNCTION_1(v1);
  while ( v3 );
  if ( v2 )
    return OUTVALUE MAY BE UNDEFINED; int v3; // w11
  OUTLINED_FUNCTION_0_0(v1);
  return OUTLINED_FUNCTION_4(v0);
}
```

#106: Outlined functions

16 Sep 2022
https://hex-rays.com/blog/igors-tip-of-the-week-106-outlined-functions/

To tell the decompiler that the calls should be inlined into the function's body, all the `OUTLINED_FUNCTION_NN` should be marked as outlined code. This can be done manually, via the Edit Function (`Alt-P`) dialog:



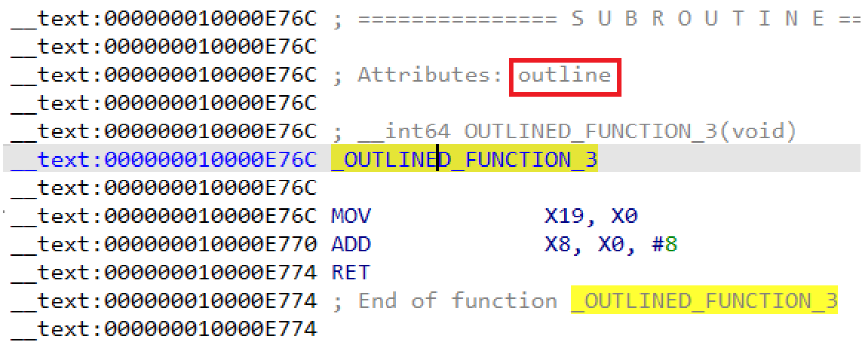
Edit function

Name of function: _OUTLINED_FUNCTION_3
Start address: xt:000000010000E76C
End address: xt:000000010000E778
Color: DEFAULT

Enter size of (in bytes)
Local variables area: 0x0
Saved registers: 0x0
Purged bytes: 0x0
Frame pointer delta: 0x0

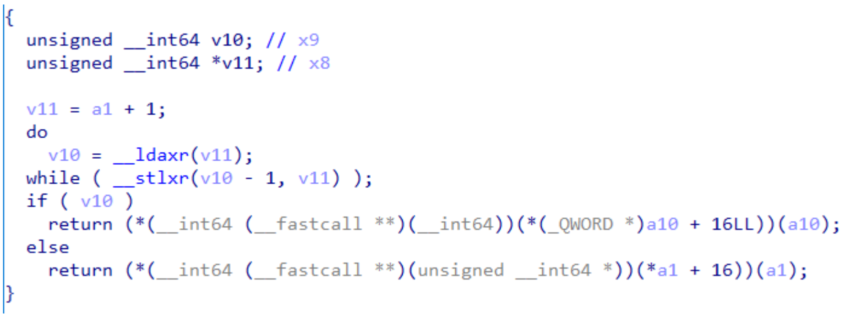
☐ Does not return
☐ Far function
☐ Library func
☐ Static func
☐ BP based frame
☐ BP equals to SP
☐ Fuzzy SP
☑ Outlined code

The added attribute is also displayed in the listing:



```
__text:000000010000E76C ; =============== S U B R O U T I N E ==
__text:000000010000E76C
__text:000000010000E76C ; Attributes: outline
__text:000000010000E76C
__text:000000010000E76C ; __int64 OUTLINED_FUNCTION_3(void)
__text:000000010000E76C _OUTLINED_FUNCTION_3
__text:000000010000E76C
__text:000000010000E76C MOV             X19, X0
__text:000000010000E770 ADD             X8, X0, #8
__text:000000010000E774 RET
__text:000000010000E774 ; End of function _OUTLINED_FUNCTION_3
__text:000000010000E774
```

Once all outlined functions are marked up, the decompiler inlines them and there are no more possibly undefined variables:



```
{
  unsigned __int64 v10; // x9
  unsigned __int64 *v11; // x8

  v11 = a1 + 1;
  do
    v10 = __ldaxr(v11);
  while ( __stlxr(v10 - 1, v11) );
  if ( v10 )
    return (*(__int64 (__fastcall **)(__int64))(*(_QWORD *)a10 + 16LL))(a10);
  else
    return (*(__int64 (__fastcall **)(unsigned __int64 *))(*a1 + 16))(a1);
}
```

Automating outlined function processing

If you have a big binary with hundreds or thousands of functions, it may become pretty tedious to mark up outlined functions manually. In such case, making a small script may speed things up. For example, if you have symbols and outlined functions have a known naming pattern, the following Python snippet should work:

#106: Outlined functions

16 Sep 2022

https://hex-rays.com/blog/igors-tip-of-the-week-106-outlined-functions/

```
import idautils
import ida_name
import ida_funcs
for f in idautils.Functions():
    nm = ida_name.get_name(f)
    if nm.startswith(“_OUTLINED_FUNCTION”) or nm.find(“.cold.”) != -1:
        print (“%08X: %s”% (f, nm))
        pfn = ida_funcs.get_func(f)
        pfn.flags |= idaapi.FUNC_OUTLINE
        ida_funcs.update_func(pfn)
```

It can be executed using File > Script command… (`Shift+F2`)

See also:
IDA Help: Edit Function[3]
IDA Help: Function flags[4]

[1] https://hex-rays.com/products/ida/news/8_0/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-87-function-chunks-and-the-decompiler/
[3] https://www.hex-rays.com/products/ida/support/idadoc/485.shtml
[4] https://www.hex-rays.com/products/ida/support/idadoc/1729.shtml

#107: Multiple return values

23 Sep 2022
https://hex-rays.com/blog/igors-tip-of-the-week-107-multiple-return-values/

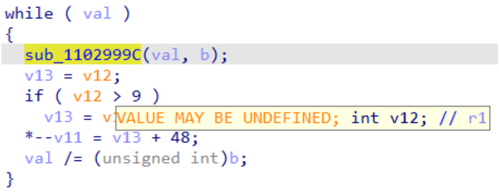
The Hex-Rays decompiler was initially created to decompile C code, so its pseudocode output uses (mostly) C syntax. However, the input binaries may be compiled using other languages: C++, Pascal, Basic, ADA, and many others. While the code of most of them can be represented in C without real issues, some have peculiarities which require language extensions[1] or have to be handled with user input[2]. Still, some languages use approaches so different from standard compiled C code that special handling for that is necessary. For example, Go[3] uses a calling convention[4] (stack-based or register-based) so different from standard C calling conventions, that custom support for it had to be added to IDA[5].

Multiple return values

Even with custom calling conventions, one fundamental limitation of IDA's type system remains (as of IDA 8.0): a function may return only a single value. However, even in otherwise C-style programs you may encounter functions which return more than one value. One example is compiler helpers like `idivmod`/`uidivmod`. They return simultaneously the quotient and remainder of a division operation. The decompiler knows about the standard ones (e.g. `__aeabi_idivmod` for ARM EABI) but you may encounter a non-standard implementation, or an unrelated function using a similar approach (e.g. a function written manually in assembly).

Because the decompiler does not expect that function returns more than one value, you may need to inspect the disassembly or look at the place of the call to recognize such functions. For example, here's a fragment of decompiled ARM32 code which seems to use an undefined register value:

The function seems to modify the `R1` register, although normally the return values (for 32-bit types) are placed in `R0`. Possibly this is an equivalent of divmod function which returns quotient in `R0` and remainder in `R1`?



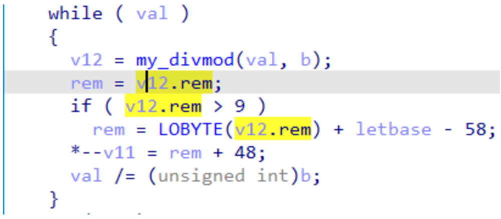
```
while ( val )
{
  sub_1102999C(val, b);
  v13 = v12;
  if ( v12 > 9 )
    v13 = vVALUE MAY BE UNDEFINED; int v12; // r1
  *--v11 = v13 + 48;
  val /= (unsigned int)b;
}
```

To handle this, we can use an artificial structure and a custom calling convention specifying the registers and/or stack locations where it should be placed. For example, add such struct to Local Types:

```
struct divmod_t
{
  int quot;
  int rem;
};
```

and set the function prototype: `divmod_t __usercall my_divmod@<R1:R0>(int@<R0>, int@<R1>);`

The decompiler then interprets the register values after the call as if they were structure fields:



```
while ( val )
{
  v12 = my_divmod(val, b);
  rem = v12.rem;
  if ( v12.rem > 9 )
    rem = LOBYTE(v12.rem) + letbase - 58;
  *--v11 = rem + 48;
  val /= (unsigned int)b;
}
```

A similar approach may be used for languages with native support for functions with multiple return values: Go, Swift, Rust etc.

See also:
Igor's tip of the week #51: Custom calling conventions[6]

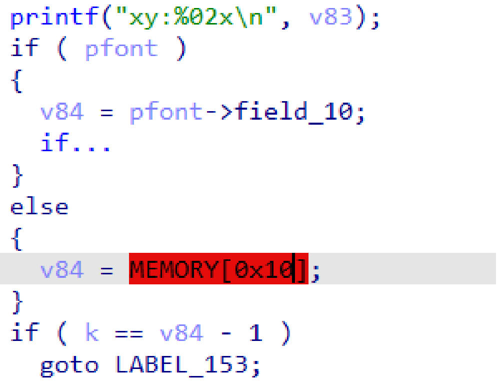
[1] https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-71-decompile-as-call/
[3] https://go.dev/
[4] https://go.dev/src/cmd/compile/abi-internal
[5] https://hex-rays.com/products/ida/news/7_6/
[6] https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/

#108: Raw memory accesses in pseudocode

30 Sep 2022

https://hex-rays.com/blog/igors-tip-of-the-week-108-raw-memory-accesses-in-pseudocode/

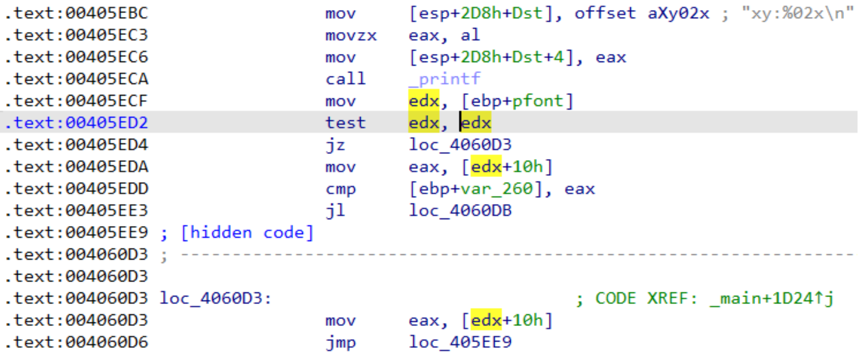
Sometimes in pseudocode you may encounter strange-looking code:



```
printf("xy:%02x\n", v83);
if ( pfont )
{
  v84 = pfont->field_10;
  if...
}
else
{
  v84 = MEMORY[0x10];
}
if ( k == v84 - 1 )
  goto LABEL_153;
```

The code seems to dereference an array called `MEMORY` and is highlighted in red. However, this variable is not defined anywhere. What is it?

Such notation is used by the decompiler when the code accesses memory addresses not present in the database. In most cases it indicates an error in the original source code. If we look at the disassembly for the example above, we'll see this:



```
.text:00405EBC                 mov     [esp+2D8h+Dst], offset aXy02x ; "xy:%02x\n"
.text:00405EC3                 movzx   eax, al
.text:00405EC6                 mov     [esp+2D8h+Dst+4], eax
.text:00405ECA                 call    _printf
.text:00405ECF                 mov     edx, [ebp+pfont]
.text:00405ED2                 test    edx, edx
.text:00405ED4                 jz      loc_4060D3
.text:00405EDA                 mov     eax, [edx+10h]
.text:00405EDD                 cmp     [ebp+var_260], eax
.text:00405EE3                 jl      loc_4060DB
.text:00405EE9 ; [hidden code]
.text:004060D3 ; ---------------------------------------------------------------------------
.text:004060D3
.text:004060D3 loc_4060D3:                             ; CODE XREF: _main+1D24↑j
.text:004060D3                 mov     eax, [edx+10h]
.text:004060D6                 jmp     loc_405EE9
```

The variable `pfont` is loaded into register `edx` which is then compared against zero using `test edx, edx/jz` sequence. The jump to `loc_4060D3` can only occur if `edx` is zero, which means that the `mov eax, [edx+10h]` instruction will try to dereference the address `0x10`. Because the database does not contain the address `0x10`, it can't be represented as a normal or a dummy variable so the decompiler represents it as a pseudo-variable `MEMORY` and uses the address as the index. The dereference is shown in red to bring attention to the potential error in the code. For example, judging by the assembly, in this binary the programmer tried reading a structure pointer even if it is NULL. A more modern compiler would probably even remove such code as dereferencing NULL pointer is undefined behavior.

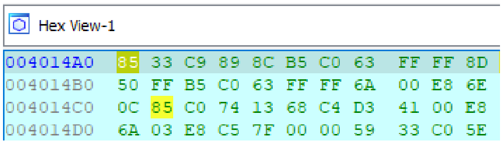
In cases where such access is **not** an error (for example, the code directly accesses memory-mapped hardware registers), creating a new segment for the accessed address range is usually the correct approach.

#109: Hex view text encoding

07 Oct 2022

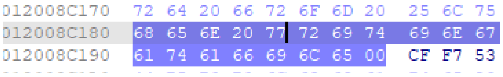
https://hex-rays.com/blog/igors-tip-of-the-week-109-hex-view-text-encoding/

The Hex view is used to display the contents of the database as a hex dump. It is also used during debugging to display memory contents.



```
004014A0  85 33 C9 89 8C B5 C0 63  FF FF 8D
004014B0  50 FF B5 C0 63 FF FF 6A  00 E8 6E
004014C0  0C 85 C0 74 13 68 C4 D3  41 00 E8
004014D0  6A 03 E8 C5 7F 00 00 59  33 C0 5E
```

By default it has a part on the right with the textual representation of the data. Usually the text part shows Latin letters or dots for unprintable characters but you may also encounter something unusual:

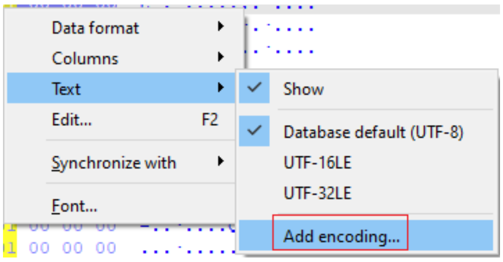


```
012008C170  72 64 20 66 72 6F 6D 20  25 6C 75
012008C180  68 65 6E 20 77 72 69 74  69 6E 67
012008C190  61 74 61 66 69 6C 65 00  CF F7 53
```

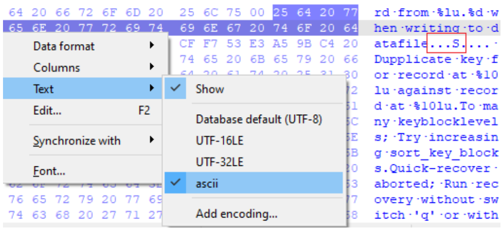
Why is there Chinese among English? Is it a hidden message and the binary actually comes from China?

In fact, the mystery has a very simple explanation: the encoding used for showing text data in hex view uses the database[1] default which is usually UTF-8, so a valid UTF-8 byte sequence may decode to Chinese, Japanese, Russian, Korean, or even emoji. If you prefer to see only the plain ASCII text, you can change the encoding using these simple steps:

1. From the hex view's context menu, invoke Text > Add encoding…

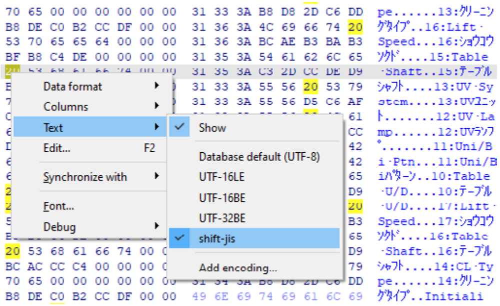


2. Enter "ascii";
3. the new encoding will be added to the list and made default, so any bytes not falling into the ASCII range will be shown as unprintable:



```
64 20 66 72 6F 6D 20  25 6C 75 00 25 64 20 77  rd·from·%lu.%d·w
65 6E 20 77 72 69 74  69 6E 67 20 74 6F 20 64  hen·writing·to·d
                      CF F7 53 E3 A5 9B C4 20  atafile...S.....·
                      74 65 20 6B 65 79 20 66  Dupplicate·key·f
                                               or·record·at·%10
                                               lu·against·recor
                                               d·at·%10lu.To·ma
                                               ny·keyblocklevel
                                               s;·Try·increasin
                                               g·sort_key_block
                                               s.Quick-recover·
                                               aborted;·Run·rec
76 65 72 79 20 77 69                           overy·without·sw
74 63 68 20 27 71 27                           itch·'q'·or·with
```

Instead of "ascii" you can use another encoding which matches the type of binary you're analyzing. For example, if you work with legacy Japanese software, encodings like "Shift-JIS", "cp932" or "EUC-JP" may help you discover otherwise hidden text.



```
70 65 00 00 00 00 00 00  31 33 3A B8 D8 2D C6 DD  pe......13:ｸﾘｰﾆﾝ
B8 DE C0 B2 CC DF 00 00  31 36 3A 4C 69 66 74 20  ｸﾞﾀｲﾌﾟ..16:Lift·
53 70 65 65 64 00 00 00  31 36 3A BC AE B3 BA B3  Speed...16:ｼｮｳｺｳ
BF B8 C4 DE 00 00 00 00  31 35 3A 54 61 62 6C 65  ｿｸﾄﾞ....15:Table
                         31 35 3A C3 2D CC DE D9  ·Shaft..15:ﾃｰﾌﾞﾙ
                         31 33 3A 55 56 20 53 79  ｼｬﾌﾄ....13:UV·Sy
                         31 33 3A 55 56 D5 C6 AF  stcm....13:UVﾕﾆｯ
                                               61  ﾄ.......12:UV·La
                                               CC  mp......12:UVﾗﾝﾌ
                                               42  ｰ.......11:Uni/B
                                               42  i·Ptn...11:Uni/B
                                               65  iﾊﾟﾀｰﾝ..10:Table
                                               D9  ·U/D....10:ﾃｰﾌﾞﾙ
                                               20  ·U/D....17:Lift·
                                               B3  Speed...17:ｼｮｳｺｳ
                                               65  ｿｸﾄﾞ....16:Table
20 53 68 61 66 74 00                           D9  ·Shaft..16:ﾃｰﾌﾞﾙ
BC AC CC C4 00 00 00                           79  ｼｬﾌﾄ....14:CL·Ty
70 65 00 00 00 00 00 00                        DD  pe......14:ｸﾘｰﾆﾝ
B8 DE C0 B2 CC DF 00 00  49 6E 69 74 69 61 6C 69  ｸﾞﾀｲﾌﾟ..Initiali
```

See also:
Igor's tip of the week #13: String literals and custom encodings[2]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/

#110: Self-relative offsets

14 Oct 2022

https://hex-rays.com/blog/igors-tip-of-the-week-110-self-relative-offsets/

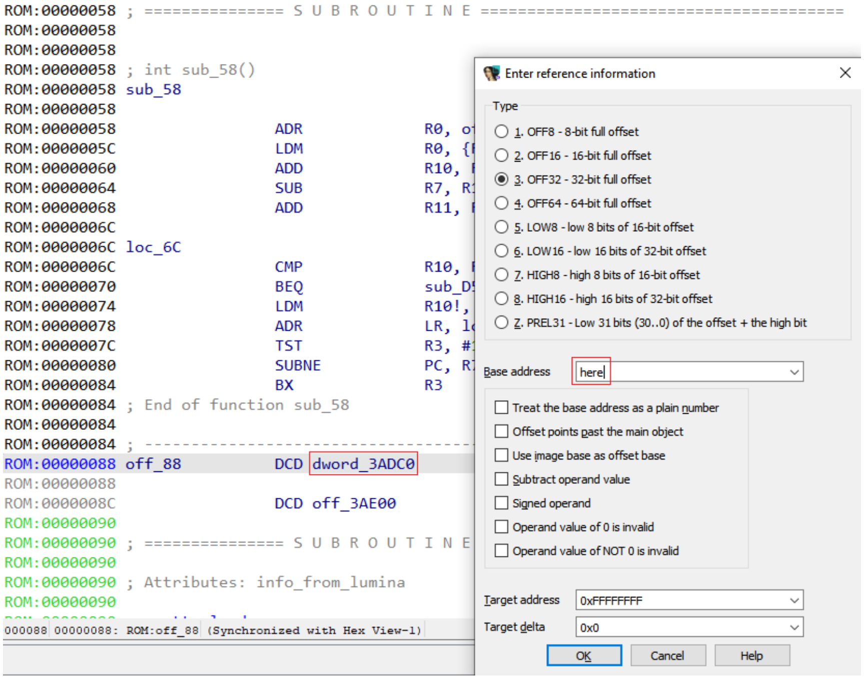
We've covered offsets with base1 previously. There is a variation of such offsets commonly used in position-independent code which can be handled easily with a little trick.

Let's consider this ARM function from an ARM32 firmware:

```
ROM:00000058 ; int sub_58()
ROM:00000058 sub_58                                  ; CODE XREF: sub_10A4:loc_50↑j
ROM:00000058                                         ; DATA XREF: sub_8D40+20↓r ...
ROM:00000058                 ADR             R0, off_88 ; R0 = 0x88
ROM:0000005C                 LDM             R0, {R10,R11} ; R10 = 0x3ADC0, R11 = 0x3AE00
ROM:00000060                 ADD             R10, R10, R0 ; R10 = 0x3ADC0+0x88
ROM:00000064                 SUB             R7, R10, #1
ROM:00000068                 ADD             R11, R11, R0 ; R11 = 0x3AE00+0x88
ROM:0000006C
ROM:0000006C loc_6C                                  ; DATA XREF: sub_58+20↓o
ROM:0000006C                 CMP             R10, R11
ROM:00000070                 BEQ             sub_D50
ROM:00000074                 LDM             R10!, {R0-R3}
ROM:00000078                 ADR             LR, loc_6C
ROM:0000007C                 TST             R3, #1
ROM:00000080                 SUBNE           PC, R7, R3
ROM:00000084                 BX              R3
ROM:00000084 ; End of function sub_58
ROM:00000084
ROM:00000084 ; ---------------------------------------------------------------------------
ROM:00000088 off_88          DCD dword_3ADC0         ; DATA XREF: sub_58↑o
ROM:00000088                                         ; sub_58+4↑o
ROM:0000008C                 DCD off_3AE00
```

IDA has converted the values at addresses 88 and 8C to offsets because they happen to be valid addresses, but if you look at what the code does (I've added comments describing what happens), we'll see that both values are added to the address from which they're loaded (0x88), i.e. they're relative to their own position (or self-relative).

To get the final value they refer to, we can use the action Edit > Operand type > Offset >Offset (user-defined) (shortcut `Ctrl-R`), and enter as the base either the address value (0x88), or, for the case of the value at `00000088`, the IDC keyword `here`, which expands to the address under the cursor.



```
ROM:00000058 ; =============== S U B R O U T I N E =======================================
ROM:00000058
ROM:00000058
ROM:00000058 ; int sub_58()
ROM:00000058 sub_58
ROM:00000058
ROM:00000058                 ADR             R0, o
ROM:0000005C                 LDM             R0, {
ROM:00000060                 ADD             R10,
ROM:00000064                 SUB             R7, R
ROM:00000068                 ADD             R11,
ROM:0000006C
ROM:0000006C loc_6C
ROM:0000006C                 CMP             R10,
ROM:00000070                 BEQ             sub_D
ROM:00000074                 LDM             R10!,
ROM:00000078                 ADR             LR, l
ROM:0000007C                 TST             R3, #
ROM:00000080                 SUBNE           PC, R
ROM:00000084                 BX              R3
ROM:00000084 ; End of function sub_58
ROM:00000084
ROM:00000084 ; ----------------------------------------
ROM:00000088 off_88          DCD dword_3ADC0
ROM:00000088
ROM:0000008C                 DCD off_3AE00
ROM:00000090
ROM:00000090 ; =============== S U B R O U T I N E
ROM:00000090
ROM:00000090 ; Attributes: info_from_lumina
ROM:00000090
000088 00000088: ROM:off_88 (Synchronized with Hex View-1)
```

Enter reference information

Type

○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
● 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset
○ Z. PREL31 - Low 31 bits (30..0) of the offset + the high bit

Base address: here

□ Treat the base address as a plain number
□ Offset points past the main object
□ Use image base as offset base
□ Subtract operand value
□ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address: 0xFFFFFFFF

Target delta: 0x0

IDA calculates the final address and replaces the value with an expression which uses a special symbol ., which denotes the current address on ARM:

#110: Self-relative offsets

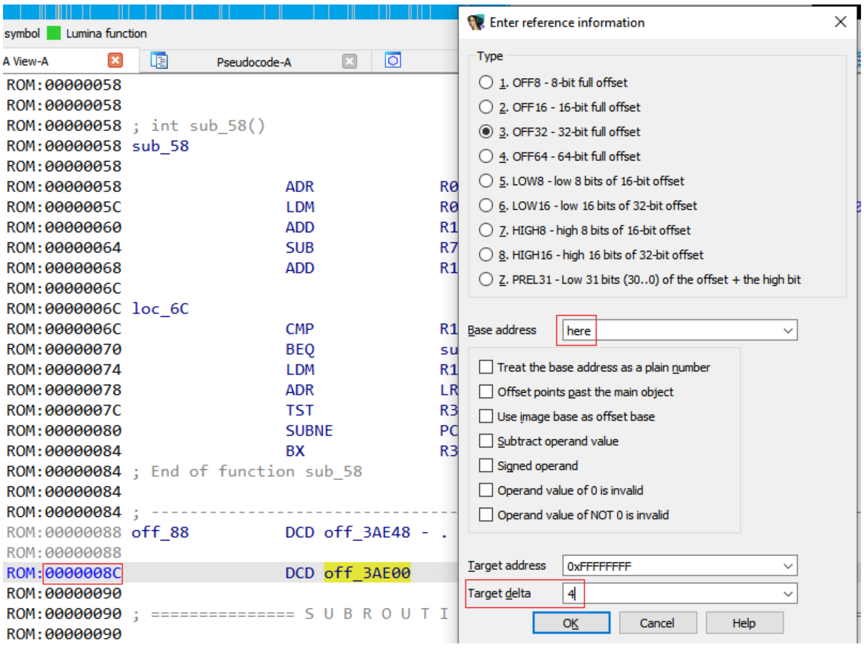
14 Oct 2022

https://hex-rays.com/blog/igors-tip-of-the-week-110-self-relative-offsets/

```
ROM:00000088 off_88          DCD off_3AE48 - .       ; DATA XREF: sub_58↑o
```

For the value at `0000008C`, `here` will not work since it expands to 0x8c while the addend is 0x88. There are several options we can use:

1. use the actual value `0x88` as the base
2. use the expression `here-4` which resolves to 0x88.
3. use `here`, but specify 4 in the *Target delta* field.



```
ROM:00000058
ROM:00000058
ROM:00000058 ; int sub_58()
ROM:00000058 sub_58
ROM:00000058
ROM:00000058                 ADR             R0
ROM:0000005C                 LDM             R0
ROM:00000060                 ADD             R1
ROM:00000064                 SUB             R7
ROM:00000068                 ADD             R1
ROM:0000006C
ROM:0000006C loc_6C
ROM:0000006C                 CMP             R1
ROM:00000070                 BEQ             su
ROM:00000074                 LDM             R1
ROM:00000078                 ADR             LR
ROM:0000007C                 TST             R3
ROM:00000080                 SUBNE           PC
ROM:00000084                 BX              R3
ROM:00000084 ; End of function sub_58
ROM:00000084
ROM:00000084 ; ---------------------------------
ROM:00000088 off_88          DCD off_3AE48 - .
ROM:00000088
ROM:0000008C                 DCD off_3AE00
ROM:00000090
ROM:00000090 ; =============== S U B R O U T I
ROM:00000090
```

Enter reference information

Type

○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
● 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset
○ Z. PREL31 - Low 31 bits (30..0) of the offset + the high bit

Base address: here

□ Treat the base address as a plain number
□ Offset points past the main object
□ Use image base as offset base
□ Subtract operand value
□ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address: 0xFFFFFFFF

Target delta: 4

IDA will use the delta as an additional adjustment for the expression:

```
ROM:0000008C                 DCD byte_3AE88+4 - .
```

Now we can see what addresses the function is actually using and analyze it further.

See also:
Igor's tip of the week #105: Offsets with custom base[2]
Igor's tip of the week #21: Calculator and expression evaluation feature in IDA[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-105-offsets-with-custom-base/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-105-offsets-with-custom-base/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-21-calculator-and-expression-evaluation-feature-in-ida/

#111: IDA Keyboard Shortcuts cheat sheet

21 Oct 2022

https://hex-rays.com/blog/igors-tip-of-the-week-111-ida-keyboard-shortcuts-cheat-sheet/

Many keyboard shortcuts[1] have been described on this blog, but they may be difficult to retain, especially if you don't use them every day. To remedy that, we have been publishing a cheat sheet with the most common ones.

You can find it linked from our documentation page[2] in HTML[3] or PDF[4] format.

NOTE: the shortcuts described are for the default configuration; you can modify them[5] to your liking.

See also:
Igor's tip of the week #01: Lesser-known keyboard shortcuts in IDA[6]
Igor's tip of the week #02: IDA UI actions and where to find them[7]

[1] https://hex-rays.com/blog/tag/shortcuts/
[2] https://hex-rays.com/documentation/
[3] https://hex-rays.com/products/ida/support/idapro_cheatsheet.html
[4] https://hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf
[5] https://hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/
[6] https://hex-rays.com/blog/igor-tip-of-the-week-01-lesser-known-keyboard-shortcuts-in-ida/
[7] https://hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/

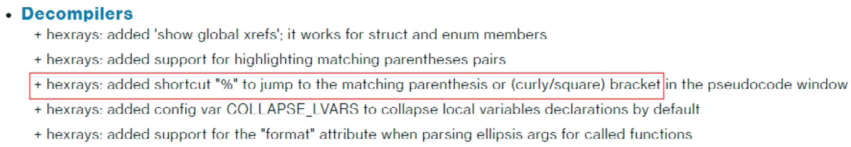
#112: Matching braces

28 Oct 2022

https://hex-rays.com/blog/igors-tip-of-the-week-112-matching-braces/

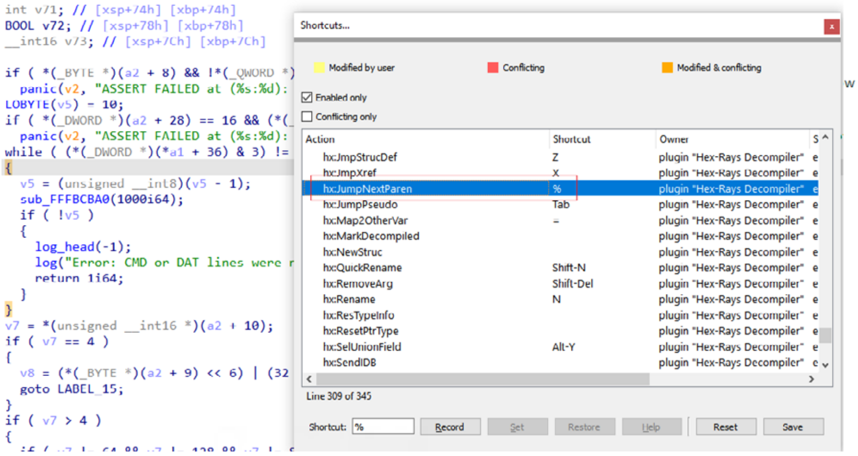
When working with big functions in the decompiler, it may be difficult to find what you need if the listing is long. While you can use cross-references[1] to jump between uses of a variable or collapse[2] parts of pseudocode to make it more compact, there is one simple shortcut which can make your life easier.

The shortcut is not currently (IDA 8.1) shown in the context menu, but it was mentioned in the release notes for IDA 7.4[3]:



- **Decompilers**
  - + hexrays: added 'show global xrefs'; it works for struct and enum members
  - + hexrays: added support for highlighting matching parentheses pairs
  - + hexrays: added shortcut "%" to jump to the matching parenthesis or (curly/square) bracket in the pseudocode window
  - + hexrays: added config var COLLAPSE_LVARS to collapse local variables declarations by default
  - + hexrays: added support for the "format" attribute when parsing ellipsis args for called functions

You can also discover it by opening the Options > Shortcuts... dialog while the cursor is positioned on a brace or parenthesis:



```
int v71; // [xsp+74h] [xbp+74h]
BOOL v72; // [xsp+78h] [xbp+78h]
__int16 v73; // [xsp+7Ch] [xbp+7Ch]

if ( *(_BYTE *)(a2 + 8) && !*(_QWORD *)
  panic(v2, "ASSERT FAILED at (%s:%d):
LOBYTE(v5) = 10;
if ( *(_DWORD *)(a2 + 28) == 16 && (*(_
  panic(v2, "ASSERT FAILED at (%s:%d):
while ( (*(_DWORD *)(*a1 + 36) & 3) !=
{
  v5 = (unsigned __int8)(v5 - 1);
  sub_FFFBCBA0(1000i64);
  if ( !v5 )
  {
    log_head(-1);
    log("Error: CMD or DAT lines were r
    return 1i64;
  }
}
v7 = *(unsigned __int16 *)(a2 + 10);
if ( v7 == 4 )
{
  v8 = (*(_BYTE *)(a2 + 9) << 6) | (32
  goto LABEL_15;
}
if ( v7 > 4 )
{
```

☑ Enabled only

☐ Conflicting only

Action	Shortcut	Owner	S
hx:JmpStrucDef	Z	plugin "Hex-Rays Decompiler"	e
hx:JmpXref	X	plugin "Hex-Rays Decompiler"	e
hx:JumpNextParen	%	plugin "Hex-Rays Decompiler"	e
hx:JumpPseudo	Tab	plugin "Hex-Rays Decompiler"	e
hx:Map2OtherVar	=	plugin "Hex-Rays Decompiler"	e
hx:MarkDecompiled		plugin "Hex-Rays Decompiler"	e
hx:NewStruc		plugin "Hex-Rays Decompiler"	e
hx:QuickRename	Shift-N	plugin "Hex-Rays Decompiler"	e
hx:RemoveArg	Shift-Del	plugin "Hex-Rays Decompiler"	e
hx:Rename	N	plugin "Hex-Rays Decompiler"	e
hx:ResTypeInfo		plugin "Hex-Rays Decompiler"	e
hx:ResetPtrType		plugin "Hex-Rays Decompiler"	e
hx:SelUnionField	Alt-Y	plugin "Hex-Rays Decompiler"	e
hx:SendIDB		plugin "Hex-Rays Decompiler"	e

Line 309 of 345

Shortcut: %

This dialog can also be used to modify the shortcut to something you may find more convenient, for example `Ctrl–]`

See also:
Igor's tip of the week #06: IDA Release notes – Hex Rays[4]
Igor's tip of the week #02: IDA UI actions and where to find them – Hex Rays[5]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-18-decompiler-and-global-cross-references/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-100-collapsing-pseudocode-parts/
[3] https://hex-rays.com/products/ida/news/7_4/
[4] https://hex-rays.com/blog/igor-tip-of-the-week-06-release-notes/
[5] https://hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/

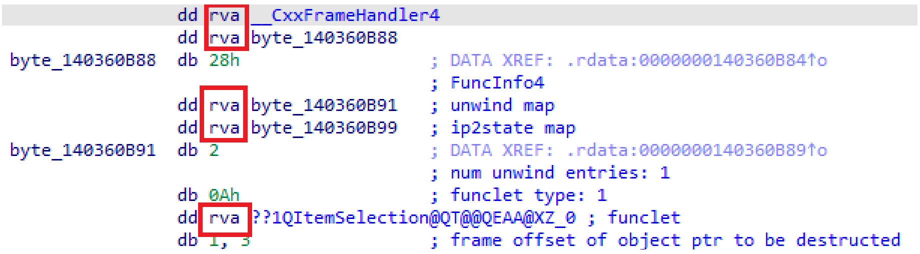
#113: Image-relative Offsets (RVA)

04 Nov 2022
https://hex-rays.com/blog/igors-tip-of-the-week-113-image-relative-offsets-rva/

Image-relative offsets are values that represent an offset from the image base of the current module (image) in memory. This means that they can be used to refer to other locations in the same module regardless of its real, final load address, and thus can be used to make the code position-independent (PIC), similarly to the self-relative offsets[1]. The alternative name RVA means “Relative virtual address” and is often used in the context of the PE file format.

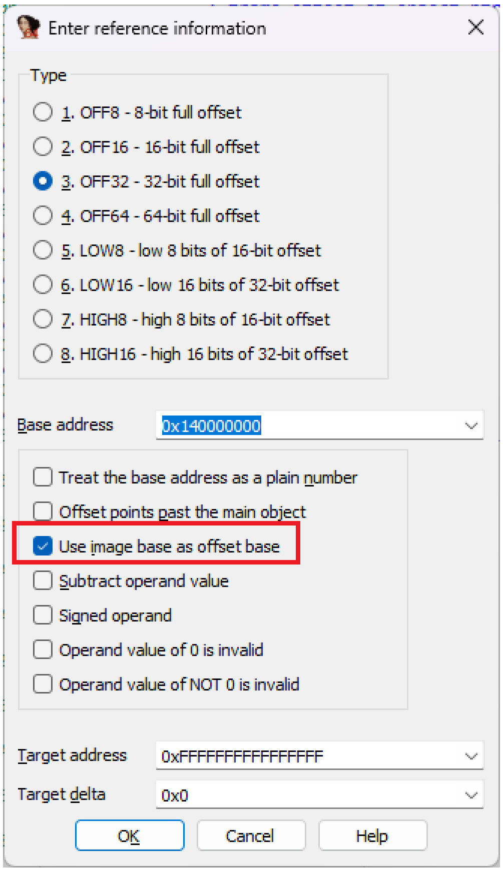
However, PIC is not the only advantage of RVAs. For example, on x64-bit platforms RVA values usually use 32 bits instead of 64 like a full pointer. While this makes their range more limited (4GiB from imagebase), the savings from pointer-type values can be substantial when accumulated over the whole binary.

For known RVA values, such as those in the PE headers or EH structures, IDA can usually convert them to an assembler-specific expression automatically:



```
                dd rva __CxxFrameHandler4
                dd rva byte_140360B88
byte_140360B88  db 28h                      ; DATA XREF: .rdata:0000000140360B84↑o
                                            ; FuncInfo4
                dd rva byte_140360B91       ; unwind map
                dd rva byte_140360B99       ; ip2state map
byte_140360B91  db 2                        ; DATA XREF: .rdata:0000000140360B89↑o
                                            ; num unwind entries: 1
                db 0Ah                      ; funclet type: 1
                dd rva ??1QItemSelection@QT@@QEAA@XZ_0 ; funclet
                db 1, 3                     ; frame offset of object ptr to be destructed
```

However, sometimes there may be a need to do it manually, for example, when dealing with another update of the file format not yet handled by IDA, or a custom format/structure which uses RVAs for addressing. In that case, you can use yet another variation of the User-defined offset[2]. The option to turn on is Use image base as offset base. When it’s enabled, IDA will ignore the entered offset base and will always use the imagebase.



Enter reference information

Type

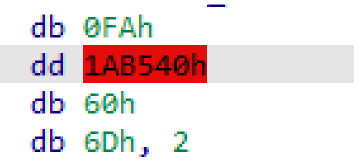
○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
● 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset

Base address: 0x140000000

☐ Treat the base address as a plain number
☐ Offset points past the main object
☑ Use image base as offset base
☐ Subtract operand value
☐ Signed operand
☐ Operand value of 0 is invalid
☐ Operand value of NOT 0 is invalid

Target address: 0xFFFFFFFFFFFFFFFF
Target delta: 0x0

However, even if you use this approach in a 64-bit program, you may fail to reach the desired effect: the value will be displayed in red to indicate an error and not show a nice expression with the final address, as expected.



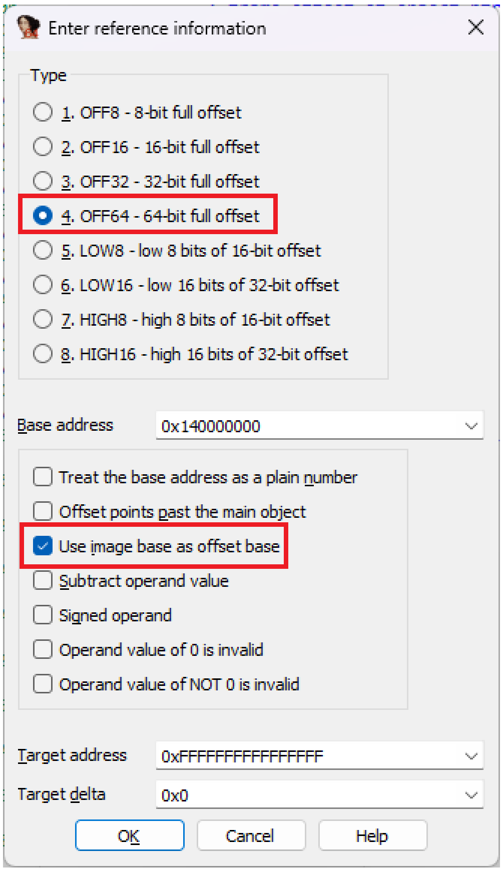
```
db 0FAh
dd 1AB540h
db 60h
db 6Dh, 2
```

#113: Image-relative Offsets (RVA)

04 Nov 2022

https://hex-rays.com/blog/igors-tip-of-the-week-113-image-relative-offsets-rva/

This happens because the command defaults to OFF32 for 32-bit values, but the final address does not fit into 32 bits. The fix is simple: select OFF64 instead of OFF32.


Enter reference information

Type

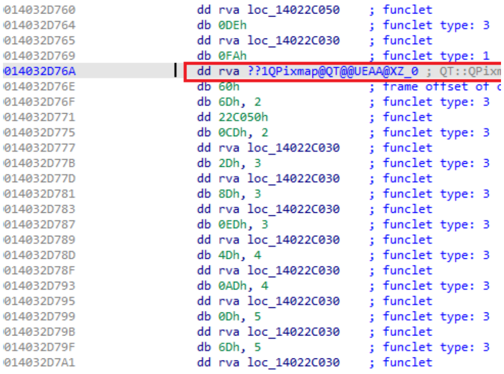
○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
○ 3. OFF32 - 32-bit full offset
● 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset

Base address: 0x140000000

□ Treat the base address as a plain number
□ Offset points past the main object
☑ Use image base as offset base
□ Subtract operand value
□ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address: 0xFFFFFFFFFFFFFFFF

Target delta: 0x0


```
014032D760               dd rva loc_14022C050    ; funclet
014032D764               db 0DEh                 ; funclet type: 3
014032D765               dd rva loc_14022C030    ; funclet
014032D769               db 0FAh                 ; funclet type: 1
014032D76A               dd rva ??1QPixmap@QT@@UEAA@XZ_0 ; QT::QPixm
014032D76E               db 60h                  ; frame offset of c
014032D76F               db 6Dh, 2               ; funclet type: 3
014032D771               dd 22C050h              ; funclet
014032D775               db 0CDh, 2              ; funclet type: 3
014032D777               dd rva loc_14022C030    ; funclet
014032D77B               db 2Dh, 3               ; funclet type: 3
014032D77D               dd rva loc_14022C030    ; funclet
014032D781               db 8Dh, 3               ; funclet type: 3
014032D783               dd rva loc_14022C030    ; funclet
014032D787               db 0EDh, 3              ; funclet type: 3
014032D789               dd rva loc_14022C030    ; funclet
014032D78D               db 4Dh, 4               ; funclet type: 3
014032D78F               dd rva loc_14022C030    ; funclet
014032D793               db 0ADh, 4              ; funclet type: 3
014032D795               dd rva loc_14022C030    ; funclet
014032D799               db 0Dh, 5               ; funclet type: 3
014032D79B               dd rva loc_14022C030    ; funclet
014032D79F               db 6Dh, 5               ; funclet type: 3
014032D7A1               dd rva loc_14022C030    ; funclet
```

NOTE: for ARM binaries, the `imagerel` keyword is used instead of `rva`.

See also:
Igor's tip of the week #105: Offsets with custom base[3]
Igor's tip of the week #110: Self-relative offsets[4]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-110-self-relative-offsets/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-105-offsets-with-custom-base/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-105-offsets-with-custom-base/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-110-self-relative-offsets/

#114: Split offsets

11 Nov 2022

https://hex-rays.com/blog/igors-tip-of-the-week-114-split-offsets/

Previously, we have covered offset expressions[1] which fit into a single instruction operand or data value. But this is not always the case, so let's see how IDA can handle offsets which may be built out of multiple parts.

8-bit processors

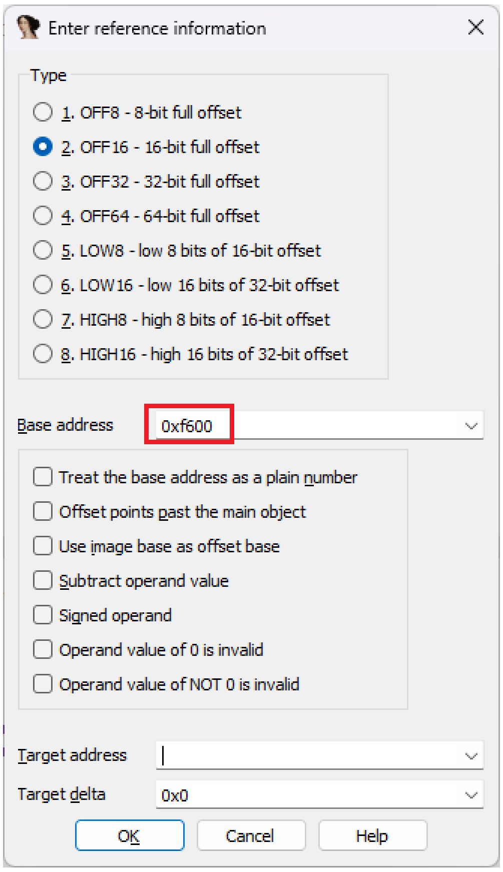
Although slowly dying out, the 8-bit processors – especially the venerable 8051 – can still appear in current hardware, and of course we'll be dealing with legacy systems for many years to come. Even though their registers can store only 8 bits af data, most of them can address 16-bit (64KiB) or more of memory which means that the addresses may need to be built by parts.

For example, consider this sequence of instructions from an 8051 firmware:

```
code:CF22    mov     R3, #0xFF
code:CF24    mov     R2, #0xF6
code:CF26    mov     R1, #0xA6
code:CF28    sjmp    code_CF36
```

The code for 8051 is often compiled using Keil C51 compiler, and this pattern is a typical way of initializing a generic pointer to code memory[2]. The address being referenced is `0xF6A6`, but can we make the instructions look "nice" and create cross references to it?

One possibility is to use offset with custom base[3] on the last move and specify the base of `0xF600`:



Type

○ 1. OFF8 - 8-bit full offset
● 2. OFF16 - 16-bit full offset
○ 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset

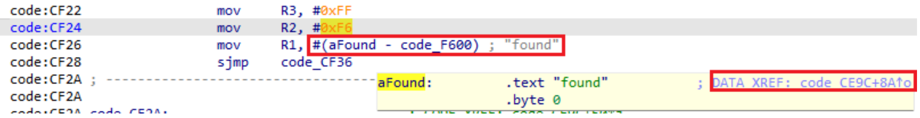
Base address: 0xf600

□ Treat the base address as a plain number
□ Offset points past the main object
□ Use image base as offset base
□ Subtract operand value
□ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address:

Target delta: 0x0

This does calculate the final address and create a cross-reference but the code is not quite "nice looking" and the other instruction remains a plain number:



```
code:CF22                   mov     R3, #0xFF
code:CF24                   mov     R2, #0xF6
code:CF26                   mov     R1, #(aFound - code_F600) ; "found"
code:CF28                   sjmp    code_CF36
code:CF2A ; --------------------------------
code:CF2A
```

```
aFound:          .text "found"          ; DATA XREF: code_CE9C+8A↑o
                 .byte 0
```

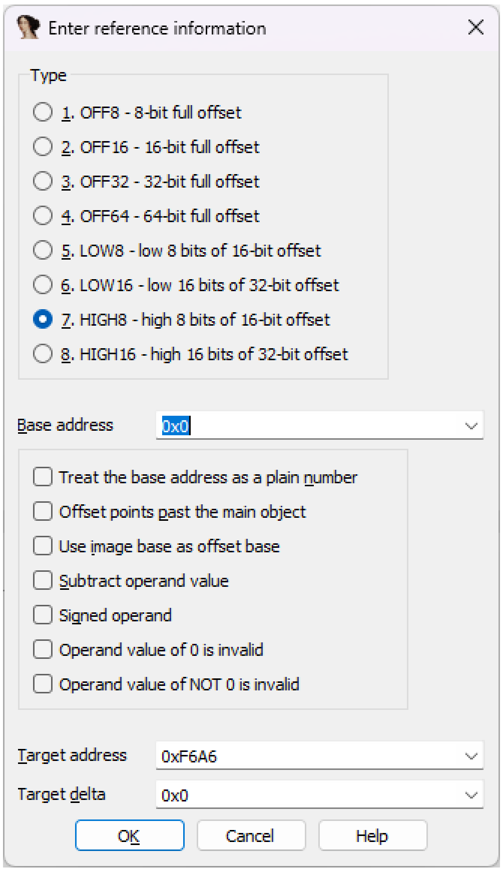
In fact, a better option is to use the high8/low8 offsets for the two instructions. Because each instruction provides only a part of the full offset, it alone cannot be used by IDA for calculating the full address which needs to be provided by the user.

#114: Split offsets

11 Nov 2022

https://hex-rays.com/blog/igors-tip-of-the-week-114-split-offsets/

R2 provides the top 8 bits of the address, so we should use the `HIGH8` offset type for it. We also need to fill in the full address (`0xF6A6`) in the Target address field. Base address should be reset to 0.



Enter reference information

Type

○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
○ 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
● 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset

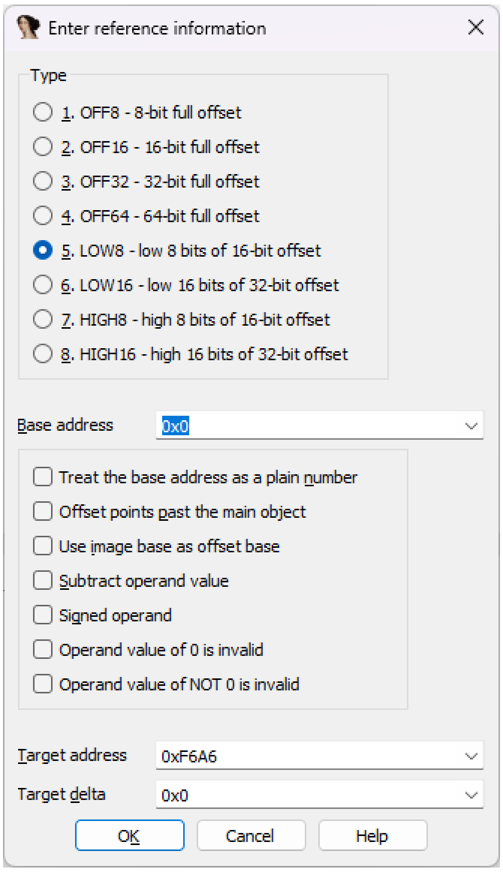
Base address: 0x0

□ Treat the base address as a plain number
□ Offset points past the main object
□ Use image base as offset base
□ Subtract operand value
□ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address: 0xF6A6

Target delta: 0x0

For R1, `LOW8` and the same target can be used:



Enter reference information

Type

○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
○ 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
● 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset

Base address: 0x0

□ Treat the base address as a plain number
□ Offset points past the main object
□ Use image base as offset base
□ Subtract operand value
□ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address: 0xF6A6

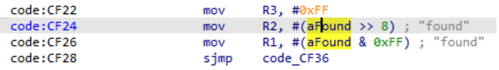
Target delta: 0x0

#114: Split offsets

11 Nov 2022

https://hex-rays.com/blog/igors-tip-of-the-week-114-split-offsets/

After applying both offsets, IDA displays them using matching assembler operators:



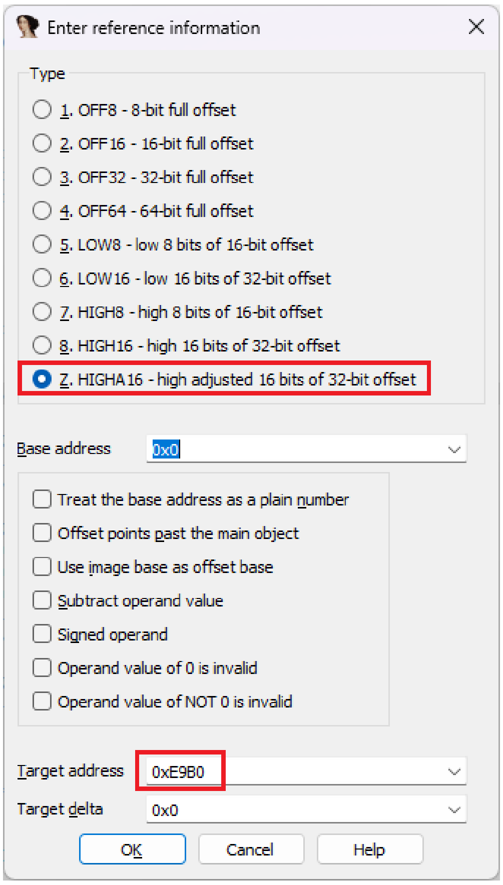
```
code:CF22                mov     R3, #0xFF
code:CF24                mov     R2, #(aFound >> 8) ; "found"
code:CF26                mov     R1, #(aFound & 0xFF) ; "found"
code:CF28                sjmp    code_CF36
```

RISC processors

RISC processors often use fixed-width instructions and may not be able to reach the full range of the address space with the limited space for the immediate operand in the instruction. This include SPARC, MIPS, PowerPC and some others. As an example, let's look at this PowerPC VLE snippet:

```
seg001:0000C156         e_lis     r3, 1 # Load Immediate Shifted
seg001:0000C15A         e_add16i  r3, r3, -0x1650 # 0xE9B0
seg001:0000C15E         se_mtlr   r3
seg001:0000C160         se_blrl
```

The code calculates an address of a function in `r3` and then calls it. IDA helpfully shows the final address in a comment, but we can also use custom offsets to represent them nicely. For the `e_add16i` instruction, we can use the `LOW16` type, as expected, but in case of e_lis, the processor-specific type `HIGHA16` should be used instead of `HIGH16`. This is because the low 16 bits are used here not as-is but as a sign-extened addend, with the high 16 bits of the final address becoming 0 after the addition (0x10000-0x1650=0xE9B0).



Enter reference information

Type

○ 1. OFF8 - 8-bit full offset
○ 2. OFF16 - 16-bit full offset
○ 3. OFF32 - 32-bit full offset
○ 4. OFF64 - 64-bit full offset
○ 5. LOW8 - low 8 bits of 16-bit offset
○ 6. LOW16 - low 16 bits of 32-bit offset
○ 7. HIGH8 - high 8 bits of 16-bit offset
○ 8. HIGH16 - high 16 bits of 32-bit offset
● Z. HIGHA16 - high adjusted 16 bits of 32-bit offset

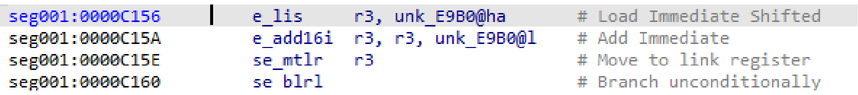
Base address: 0x0

□ Treat the base address as a plain number
□ Offset points past the main object
□ Use image base as offset base
□ Subtract operand value
□ Signed operand
□ Operand value of 0 is invalid
□ Operand value of NOT 0 is invalid

Target address: 0xE9B0

Target delta: 0x0

After converting both parts, IDA uses special assembler operators to show the final address:



```
seg001:0000C156      e_lis     r3, unk_E9B0@ha       # Load Immediate Shifted
seg001:0000C15A      e_add16i  r3, r3, unk_E9B0@l    # Add Immediate
seg001:0000C15E      se_mtlr   r3                    # Move to link register
seg001:0000C160      se_blrl                         # Branch unconditionally
```

Now we can go to the target and create a function there.

Note: specifically for PowerPC, IDA will automatically convert such sequences to offset expression if the target address exists and has instructions or data. But the manual approach can still be useful for other processors or complex situations (for example, the two instructions are too far apart).

[1] https://hex-rays.com/blog/igors-tip-of-the-week-110-self-relative-offsets/
[2] https://www.keil.com/support/man/docs/c51/c51_le_genptrs.htm
[3] https://hex-rays.com/blog/igors-tip-of-the-week-105-offsets-with-custom-base/

#115: Set callee address

18 Nov 2022

https://hex-rays.com/blog/igors-tip-of-the-week-115-set-callee-address/

Cross-references[1] is one of the most useful features of IDA. For example, they allow you to see where a particular function is being called or referenced from, helping you to see how the function is used and understand its behavior better or discover potential bugs or vulnerabilities. For direct calls, IDA adds cross-references automatically, but in modern programs there are also many indirect calls which can't always be resolved at disassembly time. In such cases, it is useful to have an option to set the target call address manually.

Indirect call types

Most instruction sets have some kind of an indirect call instruction. The most common one uses a processor register which holds the address of the function to be called:

x86/x64 and ARM can use almost any general-purpose register:

```
call edi (x86)
call rax (x64)
BLX R12 (ARM32)
BLX R3
BLR X8 (ARM64)
```

PowerPC is more limited and has to use dedicated `ctr` or `lr` registers:

```
mtlr r12
blrl

mr r12, r9
mtctr r9
bctrl
```

in MIPS, in theory any register can be used, but binaries conforming to the standard PIC ABI tend to use the register t9:

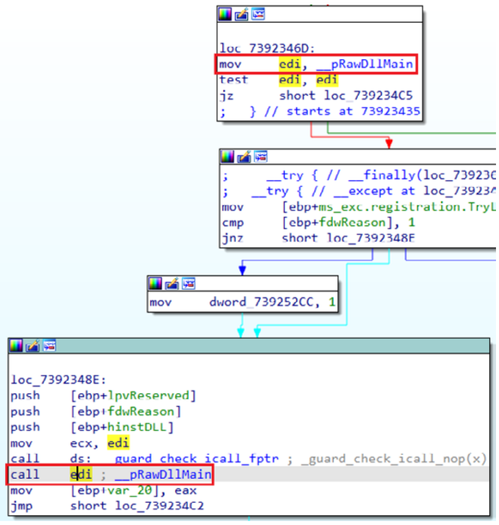
```
la      $t9, __cxa_finalize
lw      $a0, (_fdata - 0x111E0)($v0)  # void *
jalr    $t9 ; __cxa_finalize
```

In addition to simple register, some processors support more complex expressions. For example, on x86/x64 it is possible to use a register with offset, allowing to read a pointer value and jump to it in a single instruction:

```
call    dword ptr [eax+0Ch] (x86)
call    qword ptr [rax+98h] (x64)
```

Setting callee address

In some simple situations (e.g. the register is initialized shortly before the call), IDA is able to resolve it automatically and adds a comment with the target address, like in the MIPS example above, or this one:



```
loc_7392346D:
mov     edi, __pRawDllMain
test    edi, edi
jz      short loc_739234C5
;    } // starts at 73923435
```

```
;       __try { // __finally(loc_73923C
;     __try { // __except at loc_73923/
mov     [ebp+ms_exc.registration.TryL
cmp     [ebp+fdwReason], 1
jnz     short loc_7392348E
```

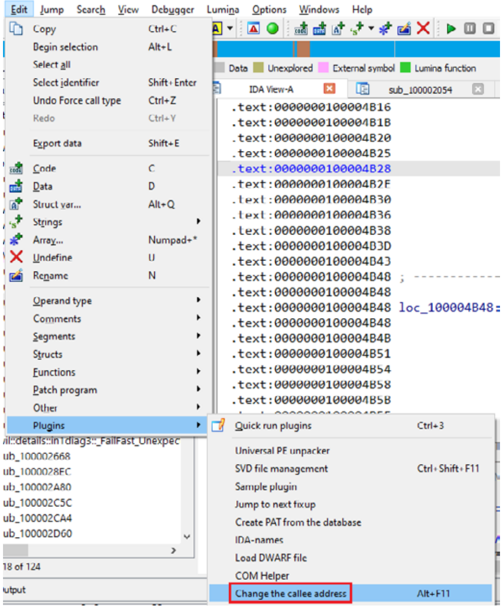
```
mov     dword_739252CC, 1
```

```
loc_7392348E:
push    [ebp+lpvReserved]
push    [ebp+fdwReason]
push    [ebp+hinstDLL]
mov     ecx, edi
call    ds:__guard_check_icall_fptr ; _guard_check_icall_nop(x)
call    edi ; __pRawDllMain
mov     [ebp+var_20], eax
jmp     short loc_739234C2
```

#115: Set callee address

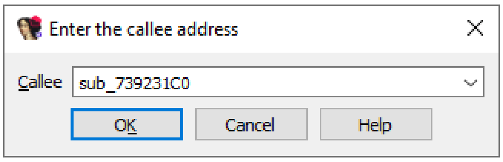
18 Nov 2022
https://hex-rays.com/blog/igors-tip-of-the-week-115-set-callee-address/

In more complicated situations, especially involving multiple memory dereferences or runtime calculations, it is possible to specify the target address manually. For this, use the standard plugin command available in Edit > Plugins > Change the callee address. The default shortcut is `Ctrl– F11`.



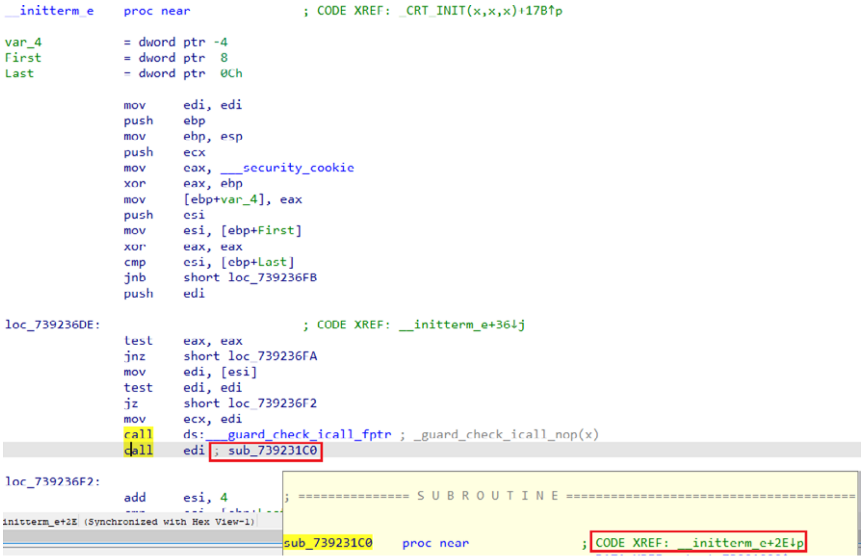
```
.text:0000000100004B16
.text:0000000100004B1B
.text:0000000100004B20
.text:0000000100004B25
.text:0000000100004B28
.text:0000000100004B2F
.text:0000000100004B30
.text:0000000100004B36
.text:0000000100004B38
.text:0000000100004B3D
.text:0000000100004B43
.text:0000000100004B48 ; -----------
.text:0000000100004B48
.text:0000000100004B48 loc_100004B48:
.text:0000000100004B48
.text:0000000100004B4B
.text:0000000100004B51
.text:0000000100004B54
.text:0000000100004B58
.text:0000000100004B5B
```

The plugin will ask you to enter the target address (you can also use a function name):



Callee: sub_739231C0

The call instruction will gain a comment with the target address, as well as a cross-reference:



```
__initterm_e    proc near               ; CODE XREF: _CRT_INIT(x,x,x)+17B↑p

var_4           = dword ptr -4
First           = dword ptr  8
Last            = dword ptr  0Ch

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    ecx
                mov     eax, ___security_cookie
                xor     eax, ebp
                mov     [ebp+var_4], eax
                push    esi
                mov     esi, [ebp+First]
                xor     eax, eax
                cmp     esi, [ebp+Last]
                jnb     short loc_739236FB
                push    edi

loc_739236DE:                           ; CODE XREF: __initterm_e+36↓j
                test    eax, eax
                jnz     short loc_739236FA
                mov     edi, [esi]
                test    edi, edi
                jz      short loc_739236F2
                mov     ecx, edi
                call    ds:___guard_check_icall_fptr ; _guard_check_icall_nop(x)
                call    edi ; sub_739231C0

loc_739236F2:
                add     esi, 4
__initterm_e+2E (Synchronized with Hex View-1)

; =============== S U B R O U T I N E =======================================

sub_739231C0    proc near               ; CODE XREF: __initterm_e+2E↓p
```

Currently the plugin is implemented for x86/x64, ARM and MIPS. If you need to set or access this information programmatically, you can check how it works by consulting the source code in the SDK, under **`plugins/callee`**.

[1] https://hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/

#116: IDA startup files

25 Nov 2022

https://hex-rays.com/blog/igors-tip-of-the-week-116-ida-startup-files/

IDA's behavior and defaults can be configured using the Options[1] dialog, saved desktop layouts[2], or config files[3]. However, sometimes the behavior you need depends on something in the input file and can't be covered by a single option, or you may want IDA to do something additional after the file is loaded. Of course, there is always the possibility of making a plugin or a loader using IDA SDK or IDAPython, but it could be an overkill for simple situations. Instead, you can make use of several startup files used by IDA every time it loads a new file or even a previously saved database, and do the necessary work there.

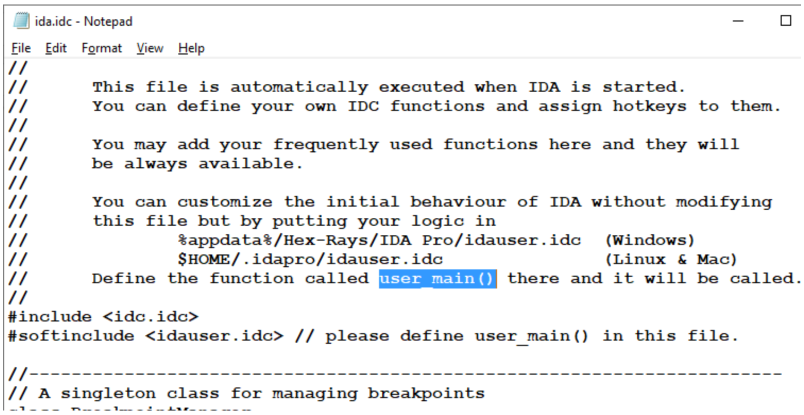
The following files can be used for such purpose:

ida.idc

This file in `idc` subdirectory if IDA's install is automatically loaded on each run of IDA and can be used to perform any actions you may need. The default implementation defines a utility class for managing breakpoints and a small helper function, but you can add there any other code you need. As an example, it has a commented call to change a global setting:

```
// uncomment this line to remove full paths in the debugger process options:
// set_inf_attr(INF_LFLAGS, LFLG_DBG_NOPATH|get_inf_attr(INF_LFLAGS));
```

Instead of editing the file itself (which may have been installed in a read-only location), you can create a file `idauser.idc` with a function `user_main()` and put it in the user directory[4]. If found, IDA will parse it and the main function of `ida.idc` will try to call `user_main()`. This feature allows you to keep the custom behaviour across multiple IDA installs and versions, without having to edit `ida.idc` every time.



```
//
//      This file is automatically executed when IDA is started.
//      You can define your own IDC functions and assign hotkeys to them.
//
//      You may add your frequently used functions here and they will
//      be always available.
//
//      You can customize the initial behaviour of IDA without modifying
//      this file but by putting your logic in
//              %appdata%/Hex-Rays/IDA Pro/idauser.idc  (Windows)
//              $HOME/.idapro/idauser.idc               (Linux & Mac)
//      Define the function called user_main() there and it will be called.
//
#include <idc.idc>
#softinclude <idauser.idc> // please define user_main() in this file.

//---------------------------------------------------------------------
// A singleton class for managing breakpoints
```

onload.idc

This file is similar to `ida.idc`, but is only executed for newly loaded files. In it you can, for example, do some additional parsing and formatting to augment the behavior of the default file loader(s). The default implementation detects when a DOS driver (EXE or COM file with `.sys` or `.drv` extension) is loaded and tries to format its header.

Similarly to `ida.idc`, instead of editing the file itself, you can create a file named `userload.idc` in the user directory and define a function `userload`.

```
//      If you want to add your own processing of newly created databases,
//      you may create a file named “userload.idc”:
//
//      #define USERLOAD_IDC
//      static userload(input_file,real_file,filetype) {
//              ... your processing here ...
//      }
//

#softinclude <userload.idc>

// Input parameteres:
//      input_file - name of loaded file
//      real_file  - name of actual file that contains the input file.
//                   usually this parameter is equal to input_file,
//                   but is different if the input file is extracted from
//                   an archive.
//      filetype   - type of loaded file. See FT_.. definitions in idc.idc
```

#116: IDA startup files

25 Nov 2022

https://hex-rays.com/blog/igors-tip-of-the-week-116-ida-startup-files/

idapythonrc.py

Unlike the previous examples, this a Python file, so it is only loaded if you have IDAPython installed and working. If the file is found in the user directory[5], it will be loaded and executed on startup of IDAPython, so you can put there any code to perform fine-tuning of IDA, add utility functions to be called from the CLI[6], or run any additional scripts.

Useful functions

Some functions which can be called from the startup files to configure IDA:

`get_inf_attr()`[7] / `set_inf_attr()`[8] / `set_flag()`[9]: read and set various flags controlling IDA's behavior. For example, `INF_AF` can be used to change various analysis options.

`process_config_directive()`[10]: change a setting using keyword=value syntax. Most settings from `ida.cfg` can be used, as well as some processor-specific or debugger-specific ones. A few examples:

- `process_config_directive("ABANDON_DATABASE=YES");`: do not save the database on exit. Please note that this setting has a side effect in that it disables most user actions which change the database, for example `MakeUnknown` (`U`) or `MakeCode` (`C`).
- `process_config_directive("PACK_DATABASE=2");`: set the default database packing option to "deflate";
- `process_config_directive("GRAPH_OPCODE_BYTES=4");`: enable display of opcode bytes in graph mode;
- for more examples, see `ida.cfg` (open it in any text editor).

[1] https://hex-rays.com/blog/igors-tip-of-the-week-25-disassembly-options/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-idas-user-directory-idausr/
[6] https://hex-rays.com/blog/igors-tip-of-the-week-73-output-window-and-logging/
[7] https://www.hex-rays.com/products/ida/support/idadoc/285.shtml
[8] https://www.hex-rays.com/products/ida/support/idadoc/285.shtml
[9] https://www.hex-rays.com/products/ida/support/idadoc/285.shtml
[10] https://www.hex-rays.com/products/ida/support/idadoc/642.shtml

#117: Reset pointer type

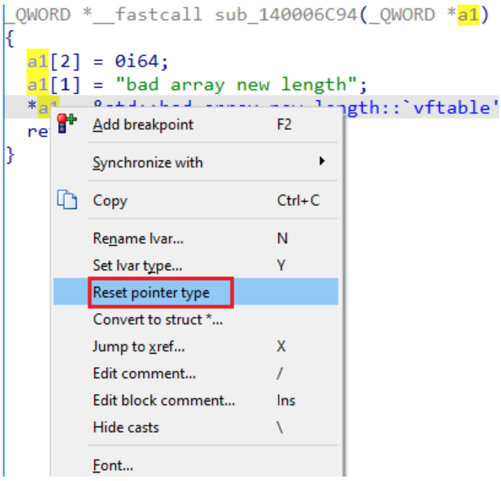
02 Dec 2022
https://hex-rays.com/blog/igors-tip-of-the-week-117-reset-pointer-type/

While currently (as of version 8.1) the Hex-Rays decompiler does not try to perform full type recovery, it does try to deduce some types based on operations done on the variables, or using the type information for the API calls from type libraries[1].

One simple type deduction performed by the decompiler is creation of typed pointers when a variable is being dereferenced, for example:

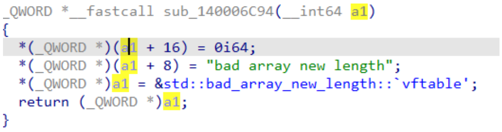
```
_QWORD *__fastcall sub_140006C94(_QWORD *a1)
{
  a1[2] = 0i64;
  a1[1] = “bad array new length”;
  *a1 = &std::bad_array_new_length::`vftable’;
  return a1;
}
```

Unfortunately, such conversions are not always correct, as can be seen in the example: we have a mix of integer and pointer elements in one array, so it’s more likely a structure. Also, due to C’s array indexing rules, the array indexes are multiplied by the element size (so, for example, `a1[2]` actually corresponds to the byte offset 16). If you prefer seeing “raw” offsets, you can change the variable’s type to a plain integer. This can, of course, be done by manually changing the variable’s type but there is a convenience command in the context menu which can be used to do it quickly:



```
_QWORD *__fastcall sub_140006C94(_QWORD *a1)
{
  a1[2] = 0i64;
  a1[1] = "bad array new length";
  *a1                        gth::`vftable'
  re
}
```

After resetting, the variable becomes a simple integer type and all dereferences now use explicit byte offsets and casts:



```
_QWORD *__fastcall sub_140006C94(__int64 a1)
{
  *(_QWORD *)(a1 + 16) = 0i64;
  *(_QWORD *)(a1 + 8) = "bad array new length";
  *(_QWORD *)a1 = &std::bad_array_new_length::`vftable';
  return (_QWORD *)a1;
}
```

Now you can, for example, create a structure corresponding to these accesses, or choose an existing one.

See also:
Hex-Rays Decompiler: Interactive operation[2]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[2] https://www.hex-rays.com/products/decompiler/manual/interactive.shtml

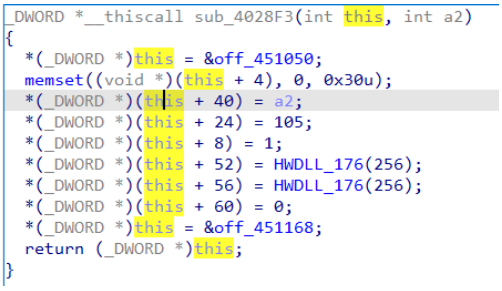
#118: Structure creation in the decompiler

09 Dec 2022

https://hex-rays.com/blog/igors-tip-of-the-week-118-structure-creation-in-the-decompiler/

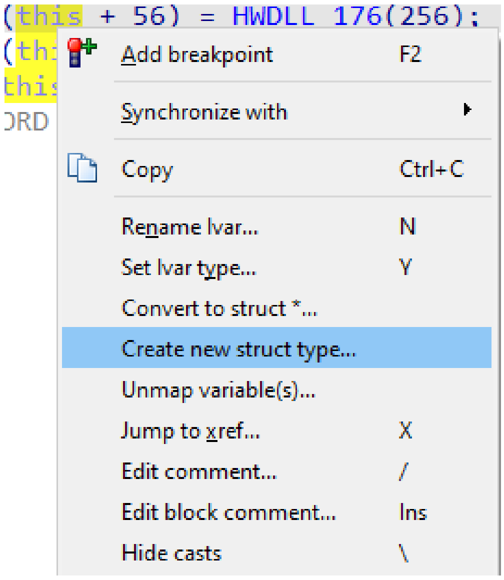
We've covered structure creation using disassembly or Local Types[1], but there is also a way of doing it from the decompiler, especially when dealing with unknown, custom types used by the program.

Whenever you see code dereferencing a variable with different offsets, it is likely a structure pointer and the function is accessing different fields of it.

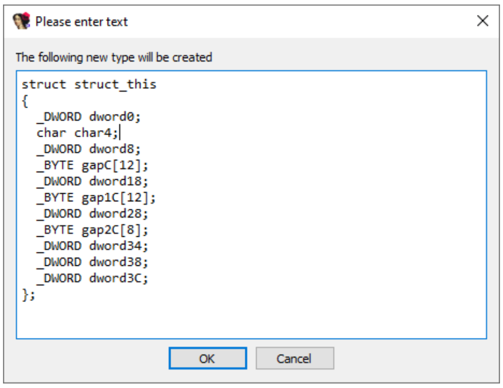


```
_DWORD *__thiscall sub_4028F3(int this, int a2)
{
  *(_DWORD *)this = &off_451050;
  memset((void *)(this + 4), 0, 0x30u);
  *(_DWORD *)(this + 40) = a2;
  *(_DWORD *)(this + 24) = 105;
  *(_DWORD *)(this + 8) = 1;
  *(_DWORD *)(this + 52) = HWDLL_176(256);
  *(_DWORD *)(this + 56) = HWDLL_176(256);
  *(_DWORD *)(this + 60) = 0;
  *(_DWORD *)this = &off_451168;
  return (_DWORD *)this;
}
```

You can, of course, create the structure manually and change the variable's type, but it is also possible to ask the decompiler to come up with a suitable layout. For this, use "Create new struct type..." from the context menu on the variable:



If you don't see the action, you may need to reset the pointer type[2] first. After you invoke it, the decompiler will analyze accesses to the variables and come up with a candidate structure type which matches them:



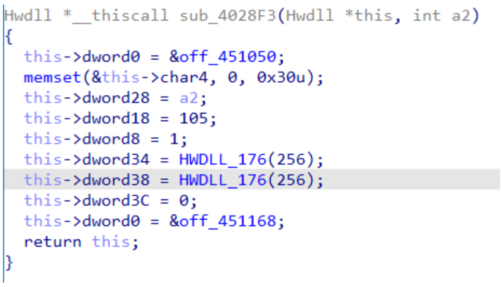
```
struct struct_this
{
  _DWORD dword0;
  char char4;
  _DWORD dword8;
  _BYTE gapC[12];
  _DWORD dword18;
  _BYTE gap1C[12];
  _DWORD dword28;
  _BYTE gap2C[8];
  _DWORD dword34;
  _DWORD dword38;
  _DWORD dword3C;
};
```

You can accept the suggestion as-is, or make any suitable adjustments (for example, change the structure name, or edit some of the fields). After confirming, the structure is added to Local Types and the variable is converted to the corresponding pointer type:

#118: Structure creation in the decompiler

09 Dec 2022

https://hex-rays.com/blog/igors-tip-of-the-week-118-structure-creation-in-the-decompiler/



```
Hwdll *__thiscall sub_4028F3(Hwdll *this, int a2)
{
  this->dword0 = &off_451050;
  memset(&this->char4, 0, 0x30u);
  this->dword28 = a2;
  this->dword18 = 105;
  this->dword8 = 1;
  this->dword34 = HWDLL_176(256);
  this->dword38 = HWDLL_176(256);
  this->dword3C = 0;
  this->dword0 = &off_451168;
  return this;
}
```

You can, of course, keep refining the structure as you continue with your analysis and discover how the fields are used in other functions and what they mean. Renaming fields can be done directly from the pseudocode view, while for adding or rearranging them you'll likely need to use Local Types or Structures window.

See also:
Hex-Rays interactive operation: Create new struct type[3]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-11-quickly-creating-structures/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-117-reset-pointer-type/
[3] https://www.hex-rays.com/products/decompiler/manual/cmd_new_struct.shtml

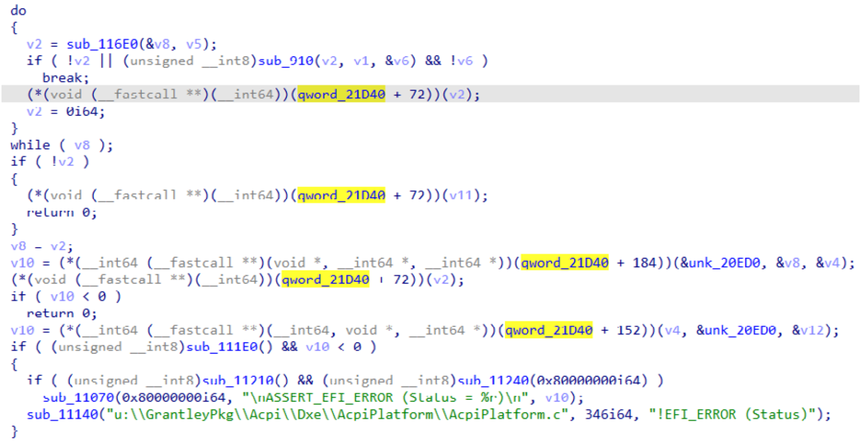
#119: Force call type

16 Dec 2022

https://hex-rays.com/blog/igors-tip-of-the-week-119-force-call-type/

When dealing with compile binary code, the decompiler lacks information present in the source code, such as function prototypes and so must guess it or rely on the information provided by the user (where its interactive features come handy).

One especially tricky situation is indirect calls: without exact information about the destination of the call, the decompiler can only try to analyze registers or stack slots initialized before the call and try to deduce the potential function prototype this way. For example, check this snippet from a UEFI module:



```
do
{
  v2 = sub_116E0(&v8, v5);
  if ( !v2 || (unsigned __int8)sub_910(v2, v1, &v6) && !v6 )
    break;
  (*(void (__fastcall **)(__int64))(qword_21D40 + 72))(v2);
  v2 = 0i64;
}
while ( v8 );
if ( !v2 )
{
  (*(void (__fastcall **)(__int64))(qword_21D40 + 72))(v11);
  return 0;
}
v8 = v2;
v10 = (*(__int64 (__fastcall **)(void *, __int64 *, __int64 *))(qword_21D40 + 184))(&unk_20ED0, &v8, &v4);
(*(void (__fastcall **)(__int64))(qword_21D40 + 72))(v2);
if ( v10 < 0 )
  return 0;
v10 = (*(__int64 (__fastcall **)(__int64, void *, __int64 *))(qword_21D40 + 152))(v4, &unk_20ED0, &v12);
if ( (unsigned __int8)sub_111E0() && v10 < 0 )
{
  if ( (unsigned __int8)sub_11210() && (unsigned __int8)sub_11240(0x80000000i64) )
    sub_11070(0x80000000i64, "\nASSERT_EFI_ERROR (Status = %r)\n", v10);
  sub_11140("u:\\GrantleyPkg\\Acpi\\Dxe\\AcpiPlatform\\AcpiPlatform.c", 346i64, "!EFI_ERROR (Status)");
}
```

For several indirect calls involving `qword_21D40`, the decompiler had to guess the arguments and add casts.

If we analyze the module from the entry point, we can find the place where the variable is initialized and figure out that it is, in fact, the standard UEFI global variable `gBS` of the type `EFI_BOOT_SERVICES *`:

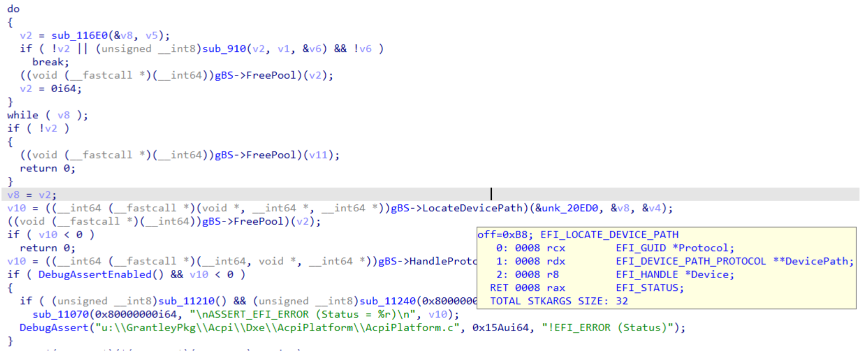
```
EFI_STATUS __fastcall UefiBootServicesTableLibConstructor(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE
*SystemTable)
{
  gImageHandle = ImageHandle;
  if ( DebugAssertEnabled() && !gImageHandle )
    DebugAssert(
      "u:\\MdePkg\\Library\\UefiBootServicesTableLib\\UefiBootServicesTableLib.c",
      0x33ui64,
      "gImageHandle != ((void *) 0)");
  gST = SystemTable;
  if ( DebugAssertEnabled() && !gST )
    DebugAssert(
      "u:\\MdePkg\\Library\\UefiBootServicesTableLib\\UefiBootServicesTableLib.c",
      0x39ui64,
      "gST != ((void *) 0)");
  // gBS was qword_21D40
  gBS = SystemTable->BootServices;
  if ( DebugAssertEnabled() && !gBS )
    DebugAssert(
      "u:\\MdePkg\\Library\\UefiBootServicesTableLib\\UefiBootServicesTableLib.c",
      0x3Fui64,
      "gBS != ((void *) 0)");
  return 0i64;
}
```

After renaming and changing the type of the global variable, the original function is slightly improved thanks to the type information from the standard UEFI type library:

#119: Force call type

16 Dec 2022

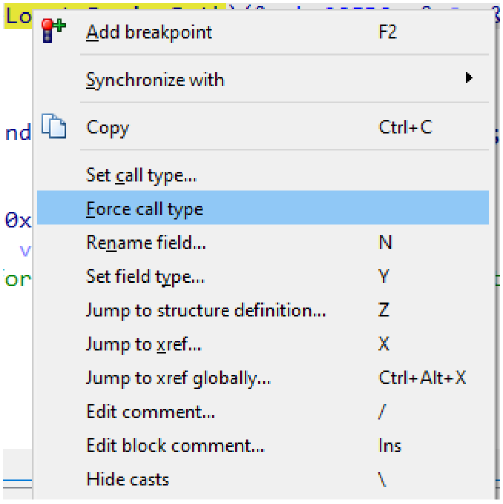
https://hex-rays.com/blog/igors-tip-of-the-week-119-force-call-type/



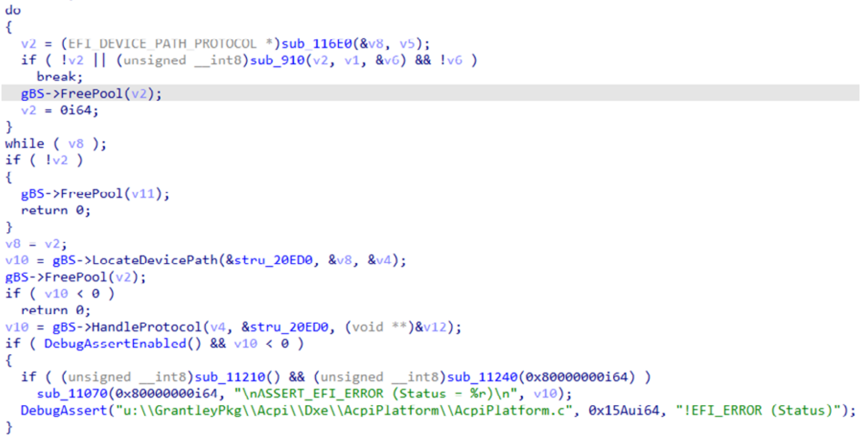
```
do
{
  v2 = sub_116E0(&v8, v5);
  if ( !v2 || (unsigned __int8)sub_910(v2, v1, &v6) && !v6 )
    break;
  ((void (__fastcall *)(__int64))gBS->FreePool)(v2);
  v2 = 0i64;
}
while ( v8 );
if ( !v2 )
{
  ((void (__fastcall *)(__int64))gBS->FreePool)(v11);
  return 0;
}
v8 = v2;
v10 = ((__int64 (__fastcall *)(void *, __int64 *, __int64 *))gBS->LocateDevicePath)(&unk_20ED0, &v8, &v4);
((void (__fastcall *)(__int64))gBS->FreePool)(v2);
if ( v10 < 0 )
  return 0;
v10 = ((__int64 (__fastcall *)(__int64, void *, __int64 *))gBS->HandleProtc
if ( DebugAssertEnabled() && v10 < 0 )
{
  if ( (unsigned __int8)sub_11210() && (unsigned __int8)sub_11240(0x8000000
    sub_11070(0x80000000i64, "\nASSERT_EFI_ERROR (Status = %r)\n", v10);
  DebugAssert("u:\\GrantleyPkg\\Acpi\\Dxe\\AcpiPlatform\\AcpiPlatform.c", 0x15Aui64, "!EFI_ERROR (Status)");
}
```

```
off=0xB8; EFI_LOCATE_DEVICE_PATH
   0: 0008 rcx        EFI_GUID *Protocol;
   1: 0008 rdx        EFI_DEVICE_PATH_PROTOCOL **DevicePath;
   2: 0008 r8         EFI_HANDLE *Device;
  RET 0008 rax        EFI_STATUS;
  TOTAL STKARGS SIZE: 32
```

Even though the decompiler now has prototypes of function pointers such as **`LocateDevicePath`** (shown in the pop-up) or **`FreePool`**, it has to add casts because the arguments which are passed to the calls do not match the prototype. To tell the decompiler to rely on the type information instead of guessing the arguments, use the command *Force call type* from the context menu:



When running the command on the indirect calls, the decompiler also uses the type information to update the types of the arguments (except those already set by the user), making the pseudocode much cleaner:



```
do
{
  v2 = (EFI_DEVICE_PATH_PROTOCOL *)sub_116E0(&v8, v5);
  if ( !v2 || (unsigned __int8)sub_910(v2, v1, &v6) && !v6 )
    break;
  gBS->FreePool(v2);
  v2 = 0i64;
}
while ( v8 );
if ( !v2 )
{
  gBS->FreePool(v11);
  return 0;
}
v8 = v2;
v10 = gBS->LocateDevicePath(&stru_20ED0, &v8, &v4);
gBS->FreePool(v2);
if ( v10 < 0 )
  return 0;
v10 = gBS->HandleProtocol(v4, &stru_20ED0, (void **)&v12);
if ( DebugAssertEnabled() && v10 < 0 )
{
  if ( (unsigned __int8)sub_11210() && (unsigned __int8)sub_11240(0x80000000i64) )
    sub_11070(0x80000000i64, "\nASSERT_EFI_ERROR (Status = %r)\n", v10);
  DebugAssert("u:\\GrantleyPkg\\Acpi\\Dxe\\AcpiPlatform\\AcpiPlatform.c", 0x15Aui64, "!EFI_ERROR (Status)");
}
```

See also:
Hex-Rays interactive operation: Force call type[1]

[1] https://www.hex-rays.com/products/decompiler/manual/cmd_force_call_type.shtml

#120: Set call type

23 Dec 2022

https://hex-rays.com/blog/igors-tip-of-the-week-120-set-call-type/

Previously we've described how to use available type info to make decompilation of calls more precise when you have type information[1], but there may be situations where you don't have it or the existing type info does not quite match the actual call arguments, and you still want to adjust the decompiler's guess.

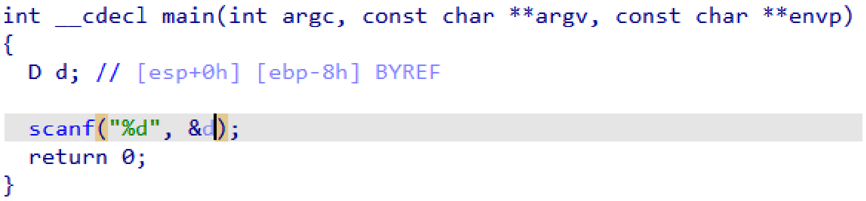
One common example is variadic functions (e.g. `printf`, `scanf` and several others from the C runtime library, as well as custom functions specific to the binary being analyzed). The decompiler knows about the standard C functions and tries to analyze the format string to guess the actually passed arguments. However, such guessing can still fail and show wrong arguments being passed.

For simple situations, adjusting variadic arguments[2] may work, but it's not always enough. For example, some calling conventions pass floating-point data in different registers from integers, so the decompiler needs to know which arguments are floating-point and which are not. You can, of course, change the prototype of the function to make the additional arguments explicit instead of variadic, but this affects all call sites instead of just the one you need.

Another difficulty can arise when dealing with the `scanf` family functions. Because the variadic arguments to such functions are usually passed by address, any variable type may be used for a specific format specifier. Consider the following example source code:

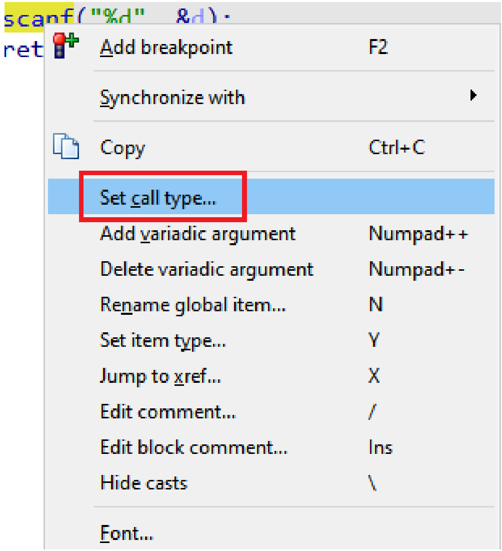
```
struct D
{
  int d;
  int e;
};


#include
int main()
{
 D d;
 scanf("%d", &d.d);
}
```

When we decompile the compiled binary, even after creating the struct and changing the local variable type, the following output is shown:

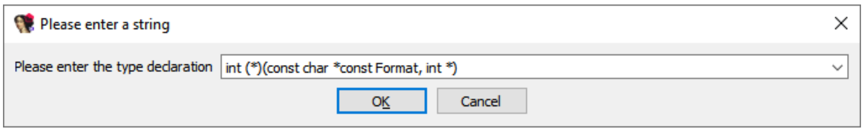


```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  D d; // [esp+0h] [ebp-8h] BYREF

  scanf("%d", &d);
  return 0;
}
```

We get `&d` instead of `&d.d` because `d` is situated at the very start of the structure so both expressions are equivalent on the binary level. To get the desired expression, we need to hint the decompiler that the extra argument is actually an `int` *. This can be done using the "Set call type…" action from the context menu on the call site:

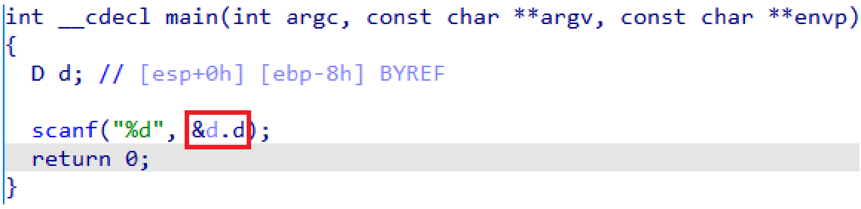


We can explicitly specify type of the extra argument:


Please enter a string

Please enter the type declaration: int (*)(const char *const Format, int *)

The decompiler takes it into account and uses the proper expression to match the new prototype:



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  D d; // [esp+0h] [ebp-8h] BYREF

  scanf("%d", &d.d);
  return 0;
}
```

See also: Hex-Rays interactive operation: Set call type[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-119-force-call-type/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-101-decompiling-variadic-function-calls/
[3] https://www.hex-rays.com/products/decompiler/manual/cmd_set_call_type.shtml

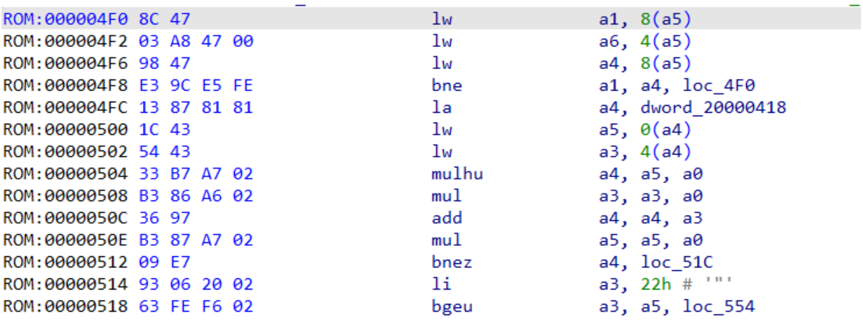
#121: Limiting search to an address range

30 Dec 2022

https://hex-rays.com/blog/igors-tip-of-the-week-121-limiting-search-to-an-address-range/

When performing a search[1] in IDA, it by default starts from the current position and continues up to the maximum address in the database (or to the minimal for searches "Up"). This works well enough for small to average files, but can get pretty slow for big ones, or especially in case of debugging where the database may include not just the input file but also multiple additional modules loaded at runtime.

To skip areas you're not interested in and improve the speed, you can limit the search to an address range. For this, IDA relies on selection. For example, consider this disassembly snippet:

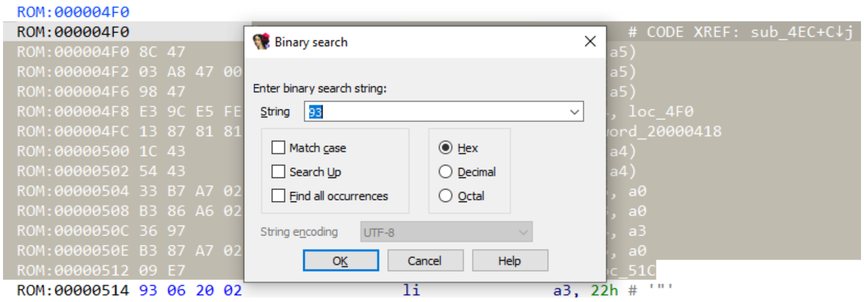


```
ROM:000004F0 8C 47                     lw             a1, 8(a5)
ROM:000004F2 03 A8 47 00               lw             a6, 4(a5)
ROM:000004F6 98 47                     lw             a4, 8(a5)
ROM:000004F8 E3 9C E5 FE               bne            a1, a4, loc_4F0
ROM:000004FC 13 87 81 81               la             a4, dword_20000418
ROM:00000500 1C 43                     lw             a5, 0(a4)
ROM:00000502 54 43                     lw             a3, 4(a4)
ROM:00000504 33 B7 A7 02               mulhu          a4, a5, a0
ROM:00000508 B3 86 A6 02               mul            a3, a3, a0
ROM:0000050C 36 97                     add            a4, a4, a3
ROM:0000050E B3 87 A7 02               mul            a5, a5, a0
ROM:00000512 09 E7                     bnez           a4, loc_51C
ROM:00000514 93 06 20 02               li             a3, 22h # '"'
ROM:00000518 63 FE F6 02               bgeu           a3, a5, loc_554
```

If you perform a binary search for the value 93, the instruction at 00000514 will be found:

```
Searching down CASE-INSENSITIVELY for binary pattern:
  93
Search completed. Found at 00000514.
```

However, if you select a range which does not include that address before invoking the search, the search will fail:



```
ROM:000004F0
ROM:000004F0                                  # CODE XREF: sub_4EC+C↓j
ROM:000004F0 8C 47                            a5)
ROM:000004F2 03 A8 47 00                      a5)
ROM:000004F6 98 47                            a5)
ROM:000004F8 E3 9C E5 FE                      , loc_4F0
ROM:000004FC 13 87 81 81                      ord_20000418
ROM:00000500 1C 43                            a4)
ROM:00000502 54 43                            a4)
ROM:00000504 33 B7 A7 02                      , a0
ROM:00000508 B3 86 A6 02                      , a0
ROM:0000050C 36 97                            , a3
ROM:0000050E B3 87 A7 02                      , a0
ROM:00000512 09 E7                            c_51C
ROM:00000514 93 06 20 02       li             a3, 22h # '"'
```

Binary search

Enter binary search string:

String 93

☐ Match case

☐ Search Up

☐ Find all occurrences

● Hex

○ Decimal

○ Octal

String encoding UTF-8

```
Searching down CASE-INSENSITIVELY for binary pattern:
  93
Search failed.
Command "AskBinaryText" failed
```

Selecting large areas with the mouse or by holding Shift can be quite tedious, so it may be more convenient to use the anchor selection[2]:

1. Move to the start or end of the intended selection and invoke Edit > Begin selection (or press `Alt-L`).
2. Navigate to the other end of the selection using any means (cursor keys, Jump actions, Functions or Sgments window, Navigation bar etc.).
3. Invoke the binary search command. The search will be performed in the selection only.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/

[2] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/

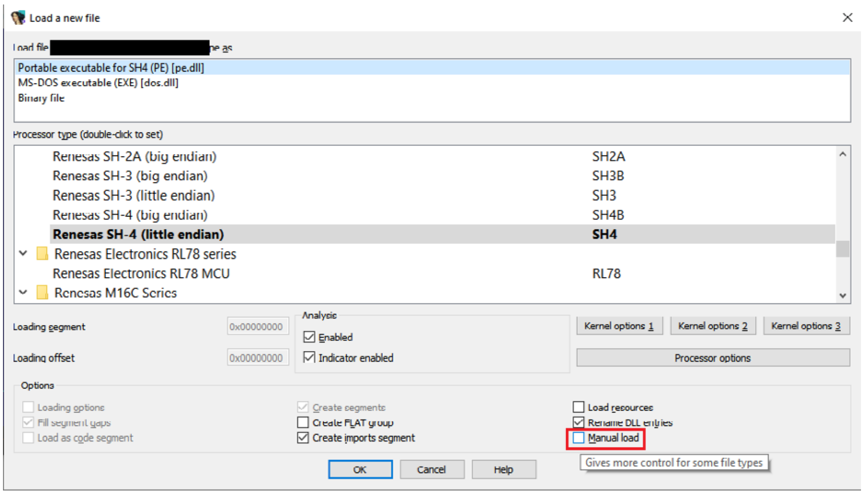
#122: Manual load

06 Jan 2023

https://hex-rays.com/blog/igors-tip-of-the-week-122-manual-load/

To save on analysis time and database size, by default IDA only tries to load relevant parts of the binary (e.g. those that are expected or known to contain code). However, there may be cases when you want to see more, or even everything the binary contains. You can always load the file as plain binary and mark it up manually, using IDA as a sort of a hybrid hex editor, but this way you lose the features handled by the built-in loaders such as names from the symbol table, automatic function boundaries from the file metadata and so on. So it may be interesting to have more granular control over the file loading process.

To support such scenarios, IDA offers the Manual load checkbox in the initial load dialog.



Load file [redacted] e as

- Portable executable for SH4 (PE) [pe.dll]
- MS-DOS executable (EXE) [dos.dll]
- Binary file

Processor type (double-click to set)

Renesas SH-2A (big endian)	SH2A
Renesas SH-3 (big endian)	SH3B
Renesas SH-3 (little endian)	SH3
Renesas SH-4 (big endian)	SH4B
Renesas SH-4 (little endian)	**SH4**
Renesas Electronics RL78 series	
Renesas Electronics RL78 MCU	RL78
Renesas M16C Series	

Loading segment 0x00000000

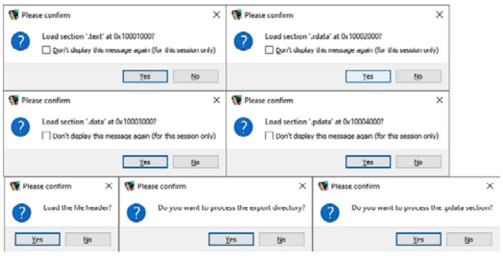
Loading offset 0x00000000

Analysis
☑ Enabled
☑ Indicator enabled

Options
□ Loading options
☑ Fill segment gaps
□ Load as code segment
☑ Create segments
□ Create FLAT group
☑ Create imports segment
□ Load resources
☑ Rename DLL entries
□ Manual load

Gives more control for some file types

What happens when the option is checked depends on the loader. For example, the PE loader may allow you to pick another load base (image base), choose which sections to load, and whether to parse some optional metadata which could, for example, be corrupted and result in bad analysis.



Load section '.text' at 0x10001000?
□ Don't display this message again (for this session only)

Load section '.rdata' at 0x10002000?
□ Don't display this message again (for this session only)

Load section '.data' at 0x10003000?
□ Don't display this message again (for this session only)

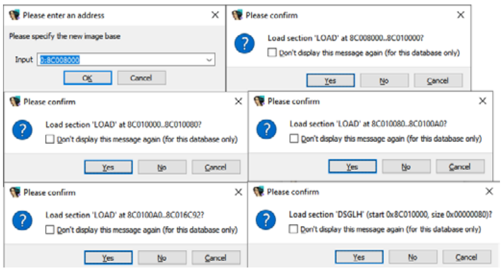
Load section '.pdata' at 0x10004000?
□ Don't display this message again (for this session only)

Load the file header?

Do you want to process the export directory?

Do you want to process the .pdata section?

The ELF loader behaves in a similar manner



Please specify the new image base

Input 0:8C008000

Load section 'LOAD' at 8C008000..8C010000?
□ Don't display this message again (for this database only)

Load section 'LOAD' at 8C010000..8C010080?
□ Don't display this message again (for this database only)

Load section 'LOAD' at 8C010080..8C0100A0?
□ Don't display this message again (for this database only)

Load section 'LOAD' at 8C0100A0..8C016C92?
□ Don't display this message again (for this database only)

Load section 'DSGLH' (start 0x8C010000, size 0x00000080)?
□ Don't display this message again (for this database only)

If you want IDA to always load all PE sections, you can edit cfg/pe.cfg and set the option PE_LOAD_ALL_SECTIONS:

```
// Always load all sections of a PE file?
// If no, sections like .reloc and .rsrc are skipped

PE_LOAD_ALL_SECTIONS = YES
```

See also: IDA Help: Load file dialog[1]

[1] https://www.hex-rays.com/products/ida/support/idadoc/242.shtml

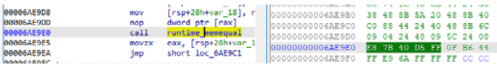
#123: Opcode bytes

13 Jan 2023

https://hex-rays.com/blog/igors-tip-of-the-week-123-opcode-bytes/

When disassembling, you are probably more interested in seeing the code (disassembly or pseudocode) rather than the raw file data, but there may be times you need to see what actually lies behind the instructions.

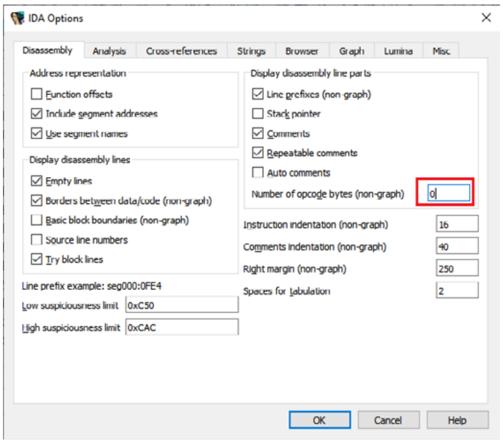
One option is to use the Hex View[1], possibly docked and synchronized with IDA View.



```
00006AE9D8                mov     [rsp+28h+var_18], r
00006AE9DD                nop     dword ptr [rax]
00006AE9E0                call    runtime_memequal
00006AE9E5                movzx   eax, [rsp+28h+var_1
00006AE9EA                jmp     short loc_6AE9C1
```

```
00000000006AE9B0  38 48 8B 5A 20 48 8B 40
00000000006AE9C0  C0 88 44 24 40 48 8B 6C
00000000006AE9D0  89 04 24 40 89 5C 24 00
00000000006AE9E0  E8 7B 40 D5 FF 0F B6 44
00000000006AE9F0  FF E9 6A FF FF FF CC CC
```

But probably a simpler solution is the disassembly option[2] *Number of opcode bytes*.



IDA Options

Disassembly | Analysis | Cross-references | Strings | Browser | Graph | Lumina | Misc

Address representation
- □ Function offsets
- ☑ Include segment addresses
- ☑ Use segment names

Display disassembly lines
- ☑ Empty lines
- ☑ Borders between data/code (non-graph)
- □ Basic block boundaries (non-graph)
- □ Source line numbers
- ☑ Try block lines

Line prefix example: seg000:0FE4

Low suspiciousness limit: 0xC50

High suspiciousness limit: 0xCAC

Display disassembly line parts
- ☑ Line prefixes (non graph)
- □ Stack pointer
- ☑ Comments
- ☑ Repeatable comments
- □ Auto comments

Number of opcode bytes (non-graph): 0

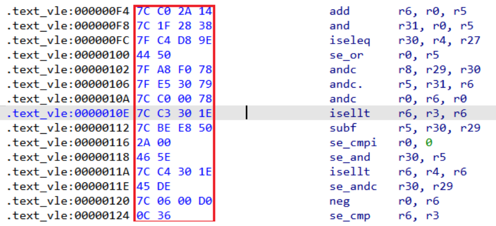
Instruction indentation (non-graph): 16

Comments indentation (non-graph): 40

Right margin (non-graph): 250

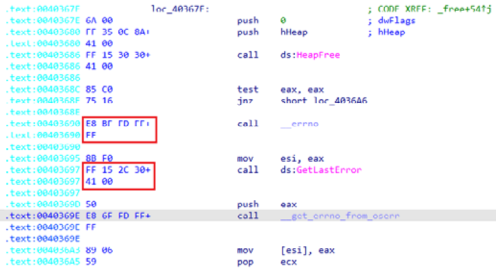
Spaces for tabulation: 2

By setting it to a non-zero value, IDA will use the specified number of columns to display the bytes of the instructions at the start of the disassembly line.



```
.text_vle:000000F4 7C C0 2A 14          add     r6, r0, r5
.text_vle:000000F8 7C 1F 28 38          and     r31, r0, r5
.text_vle:000000FC 7F C4 D8 9E          iseleq  r30, r4, r27
.text_vle:00000100 44 50                se_or   r0, r5
.text_vle:00000102 7F A8 F0 78          andc    r8, r29, r30
.text_vle:00000106 7F E5 30 79          andc.   r5, r31, r6
.text_vle:0000010A 7C C0 00 78          andc    r0, r6, r0
.text_vle:0000010E 7C C3 30 1E          isellt  r6, r3, r6
.text_vle:00000112 7C BE E8 50          subf    r5, r30, r29
.text_vle:00000116 2A 00                se_cmpi r0, 0
.text_vle:00000118 46 5E                se_and  r30, r5
.text_vle:0000011A 7C C4 30 1E          isellt  r6, r4, r6
.text_vle:0000011E 45 DE                se_andc r30, r29
.text_vle:00000120 7C 06 00 D0          neg     r0, r6
.text_vle:00000124 0C 36                se_cmp  r6, r3
```

If the instruction is longer than the specified number of bytes, extra lines will be used to display the remainder of the opcode:



```
.text:0040367F             loc_40367F:                           ; CODE XREF: _free+54↑j
.text:0040367F 6A 00                     push    0               ; dwFlags
.text:00403680 FF 35 0C 8A+              push    hHeap           ; hHeap
.text:00403680 41 00
.text:00403686 FF 15 30 30+              call    ds:HeapFree
.text:00403686 41 00
.text:00403686
.text:0040368C 85 C0                     test    eax, eax
.text:0040368E 75 16                     jnz     short loc_4036A6
.text:0040368E
.text:00403690 E8 BF FD FF+              call    __errno
.text:00403690 FF
.text:00403690
.text:00403695 8B F0                     mov     esi, eax
.text:00403697 FF 15 2C 30+              call    ds:GetLastError
.text:00403697 41 00
.text:00403697
.text:0040369D 50                        push    eax
.text:0040369E E8 6F FD FF+              call    __get_errno_from_oserr
.text:0040369E FF
.text:0040369E
.text:004036A3 89 06                     mov     [esi], eax
.text:004036A5 59                        pop     ecx
```

If you prefer to have IDA simply truncate the long opcodes instead of using extra lines, specify a negative value (e.g. -4).

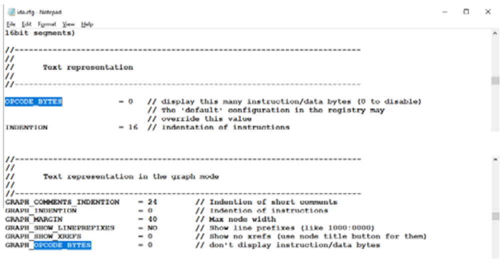
Showing opcode bytes by default

If you prefer to always see opcode bytes, you can use the `OPCODE_BYTES` setting in `ida.cfg` (either the one in your IDA install, or the override in user directory[3]). This enables opcode bytes in the text view only; for the graph view use the setting `GRAPH_OPCODE_BYTES`.

#123: Opcode bytes

13 Jan 2023

https://hex-rays.com/blog/igors-tip-of-the-week-123-opcode-bytes/



```
16bit segments)

//-----------------------------------------------------------------------
//
//      Text representation
//
//-----------------------------------------------------------------------

OPCODE_BYTES            = 0   // display this many instruction/data bytes (0 to disable)
                              // The 'default' configuration in the registry may
                              // override this value
INDENTION               = 16  // Indentation of instructions


//-----------------------------------------------------------------------
//
//      Text representation in the graph mode
//
//-----------------------------------------------------------------------
GRAPH_COMMENTS_INDENTION    = 24        // Indention of short comments
GRAPH_INDENTION             = 0         // Indention of instructions
GRAPH_MARGIN                = 40        // Max node width
GRAPH_SHOW_LINEPREFIXES     = NO        // Show line prefixes (like 1000:0000)
GRAPH_SHOW_XREFS            = 0         // Show no xrefs (use node title button for them)
GRAPH_OPCODE_BYTES          = 0         // don't display instruction/data bytes
```

Another possibility is set up the opcode bytes (and other disassembly options) as you like and save the current desktop layout as default[4]; it will be used for all new databases.

[1] https://www.hex-rays.com/products/ida/support/idadoc/605.shtml
[2] https://hex-rays.com/blog/igors-tip-of-the-week-38-hex-view/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-25-disassembly-options/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/

#124: Scripting examples

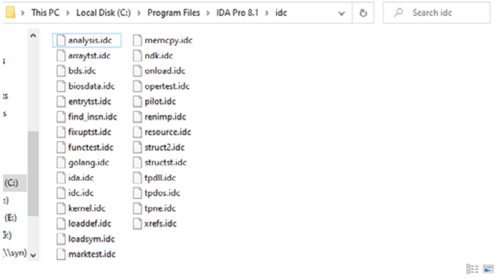
20 Jan 2023

https://hex-rays.com/blog/igors-tip-of-the-week-124-scripting-examples/

Although IDA was initially created for interactive usage and tries to automate as much of the tedious parts of RE as possible, it still cannot do everything for you and doing the still necessary work manually can take a long time. To alleviate this, IDA ships with IDC and IDAPython scripting engines, which can be used for automating some repetitive tasks. But it can be difficult to know where to start, so let's see where you can find some examples to get started.

IDC samples

Although IDC is quite old fashioned, it has the advantage of being built-in into IDA and does not require any additional software. It is also the only scripting language available in IDA Free[1]. For some sample IDC scripts, see the **idc** directory in IDA's install location:



This PC › Local Disk (C:) › Program Files › IDA Pro 8.1 › idc
Search idc
(C:)
(E:)

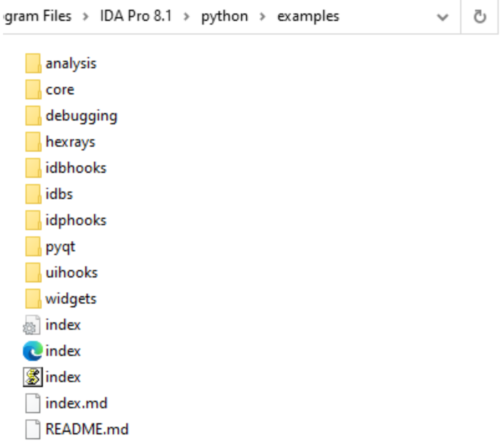
analysis.idc
arraytst.idc
bds.idc
biosdata.idc
entrytst.idc
find_insn.idc
fixuptst.idc
functest.idc
golang.idc
ida.idc
idc.idc
kernel.idc
loaddef.idc
loadsym.idc
marktest.idc
memcpy.idc
ndk.idc
onload.idc
opertest.idc
pilot.idc
renimp.idc
resource.idc
struct2.idc
structst.idc
tpdll.idc
tpdos.idc
tpne.idc
xrefs.idc

Please note that some of these files are not stand-alone scripts but are used by IDA for various tasks such as customized startup actions[2] (**ida.idc**, **onload.idc**) or batch analysis (**analysis.idc**).

A few user-contributed scripts are also available under the "User contributions" section in our Download center[3]. Note that due to their age and the big API refactoring[4] which unified IDA API and IDC, some of them may need adjustments to run in recent IDA versions.

IDAPython examples

IDAPython project had examples from the beginning, and you can find them in the source repository[5], but we're also shipping them with IDA, in the **python/examples** directory.



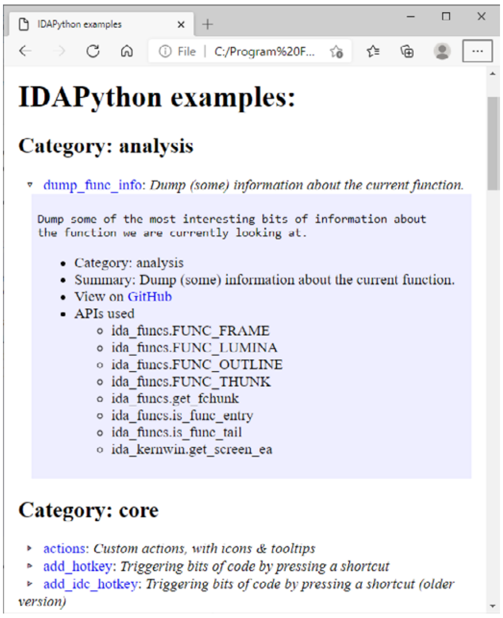
analysis
core
debugging
hexrays
idbhooks
idbs
idphooks
pyqt
uihooks
widgets
index
index
index
index.md
README.md

The provided **index.html** can be opened in a browser to see the list of the examples with short descriptions and also a list of used IDAPython APIs/keywords to help you find examples of a specific API's usage.

#124: Scripting examples

20 Jan 2023

https://hex-rays.com/blog/igors-tip-of-the-week-124-scripting-examples/



IDAPython examples:

Category: analysis

- dump_func_info: *Dump (some) information about the current function.*

```
Dump some of the most interesting bits of information about
the function we are currently looking at.
```

- Category: analysis
- Summary: Dump (some) information about the current function.
- View on GitHub
- APIs used
  - ida_funcs.FUNC_FRAME
  - ida_funcs.FUNC_LUMINA
  - ida_funcs.FUNC_OUTLINE
  - ida_funcs.FUNC_THUNK
  - ida_funcs.get_fchunk
  - ida_funcs.is_func_entry
  - ida_funcs.is_func_tail
  - ida_kernwin.get_screen_ea

Category: core

- actions: *Custom actions, with icons & tooltips*
- add_hotkey: *Triggering bits of code by pressing a shortcut*
- add_idc_hotkey: *Triggering bits of code by pressing a shortcut (older version)*

There are also countless examples of IDAPython scripts and plugins created by our users. Some of then can be found on our plugin contest pages[6] and plugin repository[7], while even more might be found on code-sharing websites (GitHub, GitLab etc.), or individual authors' websites and blogs. Oftentimes, searching for an API name on the Web can bring you to examples of its usage.

In addition to the examples made just for demonstration purposes, there are a few Python-based loaders and processors modules shipped with IDA. They can be found by looking for `.py` files under `loader` and `procs` directories of IDA.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-116-ida-startup-files/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-08-batch-mode-under-the-hood/
[3] https://hex-rays.com/download-center/
[4] https://hex-rays.com/products/ida/news/7_0/docs/api70_porting_guide/
[5] https://github.com/idapython/src/tree/master/examples
[6] https://hex-rays.com/contests/
[7] https://plugins.hex-rays.com/

#125: Structure field representation

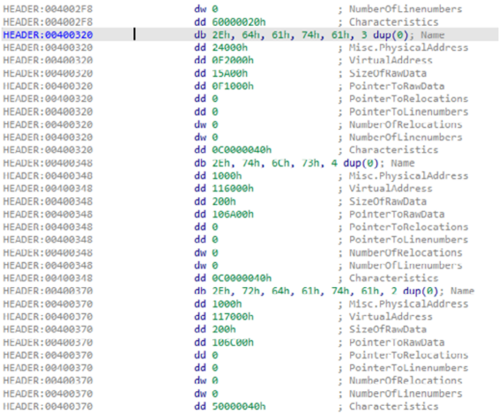
27 Jan 2023

https://hex-rays.com/blog/igors-tip-of-the-week-125-structure-fields-representation/

When dealing with structure instances in disassembly, sometimes you may want to change how IDA displays them, but how to do it is not always obvious. Let's have a look at some examples.

Win32 section headers

Let's say you have loaded the PE file header using manual load[1], or found an embedded PE file in your binary, and want to format its PE header nicely. Thanks to the standard type libraries[2], you can import standard Win32 structures such as `IMAGE_NT_HEADERS`[3] or `IMAGE_SECTION_HEADER`[4] and apply them to the header area:

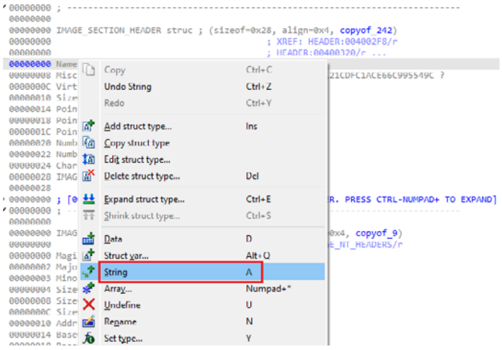


```
HEADER:004002F8                 dw 0                    ; NumberOfLinenumbers
HEADER:004002F8                 dd 60000020h            ; Characteristics
HEADER:00400320                 db 2Eh, 64h, 61h, 74h, 61h, 3 dup(0); Name
HEADER:00400320                 dd 24000h               ; Misc.PhysicalAddress
HEADER:00400320                 dd 0F2000h              ; VirtualAddress
HEADER:00400320                 dd 15A00h               ; SizeOfRawData
HEADER:00400320                 dd 0F1000h              ; PointerToRawData
HEADER:00400320                 dd 0                    ; PointerToRelocations
HEADER:00400320                 dd 0                    ; PointerToLinenumbers
HEADER:00400320                 dw 0                    ; NumberOfRelocations
HEADER:00400320                 dw 0                    ; NumberOfLinenumbers
HEADER:00400320                 dd 0C0000040h           ; Characteristics
HEADER:00400348                 db 2Eh, 74h, 6Ch, 73h, 4 dup(0); Name
HEADER:00400348                 dd 1000h                ; Misc.PhysicalAddress
HEADER:00400348                 dd 116000h              ; VirtualAddress
HEADER:00400348                 dd 200h                 ; SizeOfRawData
HEADER:00400348                 dd 106A00h              ; PointerToRawData
HEADER:00400348                 dd 0                    ; PointerToRelocations
HEADER:00400348                 dd 0                    ; PointerToLinenumbers
HEADER:00400348                 dw 0                    ; NumberOfRelocations
HEADER:00400348                 dw 0                    ; NumberOfLinenumbers
HEADER:00400348                 dd 0C0000040h           ; Characteristics
HEADER:00400370                 db 2Eh, 72h, 64h, 61h, 74h, 61h, 2 dup(0); Name
HEADER:00400370                 dd 1000h                ; Misc.PhysicalAddress
HEADER:00400370                 dd 117000h              ; VirtualAddress
HEADER:00400370                 dd 200h                 ; SizeOfRawData
HEADER:00400370                 dd 106C00h              ; PointerToRawData
HEADER:00400370                 dd 0                    ; PointerToRelocations
HEADER:00400370                 dd 0                    ; PointerToLinenumbers
HEADER:00400370                 dw 0                    ; NumberOfRelocations
HEADER:00400370                 dw 0                    ; NumberOfLinenumbers
HEADER:00400370                 dd 50000040h            ; Characteristics
```

However, because the `Name` field is declared simply as a `BYTE` array in the original structure, IDA shows them as bytes instead of nice readable string. Without the struct, we could use the Create string (`A`) command, but it is also possible to show the string as part of the structure instance.

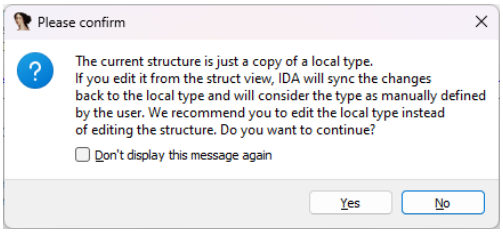
Changing structure field representation

To change how a specific fiield should be formatted in the disassembly, go to it in the structure definition in the Structures window and use Edit or the context menu. For example, use the String (`A`) action to have IDA format the Name byte array as a string.



```
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 IMAGE_SECTION_HEADER struc ; (sizeof=0x28, align=0x4, copyof_242)
00000000                                         ; XREF: HEADER:004002F8/r
00000000                                         ; HEADER:00400320/r ...
00000000 Name
00000008 Misc                                    21CDFC1ACE66C995549C ?
0000000C Virt
00000010 Size
00000014 Poin
00000018 Poin
0000001C Poin
00000020 Numb
00000022 Numb
00000024 Char
00000028 IMAG
00000028
00000000 ; [0                                    R. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ;
00000000
00000000 IMAG                                    0x4, copyof_9)
00000000                                         E_NT_HEADERS/r
00000000 Magi
00000002 Majo
00000003 Mino
00000004 Size
00000008 Size
0000000C Size
00000010 Addr
00000014 Base
```

When you edit an imported structure for the first time, you may get this warning:



Please confirm

The current structure is just a copy of a local type.
If you edit it from the struct view, IDA will sync the changes
back to the local type and will consider the type as manually defined
by the user. We recommend you to edit the local type instead
of editing the structure. Do you want to continue?

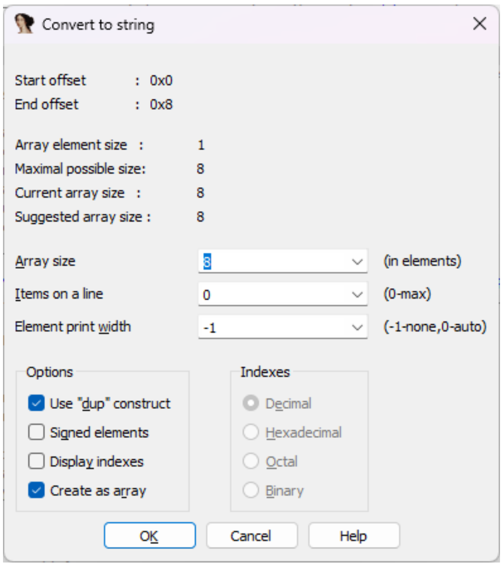
☐ Don't display this message again

Because the field type representation cannot be specified in Local Types, we have to edit the structure, so answer Yes to continue. A dialog to specify the string length will be displayed, just confirm it:

#125: Structure field representation

27 Jan 2023

https://hex-rays.com/blog/igors-tip-of-the-week-125-structure-fields-representation/



Convert to string

Start offset : 0x0
End offset : 0x8

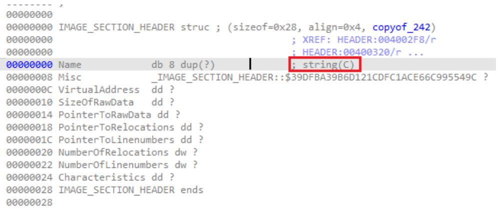
Array element size : 1
Maximal possible size: 8
Current array size : 8
Suggested array size : 8

Array size: 8 (in elements)
Items on a line: 0 (0-max)
Element print width: -1 (-1-none,0-auto)

Options
- ☑ Use "dup" construct
- ☐ Signed elements
- ☐ Display indexes
- ☑ Create as array

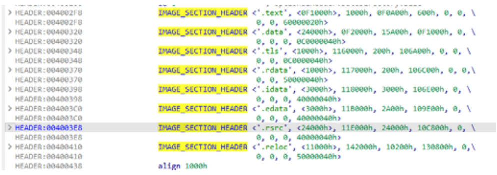
Indexes
- ● Decimal
- ○ Hexadecimal
- ○ Octal
- ○ Binary

The field will gain a comment indicating that the array is now a string:



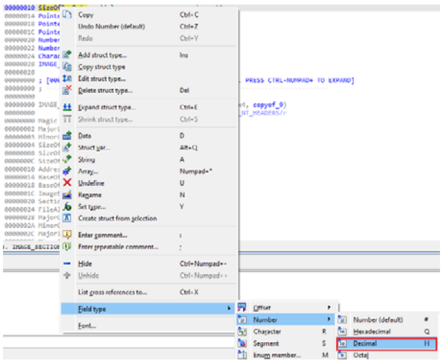
```
00000000
00000000 IMAGE_SECTION_HEADER struc ; (sizeof=0x28, align=0x4, copyof_242)
00000000                                         ; XREF: HEADER:004002F8/r
00000000                                         ; HEADER:00400320/r ...
00000000 Name            db 8 dup(?)             ; string(C)
00000008 Misc            _IMAGE_SECTION_HEADER::$39DFBA39B6D121CDFC1ACE66C995549C ?
0000000C VirtualAddress  dd ?
00000010 SizeOfRawData   dd ?
00000014 PointerToRawData dd ?
00000018 PointerToRelocations dd ?
0000001C PointerToLinenumbers dd ?
00000020 NumberOfRelocations dw ?
00000022 NumberOfLinenumbers dw ?
00000024 Characteristics dd ?
00000028 IMAGE_SECTION_HEADER ends
00000028
```

And the struct instances in the binary will now show the first field as a string:



```
> HEADER:004002F8          IMAGE_SECTION_HEADER <'.text', <0F1000h>, 1000h, 0F0A00h, 600h, 0, 0, \
  HEADER:004002F8                                0, 0, 60000020h>
> HEADER:00400320          IMAGE_SECTION_HEADER <'.data', <24000h>, 0F2000h, 15A00h, 0F1000h, 0, \
  HEADER:00400320                                0, 0, 0, 0C0000040h>
> HEADER:00400348          IMAGE_SECTION_HEADER <'.tls', <1000h>, 116000h, 200h, 106A00h, 0, 0, \
  HEADER:00400348                                0, 0, 0C0000040h>
> HEADER:00400370          IMAGE_SECTION_HEADER <'.rdata', <1000h>, 117000h, 200h, 106C00h, 0, 0,\
  HEADER:00400370                                0, 0, 50000040h>
> HEADER:00400398          IMAGE_SECTION_HEADER <'.idata', <3000h>, 118000h, 3000h, 106E00h, 0, \
  HEADER:00400398                                0, 0, 0, 40000040h>
> HEADER:004003C0          IMAGE_SECTION_HEADER <'.edata', <3000h>, 11B000h, 2A00h, 109E00h, 0, \
  HEADER:004003C0                                0, 0, 0, 40000040h>
> HEADER:004003E8          IMAGE_SECTION_HEADER <'.rsrc', <24000h>, 11E000h, 24000h, 10C800h, 0, \
  HEADER:004003E8                                0, 0, 0, 40000040h>
> HEADER:00400410          IMAGE_SECTION_HEADER <'.reloc', <11000h>, 142000h, 10200h, 130800h, 0,\
  HEADER:00400410                                0, 0, 0, 50000040h>
  HEADER:00400438          align 1000h
```

In addition to strings, you can ofcourse change representation of other structure fields similarly to operand representation[5] for instructions. For example, you can change the **`SizeOfRawData`** field to be printed in decimal instead of the default hex.



See also:
IDA Help: Assembler level and C level types[6]
Igor's tip of the week #46: Disassembly operand representation[7]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-122-manual-load/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[3] https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_nt_headers32
[4] https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_section_header
[5] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/
[6] https://www.hex-rays.com/products/ida/support/idadoc/1042.shtml
[7] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/

#126: Non-returning functions

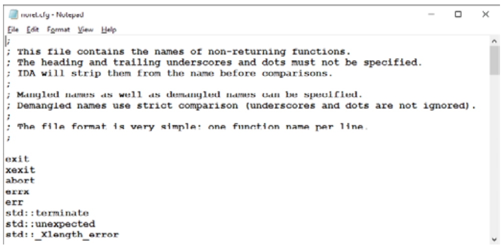
03 Feb 2023

https://hex-rays.com/blog/igors-tip-of-the-week-126-non-returning-functions/

Some functions in programs do not return to caller: well-known examples include C runtime functions like `exit()`, `abort()`, `assert()` but also many others. Modern compilers can exploit this knowledge to optimize the code better: for example, the code which would normally follow such a function call does not need to be generated which decreases the program size. Other functions, which call non-returning functions unconditionally also become non-returning, which can lead to further optimizations.

Well-known functions

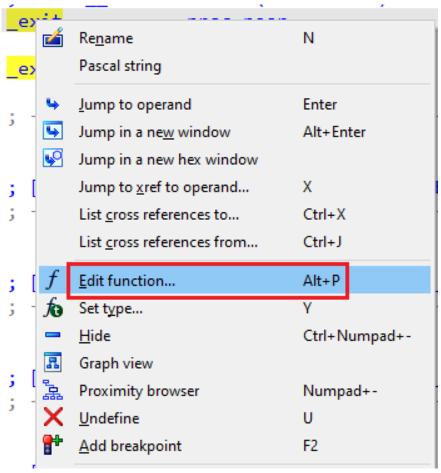
IDA uses function names to mark well-known non-returning functions. The list of such names is stored in the file `cfg/noret.cfg`, which can be edited to add more names if necessary:

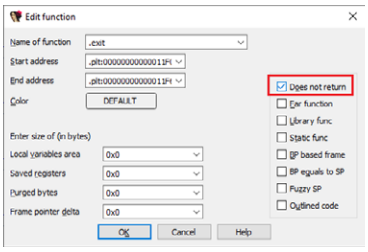


```
;
; This file contains the names of non-returning functions.
; The heading and trailing underscores and dots must not be specified.
; IDA will strip them from the name before comparisons.
;
; Mangled names as well as demangled names can be specified.
; Demangled names use strict comparison (underscores and dots are not ignored).
;
; The file format is very simple: one function name per line.
;

exit
xexit
abort
errx
err
std::terminate
std::unexpected
std::_Xlength_error
```

Marking non-returning functions manually

Instead of editing `noret.cfg`, you can also mark a function as non-returning manually on a case-by-case basis. This can be done by editing function properties: *Edit > Functions > Edit Function…* in the main menu, *Edit Function…* in the context menu or the `Alt–P` shortcut.





Name of function: .exit
Start address: .plt:00000000000011F(
End address: .plt:00000000000011F(
Color: DEFAULT
Enter size of (in bytes)
Local variables area: 0x0
Saved registers: 0x0
Purged bytes: 0x0
Frame pointer delta: 0x0

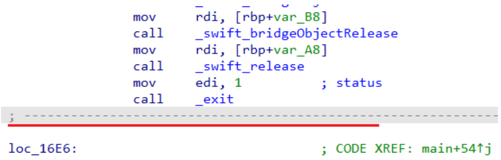
☑ Does not return
☐ Far function
☐ Library func
☐ Static func
☐ BP based frame
☐ BP equals to SP
☐ Fuzzy SP
☐ Outlined code

Another option is to edit the function's prototype and add the `__noreturn keyword`[1].

Identifying no-return calls

Incorrectly identified non-returning calls may lead to various problems during analysis: functions being truncated too early; decompiled pseudocode missing big parts of the function and so on. One option is to inspect each function being called to see if it has the Does not return flag set (or `Attributes: noreturn` mentioned in a comment) but this can take a long time with many calls. So there are indicators which may be easier to spot:

- In the text view, look for dashed line after a call; it indicates a break in the code flow which means that the execution does not continue after the call, i.e. it does not return.



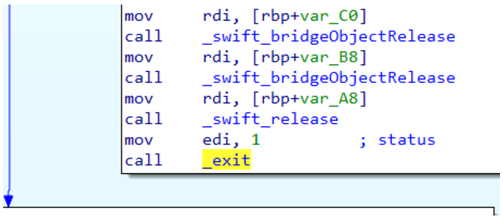
```
                mov     rdi, [rbp+var_B8]
                call    _swift_bridgeObjectRelease
                mov     rdi, [rbp+var_A8]
                call    _swift_release
                mov     edi, 1          ; status
                call    _exit
; ---------------------------------------------------------------------------

loc_16E6:                               ; CODE XREF: main+54↑j
```

#126: Non-returning functions

03 Feb 2023

https://hex-rays.com/blog/igors-tip-of-the-week-126-non-returning-functions/

- In the graph view, when a node which ends with a call has no outgoing edge, this means that the call does not return.

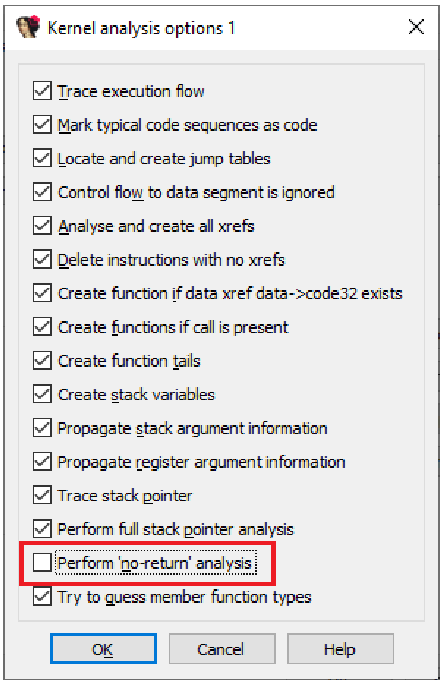


```
mov     rdi, [rbp+var_C0]
call    _swift_bridgeObjectRelease
mov     rdi, [rbp+var_B8]
call    _swift_bridgeObjectRelease
mov     rdi, [rbp+var_A8]
call    _swift_release
mov     edi, 1          ; status
call    _exit
```

- In the pseudocode it's not always obvious, but calls to no-ret functions usually end a compound statement or the whole function. You can also switch to the disassembly if the function looks suspiciously short and look for the above tell-tales.

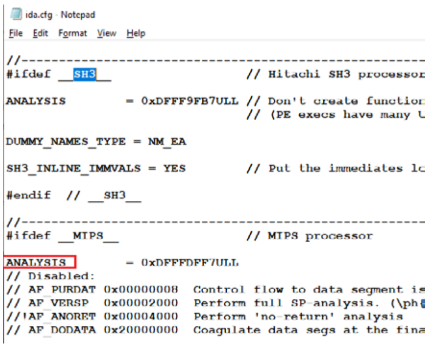
Enabling or disabling no-return analysis

If you find that IDA's treatment of non-returning functions does not work well with your specific binary or set of binaries, you can turn it off. This can be done in the first set of the analysis options[2] at the initial load time or afterwards. Conversely, you can enable it for processors which do not enable it by default.



Kernel analysis options 1

☑ Trace execution flow
☑ Mark typical code sequences as code
☑ Locate and create jump tables
☑ Control flow to data segment is ignored
☑ Analyse and create all xrefs
☑ Delete instructions with no xrefs
☑ Create function if data xref data->code32 exists
☑ Create functions if call is present
☑ Create function tails
☑ Create stack variables
☑ Propagate stack argument information
☑ Propagate register argument information
☑ Trace stack pointer
☑ Perform full stack pointer analysis
☐ Perform 'no-return' analysis
☑ Try to guess member function types



ida.cfg - Notepad

```
//-------------------------------------------------------------
#ifdef __SH3__                        // Hitachi SH3 processor

ANALYSIS        = 0xDFFF9FB7ULL // Don't create function
                                // (PE execs have many t

DUMMY_NAMES_TYPE = NM_EA

SH3_INLINE_IMMVALS = YES         // Put the immediates lo

#endif  // __SH3__

//-------------------------------------------------------------
#ifdef __MIPS__                       // MIPS processor

ANALYSIS        = 0xDFFFDFF7ULL
// Disabled:
// AF_PURDAT 0x00000008  Control flow to data segment is
// AF_VERSP  0x00002000  Perform full SP-analysis. (\ph
//!AF_ANORET 0x00004000  Perform 'no-return' analysis
// AF_DODATA 0x20000000  Coagulate data segs at the fina
```

If you need to permanently enable or disable it for all new databases, edit the **ANALYSIS** value in **ida.cfg** to include or not the **AF_ANORET** flag. NB: you should edit the value under #ifdef for the specific processor you need.

See also: IDA Help: Function flags[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-52-special-attributes/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-98-analysis-options/
[3] https://www.hex-rays.com/products/ida/support/idadoc/1729.shtml

#127: Changing function bounds

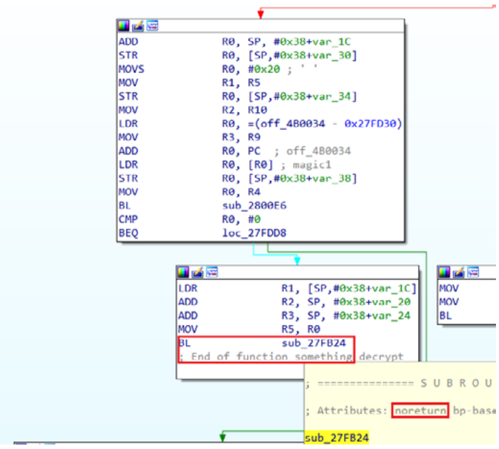
10 Feb 2023

https://hex-rays.com/blog/igors-tip-of-the-week-127-changing-function-bounds/

When analyzing regular, well-formed binaries, you can usually rely on IDA's autoanalysis to create functions and detect their boundaries correctly. However, there may be situations when IDA's guesses need to be adjusted.

Non-returning calls

One example could be calls to non-returning functions[1]. Let's say a function has been misdetected by IDA as non-returning:



```
ADD     R0, SP, #0x38+var_1C
STR     R0, [SP,#0x38+var_30]
MOVS    R0, #0x20 ; ' '
MOV     R1, R5
STR     R0, [SP,#0x38+var_34]
MOV     R2, R10
LDR     R0, =(off_4B0034 - 0x27FD30)
MOV     R3, R9
ADD     R0, PC  ; off_4B0034
LDR     R0, [R0] ; magic1
STR     R0, [SP,#0x38+var_38]
MOV     R0, R4
BL      sub_2800E6
CMP     R0, #0
BEQ     loc_27FDD8
```

```
LDR     R1, [SP,#0x38+var_1C]
ADD     R2, SP, #0x38+var_20
ADD     R3, SP, #0x38+var_24
MOV     R5, R0
BL      sub_27FB24
; End of function something_decrypt
```

```
MOV
MOV
BL
```

```
; =============== S U B R O U
; Attributes: noreturn bp-base
sub_27FB24
```

But on further analysis you realize that it actually returns and remove the no-return flag. However, IDA has already truncated the function after the call and now you need to extend it to include the code after call. How to do it?

Recreating the function

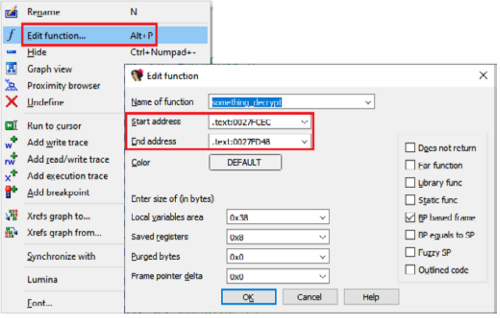
This is probably the quickest approach which can be used in simple situations:

1. Go to the start of the function (for example, by double-clicking the function in the Functions list[2]), or via key sequence `Ctrl– P`, `Enter`.
2. Delete the function (from the Functions list), or `Ctrl–P`, `Del`. If you were in Graph view, IDA will switch to the text view.
3. Create it again (Create function... from context menu), or press `P`.

This works well if the changes were enough to fix the original problem. You may need to repeat this a few times when fixing problems one by one. Note that deleting the function may destroy some of the information attached to it (such as the function comment), so this is not always the best choice.

Editing function bounds

The Edit function dialog has fields for function's start and end addresses:



Name of function: something_decrypt

Start address: .text:0027FCEC

End address: .text:0027FD48

Color: DEFAULT

Enter size of (in bytes)

Local variables area: 0x38

Saved registers: 0x8

Purged bytes: 0x0

Frame pointer delta: 0x0

□ Does not return
□ Far function
□ Library func
□ Static func
☑ BP based frame
□ BP equals to SP
□ Fuzzy SP
□ Outlined code

They can be edited to expand or shrink the function, but there are some limitations:

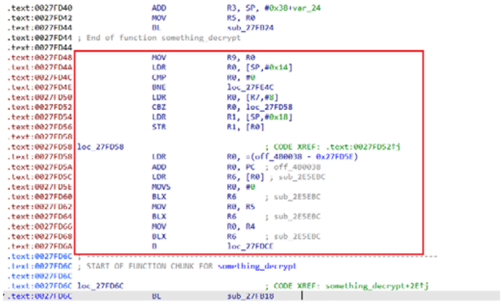
1. The new function bounds may not intersect with another function or a function chunk[3]. They also may not cross a segment boundary.
2. The function start must be a valid instruction.

Keep in mind that the end address is exclusive, i.e. it is the address **after** the last instruction of the function.

Changing the function end

To move the current or preceding function's end only, you can use the hotkey `E` (Set function end). If there is a function or a chunk at the current address, it is truncated to end just after the current instruction. If the current address does not belong to a function, the nearest preceding function or chunk is extended instead. If the extension causes function chunks to be immediately next to each other, they're merged together.

For example, consider this situation:



```
.text:0027FD40                 ADD             R3, SP, #0x38+var_24
.text:0027FD42                 MOV             R5, R0
.text:0027FD44                 BL              sub_27FB24
.text:0027FD44 ; End of function something_decrypt
.text:0027FD44
.text:0027FD48                 MOV             R9, R0
.text:0027FD4A                 LDR             R0, [SP,#0x14]
.text:0027FD4C                 CMP             R0, #0
.text:0027FD4E                 BNE             loc_27FE4C
.text:0027FD50                 LDR             R0, [R7,#8]
.text:0027FD52                 CBZ             R0, loc_27FD58
.text:0027FD54                 LDR             R1, [SP,#0x18]
.text:0027FD56                 STR             R1, [R0]
.text:0027FD58
.text:0027FD58 loc_27FD58                              ; CODE XREF: .text:0027FD52↑j
.text:0027FD58                 LDR             R0, =(off_4B0038 - 0x27FD5E)
.text:0027FD5A                 ADD             R0, PC  ; off_4B0038
.text:0027FD5C                 LDR             R6, [R0] ; sub_2E5EBC
.text:0027FD5E                 MOVS            R0, #0
.text:0027FD60                 BLX             R6      ; sub_2E5EBC
.text:0027FD62                 MOV             R0, R5
.text:0027FD64                 BLX             R6      ; sub_2E5EBC
.text:0027FD66                 MOV             R0, R4
.text:0027FD68                 BLX             R6      ; sub_2E5EBC
.text:0027FD6A                 B               loc_27FDCE
.text:0027FD6C ; ---------------------------------------------------------------------------
.text:0027FD6C ; START OF FUNCTION CHUNK FOR something_decrypt
.text:0027FD6C
.text:0027FD6C loc_27FD6C                              ; CODE XREF: something_decrypt+2E↑j
.text:0027FD6C                 BL              sub_27FB18
```

The instructions in the red rectangle should be part of the function but they're currently "independent" (this can also be seen by the color of the address prefix which is brown and not black like for instructions inside a function). To make them part of the function, we can move its end to the last one (`0027FD6A`). Putting the cursor there and invoking Edit > Functions > Set function end (shortcut `E`) will move the function end from `0027FD44` to `0027FD6A`. Because this makes the function adjacent to its own chunk, IDA merges the chunk with the function and the function is expanded to cover all newly reachable instructions.

See also:
IDA Help: Edit Function[4]
IDA Help: Set Function End[5]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-126-non-returning-functions/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-28-functions-list/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-86-function-chunks/
[4] https://www.hex-rays.com/products/ida/support/idadoc/485.shtml
[5] https://www.hex-rays.com/products/ida/support/idadoc/487.shtml

#128: String list

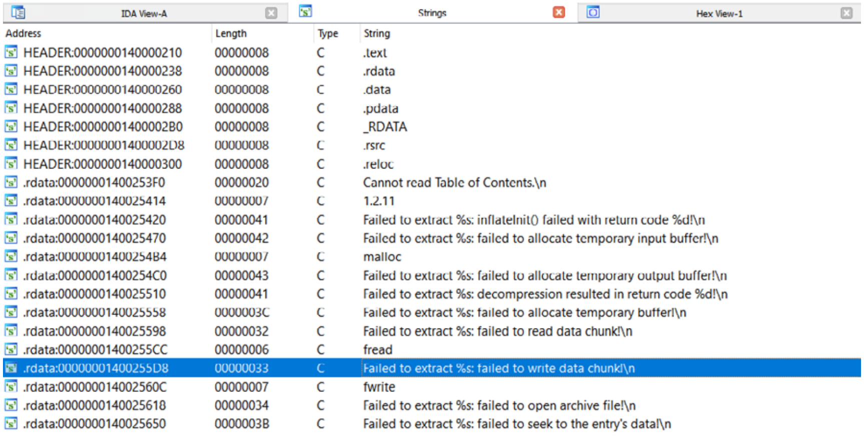
17 Feb 2023

https://hex-rays.com/blog/igors-tip-of-the-week-128-strings-list/

When exploring an unfamiliar binary, it may be difficult to find interesting places to start from. One common approach is to check what strings are present in the program – this might give some hints about its functionality and maybe some starting places for analysis. While you can scroll through the listing and look at the strings as you come across them, it is probably more convenient to see them all in one place. IDA offer this functionality as the *Strings* view.

Opening String list

To open the list, use the menu View > Open subviews > Strings, or the shortcut `Shift– F12`. Note that the first time IDA will scan the whole database so it may take some time on big files. If you have a really big binary, it may be useful to select a range[1] before invoking the command will so that the scan is limited to the selection.

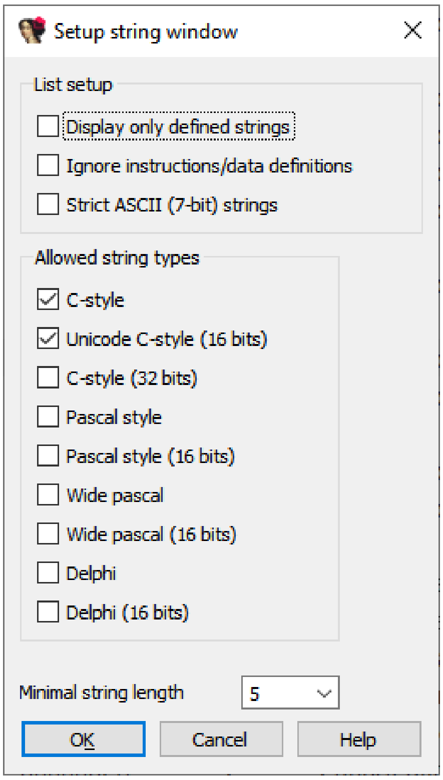


Address	Length	Type	String
HEADER:0000000140000210	00000008	C	.text
HEADER:0000000140000238	00000008	C	.rdata
HEADER:0000000140000260	00000008	C	.data
HEADER:0000000140000288	00000008	C	.pdata
HEADER:00000001400002B0	00000008	C	_RDATA
HEADER:00000001400002D8	00000008	C	.rsrc
HEADER:0000000140000300	00000008	C	.reloc
.rdata:00000001400253F0	00000020	C	Cannot read Table of Contents.\n
.rdata:0000000140025414	00000007	C	1.2.11
.rdata:0000000140025420	00000041	C	Failed to extract %s: inflateInit() failed with return code %d!\n
.rdata:0000000140025470	00000042	C	Failed to extract %s: failed to allocate temporary input buffer!\n
.rdata:00000001400254B4	00000007	C	malloc
.rdata:00000001400254C0	00000043	C	Failed to extract %s: failed to allocate temporary output buffer!\n
.rdata:0000000140025510	00000041	C	Failed to extract %s: decompression resulted in return code %d!\n
.rdata:0000000140025558	0000003C	C	Failed to extract %s: failed to allocate temporary buffer!\n
.rdata:0000000140025598	00000032	C	Failed to extract %s: failed to read data chunk!\n
.rdata:00000001400255CC	00000006	C	fread
.rdata:00000001400255D8	00000033	C	Failed to extract %s: failed to write data chunk!\n
.rdata:000000014002560C	00000007	C	fwrite
.rdata:0000000140025618	00000034	C	Failed to extract %s: failed to open archive file!\n
.rdata:0000000140025650	0000003B	C	Failed to extract %s: failed to seek to the entry's data!\n

The view includes the string's address, length (in characters, including the terminating one), type (e.g. `C` for standard 8-bit strings or `C16` for Unicode (UTF-16)), and the text of the string. Double-clicking an entry will jump to the string in the binary, and you can, for example, check the cross-references[2] to see where it's used.

String list options

The default settings are somewhat conservative so if you think some items are missing (or, conversely, you see a lot of useless entries), changing scan options can be useful. For this, use "Setup.." from the context menu.



List setup

□ Display only defined strings
□ Ignore instructions/data definitions
□ Strict ASCII (7-bit) strings

Allowed string types

☑ C-style
☑ Unicode C-style (16 bits)
□ C-style (32 bits)
□ Pascal style
□ Pascal style (16 bits)
□ Wide pascal
□ Wide pascal (16 bits)
□ Delphi
□ Delphi (16 bits)

Minimal string length: 5

- Display only defined strings will have IDA include only explicitly defined string literals (e.g. strings discovered in a middle of undefined areas won't be included).
- Ignore instructions/data definitions makes IDA look for text inside code or non-string data.
- Strict ASCII (7-bit) strings option shows only strings with characters in the basic ASCII range.
- Allowed string types lets you choose what string types you are interested in.
- Minimal string length sets the lower limit on the length the string must have to be included in the list. Raising the limit may be useful to filter out false positives.

Note that you will likely need to invoke "Rebuild…" from the context menu to refresh the list after changing the options.

See also: IDA Help: Strings window[3]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/
[3] https://www.hex-rays.com/products/ida/support/idadoc/1379.shtml

#129: Searching for text in database

24 Feb 2023

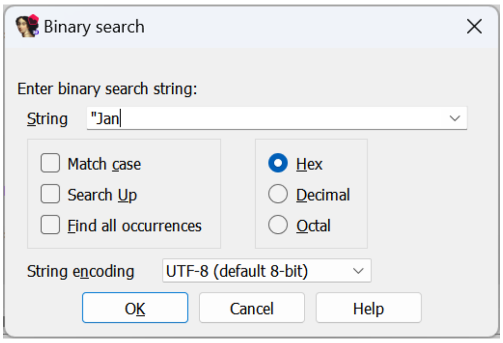
https://hex-rays.com/blog/igors-tip-of-the-week-129-searching-for-text-in-database/

Using the string list[1] is one way to look for text in the binary but it has its downsides: building the list takes time for big binaries, some strings may be missing initially so you may need several tries to get the options right, and then you need to actually find what you need in the list.

If you already know the text you want to find (e.g. from the output of the program), there is a quicker way.

Using binary search for text

The binary search action can be invoked via Search > Sequence of bytes… menu, or the `Alt– B` shortcut. Although its primary use is for binding known byte sequences, you can also use it for finding text embedded in the binary. For this, surround the text string with double quotes ("). The closing quote is optional.


Binary search

Enter binary search string:

String: "Jan

☐ Match case
☐ Search Up
☐ Find all occurrences

● Hex
○ Decimal
○ Octal

String encoding: UTF-8 (default 8-bit)

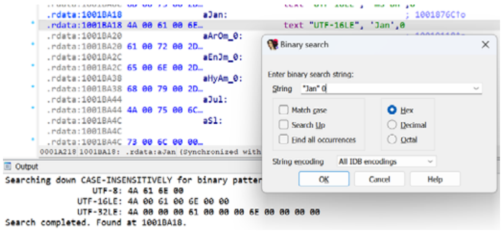
Once a quote is present in the input box, the String encoding dropdown is enabled. It allows you to choose in which encoding[2](s) to look for the string.

After confirming, IDA will print in the Output window the exact byte patterns it's looking for:

Searching down CASE-INSENSITIVELY for binary patterns:

```
UTF-8: 4A 61 6E
UTF-16LE: 4A 00 61 00 6E 00
UTF-32LE: 4A 00 00 00 61 00 00 00 6E 00 00 00
Search completed. Found at 1001A9C4.
```

You can also mix string literals and byte values. For example, to find "Jan" but not "January", add `0` for the C string terminator:



```
.rdata:1001BA18                 aJan:                                  ; 100187GCto
.rdata:1001BA18 4A 00 61 00 6E…                 text "UTF-16LE", 'Jan',0
.rdata:1001BA20                 aArOm_0:
.rdata:1001BA20 61 00 72 00 2D…
.rdata:1001BA2C                 aEnJm_0:
.rdata:1001BA2C 65 00 6E 00 2D…
.rdata:1001BA38                 aHyAm_0:
.rdata:1001BA38 68 00 79 00 2D…
.rdata:1001BA44                 aJul:
.rdata:1001BA44 4A 00 75 00 6C…
.rdata:1001BA4C                 aS1:
.rdata:1001BA4C
.rdata:1001BA4C 73 00 6C 00 00…
0001A218 1001BA18: .rdata:aJan (Synchronized with
```

Binary search

Enter binary search string:

String: "Jan" 0

☐ Match case
☐ Search Up
☐ Find all occurrences

● Hex
○ Decimal
○ Octal

String encoding: All IDB encodings

```
Searching down CASE-INSENSITIVELY for binary patter
          UTF-8: 4A 61 6E 00
       UTF-16LE: 4A 00 61 00 6E 00 00
       UTF-32LE: 4A 00 00 00 61 00 00 00 6E 00 00 00 00
Search completed. Found at 1001BA18.
```

To continue the search, use Search > Next sequence of bytes…, or shortcut `Ctrl– B`.

See also:
Igor's tip of the week #48: Searching in IDA[3]
IDA Help: Search for substring in the file[4]
IDA Help: Binary string format[5]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-128-strings-list/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/
[4] https://www.hex-rays.com/products/ida/support/idadoc/579.shtml
[5] https://www.hex-rays.com/products/ida/support/idadoc/528.shtml

#130: Source line numbers

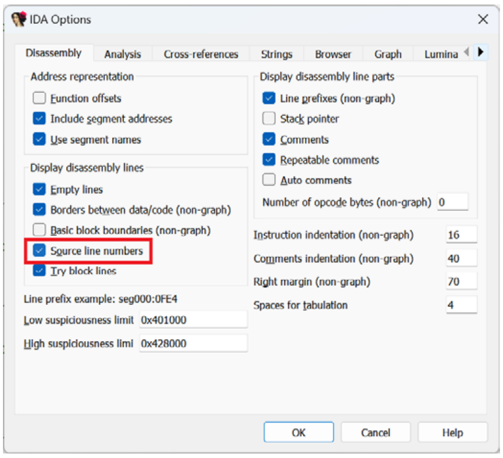
03 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-130-source-line-numbers/

Debug information, whether present in the binary or loaded separately[1], can contain not only symbols such as function or variable names, but also mapping of binary's instructions to the original source files. It can be used by IDA's debugger for source-level debugging[2], but what if you want to see this mapping during static analysis?

Enabling source line number display

Assuming the line number info was available and has been imported, it can be enabled in the Options > General... dialog, Disassembly tab:



Address representation
- ☐ Function offsets
- ☑ Include segment addresses
- ☑ Use segment names

Display disassembly lines
- ☑ Empty lines
- ☑ Borders between data/code (non-graph)
- ☐ Basic block boundaries (non-graph)
- ☑ Source line numbers
- ☑ Try block lines

Line prefix example: seg000:0FE4

Low suspiciousness limit 0x401000

High suspiciousness limi 0x428000

Display disassembly line parts
- ☑ Line prefixes (non-graph)
- ☐ Stack pointer
- ☑ Comments
- ☑ Repeatable comments
- ☐ Auto comments

Number of opcode bytes (non-graph) 0

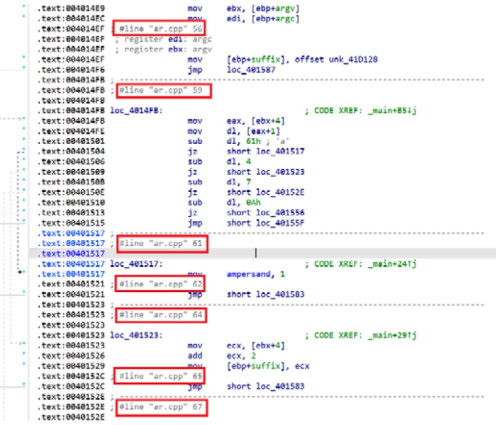
Instruction indentation (non-graph) 16

Comments indentation (non-graph) 40

Right margin (non-graph) 70

Spaces for tabulation 4

Once enabled, IDA will add automatic comments with the file name and line number in the disassembly listing:



```
.text:004014E9                 mov     ebx, [ebp+argv]
.text:004014EC                 mov     edi, [ebp+argc]
.text:004014EF ; #line "ar.cpp" 56
.text:004014EF  ; register edi: argc
.text:004014EF  ; register ebx: argv
.text:004014EF                 mov     [ebp+suffix], offset unk_41D128
.text:004014F6                 jmp     loc_401587
.text:004014FB ; ---------------------------------------------------------------------------
.text:004014FB ; #line "ar.cpp" 59
.text:004014FB
.text:004014FB loc_4014FB:                             ; CODE XREF: _main+B5↓j
.text:004014FB                 mov     eax, [ebx+4]
.text:004014FE                 mov     dl, [eax+1]
.text:00401501                 sub     dl, 61h ; 'a'
.text:00401504                 jz      short loc_401517
.text:00401506                 sub     dl, 4
.text:00401509                 jz      short loc_401523
.text:0040150B                 sub     dl, 7
.text:0040150E                 jz      short loc_40152E
.text:00401510                 sub     dl, 0Ah
.text:00401513                 jz      short loc_401556
.text:00401515                 jmp     short loc_40155F
.text:00401517 ; ---------------------------------------------------------------------------
.text:00401517 ; #line "ar.cpp" 61
.text:00401517
.text:00401517 loc_401517:                             ; CODE XREF: _main+24↑j
.text:00401517                 mov     ampersand, 1
.text:00401521 ; #line "ar.cpp" 62
.text:00401521                 jmp     short loc_401583
.text:00401523 ; ---------------------------------------------------------------------------
.text:00401523 ; #line "ar.cpp" 64
.text:00401523
.text:00401523 loc_401523:                             ; CODE XREF: _main+29↑j
.text:00401523                 mov     ecx, [ebx+4]
.text:00401526                 add     ecx, 2
.text:00401529                 mov     [ebp+suffix], ecx
.text:0040152C ; #line "ar.cpp" 65
.text:0040152C                 jmp     short loc_401583
.text:0040152E ; ---------------------------------------------------------------------------
.text:0040152E ; #line "ar.cpp" 67
.text:0040152E
```

To enable this for all new databases by default, change `SHOW_SOURCE_LINNUM` setting in `ida.cfg`.

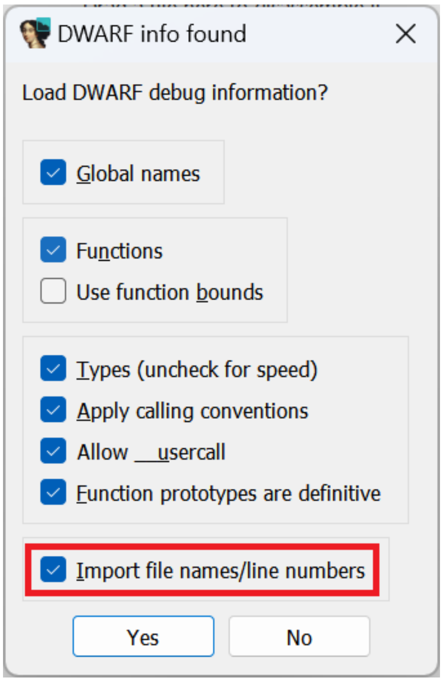
Importing line numbers from DWARF

DWARF debug format can also include line number information, but by default it's skipped because it's rarely needed in the database itself and can take a long time to load for big files. If you do need it, you should enable the corresponding option when prompted by IDA:

#130: Source line numbers

03 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-130-source-line-numbers/



DWARF info found

Load DWARF debug information?

☑ Global names

☑ Functions
□ Use function bounds

☑ Types (uncheck for speed)
☑ Apply calling conventions
☑ Allow __usercall
☑ Function prototypes are definitive

☑ Import file names/line numbers

To always import line numbers from DWARF debug info, enable `DWARF_IMPORT_LNNUMS` in `cfg/dwarf.cfg`.

See also:
Igor's tip of the week #55: Using debug symbols[3]
Igor's tip of the week #85: Source-level debugging[4]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-55-using-debug-symbols/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-85-source-level-debugging/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-55-using-debug-symbols/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-85-source-level-debugging/

#131: Advanced filters in choosers

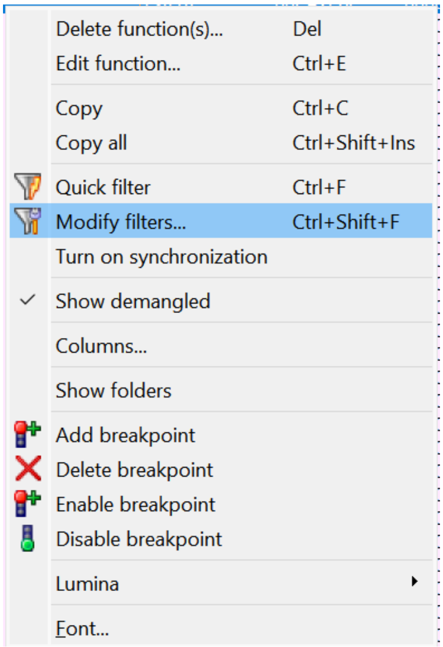
10 Mar 2023

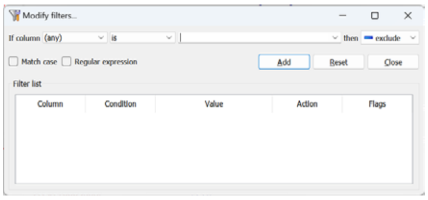
https://hex-rays.com/blog/igors-tip-of-the-week-131-advanced-filters-in-choosers/

We've covered choosers previously[1] and talked about searching, sorting and filtering. The default filter (`Ctrl– F` shortcut) is pretty simple: it performs case-insensitive match on any column of the list.

Advanced filters

Advanced filter dialog is accessible via the context menu entry "Modify filters…" or the shortcut `Ctrl–Shift–F`



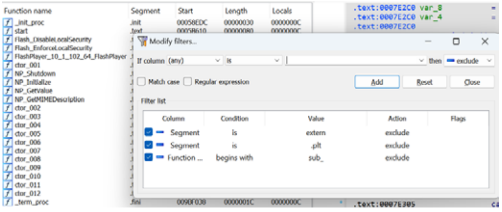


In the dialog you can:

- match any or a specific column;
- perform an exact match (is/is not) or partial (contains/doesn't contain, begins/ends with);
- perform a lexicographical comparison (less than/more than);
- decide whether a specific filter excludes, includes, or highlights matches;
- disable and enable filters individually;
- use case-sensitive matching or regular expressions.

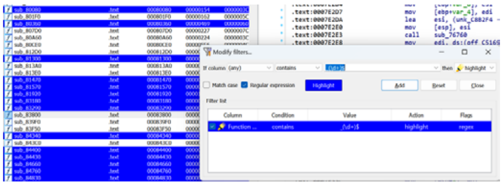
Examples

The following set of filters excludes functions which start with `sub_`, or situated in segments `extern` (external functions) and `.plt` (PLT thunks for external functions). This way only the functions defined inside the binary which have non-dummy names[2] are shown:


Modify filters...
If column (any) is then exclude
□ Match case □ Regular expression
Add
Reset
Close
Filter list

Column	Condition	Value	Action	Flags
☑ Segment	is	extern	exclude	
☑ Segment	is	.plt	exclude	
☑ Function ...	begins with	sub_	exclude	

Highlight any function with name ending in _NNN where NNN is a sequence of decimal numbers:


Modify filters...
If column (any) contains then highlight
□ Match case ☑ Regular expression
Highlight
Add
Reset
Close
Filter list

Column	Condition	Value	Action	Flags
☑ Function ...	contains	_(\d+)$	highlight	regex

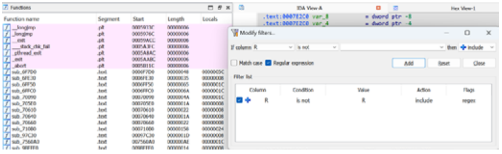
The highlight color can be changed by clicking the "Highlight button".

Show only functions which were detected by IDA as non-returning[3]:

#131: Advanced filters in choosers

10 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-131-advanced-filters-in-choosers/



Functions

Function name	Segment	Start	Length	Locals
__longjmp	.plt	0005913C	00000006	
_longjmp	.plt	0005976C	00000006	
__exit	.plt	00059ACC	00000006	
___stack_chk_fail	.plt	0005A3FC	00000006	
_pthread_exit	.plt	0005A8AC	00000006	
_exit	.plt	0005AABC	00000006	
_abort	.plt	0005B01C	00000006	
sub_6F7D0	.text	0006F7D0	00000048	0000005C
sub_6FE30	.text	0006FE30	00000035	00000008
sub_6FF50	.text	0006FF50	00000065	0000001C
sub_6FFC0	.text	0006FFC0	0000006A	0000001C
sub_70090	.text	00070090	0000004A	0000001C
sub_705F8	.text	000705F8	0000001A	00000008
sub_70610	.text	00070610	00000022	00000008
sub_70640	.text	00070640	0000001A	00000008
sub_70660	.text	00070660	00000022	00000008
sub_71080	.text	00071080	00000158	00000024
sub_97C30	.text	00097C30	0000001D	00000008
sub_7560A0	.text	007560A0	000000AE	0000001C
sub_98FEF0	.text	0098FEF0	00000014	00000008

IDA View-A

```
.text:0007E2C0 var_8          = dword ptr -8
.text:0007E2C0 var_4          = dword ptr -4
```

Hex View-1

Modify filters...

If column R is not then include

□ Match case ☑ Regular expression

Add Reset Close

Filter list

	Column	Condition	Value	Action	Flags
☑	R	is not	R	include	regex

NOTE: the examples listed apply to the Functions list but these filters are available in any chooser (list view) in IDA: Imports, Exports, Names, Local Types etc.

See also: Igor's tip of the week #36: Working with list views in IDA[4]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-36-working-with-list-views-in-ida/

[2] https://hex-rays.com/blog/igors-tip-of-the-week-34-dummy-names/

[3] https://hex-rays.com/blog/igors-tip-of-the-week-126-non-returning-functions/

[4] https://hex-rays.com/blog/igors-tip-of-the-week-36-working-with-list-views-in-ida/

#132: Finding “hidden” cross-references

17 Mar 2023

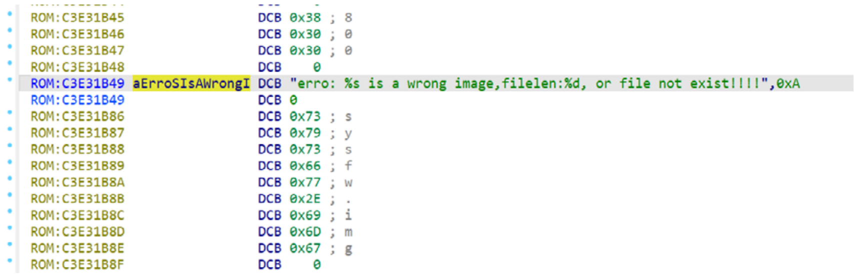
https://hex-rays.com/blog/igors-tip-of-the-week-132-finding-hidden-cross-references/

When analyzing firmware or other binaries without metadata, IDA is not always able to discover and analyze all functions which means the cross-references can be missing. Let’s say you found a string in the binary (e.g. in the String list[1]) which has no cross references, but you’re reasonably sure it’s actually used. How to discover where?

Finding addresses using binary search

One possibility is that the string is referred to by its address value, either from a pointer somewhere, or as an immediate value embedded directly in the instruction (the latter case is more common for CISC instruction sets such as x86). In such case, looking for the address value should discover it.

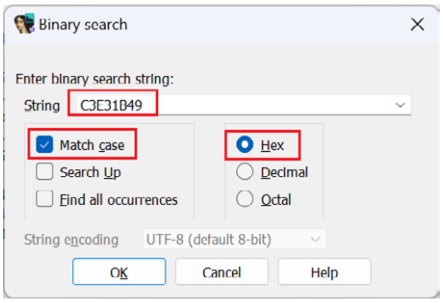
For example, here’s a string in an ARM firmware which currently has no cross-references:



```
ROM:C3E31B45                    DCB 0x38 ; 8
ROM:C3E31B46                    DCB 0x30 ; 0
ROM:C3E31B47                    DCB 0x30 ; 0
ROM:C3E31B48                    DCB    0
ROM:C3E31B49 aErroSIsAWrongI DCB "erro: %s is a wrong image,filelen:%d, or file not exist!!!!",0xA
ROM:C3E31B49                    DCB 0
ROM:C3E31B86                    DCB 0x73 ; s
ROM:C3E31B87                    DCB 0x79 ; y
ROM:C3E31B88                    DCB 0x73 ; s
ROM:C3E31B89                    DCB 0x66 ; f
ROM:C3E31B8A                    DCB 0x77 ; w
ROM:C3E31B8B                    DCB 0x2E ; .
ROM:C3E31B8C                    DCB 0x69 ; i
ROM:C3E31B8D                    DCB 0x6D ; m
ROM:C3E31B8E                    DCB 0x67 ; g
ROM:C3E31B8F                    DCB    0
```

We can try the following:

1. Select and copy to clipboard the string’s address (`C3E31B49`);
2. Go to the start of the database (`Ctrl–PgUp` or `Home`, `Home`, `Home`);
3. Invoke binary search (Search > Sequence of bytes..., or `Alt–B`);
4. Paste the address and make sure that Hex is selected. It is also recommended to enable Match case to avoid false positives:



Binary search

Enter binary search string:

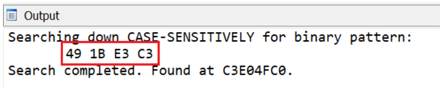
String: C3E31B49

☑ Match case
□ Search Up
□ Find all occurrences

● Hex
○ Decimal
○ Octal

String encoding: UTF-8 (default 8-bit)

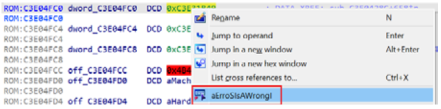
5. Click OK. IDA will automatically convert the value into a byte sequence corresponding to the processor endianness and look for it in the database:



Output

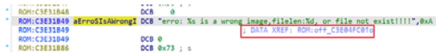
```
Searching down CASE-SENSITIVELY for binary pattern:
        49 1B E3 C3
Search completed. Found at C3E04FC0.
```

The value may be initially displayed as a raw number or even separate bytes. To convert it to an offset so that xref is created you can usually use the `O` or `Ctrl–O` shortcuts, or the context menu:



```
ROM:C3E04FC0 dword_C3E04FC0  DCD 0xC3E31B49
ROM:C3E04FC0
ROM:C3E04FC4 dword_C3E04FC4  DCD 0xC3E
ROM:C3E04FC4
ROM:C3E04FC8 dword_C3E04FC8  DCD 0xC3E
ROM:C3E04FC8
ROM:C3E04FCC off_C3E04FCC    DCD 0x404
ROM:C3E04FD0 off_C3E04FD0    DCD aMach
ROM:C3E04FD0
ROM:C3E04FD4 off_C3E04FD4    DCD aHard
```

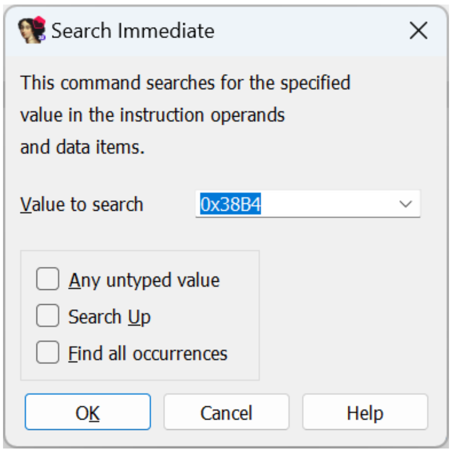
Now the string has a cross-reference and you can look further at where exactly it is used:



```
ROM:C3E31B48                    DCB    0
ROM:C3E31B49 aErroSIsAWrongI DCB "erro: %s is a wrong image,filelen:%d, or file not exist!!!!",0xA
ROM:C3E31B49                                        ; DATA XREF: ROM:off_C3E04FC0↑o
ROM:C3E31B49                    DCB 0
ROM:C3E31B86                    DCB 0x73 ; s
```

Finding addresses using immediate search

Binary search works for addresses embedded as-is into the binary. However, there may be situations where an address is embedded into an instruction not on a byte boundary, or split between several instructions. For example, RISC-V usually has to use at least two instructions to load a 32-bit value into a register (high 20 bits and low 12 bits). In case these instructions are next to each other, IDA can combine them into a single macroinstruction and calculate the full value, but because it’s split between two instructions, binary search won’t find it. However, immediate search (Search > Immediate value…, or `Alt– I`) should work. Note that if you copy the address from the listing, you’ll need to add 0x so that it can be parsed as hexadecimal by IDA.

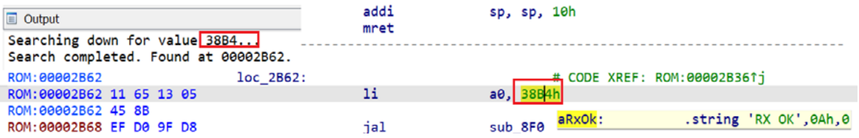


Search Immediate

This command searches for the specified value in the instruction operands and data items.

Value to search: 0x38B4

□ Any untyped value
□ Search Up
□ Find all occurrences



```
Output
Searching down for value 38B4...
Search completed. Found at 00002B62.

                                        addi            sp, sp, 10h
                                        mret
ROM:00002B62              loc_2B62:                          # CODE XREF: ROM:00002B36↑j
ROM:00002B62 11 65 13 05                li              a0, 38B4h
ROM:00002B62 45 8B                                          aRxOk:          .string 'RX OK',0Ah,0
ROM:00002B68 EF D0 9F D8                jal             sub_8F0
```

NOTE: this approach will succeed only under the following conditions:

1. the instruction(s) using the address were actually decoded. You can try the approach described in Tip #04[2] to try disassembling the whole binary before looking for cross-references;
2. the instructions were actually combined into a macro with the full address. For example, if they are interleaved with unrelated instructions, IDA won’t be able to combine them and you may need to look for each part separately.

Unfortunately, even the methods described here are not always enough. For example, self-relative offsets[3] will likely require analyzing the code to figure out what they refer to.

See also:
Igor’s tip of the week #95: Offsets[4]
Igor’s Tip of the Week #114: Split offsets[5]

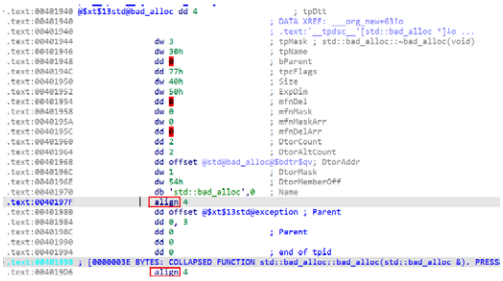
[1] https://hex-rays.com/blog/igors-tip-of-the-week-128-strings-list/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-04-more-selection/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-110-self-relative-offsets/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-95-offsets/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-114-split-offsets/

#133: Alignment items

24 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-133-alignment-items/

Sometimes you may see mysterious `align` keywords in the disassembly, which can appear both in code and data areas:



```
.text:00401940 @$xt$13std@bad_alloc dd 4                ; tpDtt
.text:00401940                                          ; DATA XREF: ___org_new+63↑o
.text:00401940                                          ; .text:'__tpdsc__'[std::bad_alloc *]↓o ...
.text:00401944                 dw 3                     ; tpMask ; std::bad_alloc::~bad_alloc(void)
.text:00401946                 dw 30h                   ; tpName
.text:00401948                 dd 0                     ; bParent
.text:0040194C                 dd 77h                   ; tpcFlags
.text:00401950                 dw 40h                   ; Size
.text:00401952                 dw 50h                   ; ExpDim
.text:00401954                 dd 0                     ; mfnDel
.text:00401958                 dw 0                     ; mfnMask
.text:0040195A                 dw 0                     ; mfnMaskArr
.text:0040195C                 dd 0                     ; mfnDelArr
.text:00401960                 dd 2                     ; DtorCount
.text:00401964                 dd 2                     ; DtorAltCount
.text:00401968                 dd offset @std@bad_alloc@$bdtr$qv; DtorAddr
.text:0040196C                 dw 1                     ; DtorMask
.text:0040196E                 dw 54h                   ; DtorMemberOff
.text:00401970                 db 'std::bad_alloc',0    ; Name
.text:0040197F                 align 4
.text:00401980                 dd offset @$xt$13std@exception ; Parent
.text:00401984                 dd 0, 3
.text:0040198C                 dd 0                     ; Parent
.text:00401990                 dd 0
.text:00401994                 dd 0                     ; end of tpid
.text:00401998 ; [0000003E BYTES: COLLAPSED FUNCTION std::bad_alloc::bad_alloc(std::bad_alloc &). PRESS
.text:004019D6                 align 4
```

Usually they're only apparent in the text view.

These directives are used by many assemblers to indicate alignment to a specific address boundary, usually a power of two. IDA uses it to replace potentially irrelevant bytes by a short one-liner, both for more compact listing and to indicate that this part of the binary is probably not interesting.

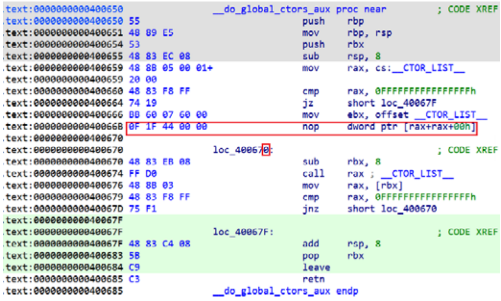
Depending on the processor and the assembler chosen, different keyword can be used (e.g. `align` or `.align`), and the number after the directive can mean either the number of bytes or the power of two (i.e. 1 means aligning to two bytes, 2 to four, 4 to sixteen and so on).

The alignment items can appear in the following situations:

Code alignment padding

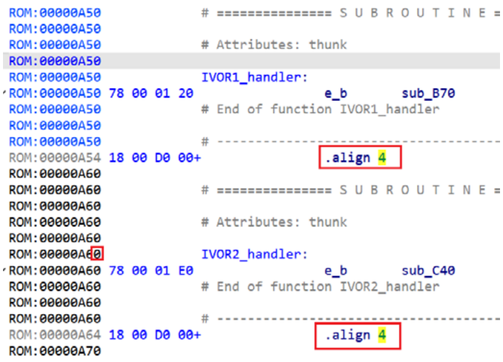
Many processors use instruction caches which speed up execution of often-executed code (for example, loops). This is why it may be useful to ensure that start of a loop is aligned on a specific address boundary (usually 16 bytes). For this, the compiler needs to insert instructions which do not affect the behavior of the function, i.e. NOP (no-operation) instructions. Which specific instructions are used depends on the processor and compiler.

For example, here GCC used a so-called "long NOP" to align the loop on 16 bytes (obvious thanks to the hexadecimal address ending with 0). Because this instruction is actually executed, IDA shows it as code and not as an align expression (which is considered non-executable and would break disassembly), but you can still convert it manually.



```
.text:0000000000400650                   __do_global_ctors_aux proc near        ; CODE XREF
.text:0000000000400650 55                               push    rbp
.text:0000000000400651 48 89 E5                         mov     rbp, rsp
.text:0000000000400654 53                               push    rbx
.text:0000000000400655 48 83 EC 08                      sub     rsp, 8
.text:0000000000400659 48 8B 05 00 01+                  mov     rax, cs:__CTOR_LIST__
.text:0000000000400659 20 00
.text:0000000000400660 48 83 F8 FF                      cmp     rax, 0FFFFFFFFFFFFFFFFh
.text:0000000000400664 74 19                            jz      short loc_40067F
.text:0000000000400666 BB 60 07 60 00                   mov     ebx, offset __CTOR_LIST__
.text:000000000040066B 0F 1F 44 00 00                   nop     dword ptr [rax+rax+00h]
.text:0000000000400670
.text:0000000000400670                   loc_400670:                            ; CODE XREF
.text:0000000000400670 48 83 EB 08                      sub     rbx, 8
.text:0000000000400674 FF D0                            call    rax ; __CTOR_LIST__
.text:0000000000400676 48 8B 03                         mov     rax, [rbx]
.text:0000000000400679 48 83 F8 FF                      cmp     rax, 0FFFFFFFFFFFFFFFFh
.text:000000000040067D 75 F1                            jnz     short loc_400670
.text:000000000040067F
.text:000000000040067F                   loc_40067F:                            ; CODE XREF
.text:000000000040067F 48 83 C4 08                      add     rsp, 8
.text:0000000000400683 5B                               pop     rbx
.text:0000000000400684 C9                               leave
.text:0000000000400685 C3                               retn
.text:0000000000400685                   __do_global_ctors_aux endp
```

There may also be hardware requirements. On some processors the interrupt handlers must be aligned, like in this example from PowerPC:



```
ROM:00000A50             # =============== S U B R O U T I N E =
ROM:00000A50
ROM:00000A50             # Attributes: thunk
ROM:00000A50
ROM:00000A50             IVOR1_handler:
ROM:00000A50 78 00 01 20                 e_b          sub_B70
ROM:00000A50             # End of function IVOR1_handler
ROM:00000A50
ROM:00000A50             # ------------------------------------
ROM:00000A54 18 00 D0 00+               .align 4
ROM:00000A60
ROM:00000A60             # =============== S U B R O U T I N E =
ROM:00000A60
ROM:00000A60             # Attributes: thunk
ROM:00000A60
ROM:00000A60             IVOR2_handler:
ROM:00000A60 78 00 01 E0                 e_b          sub_C40
ROM:00000A60             # End of function IVOR2_handler
ROM:00000A60
ROM:00000A60             # ------------------------------------
ROM:00000A64 18 00 D0 00+               .align 4
ROM:00000A70
```

Here, 4 is a power-of-two value, i.e. alignment to 16-byte boundary.

#133: Alignment items

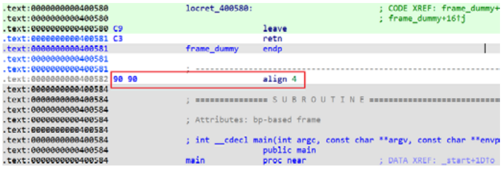
24 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-133-alignment-items/

Function padding

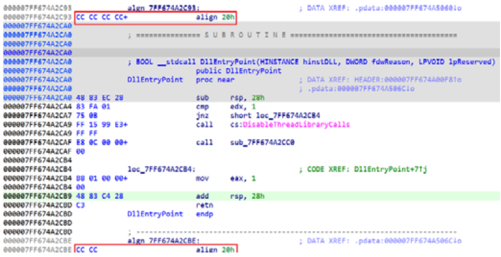
Similarly to loops, whole functions can benefit from the alignment, so they're commonly (but not always!) aligned to at least four bytes. Because the functions are usually placed one after the other but the function size is not always a multiple of the alignment, extra padding has to be inserted by the compiler and/or the linker. Two common approaches are used:

1. executable NOP instructions, just like for the loop alignment. This is the approach commonly used by GCC and derived compilers:



```
.text:0000000000400580                  locret_400580:                          ; CODE XREF: frame_dummy+
.text:0000000000400580                                                          ; frame_dummy+16↑j
.text:0000000000400580 C9                               leave
.text:0000000000400581 C3                               retn
.text:0000000000400581                  frame_dummy     endp
.text:0000000000400581
.text:0000000000400581                  ; ---------------------------------------------------------------
.text:0000000000400582 90 90                            align 4
.text:0000000000400584
.text:0000000000400584                  ; =============== S U B R O U T I N E =========================
.text:0000000000400584
.text:0000000000400584                  ; Attributes: bp-based frame
.text:0000000000400584
.text:0000000000400584                  ; int __cdecl main(int argc, const char **argv, const char **envp
.text:0000000000400584                                  public main
.text:0000000000400584                  main            proc near               ; DATA XREF: _start+1D↑o
```

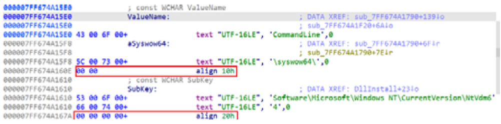
2. invalid or trapping instructions. This can be useful to catch bugs where execution is diverted to an address between functions, for example due to a bug or an exploit. Microsoft Visual C++, for example, tends to use 0xCC (breakpoint instruction) to pad the space between functions on x86:



```
000007FF674A2C93             algn_7FF674A2C93:                       ; DATA XREF: .pdata:000007FF674A5060↓o
000007FF674A2C93 CC CC CC CC+                align 20h
000007FF674A2CA0
000007FF674A2CA0             ; =============== S U B R O U T I N E =======================================
000007FF674A2CA0
000007FF674A2CA0
000007FF674A2CA0             ; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
000007FF674A2CA0                             public DllEntryPoint
000007FF674A2CA0             DllEntryPoint   proc near               ; DATA XREF: HEADER:000007FF674A00F8↑o
000007FF674A2CA0                                                     ; .pdata:000007FF674A506C↓o
000007FF674A2CA0 48 83 EC 28                 sub     rsp, 28h
000007FF674A2CA4 83 FA 01                    cmp     edx, 1
000007FF674A2CA7 75 0B                       jnz     short loc_7FF674A2CB4
000007FF674A2CA9 FF 15 99 E3+                call    cs:DisableThreadLibraryCalls
000007FF674A2CA9 FF FF
000007FF674A2CAF E8 0C 00 00+                call    sub_7FF674A2CC0
000007FF674A2CAF 00
000007FF674A2CB4
000007FF674A2CB4             loc_7FF674A2CB4:                        ; CODE XREF: DllEntryPoint+7↑j
000007FF674A2CB4 B8 01 00 00+                mov     eax, 1
000007FF674A2CB4 00
000007FF674A2CB9 48 83 C4 28                 add     rsp, 28h
000007FF674A2CBD C3                          retn
000007FF674A2CBD             DllEntryPoint   endp
000007FF674A2CBD
000007FF674A2CBD             ; ---------------------------------------------------------------------------
000007FF674A2CBE             algn_7FF674A2CBE:                       ; DATA XREF: .pdata:000007FF674A506C↓o
000007FF674A2CBE CC CC                       align 20h
```

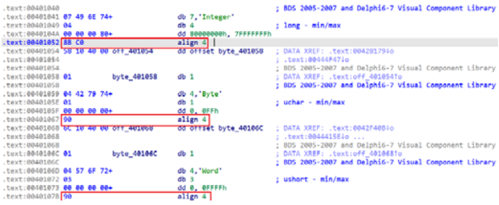
Data alignment padding

Many processors have alignment requirements: some can't even load data from unaligned addresses, and others can usually fetch aligned data faster. So the compilers often try to ensure that data items are placed on an aligned address boundary (usually at least 4 bytes). Most commonly, zero-fill padding is used:



```
000007FF674A15E0             ; const WCHAR ValueName
000007FF674A15E0             ValueName:                              ; DATA XREF: sub_7FF674A1790+139↓o
000007FF674A15E0                                                     ; sub_7FF674A1F20+6A↓o
000007FF674A15E0 43 00 6F 00+                text "UTF-16LE", 'CommandLine',0
000007FF674A15F8             aSyswow64:                              ; DATA XREF: sub_7FF674A1790+6F↓r
000007FF674A15F8                                                     ; sub_7FF674A1790+7E↓r
000007FF674A15F8 5C 00 73 00+                text "UTF-16LE", '\syswow64\',0
000007FF674A160E 00 00                       align 10h
000007FF674A1610             ; const WCHAR SubKey
000007FF674A1610             SubKey:                                 ; DATA XREF: DllInstall+23↓o
000007FF674A1610 53 00 6F 00+                text "UTF-16LE", 'Software\Microsoft\Windows NT\CurrentVersion\NtVdm6'
000007FF674A1610 66 00 74 00+                text "UTF-16LE", '4',0
000007FF674A167A 00 00 00 00+                align 20h
```

Although NOP-like fillers may be used by some compilers too, especially for constant data placed in executable areas:



```
.text:00401040                                      ; BDS 2005-2007 and Delphi6-7 Visual Component Library
.text:00401041 07 49 6E 74+            db 7,'Integer'
.text:00401049 04                      db 4         ; long - min/max
.text:0040104A 00 00 00 80+            dd 80000000h, 7FFFFFFFh
.text:00401052 8B C0                   align 4
.text:00401054 58 10 40 00 off_401054  dd offset byte_401058 ; DATA XREF: .text:0042B179↓o
.text:00401054                                      ; .text:00444F47↓o
.text:00401054                                      ; BDS 2005-2007 and Delphi6-7 Visual Component Library
.text:00401058 01          byte_401058 db 1         ; DATA XREF: .text:off_401054↑o
.text:00401058                                      ; BDS 2005-2007 and Delphi6-7 Visual Component Library
.text:00401059 04 42 79 74+            db 4,'Byte'
.text:0040105E 01                      db 1         ; uchar - min/max
.text:0040105F 00 00 00 00+            dd 0, 0FFh
.text:00401067 90                      align 4
.text:00401068 6C 10 40 00 off_401068  dd offset byte_40106C ; DATA XREF: .text:0042F40B↓o
.text:00401068                                      ; .text:004415E↓o ...
.text:00401068                                      ; BDS 2005-2007 and Delphi6-7 Visual Component Library
.text:0040106C 01          byte_40106C db 1         ; DATA XREF: .text:off_401068↑o
.text:0040106C                                      ; BDS 2005-2007 and Delphi6-7 Visual Component Library
.text:0040106D 04 57 6F 72+            db 4,'Word'
.text:00401072 03                      db 3         ; ushort - min/max
.text:00401073 00 00 00 00+            dd 0, 0FFFFh
.text:0040107B 90                      align 4
```

Converting alignment items

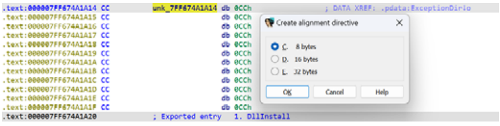
While rare, it may be necessary for you to change IDA's decision concerning an alignment item. Because they're mostly equivalent to data items, you can use the usual shortcut `U` to undefine them (convert to plain bytes), and then `C` to convert to code (in case they correspond to valid instructions).

To go the other way (convert instructions or undefined bytes) to an alignment item, use Edit > Other > Create alignment directive..., or just the shortcut `L`. IDA will check at what address is the next defined instruction or data item and will offer possibly several alignment options depending on its address. For example, in this situation:

#133: Alignment items

24 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-133-alignment-items/



```
.text:000007FF674A1A14 CC          unk_7FF674A1A14 db 0CCh               ; DATA XREF: .pdata:ExceptionDir↓o
.text:000007FF674A1A15 CC                          db 0CCh
.text:000007FF674A1A16 CC                          db 0CCh
.text:000007FF674A1A17 CC                          db 0CCh
.text:000007FF674A1A18 CC                          db 0CCh
.text:000007FF674A1A19 CC                          db 0CCh
.text:000007FF674A1A1A CC                          db 0CCh
.text:000007FF674A1A1B CC                          db 0CCh
.text:000007FF674A1A1C CC                          db 0CCh
.text:000007FF674A1A1D CC                          db 0CCh
.text:000007FF674A1A1E CC                          db 0CCh
.text:000007FF674A1A1F CC                          db 0CCh
.text:000007FF674A1A20             ; Exported entry   1. DllInstall
```

Create alignment directive

● C. 8 bytes
○ D. 16 bytes
○ E. 32 bytes

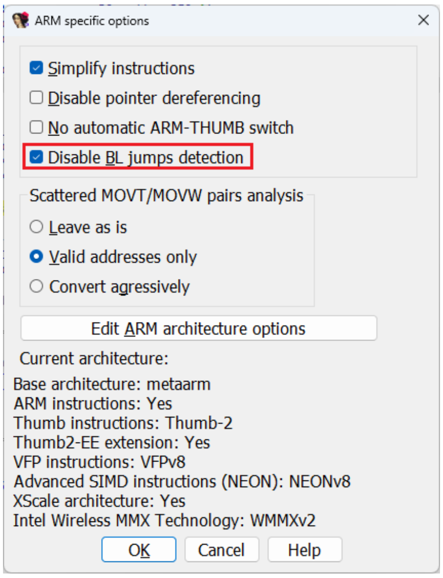
The current address is divisible by 4 so any alignment less than 4 is not applicable. The following defined address (`7FF674A1A20`) is divisible by 32, so IDA offers options 8, 16 and 32. Note that if you choose 8, the alignment item will only cover the first 4 bytes (up to `7FF674A1A18`), so in this situation 16 or 32 makes the most sense.

#134: ARM BL jumps

31 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-134-arm-bl-jumps/

If you ever looked at IDA ARM module's processor-specific settings[1], you may have been puzzled by the option "Disable BL jumps detection".



ARM specific options

☑ Simplify instructions
☐ Disable pointer dereferencing
☐ No automatic ARM-THUMB switch
☑ Disable BL jumps detection

Scattered MOVT/MOVW pairs analysis
○ Leave as is
● Valid addresses only
○ Convert agressively

Current architecture:

Base architecture: metaarm
ARM instructions: Yes
Thumb instructions: Thumb-2
Thumb2-EE extension: Yes
VFP instructions: VFPv8
Advanced SIMD instructions (NEON): NEONv8
XScale architecture: Yes
Intel Wireless MMX Technology: WMMXv2

What is it and when to use it?

Background

The ARM instruction set initially used fixed-width 32-bit instructions. The relative branch instruction, **B,** allocated 24 bits for the offset, giving it a range of ±32MB.

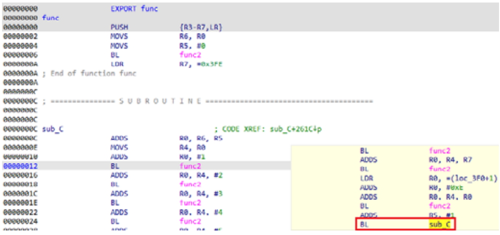
Some time later, ARM introduced a a compact 16-bit encoding for a subset of instructions, called Thumb. Because most relative branches occur in the same function, the ±2KB range available for 16-bit **B** instructions was usually enough. In case longer distance was needed, a longer instruction sequence would have to be generated.

Some compiler writers realized, that the **BL** instruction, normally used for function calls, can be used for simple branches as well. On ARM, the function calls do not use the stack, so the only side effect of BL as opposed to simple branch is that it sets the LR register to the address following the BL instruction. If the LR is saved at the start of the current function, it does not matter that if LR is clobbered by the intermediate BL instructions, since it can be restored from the saved area to return to the caller. The BL is encoded as pair of 16-bit instructions, which gives it a range of ±4MB.

A later extension of the Thumb, called Thumb-2, introduced a 32-bit version of B, giving it a range of ±16MB, so there is less need of such tricks in code compiled for modern processors which support Thumb-2. However, old code still needs to be analyzed sometimes, so it may be necessary to support such usage of **BL**.

Example

Here's an example of a Thumb mode program which looks a little strange…



```
00000000                 EXPORT func
00000000 func
00000000                 PUSH            {R3-R7,LR}
00000002                 MOVS            R6, R0
00000004                 MOVS            R5, #0
00000006                 BL              func2
0000000A                 LDR             R7, =0x3FE
0000000A ; End of function func
0000000A
0000000C
0000000C ; =============== S U B R O U T I N E =======================================
0000000C
0000000C
0000000C sub_C                                   ; CODE XREF: sub_C+261C↓p
0000000C                 ADDS            R0, R6, R5
0000000E                 MOVS            R4, R0
00000010                 ADDS            R0, #1
00000012                 BL              func2
00000016                 ADDS            R0, R4, #2
00000018                 BL              func2
0000001C                 ADDS            R0, R4, #3
0000001E                 BL              func2
00000022                 ADDS            R0. R4. #4
00000024                 BL              func2
```

```
                 BL              func2
                 ADDS            R0, R4, R7
                 BL              func2
                 LDR             R0, =(loc_3F0+1)
                 ADDS            R0, #0xE
                 ADDS            R0. R4. R0
                 BL              func2
                 ADDS            R5, #1
                 BL              sub_C
```

IDA has created a function because of the BL instruction which normally implies a function call. But we see that func is not complete, so most likely `sub_C` is actually its continuation and BL is used only as a branch. Also, func saves LR on the stack, so BL clobbering it does not matter.

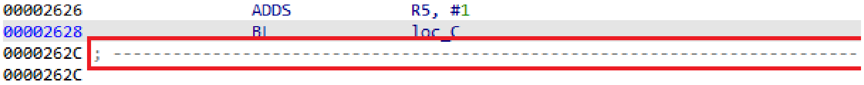
#134: ARM BL jumps

31 Mar 2023

https://hex-rays.com/blog/igors-tip-of-the-week-134-arm-bl-jumps/

Marking single instructions

If the BL-as-branch approach is used only in few cases, you can handle them manually. For this, place the cursor on the line with BL and use Edit > Other > Force BL jump menu item. IDA will take this into account and indicate that this BL does not continue to the next instruction by adding a dashed comment line after it[2].



```
00002626                ADDS            R5, #1
00002628                BL              loc_C
0000262C ; ---------------------------------------------------------------------------
0000262C
```

You can then delete the wrongly created function and extend[3] or recreate the original one which had been truncated.

Changing analysis behavior

If the binary has multiple functions which use this technique, it may be worth it to let the analyzer check each BL destination before creating functions. For this, turn off Disable BL jumps detection in the processor specific options and reanalyze the program[4]. Note that you will likely have to delete the wrongly created functions, so it may be better to reload the file, changing the options in the initial Load File dialog.

To set this by default, change `ARM_DISABLE_BL_JUMPS` value in `ida.cfg`.

In cases where the BL jumps detection fails (it marks a BL as a jump where it should be a call, or vice versa), you can always override its decision using Force BL jump and Force BL call menu options. In case you discover a specific code pattern and need to script it, you can also use IDC functions[5] `force_bl_jump(ea)` and `force_bl_call(ea)`.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-98-analysis-options/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-126-non-returning-functions/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-127-changing-function-bounds/
[4] https://hex-rays.com/blog/igor-tip-of-the-week-09-reanalysis/
[5] https://www.hex-rays.com/products/ida/support/idadoc/681.shtml

#135: Exporting disassembly from IDA

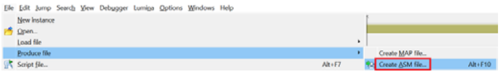
07 Apr 2023

https://hex-rays.com/blog/igors-tip-of-the-week-135-exporting-disassembly-from-ida/

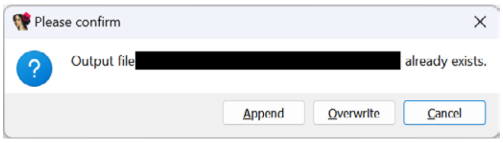
Although most of the time you can probably do all of the reversing inside IDA, occasionally you may need to continue it using other tools. While sometimes it may be enough to analyze the input file with another tool, or use the Export Data[1] feature, the disassembly listing is more convenient in many cases. Of course, you can use the clipboard to copy some snippets, but this can be awkward and slow if you need big chunks of the listing, or need to remove unnecessary parts of the listing such as the address prefixes.

ASM file

ASM files can be generated by using the menu entry File > Produce File > Create ASM File..., or the shortcut `Alt-F10`.



By default, the contents of the whole database is exported, but you can select a range[2] before invoking the command to limit it to just what you need. If you need multiple fragments, you can repeat the action several time, choosing “Append” when IDA informs you that the file already exists.

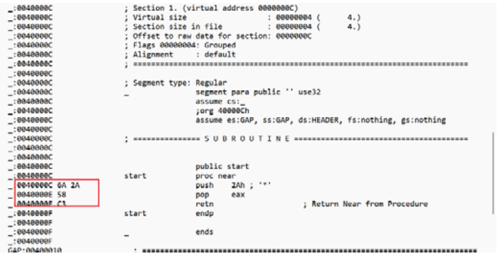


Output file [redacted] already exists.

In ideal circumstances, the ASM listing can be passed to the assembler to generate code equivalent to the original binary. It means it does not contain extra annotations which may be present in IDA, such as address prefixes or opcode bytes[3]. Of course, the reality is often not so simple, but minor modification to the ASM file may be enough to solve your problem.

LST file

The LST file can be generated via the menu entry File > Produce File > Create LST File... (no default shortcut). Unlike the ASM file, it contains all the information present in IDA’s text view, so it can be useful if you want to see opcode bytes[3] or address prefixes.



```
_:0040000C                 ; Section 1. (virtual address 0000000C)
_:0040000C                 ; Virtual size                  : 00000004 (      4.)
_:0040000C                 ; Section size in file          : 00000004 (      4.)
_:0040000C                 ; Offset to raw data for section: 0000000C
_:0040000C                 ; Flags 00000004: Grouped
_:0040000C                 ; Alignment     : default
_:0040000C                 ; ===========================================================================
_:0040000C
_:0040000C                 ; Segment type: Regular
_:0040000C                 _               segment para public '' use32
_:0040000C                                 assume cs:_
_:0040000C                                 ;org 40000Ch
_:0040000C                                 assume es:GAP, ss:GAP, ds:HEADER, fs:nothing, gs:nothing
_:0040000C
_:0040000C                 ; =============== S U B R O U T I N E =======================================
_:0040000C
_:0040000C
_:0040000C                                 public start
_:0040000C                 start           proc near
_0040000C 6A 2A                            push    2Ah ; '*'
_0040000E 58                               pop     eax
_0040000F C3                               retn                    ; Return Near from Procedure
_:0040000F                 start           endp
_:0040000F
_:0040000F                 _               ends
_:0040000F
GAP:00400010                ; ===========================================================================
```

Protip

The ASM or LST file usually needs at least one line of text per each instruction or data item. If your database contains large data areas, converting them to arrays[4] before exporting can reduce the size of the output files significantly. Hiding or collapsing[5] uninteresting areas or whole segments is another option.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-39-export-data/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-123-opcode-bytes/
[4] https://hex-rays.com/blog/igor-tip-of-the-week-10-working-with-arrays/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-31-hiding-and-collapsing/

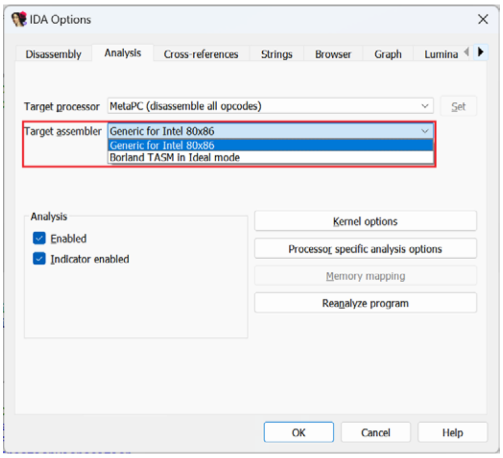
#136: Changing assembler syntax

14 Apr 2023

https://hex-rays.com/blog/igors-tip-of-the-week-136-changing-assembler-syntax/

When exporting disassembly[1], sometimes you need to modify it so that it is accepted by a specific assembler you're using. One little-known fact is that some of IDA's processor modules support different assembler syntaxes, so it may be useful to try a different one to see if it matches your needs better.

The assembler can be changed via Options > General..., Analysis tab:



Target processor: MetaPC (disassemble all opcodes)

Target assembler: Generic for Intel 80x86

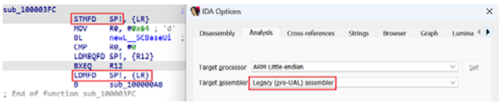
- Generic for Intel 80x86
- Borland TASM in Ideal mode

☑ Enabled

☑ Indicator enabled

For example, on x86 the TASM Ideal syntax may be selected instead of the default Generic one (based on MASM). One feature of this syntax is that it always uses brackets for instructions which dereference memory pointers.

For ARM, you can choose a legacy assembler, which was used before introduction of UAL (unified assembly language) with Thumb-2. For example, it used explicit `STMFD` and `LDMFD` instructions instead of the more convenient `PUSH` and `POP` introduced for Thumb.

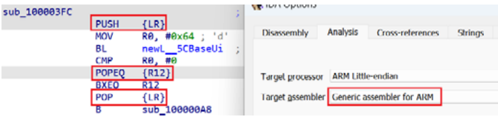


```
sub_100003FC
                STMFD   SP!, {LR}
                MOV     R0, #0x64 ; 'd'
                BL      newL__5CBaseUi
                CMP     R0, #0
                LDMEQFD SP!, {R12}
                BXEQ    R12
                LDMFD   SP!, {LR}
                B       sub_100000A8
; End of function sub_100003FC
```

Target processor: ARM Little-endian

Target assembler: Legacy (pre-UAL) assembler

Nowadays, IDA defaults to the generic UAL assembler which is de-facto standard and easier to read.

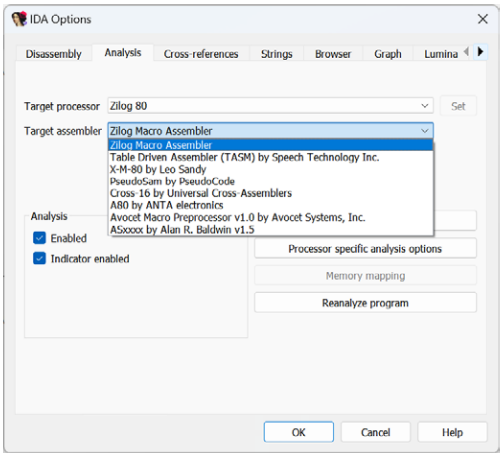


```
sub_100003FC
                PUSH    {LR}
                MOV     R0, #0x64 ; 'd'
                BL      newL__5CBaseUi
                CMP     R0, #0
                POPEQ   {R12}
                BXEQ    R12
                POP     {LR}
                B       sub_100000A8
```

Target processor: ARM Little-endian

Target assembler: Generic assembler for ARM

For some of the older processors the selection of assemblers can be quite extensive; they often didn't have a freely available official assembler so many third-party alternatives were available.



Target processor: Zilog 80

Target assembler: Zilog Macro Assembler

- Zilog Macro Assembler
- Table Driven Assembler (TASM) by Speech Technology Inc.
- X-M-80 by Leo Sandy
- PseudoSam by PseudoCode
- Cross-16 by Universal Cross-Assemblers
- A80 by ANTA electronics
- Avocet Macro Preprocessor v1.0 by Avocet Systems, Inc.
- ASxxxx by Alan R. Baldwin v1.5

☑ Enabled

☑ Indicator enabled

[1] https://hex-rays.com/blog/igors-tip-of-the-week-135-exporting-disassembly-from-ida/

#137: Processor modes and segment registers

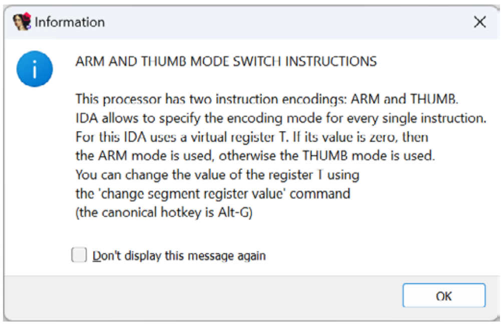
21 Apr 2023

https://hex-rays.com/blog/igors-tip-of-the-week-137-processor-modes-and-segment-registers/

Some of the processors supported by IDA support different ISA variants, in particular:

- ARM processor module supports the classic 32-bit ARM instructions (A32), 16-bit Thumb or mixed 16/32-bit Thumb32 (T32) , as well as 64-bit A64 instructions (A64)
- PPC processor module supports the standard 32-bit PowerPC instructions and mixed 16/32-bit Variable Length Environment (VLE)
- MIPS module supports the classic 32-bit instructions as well as the compressed variants MIPS16 and microMIPS

Because sometimes these instructions sets may be present in the same binary, IDA needs a way to determine which subset to use. For this, it repurposes segment registers, originally used on 16-bit x86 processors to extend the 16-bit addressing. For example, if you load an ARM firmware binary, you will see the following informational box:



Information

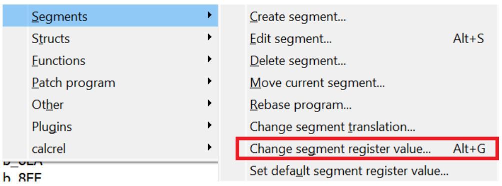
ARM AND THUMB MODE SWITCH INSTRUCTIONS

This processor has two instruction encodings: ARM and THUMB.
IDA allows to specify the encoding mode for every single instruction.
For this IDA uses a virtual register T. If its value is zero, then
the ARM mode is used, otherwise the THUMB mode is used.
You can change the value of the register T using
the 'change segment register value' command
(the canonical hotkey is Alt-G)

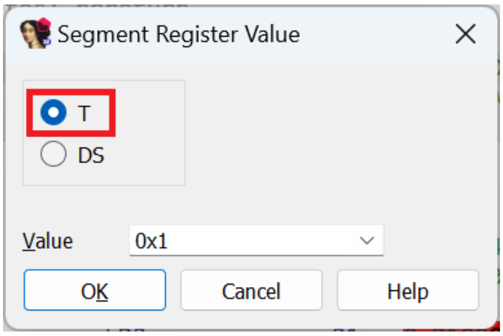
☐ Don't display this message again

OK

In many cases, IDA is able to determine the correct processor mode by analyzing the code and determining mode switch sequences (e.g. BX/BLX instructions), but you can also force its decision by using the described shortcut `Alt-G` (if you prefer menus, you can find it in Edit > Segments > Change segment register value…).



In the dialog, select the **T** register and specify `0` for ARM mode or `1` for Thumb (includes Thumb32 aka Thumb-2).



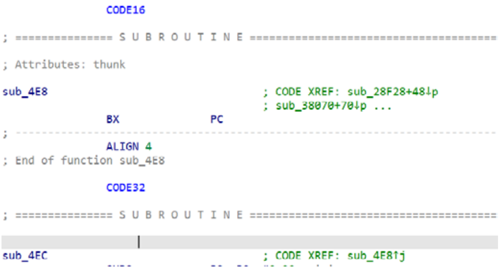
Segment Register Value

● T
○ DS

Value 0x1

OK Cancel Help

You can observe mode switches in the disassembly listing by the `CODE32`/`CODE16` directives (usually text view only):



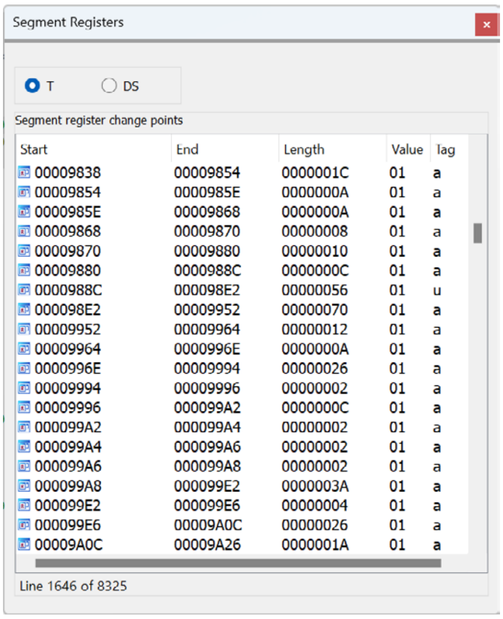
```
                CODE16

; =============== S U B R O U T I N E =======================================

; Attributes: thunk

sub_4E8                                 ; CODE XREF: sub_28F28+48↓p
                                        ; sub_38070+70↓p ...
                BX              PC
; ---------------------------------------------------------------------------
                ALIGN 4
; End of function sub_4E8

                CODE32

; =============== S U B R O U T I N E =======================================

sub_4EC                                 ; CODE XREF: sub_4E8↑j
```

#137: Processor modes and segment registers

21 Apr 2023

https://hex-rays.com/blog/igors-tip-of-the-week-137-processor-modes-and-segment-registers/

If you need a global overview, use the View> Open subviews > Segment registers…. (`Shift– F8`) view or its modal version Jump > Jump to segment (`Ctrl– G`):



● T ○ DS

Segment register change points

Start	End	Length	Value	Tag
00009838	00009854	0000001C	01	a
00009854	0000985E	0000000A	01	a
0000985E	00009868	0000000A	01	a
00009868	00009870	00000008	01	a
00009870	00009880	00000010	01	a
00009880	0000988C	0000000C	01	a
0000988C	000098E2	00000056	01	u
000098E2	00009952	00000070	01	a
00009952	00009964	00000012	01	a
00009964	0000996E	0000000A	01	a
0000996E	00009994	00000026	01	a
00009994	00009996	00000002	01	a
00009996	000099A2	0000000C	01	a
000099A2	000099A4	00000002	01	a
000099A4	000099A6	00000002	01	a
000099A6	000099A8	00000002	01	a
000099A8	000099E2	0000003A	01	a
000099E2	000099E6	00000004	01	a
000099E6	00009A0C	00000026	01	a
00009A0C	00009A26	0000001A	01	a

Line 1646 of 8325

The Tag column gives a hint on how the specific changepoint was created: **a** denotes a changepoint added by IDA during autoanalysis while u is used for those specified by the user (or, sometimes a plugin).

If necessary, wrong changepoints can be deleted from the list (even many at a time, using the selection). When a change point is deleted, IDA uses the value of a preceding one (or the default for the current segment).

For MIPS, the **mips16** pseudoregister is used to switch between standard MIPS and MIPS16 or microMIPS, and for PPC, vle is used to enable decoding of **VLE** instructions.

See also:
IDA Help: Segment Register Change Points[1]
IDA Help: Jump to the specified segment register change point[2]

[1] https://www.hex-rays.com/products/ida/support/idadoc/524.shtml
[2] https://www.hex-rays.com/products/ida/support/idadoc/547.shtml

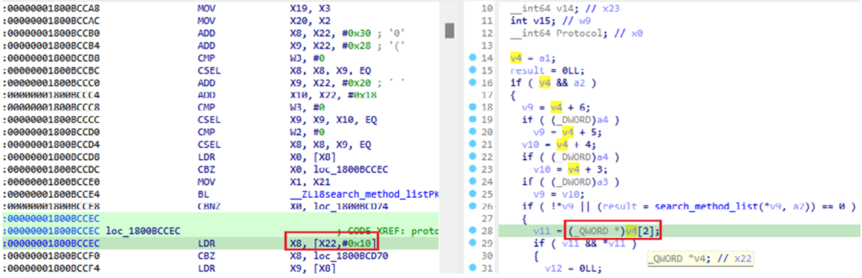
#138: Pointer math in the decompiler

28 Apr 2023

https://hex-rays.com/blog/igors-tip-of-the-week-138-pointer-math-in-the-decompiler/

While working with decompiled code and retyping variables (or sometimes when they get typed by the decompiler automatically), you might be puzzled by the discrepancies between pseudocode and disassembly.

Consider the following example:



```
:00000001800BCCA8                 MOV             X19, X3
:00000001800BCCAC                 MOV             X20, X2
:00000001800BCCB0                 ADD             X8, X22, #0x30 ; '0'
:00000001800BCCB4                 ADD             X9, X22, #0x28 ; '('
:00000001800BCCB8                 CMP             W3, #0
:00000001800BCCBC                 CSEL            X8, X8, X9, EQ
:00000001800BCCC0                 ADD             X9, X22, #0x20 ; ' '
:00000001800BCCC4                 ADD             X10, X22, #0x18
:00000001800BCCC8                 CMP             W3, #0
:00000001800BCCCC                 CSEL            X9, X9, X10, EQ
:00000001800BCCD0                 CMP             W2, #0
:00000001800BCCD4                 CSEL            X8, X8, X9, EQ
:00000001800BCCD8                 LDR             X0, [X8]
:00000001800BCCDC                 CBZ             X0, loc_1800BCCEC
:00000001800BCCE0                 MOV             X1, X21
:00000001800BCCE4                 BL              __ZL18search_method_listPK
:00000001800BCCE8                 CBNZ            X0, loc_1800BCD74
:00000001800BCCEC
:00000001800BCCEC loc_1800BCCEC                          ; CODE XREF: protc
:00000001800BCCEC                 LDR             X8, [X22,#0x10]
:00000001800BCCF0                 CBZ             X8, loc_1800BCD70
:00000001800BCCF4                 LDR             X9, [X8]
```

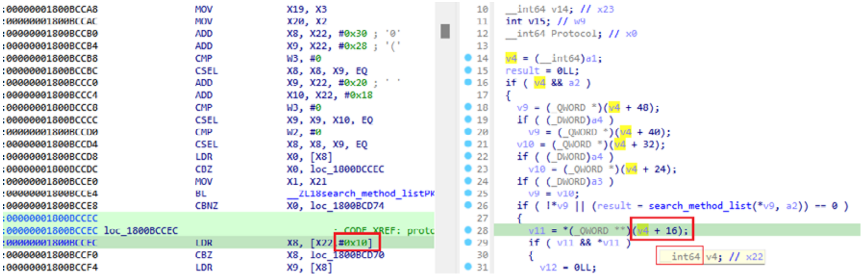
```
__int64 v14; // x23
int v15; // w9
__int64 Protocol; // x0

v4 = a1;
result = 0LL;
if ( v4 && a2 )
{
  v9 = v4 + 6;
  if ( (_DWORD)a4 )
    v9 = v4 + 5;
  v10 = v4 + 4;
  if ( (_DWORD)a4 )
    v10 = v4 + 3;
  if ( (_DWORD)a3 )
    v9 = v10;
  if ( !*v9 || (result = search_method_list(*v9, a2)) == 0 )
  {
    v11 = (_QWORD *)v4[2];
    if ( v11 && *v11 )
    {
      v12 = 0LL;
```

_QWORD *v4; // x22

We see that `X22` is accessed with offset 0x10 (16) in the disassembly but 2 in the pseudocode. Is there a bug in the decompiler?

In fact, there is no bug. The difference is explained by the C/C++pointer/array referencing rules: the array indexing or integer addition operation advances the pointer value by the value of index *multiplied by the element size*. In this case, the type of `v4` is `_QWORD*`, which means that elements are `_QWORD`s (64-bit or 8-byte integers). Thus, 2*8=16(0x10), which matches the assembly code.

To confirm what's really going on, you can do "Reset pointer type" on the variable so that it reverts to the generic integer variable and the decompiler is forced to use raw byte offsets:



```
:00000001800BCCA8                 MOV             X19, X3
:00000001800BCCAC                 MOV             X20, X2
:00000001800BCCB0                 ADD             X8, X22, #0x30 ; '0'
:00000001800BCCB4                 ADD             X9, X22, #0x28 ; '('
:00000001800BCCB8                 CMP             W3, #0
:00000001800BCCBC                 CSEL            X8, X8, X9, EQ
:00000001800BCCC0                 ADD             X9, X22, #0x20 ; ' '
:00000001800BCCC4                 ADD             X10, X22, #0x18
:00000001800BCCC8                 CMP             W3, #0
:00000001800BCCCC                 CSEL            X9, X9, X10, EQ
:00000001800BCCD0                 CMP             W2, #0
:00000001800BCCD4                 CSEL            X8, X8, X9, EQ
:00000001800BCCD8                 LDR             X0, [X8]
:00000001800BCCDC                 CBZ             X0, loc_1800BCCEC
:00000001800BCCE0                 MOV             X1, X21
:00000001800BCCE4                 BL              __ZL18search_method_listPK
:00000001800BCCE8                 CBNZ            X0, loc_1800BCD74
:00000001800BCCEC
:00000001800BCCEC loc_1800BCCEC                          ; CODE XREF: protc
:00000001800BCCEC                 LDR             X8, [X22,#0x10]
:00000001800BCCF0                 CBZ             X8, loc_1800BCD70
:00000001800BCCF4                 LDR             X9, [X8]
```

```
__int64 v14; // x23
int v15; // w9
__int64 Protocol; // x0

v4 = (__int64)a1;
result = 0LL;
if ( v4 && a2 )
{
  v9 = (_QWORD *)(v4 + 48);
  if ( (_DWORD)a4 )
    v9 = (_QWORD *)(v4 + 40);
  v10 = (_QWORD *)(v4 + 32);
  if ( (_DWORD)a4 )
    v10 = (_QWORD *)(v4 + 24);
  if ( (_DWORD)a3 )
    v9 = v10;
  if ( !*v9 || (result = search_method_list(*v9, a2)) == 0 )
  {
    v11 = *(_QWORD **)(v4 + 16);
    if ( v11 && *v11 )
    {
      v12 = 0LL;
```

__int64 v4; // x22

See also:

Igor's Tip of the Week #117: Reset pointer type[1]

Igor's tip of the week #42: Renaming and retyping in the decompiler[2]

Igor's Tip of the Week #118: Structure creation in the decompiler[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-117-reset-pointer-type/

[2] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/

[3] https://hex-rays.com/blog/igors-tip-of-the-week-118-structure-creation-in-the-decompiler/

#139: License borrowing

05 May 2023

https://hex-rays.com/blog/igors-tip-of-the-week-139-license-borrowing/

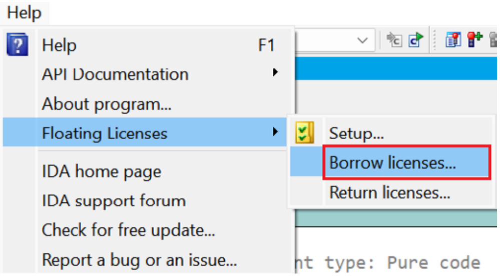
Floating licenses allow additional flexibility for companies with many IDA users: IDA can be installed on as many computers as required, but only a limited number of copies can run simultaneously.

This flexibility its downsides: IDA needs to have permanent connection to your organization’s license server which may make things problematic in some situations (e.g. working on an isolated network or in the field/while traveling). While the first issue can be handled by placing the license server inside that network, accessing the company network during travel may be problematic or impossible. In such situations, you can use license borrowing.

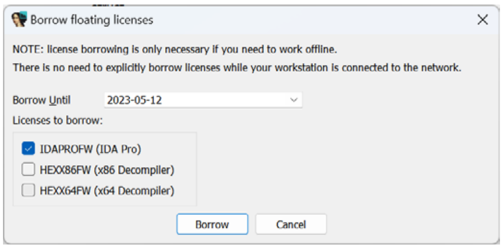
Borrowing allows the user to check out the license for a fixed period and work without connection to the server during that time.

Borrowing licenses

To borrow a license, in a floating-license IDA go to Help > Floating licenses > Borrow licenses...



You will get a dialog like the following:



Borrow floating licenses

NOTE: license borrowing is only necessary if you need to work offline.
There is no need to explicitly borrow licenses while your workstation is connected to the network.

Borrow Until 2023-05-12

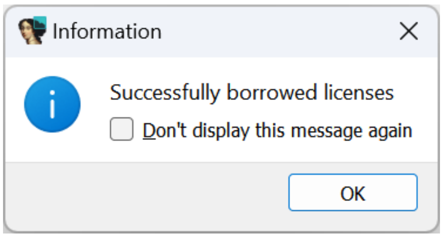
Licenses to borrow:

☑ IDAPROFW (IDA Pro)
☐ HEXX86FW (x86 Decompiler)
☐ HEXX64FW (x64 Decompiler)

Borrow Cancel

Here you can pick which licenses you want to borrow and the borrow period end date. By default, IDA offers one week but you can make it shorter or longer (by default we limit the maximum borrow period to 6 months but it can be limited further by the license server administrator).

If you click “Borrow”, you should see this confirmation:



Information

Successfully borrowed licenses

☐ Don't display this message again

OK

and the details in the Output window:

```
Successfully borrowed licenses:
IDAPROFW (IDA Pro) [currently borrowed until 2023-05-12 23:59]
```

After this, you can disconnect from the network and IDA will continue working until the specified date.

NB: once borrowed, the license(s) remain checked out (“In Use”) on the license server and will not become available for others until the end of the borrow period or early return.

#139: License borrowing

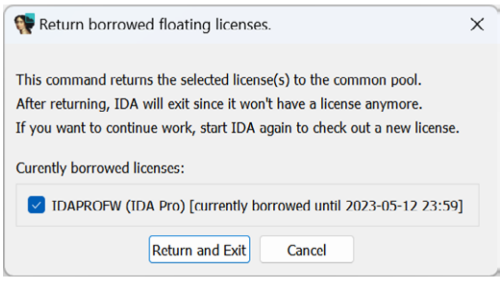
05 May 2023

https://hex-rays.com/blog/igors-tip-of-the-week-139-license-borrowing/

Returning licenses

If you need to return borrowed licenses early (before the end of the borrow period):

1. Reconnect to the network with the server from which you borrowed the license
2. Go to Help > Floating licenses > Return licenses



This command returns the selected license(s) to the common pool.
After returning, IDA will exit since it won't have a license anymore.
If you want to continue work, start IDA again to check out a new license.

Curently borrowed licenses:

☑ IDAPROFW (IDA Pro) [currently borrowed until 2023-05-12 23:59]

3. select the license(s) to return and click “Return and Exit”.
4. IDA will exit since it has returned the license, but you can start it again to use the license server in online mode or borrow again for another period.

Borrowing and returning licenses from command line

If you prefer using command line, check the corresponding section on our support page[1].

See also: Floating Licenses[2]

[1] https://hex-rays.com/products/ida/support/flexlm/#borrow
[2] https://hex-rays.com/products/ida/support/flexlm/

#140: Loading PDB types

12 May 2023

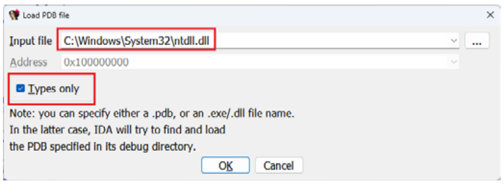
https://hex-rays.com/blog/igors-tip-of-the-week-140-loading-pdb-types/

While IDA comes with a rich set of type libraries[1] for Windows API, they don't cover the whole set of types used in Windows. Our libraries are based on the official Windows SDK/DDK headers, which tend to only include public, stable information which is common to multiple Windows versions. A new Windows build may introduce new types or use some of the previously reserved fields. Because some of these structures are critical for proper debugging, Microsoft usually publishes a subset of actual, up-to-date types in the PDBs for the core Windows modules (`kernel32.dll` and `ntdll.dll` for user mode, `ntoskrnl.exe` for kernel mode). Thus, usually you can use these files to get types matching the Windows version you're analyzing.

Loading types from PDB

To load an additional PDB file, use File > Load file > PDB File…

Here, you can specify either an already downloaded PDB, or a path to .exe or .dll. In the latter case, IDA will try to fetch the matching PDB from the symbol servers. Because we're loading the PDB which does not actually match the currently loaded file, check "Types only" so that the global symbols from it are not applied unnecessarily.



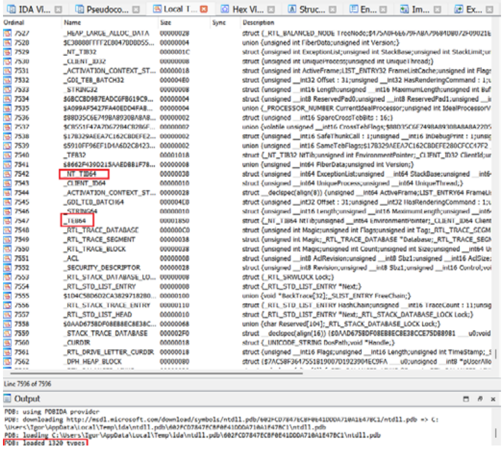
Input file: C:\Windows\System32\ntdll.dll

Address: 0x100000000

☑ Types only

Note: you can specify either a .pdb, or an .exe/.dll file name.
In the latter case, IDA will try to find and load
the PDB specified in its debug directory.

After downloading and processing the PDB, the new types can be consulted in the Local Types view.



Ordinal	Name	Size	Sync	Description
7527	_HEAP_LARGE_ALLOC_DATA	00000028		struct {_RTL_BALANCED_NODE TreeNode;$475A0F6E679FABA79684D8072F09021E
7528	$E38888FFFF2E8047BD8D55...	00000004		union {unsigned int FiberData;unsigned int Version;}
7529	_NT_TIB32	0000001C		struct {unsigned int ExceptionList;unsigned int StackBase;unsigned int StackLimit;ur
7530	_CLIENT_ID32	00000008		struct {unsigned int UniqueProcess;unsigned int UniqueThread;}
7531	_ACTIVATION_CONTEXT_ST...	00000018		struct {unsigned int ActiveFrame;LIST_ENTRY32 FrameListCache;unsigned int Flags
7532	_GDI_TEB_BATCH32	000004E0		struct {unsigned __int32 Offset : 31;unsigned __int32 HasRenderingCommand : 1;u
7533	_STRING32	00000008		struct {unsigned __int16 Length;unsigned __int16 MaximumLength;unsigned int Buf
7534	$6BCC8D987EADC6FB619C9...	00000004		struct {unsigned __int8 ReservedPad0;unsigned __int8 ReservedPad1;unsigned __i
7535	$A099AF5427FA40EDD4FAB...	00000004		union {_PROCESSOR_NUMBER CurrentIdealProcessor;unsigned int IdealProcessorV
7536	$88D35C6E749BA8930BA8A8...	00000002		struct {unsigned __int16 SpareCrossTebBits : 16;}
7537	$CB551F47A7D672B4CB2B6F...	00000002		union {volatile unsigned __int16 CrossTebFlags;$88D35C6E749BA8930BA8A8A22D5
7538	$17B329AEEA7C162CBDEFE2...	00000002		struct {unsigned __int16 SafeThunkCall : 1;unsigned __int16 InDebugPrint : 1;unsig
7539	$5910FF96EF1D4A6D2C8423...	00000002		union {unsigned __int16 SameTebFlags;$17B329AEEA7C162CBDEFE280CFCC47F2
7540	_TEB32	00001018		struct {_NT_TIB32 NtTib;unsigned int EnvironmentPointer;_CLIENT_ID32 ClientId;ur
7541	$8662F439D215AAEDBB1F78...	00000008		union {unsigned __int64 FiberData;unsigned int Version;}
7542	_NT_TIB64	00000038		struct {unsigned __int64 ExceptionList;unsigned __int64 StackBase;unsigned __int6
7543	_CLIENT_ID64	00000010		struct {unsigned __int64 UniqueProcess;unsigned __int64 UniqueThread;}
7544	_ACTIVATION_CONTEXT_ST...	00000028		struct __declspec(align(8)) {unsigned __int64 ActiveFrame;LIST_ENTRY64 FrameLi
7545	_GDI_TEB_BATCH64	000004E8		struct {unsigned __int32 Offset : 31;unsigned __int32 HasRenderingCommand : 1;u
7546	_STRING64	00000010		struct {unsigned __int16 Length;unsigned __int16 MaximumLength;unsigned __int6
7547	_TEB64	00001850		struct {_NT_TIB64 NtTib;unsigned __int64 EnvironmentPointer;_CLIENT_ID64 Client
7548	_RTL_TRACE_DATABASE	000000C0		struct {unsigned int Magic;unsigned int Flags;unsigned int Tag;_RTL_TRACE_SEGM
7549	_RTL_TRACE_SEGMENT	00000038		struct {unsigned int Magic;_RTL_TRACE_DATABASE *Database;_RTL_TRACE_SEG
7550	_RTL_TRACE_BLOCK	00000038		struct {unsigned int Magic;unsigned int Count;unsigned int Size;unsigned __int64 U
7551	_ACL	00000008		struct {unsigned __int8 AclRevision;unsigned __int8 Sbz1;unsigned __int16 AclSize;
7552	_SECURITY_DESCRIPTOR	00000028		struct {unsigned __int8 Revision;unsigned __int8 Sbz1;unsigned __int16 Control;voi
7553	_RTL_STACK_DATABASE_LO...	00000008		struct {_RTL_SRWLOCK Lock;}
7554	_RTL_STD_LIST_ENTRY	00000008		struct {_RTL_STD_LIST_ENTRY *Next;}
7555	$1D4C58D6D2CA3829718280...	00000100		union {void *BackTrace[32];_SLIST_ENTRY FreeChain;}
7556	_RTL_STACK_TRACE_ENTRY	00000110		struct {_RTL_STD_LIST_ENTRY HashChain;unsigned __int16 TraceCount : 11;unsig
7557	_RTL_STD_LIST_HEAD	00000010		struct {_RTL_STD_LIST_ENTRY *Next;_RTL_STACK_DATABASE_LOCK Lock;}
7558	$0AAD6758DF08EB8EC8E38C...	00000068		union {char Reserved[104];_RTL_STACK_DATABASE_LOCK Lock;}
7559	_STACK_TRACE_DATABASE	000002F0		struct __declspec(align(16)) {$0AAD6758DF08EB8EC8E38CCE75D88981 __u0;void
7560	_CURDIR	00000018		struct {_UNICODE_STRING DosPath;void *Handle;}
7561	_RTL_DRIVE_LETTER_CURDIR	00000018		struct {unsigned __int16 Flags;unsigned __int16 Length;unsigned int TimeStamp;_S
7562	_DPH_HEAP_BLOCK	00000080		struct {$7AC58F3647551819007D1923904EC9FA __u0;unsigned __int8 *pUserAllo

Line 7596 of 7596

```
PDB: using PDBIDA provider
PDB: downloading http://msdl.microsoft.com/download/symbols/ntdll.pdb/602FCD7847EC8F0E41DDDA710A1E47BC1/ntdll.pdb => C:
\Users\Igor\AppData\Local\Temp\ida\ntdll.pdb\602FCD7847EC8F0E41DDDA710A1E47BC1\ntdll.pdb
PDB: loading C:\Users\Igor\AppData\Local\Temp\ida\ntdll.pdb\602FCD7847EC8F0E41DDDA710A1E47BC1\ntdll.pdb
PDB: loaded 1320 types
```

See also:
Igor's tip of the week #55: Using debug symbols[2]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-55-using-debug-symbols/

#141: Parsing C files

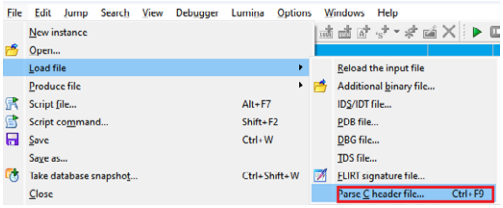
19 May 2023

https://hex-rays.com/blog/igors-tip-of-the-week-141-parsing-c-files/

Previosuly, we've covered creating structures from C code using the Local Types window[1], however this may be not very convenient when you have complex types with many dependencies (especially of scattered over several fiels or depending on preprocessor defines). In such case it may be nore convenient to parse the original header file(s) on disk.

Parsing header files

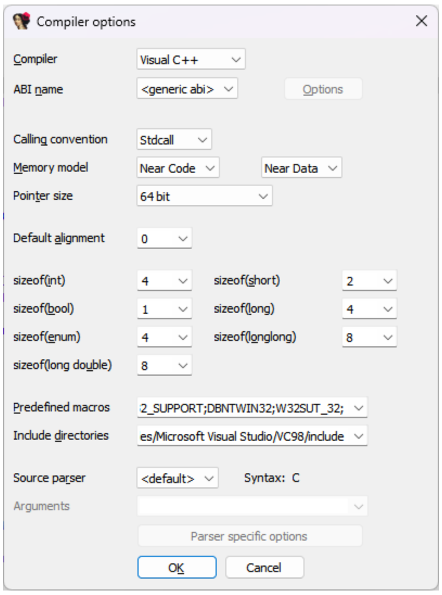
If you happen to have the types you need in a header file, you can try using IDA's built-in C parser via the File > Load file > Parse C header file… (shortcut `Ctrl+ F9`).



Just like a compiler, IDA will handle the preprocessor directives (`#include`, `#define`, `#ifdef` and so on), and add any types discovered to the Local Types list, from where they can be used in the decompiler (or the disassembly, after importing into the IDB).

Setting compiler options

IDA's built-in parser can mimic several popular compilers, including Visual C++, GCC (and compatibles), Borland C++ Builder, or Watcom. For many stuctured files the compiler is preset to a detected or guessed value, but you can also change or set it via Options > Compiler… dialog:


Compiler options

Compiler: Visual C++

ABI name: <generic abi>

Calling convention: Stdcall

Memory model: Near Code, Near Data

Pointer size: 64 bit

Default alignment: 0

sizeof(int): 4
sizeof(short): 2
sizeof(bool): 1
sizeof(long): 4
sizeof(enum): 4
sizeof(longlong): 8
sizeof(long double): 8

Predefined macros: 2_SUPPORT;DBNTWIN32;W32SUT_32;

Include directories: es/Microsoft Visual Studio/VC98/include

Source parser: <default> Syntax: C

Arguments:

In this dialog you can also adjust settings necessary for the preprocessing step, such as the predefined preprocessor macros (#defines) or the include paths for the #include directives. They are pre-filled from the `CC_PARMS` setting in `ida.cfg`.

Clang parser

The built-in parser is quite basic and handles mostly simple C syntax or very basic C++ (e.g templates are not supported). If you have complex files employing new, modern C or C++ features, you may have more luck using the Clang-based parser added in IDA 7.7. It can be selected in the Source parser dropdown of the compiler options dialog and will be used next time you invoke the Parse C header file command. For the details on using it, see the dedicated IDAClang tutorial[2].

See also:
IDA Help: Load C header[3]
IDA Help: Compiler[4]
Igor's tip of the week #62: Creating custom type libraries[5]
Introducing the IDAClang Tutorial[2]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-11-quickly-creating-structures/
[2] https://hex-rays.com/tutorials/idaclang/
[3] https://www.hex-rays.com/products/ida/support/idadoc/1367.shtml
[4] https://www.hex-rays.com/products/ida/support/idadoc/1354.shtml
[5] https://hex-rays.com/blog/igors-tip-of-the-week-62-creating-custom-type-libraries/

#142: Mapping local types

26 May 2023

https://hex-rays.com/blog/igors-tip-of-the-week-142-mapping-local-types/

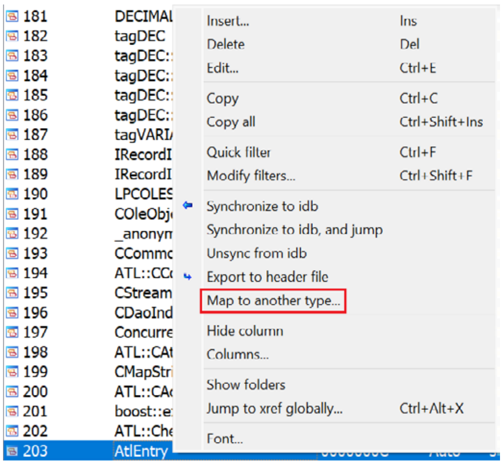
When working on a binary, you often recover types used in it from many sources:

- creating structures manually, from data[1], or using decompiler[2];
- parsing header files[3];
- importing them from type libraries[4] or debug information[5];

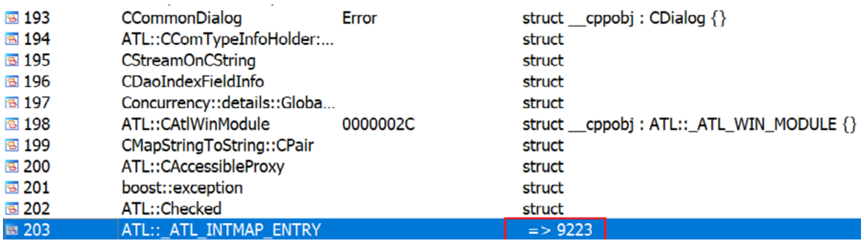
However, it may happen that eventually you discover duplicates. For example, you find out that the "custom" structure you've been building up is actually a well-known type and you found the correct definition in debug info or header files. Or, after analyzing two different functions, you only find out later that two structures are, in fact, one and the same. Of course, you can go and replace all references to the "wrong" one manually, which is doable if you discover this early, but if you already have hundreds of functions or other types referring to it, the process can become tedious.

Type mapping

To map a type to another, open the Local Types window (`Shift-F1`), and choose "Map to another type..." from the context menu on the type you want to map.


181	DECIMAL
182	tagDEC
183	tagDEC:
184	tagDEC:
185	tagDEC:
186	tagDEC:
187	tagVARI
188	IRecordI
189	IRecordI
190	LPCOLES
191	COleObj
192	_anonym
193	CCommc
194	ATL::CC
195	CStream
196	CDaoInd
197	Concurre
198	ATL::CA
199	CMapStr
200	ATL::CA
201	boost::e
202	ATL::Che
203	AtlEntry

After choosing the type to replace it, the original type is deleted and all references to it are redirected to the new one. This is indicated by the arrow sign pointing to the new type's definition.


193	CCommonDialog	Error	struct __cppobj : CDialog {}
194	ATL::CComTypeInfoHolder:...		struct
195	CStreamOnCString		struct
196	CDaoIndexFieldInfo		struct
197	Concurrency::details::Globa...		struct
198	ATL::CAtlWinModule	0000002C	struct __cppobj : ATL::_ATL_WIN_MODULE {}
199	CMapStringToString::CPair		struct
200	ATL::CAccessibleProxy		struct
201	boost::exception		struct
202	ATL::Checked		struct
203	ATL::_ATL_INTMAP_ENTRY		=> 9223

All uses of the old type in the function prototypes, local variable types etc. are replaced by the new type automatically.

See also:

IDA Help: Local types window[6]

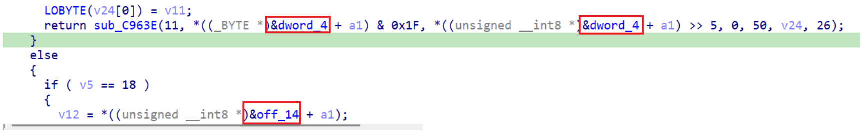
[1] https://hex-rays.com/blog/igor-tip-of-the-week-11-quickly-creating-structures/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-118-structure-creation-in-the-decompiler/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-141-parsing-c-files/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-140-loading-pdb-types/
[6] https://www.hex-rays.com/products/ida/support/idadoc/1259.shtml

#143: Fixing wrong address references in the decompiler

02 Jun 2023

https://hex-rays.com/blog/igors-tip-of-the-week-143-fixing-wrong-address-references-in-the-decompiler/

When decompiling code without high-level metadata (especially firmware), you may observe strange-looking address expressions which do not seem to make sense.



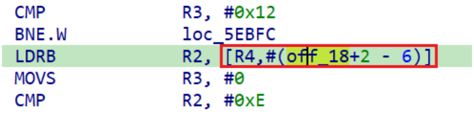
```
    LOBYTE(v24[0]) = v11;
    return sub_C963E(11, *((_BYTE *)&dword_4 + a1) & 0x1F, *((unsigned __int8 *)&dword_4 + a1) >> 5, 0, 50, v24, 26);
  }
  else
  {
    if ( v5 == 18 )
    {
      v12 = *((unsigned __int8 *)&off_14 + a1);
```

What are these and how to fix/improve the pseudocode?

Because on the CPU level there is no difference between an address and a simple number[1], distinguishing addresses and plain numbers is a difficult task which is not solvable in general case without actually executing the code. IDA uses some heuristics to try and detect when a number looks like an address and convert such numbers to offsets[2], but such heuristics are not always reliable and may lead to false positives. This can be especially bad when the database has valid addresses around 0, because then many small numbers look like addresses. The decompiler relies on IDA's analysis and uses the information provided by it to produce the pseudocode which is supposed to faithfully represent behavior of the machine code. However, this can backfire in case the analysis made a mistake. Thankfully, IDA is interactive and allows you to fix almost anything.

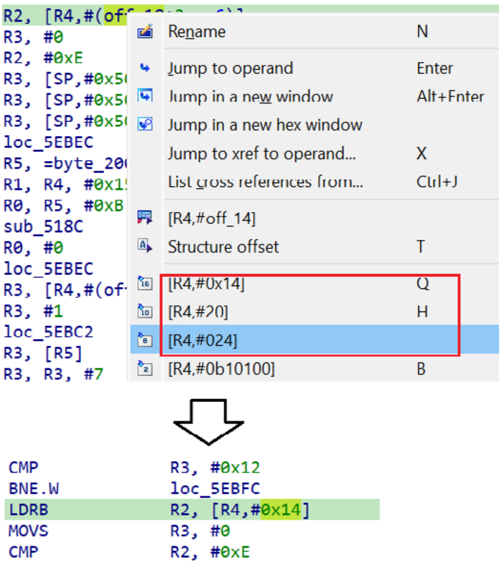
In situation like above, usually the simplest algorithm is as follows:

1. position cursor on the wrong address expression
2. press `Tab` to switch to disassembly. You should land on or close to the wrong offset expression. Note that it does not always match what you see in the pseudocode.



```
CMP                 R3, #0x12
BNE.W               loc_5EBFC
LDRB                R2, [R4,#(off_18+2 - 6)]
MOVS                R3, #0
CMP                 R2, #0xE
```

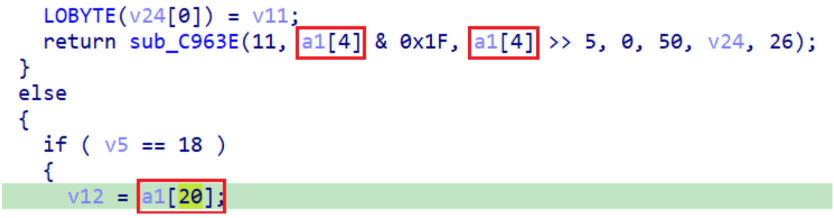
3. convert it to a plain number, e.g. by pressing `Q` (hex), `H` (decimal) or `#` (default).



```
R2, [R4,#(of
R3, #0
R2, #0xE
R3, [SP,#0x5
R3, [SP,#0x5
R3, [SP,#0x5
loc_5EBEC
R5, =byte_20
R1, R4, #0x1
R0, R5, #0xB
sub_518C
R0, #0
loc_5EBEC
R3, [R4,#(of
R3, #1
loc_5EBC2
R3, [R5]
R3, R3, #7
```

```
CMP               R3, #0x12
BNE.W             loc_5EBFC
LDRB              R2, [R4,#0x14]
MOVS              R3, #0
CMP               R2, #0xE
```

4. press `Tab` to switch back to pseudocode and `F5` to refresh it. The wrong expression should be converted to plain number or another context-dependent expression.



```
    LOBYTE(v24[0]) = v11;
    return sub_C963E(11, a1[4] & 0x1F, a1[4] >> 5, 0, 50, v24, 26);
  }
  else
  {
    if ( v5 == 18 )
    {
      v12 = a1[20];
```

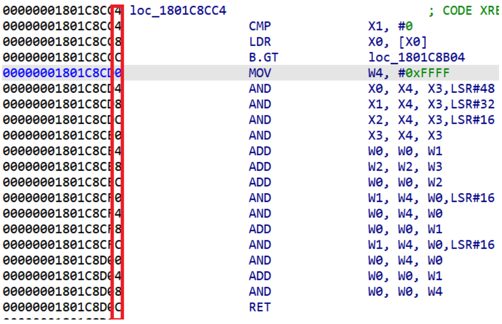
[1] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-95-offsets/

#144: Macros and simplified instructions

16 Jun 2023

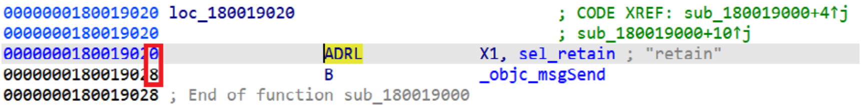
https://hex-rays.com/blog/igors-tip-of-the-week-144-macros-and-simplified-instructions/

Many processors (especially RISC based) use instruction sets with fixed size (most commonly 4 bytes). Among examples are ARM, PPC, MIPS and a few others. This is also obvious in the disassembly when observing the instructions' addresses – they increase by a fixed amount:



```
00000001801C8CC4 loc_1801C8CC4                              ; CODE XRE
00000001801C8CC4                CMP             X1, #0
00000001801C8CC8                LDR             X0, [X0]
00000001801C8CCC                B.GT            loc_1801C8B04
00000001801C8CD0                MOV             W4, #0xFFFF
00000001801C8CD4                AND             X0, X4, X3,LSR#48
00000001801C8CD8                AND             X1, X4, X3,LSR#32
00000001801C8CDC                AND             X2, X4, X3,LSR#16
00000001801C8CE0                AND             X3, X4, X3
00000001801C8CE4                ADD             W0, W0, W1
00000001801C8CE8                ADD             W2, W2, W3
00000001801C8CEC                ADD             W0, W0, W2
00000001801C8CF0                AND             W1, W4, W0,LSR#16
00000001801C8CF4                AND             W0, W4, W0
00000001801C8CF8                ADD             W0, W0, W1
00000001801C8CFC                AND             W1, W4, W0,LSR#16
00000001801C8D00                AND             W0, W4, W0
00000001801C8D04                ADD             W0, W0, W1
00000001801C8D08                AND             W0, W0, W4
00000001801C8D0C                RET
```

However, occasionally you may come across larger instructions:



```
0000000180019020 loc_180019020                          ; CODE XREF: sub_180019000+4↑j
0000000180019020                                        ; sub_180019000+10↑j
0000000180019020                ADRL            X1, sel_retain ; "retain"
0000000180019028                B               _objc_msgSend
0000000180019028 ; End of function sub_180019000
```

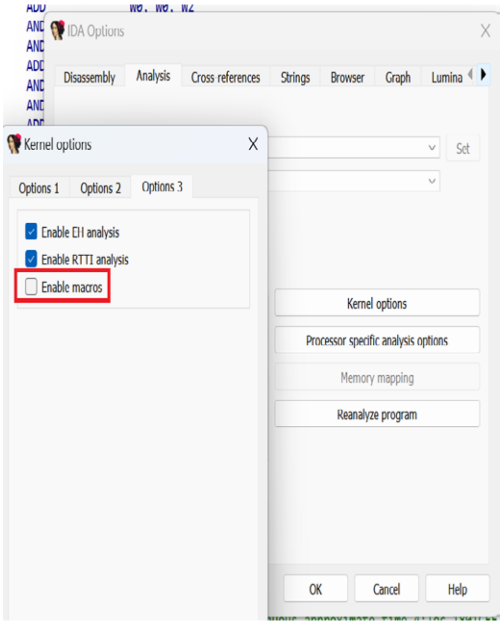
What is this? Does A64 ISA have 8-byte instructions?

In fact, if you check ARM's documentation[1], you'll discover that ADRL is a pseudo-instruction which generates two machine instructions, ADRP and ADD. IDA combines them to provide more compact disassembly and improve cross-references.

In IDA's terminology, a pseudo-instruction which replaces several simpler instructions is called a macro instruction.

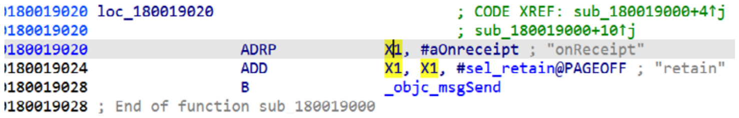
Disabling macros

If you prefer to see the actual instructions, you can disable macros. This can be done in the Kernel Options 3 group of settings:



☑ Enable EH analysis
☑ Enable RTTI analysis
☐ Enable macros

And now IDA no longer uses ADRL:



```
180019020 loc_180019020                          ; CODE XREF: sub_180019000+4↑j
180019020                                        ; sub_180019000+10↑j
180019020                ADRP            X1, #aOnreceipt ; "onReceipt"
180019024                ADD             X1, X1, #sel_retain@PAGEOFF ; "retain"
180019028                B               _objc_msgSend
180019028 ; End of function sub_180019000
```

#144: Macros and simplified instructions

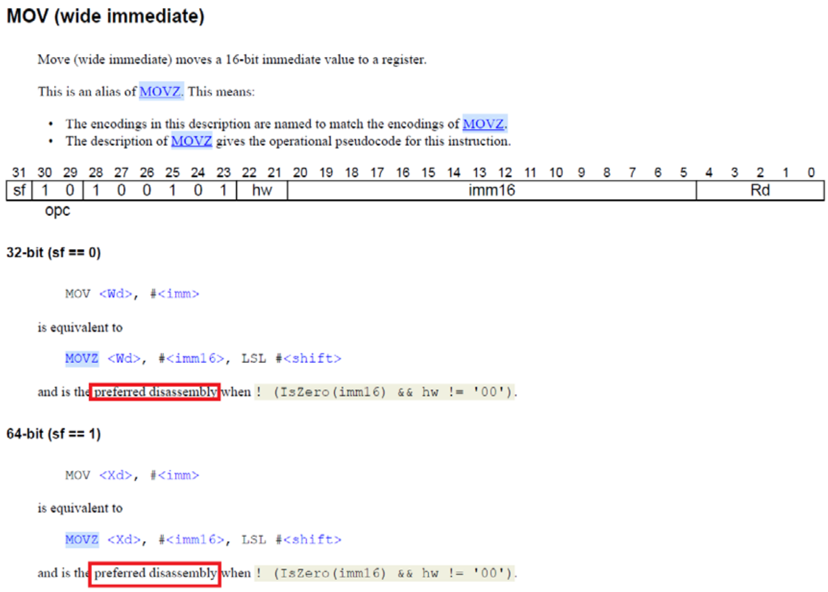
16 Jun 2023

https://hex-rays.com/blog/igors-tip-of-the-week-144-macros-and-simplified-instructions/

As can be seen in this example, it can produce misleading disassembly (ADRP can only use page-aligned addresses which is why it seems to refer to some unrelated string)

Simplified instructions

In addition to macros, sometimes IDA may transform single instructions to improve readability or make their behavior more obvious. For example, on ARM some instructions have preferred disassembly form and by default IDA uses it.



MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

31	30 29	28 27 26 25 24 23	22 21	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4 3 2 1 0
sf	1 0	1 0 0 1 0 1	hw	imm16	Rd
	opc				

32-bit (sf == 0)

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

64-bit (sf == 1)

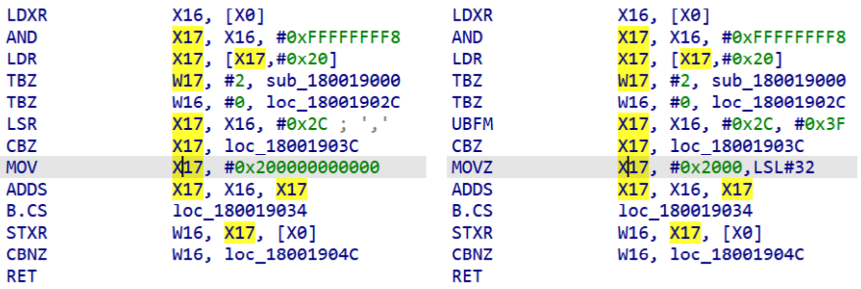
```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

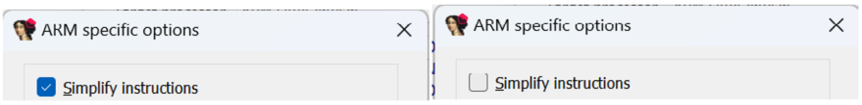
and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Instruction simplification feature is usually controlled by a processor-specific option.



```
LDXR            X16, [X0]
AND             X17, X16, #0xFFFFFFFF8
LDR             X17, [X17,#0x20]
TBZ             W17, #2, sub_180019000
TBZ             W16, #0, loc_18001902C
LSR             X17, X16, #0x2C ; ','
CBZ             X17, loc_18001903C
MOV             X17, #0x200000000000
ADDS            X17, X16, X17
B.CS            loc_180019034
STXR            W16, X17, [X0]
CBNZ            W16, loc_18001904C
RET
```

```
LDXR            X16, [X0]
AND             X17, X16, #0xFFFFFFFF8
LDR             X17, [X17,#0x20]
TBZ             W17, #2, sub_180019000
TBZ             W16, #0, loc_18001902C
UBFM            X17, X16, #0x2C, #0x3F
CBZ             X17, loc_18001903C
MOVZ            X17, #0x2000,LSL#32
ADDS            X17, X16, X17
B.CS            loc_180019034
STXR            W16, X17, [X0]
CBNZ            W16, loc_18001904C
RET
```



ARM specific options

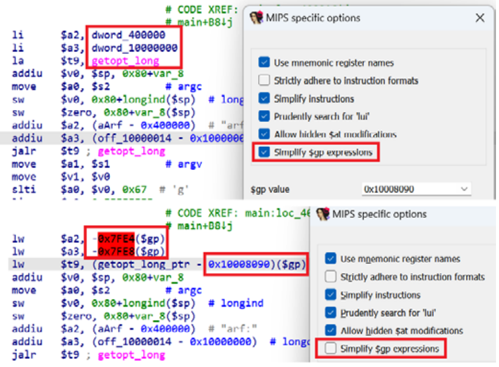
☑ Simplify instructions

ARM specific options

□ Simplify instructions

Other disassembly improvements

Some processor modules may have other options which may change disassembly to improve readability even if it sometimes means the resulting listing is not strictly conforming. For example, MIPS has an option to simplify instructions which use the global register `$gp` which usually has a fixed value and using it makes disassembly much easier to read:



```
                         # CODE XREF:
                         # main+B8↓j
li      $a2, dword_400000
li      $a3, dword_10000000
la      $t9, getopt_long
addiu   $v0, $sp, 0x80+var_8
move    $a0, $s2         # argc
sw      $v0, 0x80+longind($sp)  # long
sw      $zero, 0x80+var_8($sp)
addiu   $a2, (aArf - 0x400000)  # "arf
addiu   $a3, (off_10000014 - 0x1000000
jalr    $t9 ; getopt_long
move    $a1, $s1         # argv
move    $v1, $v0
slti    $a0, $v0, 0x67  # 'g'
```

MIPS specific options

☑ Use mnemonic register names
□ Strictly adhere to instruction formats
☑ Simplify instructions
☑ Prudently search for 'lui'
☑ Allow hidden $at modifications
☑ Simplify $gp expressions

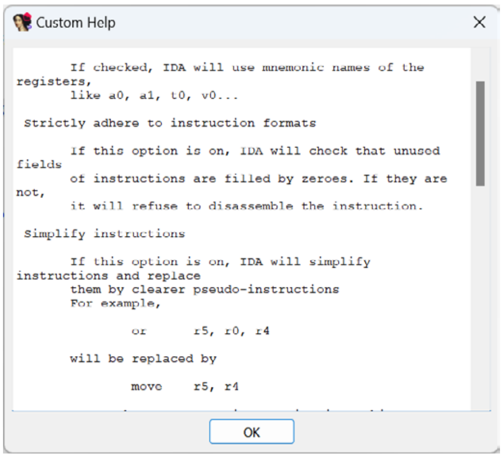
$gp value: 0x10008090

```
                         # CODE XREF: main:loc_4
                         # main+B8↓j
lw      $a2, -0x7FE4($gp)
lw      $a3, -0x7FE8($gp)
lw      $t9, (getopt_long_ptr - 0x10008090)($gp)
addiu   $v0, $sp, 0x80+var_8
move    $a0, $s2         # argc
sw      $v0, 0x80+longind($sp)  # longind
sw      $zero, 0x80+var_8($sp)
addiu   $a2, (aArf - 0x400000)  # "arf:"
addiu   $a3, (off_10000014 - 0x10000000)  # long
jalr    $t9 ; getopt_long
```

MIPS specific options

☑ Use mnemonic register names
□ Strictly adhere to instruction formats
☑ Simplify instructions
☑ Prudently search for 'lui'
☑ Allow hidden $at modifications
□ Simplify $gp expressions

If you are curious about what the options in the dialog do, clicking “Help” shows a short explanation:



```
        If checked, IDA will use mnemonic names of the
registers,
        like a0, a1, t0, v0...

 Strictly adhere to instruction formats

        If this option is on, IDA will check that unused
fields
        of instructions are filled by zeroes. If they are
not,
        it will refuse to disassemble the instruction.

 Simplify instructions

        If this option is on, IDA will simplify
instructions and replace
        them by clearer pseudo-instructions
        For example,

                or      r5, r0, r4

        will be replaced by

                move    r5, r4
```

See also:
Igor’s Tip of the Week #137: Processor modes and segment registers[2]
Igor’s tip of the week #98: Analysis options[3]

[1] https://developer.arm.com/documentation/dui0801/e/A64-General-Instructions/ADRL-pseudo-instruction?lang=en
[2] https://hex-rays.com/blog/igors-tip-of-the-week-137-processor-modes-and-segment-registers/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-98-analysis-options/

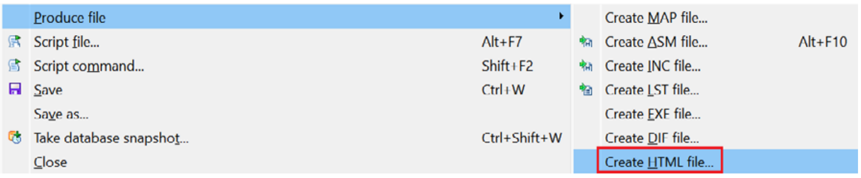
#145: HTML export

23 Jun 2023
https://hex-rays.com/blog/igors-tip-of-the-week-145-html-export/

We've covered exporting disassembly from IDA[1] before but it was in context of interoperability, when simple text is enough. If you want to preserve formatting and coloring of IDA View (e.g. for a web page or blog post), taking a screenshot is one option, but that has its downsides (e.g. no indexing for search engines). There is an alternative you can use instead.

HTML export

To export a fragment of disassembly as HTML, select[2] the desired address range in the listing and invoke File > Produce file > Create HTML file…



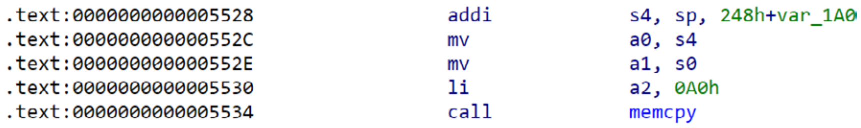
IDA will ask you for a filename and write the formatted text to it. The result will look like similar to the following:

```
<!DOCTYPE HTML PUBLIC “-//W3C//DTD HTML 4.01 Transitional//EN” “http://www.w3.org/TR/html4/loose.dtd”>
<html>
<head>
<meta http-equiv=”Content-Type” content=”text/html; charset=ISO-8859-1”>
<title>IDA - riscv_lscolors64.elf </title>
</head>
<body class=”c41”>
<span style=”white-space: pre; font-family: Consolas,monospace;” class=”c1 c41”>
<span class=”c44”>.text:0000000000005528 </span><span class=”c5”>addi </span><span class=”c33”>s4</
span><span class=”c9”>, </span><span class=”c33”>sp</span><span class=”c9”>, </span><span
class=”c12”>248h</span><span class=”c9”>+</span><span class=”c25”>var_1A0
</span><span class=”c44”>.text:000000000000552C </span><span class=”c5”>mv </span><span class=”c33”>a0</
span><span class=”c9”>, </span><span class=”c33”>s4
</span><span class=”c44”>.text:000000000000552E </span><span class=”c5”>mv </span><span class=”c33”>a1</
span><span class=”c9”>, </span><span class=”c33”>s0
</span><span class=”c44”>.text:0000000000005530 </span><span class=”c5”>li </span><span class=”c33”>a2</
span><span class=”c9”>, </span><span class=”c12”>0A0h
</span><span class=”c44”>.text:0000000000005534 </span><span class=”c5”>call </span><span class=”c37”>mem-
cpy
</span>
</span><style type=”text/css”>
/* line-fg-default */
.c1 { color: blue; }
/* line-bg-default */
.c41 { background-color: white; }
/* line-pfx-func */
.c44 { color: black; }
/* line-fg-insn */
.c5 { color: navy; }
/* line-fg-register-name */
.c33 { color: navy; }
/* line-fg-punctuation */
.c9 { color: navy; }
/* line-fg-numlit-in-insn */
.c12 { color: green; }
/* line-fg-locvar */
.c25 { color: green; }
/* line-fg-code-name */
.c37 { color: blue; }
</style></body></html>
```

As you can see, the color tags are represented by CSS classes which can be adjusted if necessary. When opened in browser, the result should look pretty close to IDA View:

#145: HTML export

23 Jun 2023
https://hex-rays.com/blog/igors-tip-of-the-week-145-html-export/

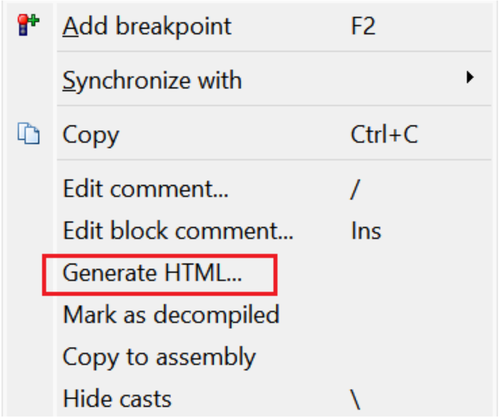


```
.text:0000000000005528                 addi            s4, sp, 248h+var_1A0
.text:000000000000552C                 mv              a0, s4
.text:000000000000552E                 mv              a1, s0
.text:0000000000005530                 li              a2, 0A0h
.text:0000000000005534                 call            memcpy
```

We use this feature on our web site to display disassembly snippets for the processor gallery[3].

Pseudocode to HTML

HTML can be generated not only for disassembly but also for the decompiled pseudocode; for this use “Generate HTML…” from the context menu in the Pseudocode view.



See also:
IDA Help: Create HTML File[4]
Hex-Rays interactive operation: Generate HTML file[5]
Hack of the day #0: Somewhat-automating pseudocode HTML generation, with IDAPython.[6]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-135-exporting-disassembly-from-ida/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/
[3] https://hex-rays.com/products/ida/processor-gallery/
[4] https://www.hex-rays.com/products/ida/support/idadoc/1504.shtml
[5] https://www.hex-rays.com/products/decompiler/manual/cmd_html.shtml
[6] https://hex-rays.com/blog/hack-of-the-day-0-somewhat-automating-pseudocode-html-generation-with-idapython/

#146: Graph printing

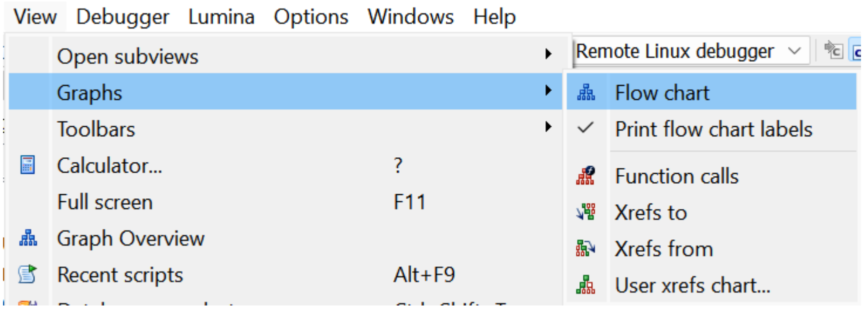
30 Jun 2023

https://hex-rays.com/blog/igors-tip-of-the-week-146-graph-printing/

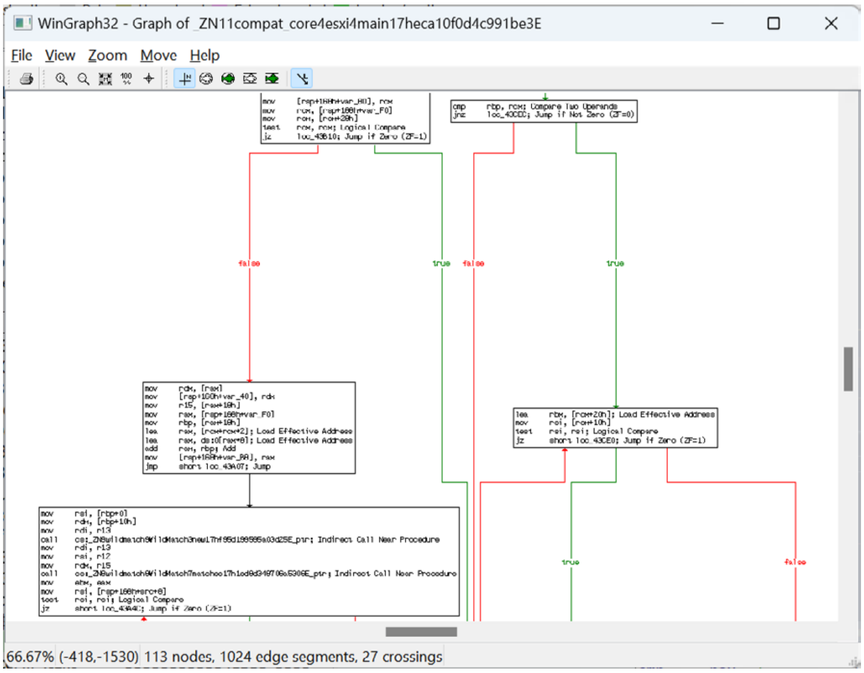
While exporting text disassembly is enough in many cases, many users nowadays prefer IDA's graph view[1], and saving its representation may be necessary. What other options are there besides screenshots?

WinGraph

WinGraph is an external program shipped with IDA which can display graphs. It was used to show function (and other) graphs before introduction of the built-in graph view in IDA 5.0 (2006). You can still use it via the View > Graphs menu.



For example, Flowchart action displays the graph of the current function.


WinGraph32 - Graph of _ZN11compat_core4esxi4main17heca10f0d4c991be3E
File
View
Zoom
Move
Help
false
true
false
true
true
false

```
mov     [rsp+168h+var_H0], rcx
mov     rcx, [rsp+168h+var_F0]
mov     rcx, [rcx+28h]
test    rcx, rcx; Logical Compare
jz      loc_43B10; Jump if Zero (ZF=1)
```

```
cmp     rbp, rcx; Compare Two Operands
jnz     loc_43CDC; Jump if Not Zero (ZF=0)
```

```
mov     rdx, [rax]
mov     [rsp+100h+var_40], rdx
mov     r15, [rax+18h]
mov     rax, [rsp+168h+var_F0]
mov     rbp, [rax+18h]
lea     rax, [rcx+rcx*2]; Load Effective Address
lea     rax, ds:0[rax*8]; Load Effective Address
add     rcx, rbp; Add
mov     [rsp+168h+var_A8], rax
jmp     short loc_43A07; Jump
```

```
lea     rbx, [rcx+20h]; Load Effective Address
mov     rsi, [rcx+10h]
test    rsi, rsi; Logical Compare
jz      short loc_43CE0; Jump if Zero (ZF=1)
```

```
mov     rsi, [rbp+0]
mov     rdx, [rbp+10h]
mov     rdi, r13
call    cs:_ZN8wildmatch9WildMatch3new17hf95d199595a03d25E_ptr; Indirect Call Near Procedure
mov     rdi, r13
mov     rsi, r12
mov     rdx, r15
call    cs:_ZN8wildmatch9WildMatch7matches17h1cd0d349706a5306E_ptr; Indirect Call Near Procedure
mov     ebx, eax
mov     rsi, [rsp+168h+src+8]
test    rsi, rsi; Logical Compare
jz      short loc_43A4C; Jump if Zero (ZF=1)
```

66.67% (-418,-1530) 113 nodes, 1024 edge segments, 27 crossings

Once the graph is displayed in WinGraph, you can print it using File > Print... or the first toolbar button. On most platforms this supports printing to PDF in addition to real printers.

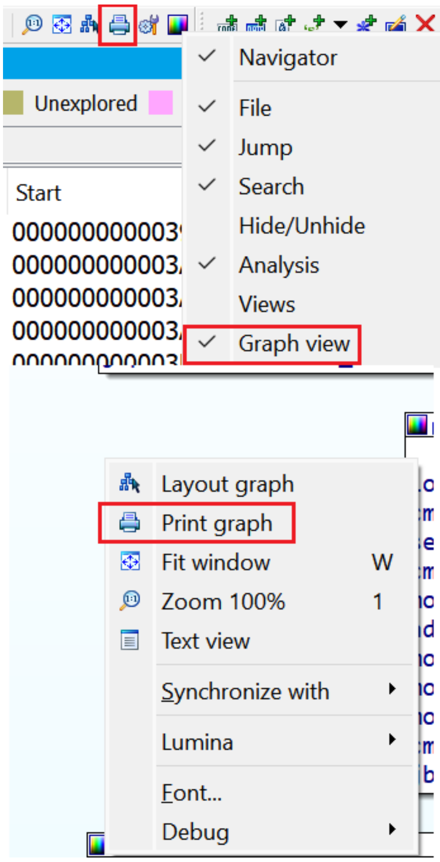
IDA graph view

If you prefer IDA's graph layout, or have customized it to your liking (groups or custom layouts are ignored by WinGraph), you can also print it directly from IDA. For this, use the print buttion on the Graph View toolbar, or the context menu by right-clicking outside of any node.

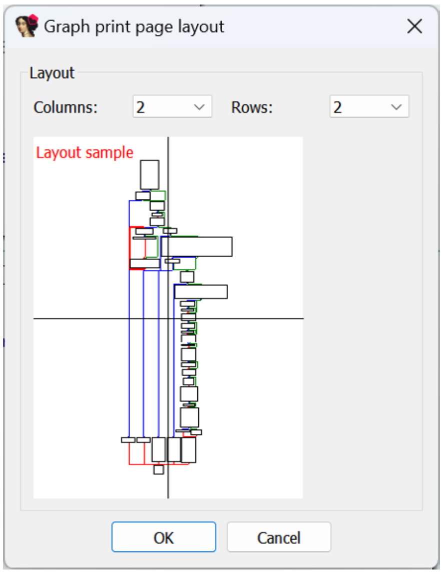
#146: Graph printing

30 Jun 2023

https://hex-rays.com/blog/igors-tip-of-the-week-146-graph-printing/



You will be asked about the page layout – this can be useful when printing large graphs



Columns: 2 Rows: 2

See also:
Igor's tip of the week #23: Graph view[2]
Igor's Tip of the Week #145: HTML export[3]
Igor's Tip of the Week #135: Exporting disassembly from IDA[4]

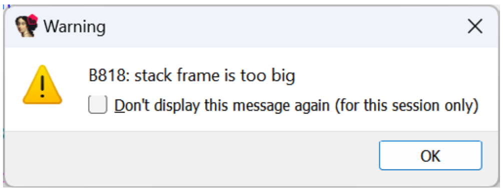
[1] https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-145-html-export/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-135-exporting-disassembly-from-ida/

#147: Fixing “stack frame is too big”

07 Jul 2023

https://hex-rays.com/blog/igors-tip-of-the-week-147-fixing-stack-frame-is-too-big/

The Hex-Rays decompiler has been designed to decompile compiler-generated code, so while it can usually handle hand-written or unusual assembly, occasionally you may run into a failure, especially if the code has been modified to hinder decompilation. Here is one of such errors:



Warning

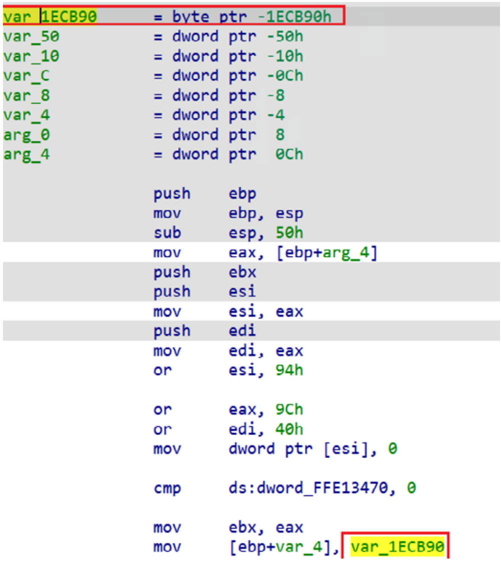
B818: stack frame is too big

□ Don't display this message again (for this session only)

If you have a genuine function with a huge stack frame, you'll probably have to give up and RE it the hard way – from the disassembly. However, in some situations it is possible to fix the code and get the function decompiled.

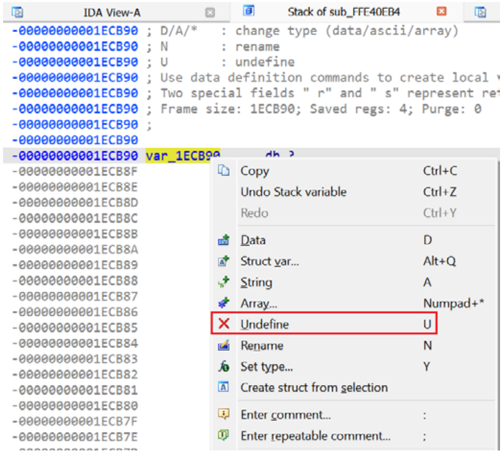
Bogus stack variables

Stack variable with a large offset may be created by mistake (e.g. pressing `K` on an immediate operand), or induced deliberately (e.g. junk code referring to large stack offsets which are not used in reality). The fastest way to check for them is to look at the stack variable definitions at the start of the function and look for unusually large offsets:



```
var_1ECB90      = byte ptr -1ECB90h
var_50          = dword ptr -50h
var_10          = dword ptr -10h
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, 50h
                mov     eax, [ebp+arg_4]
                push    ebx
                push    esi
                mov     esi, eax
                push    edi
                mov     edi, eax
                or      esi, 94h

                or      eax, 9Ch
                or      edi, 40h
                mov     dword ptr [esi], 0

                cmp     ds:dword_FFE13470, 0

                mov     ebx, eax
                mov     [ebp+var_4], var_1ECB90
```

To fix, double-click the variable or press `Ctrl-K` to open the stack frame editor[1], then undefine (`U`) the wrong stackvar(s).



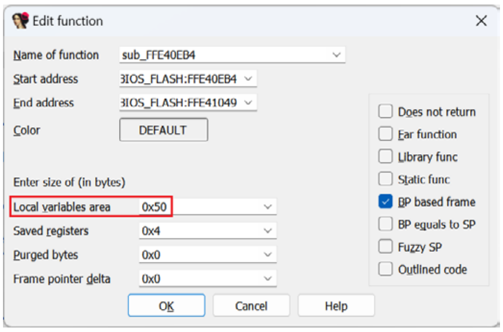
```
-000000000001ECB90 ; D/A/*   : change type (data/ascii/array)
-000000000001ECB90 ; N       : rename
-000000000001ECB90 ; U       : undefine
-000000000001ECB90 ; Use data definition commands to create local 
-000000000001ECB90 ; Two special fields " r" and " s" represent re
-000000000001ECB90 ; Frame size: 1ECB90; Saved regs: 4; Purge: 0
-000000000001ECB90 ;
-000000000001ECB90
-000000000001ECB90 var_1ECB90      db ?
-000000000001ECB8F
-000000000001ECB8E
-000000000001ECB8D
-000000000001ECB8C
-000000000001ECB8B
-000000000001ECB8A
-000000000001ECB89
-000000000001ECB88
-000000000001ECB87
-000000000001ECB86
-000000000001ECB85
-000000000001ECB84
-000000000001ECB83
-000000000001ECB82
-000000000001ECB81
-000000000001ECB80
-000000000001ECB7F
-000000000001ECB7E
```

Then you need to edit the function properties[2] (`Alt– P`) and reduce the local variables area to the actually used size (usually equival to the offset of the bottom-most actually used variable):

#147: Fixing “stack frame is too big”

07 Jul 2023

https://hex-rays.com/blog/igors-tip-of-the-week-147-fixing-stack-frame-is-too-big/



Name of function: sub_FFE40EB4
Start address: 3IOS_FLASH:FFE40EB4
End address: 3IOS_FLASH:FFE41049
Color: DEFAULT

Enter size of (in bytes)
Local variables area: 0x50
Saved registers: 0x4
Purged bytes: 0x0
Frame pointer delta: 0x0

☐ Does not return
☐ Far function
☐ Library func
☐ Static func
☑ BP based frame
☐ BP equals to SP
☐ Fuzzy SP
☐ Outlined code

If you still get the error message after all that, the bogus variables may have been re-added during autoanalysis, so it may be necessary to patch out[3] or otherwise exclude from analysis the instructions which refer to them.

Unusual stack pointer manipulation

This trick may cause IDA to decide that the stack pointer changes by a huge value, or not detect stack changes, causing it to grow the stack frame unnecessarily. This can be dealt with by adjusting the stack pointer delta[4] manually, or patching the instructions involved.

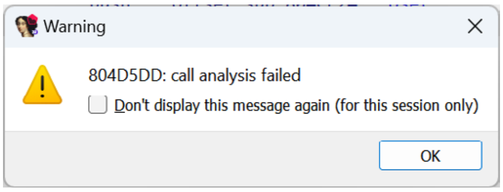
See also:
Igor’s tip of the week #27: Fixing the stack pointer[5]
Decompiler Manual: Failures and troubleshooting)[6]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-127-changing-function-bounds/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-37-patching/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-27-fixing-the-stack-pointer/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-27-fixing-the-stack-pointer/
[6] https://www.hex-rays.com/products/decompiler/manual/failures.shtml

#148: Fixing “call analysis failed”

14 Jul 2023
https://hex-rays.com/blog/igors-tip-of-the-week-148-fixing-call-analysis-failed/

This error is not very common but may appear in some situations.



Warning

804D5DD: call analysis failed

☐ Don't display this message again (for this session only)

Such errors happen when there is a function call in the code, but the decompiler fails to convert it to a high-level function call, e.g.:

1. the target function’s prototype is wrong;
2. the decompiler failed to figure out the function arguments: how many of them, or how exactly they’re being passed to the callee;
3. the usage of the stack by the call does not make sense.

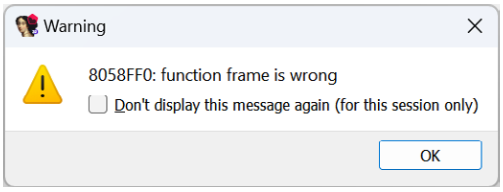
Let’s look at some examples where it happens and how to fix it.

Wrong function info

The first action on seeing the error should be to inspect the address mentioned and the surrounding code. For example, here’s the snippet around the address in the first screenshot:

```
.text:0804D5CD                 push    [ebp+var_10]
.text:0804D5D0                 push    offset sub_804D6E8
.text:0804D5D5                 push    [ebp+var_28]
.text:0804D5D8                 push    offset sub_804CF24 ; oset
.text:0804D5DD                 call    sub_8058FF0
.text:0804D5E2                 mov     edx, [ebp+var_14]
.text:0804D5E5                 or      dword ptr [edx+28h], 10h
.text:0804D5E9                 mov     eax, [ebp+var_18]
.text:0804D5EC                 add     esp, 10h
.text:0804D5EF                 test    eax, eax
.text:0804D5F1                 jz      loc_804D1D3
.text:0804D5F7                 sub     esp, 0Ch
.text:0804D5FA                 push    [ebp+var_18]
.text:0804D5FD                 call    sub_8055A0C
```

At the first glance, there doesn’t seem to be anything unusual: four arguments are pushed on the stack before calling the function sub_8058FF0. However, if we go inside the function and try to decompile it, we get another error:



Warning

8058FF0: function frame is wrong

☐ Don't display this message again (for this session only)

Also, the header of the function looks strange:

```
.text:08058FF0 ; =============== S U B R O U T I N E =======================================
.text:08058FF0
.text:08058FF0 ; Attributes: bp-based frame
.text:08058FF0
.text:08058FF0 ; int __cdecl sub_8058FF0(sigset_t oset)
.text:08058FF0 sub_8058FF0     proc near               ; CODE XREF: sub_804CF6C+671↑p
.text:08058FF0                                         ; sub_804F798+126↑p ...
.text:08058FF0
.text:08058FF0 var_48          = dword ptr -48h
.text:08058FF0 oset            = sigset_t ptr -38h
```

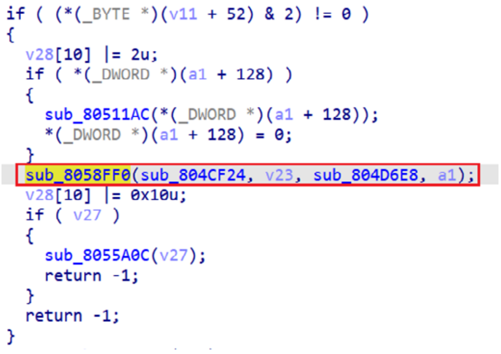
I.e. the function was detected not to take four arguments, but one structure by value. While this can indeed happen in some cases, the argument is in a wrong location: the local variables area (note the negative offset).

#148: Fixing “call analysis failed”

14 Jul 2023

https://hex-rays.com/blog/igors-tip-of-the-week-148-fixing-call-analysis-failed/

Fixing the function itself is a topic for another post, but a quick fix for the original issue would be to delete the current prototype and let the decompiler fall back to guessing the arguments. For this, put the cursor on the function name or its first line, then press `Y` (edit type[1]), `Del`, `Enter`. This will clear the wrong prototype and decompilation should succeed, showing the four arguments we’ve seen in the disassembly:



```
if ( (*(_BYTE *)(v11 + 52) & 2) != 0 )
{
  v28[10] |= 2u;
  if ( *(_DWORD *)(a1 + 128) )
  {
    sub_80511AC(*(_DWORD *)(a1 + 128));
    *(_DWORD *)(a1 + 128) = 0;
  }
  sub_8058FF0(sub_804CF24, v23, sub_804D6E8, a1);
  v28[10] |= 0x10u;
  if ( v27 )
  {
    sub_8055A0C(v27);
    return -1;
  }
  return -1;
}
```

Sometimes the decompiler’s guessing of the prototype still fails, so try to specify one based on the actual arguments being passed to the call (look at the assembly around the call). In some cases this may require the `__usercall` calling convention[2].

Indirect calls

Instead of the direct function address, indirect calls use a register or a memory location which holds the destination address to perform the call. For example, on x86 it may look like one of the following:

```
call eax
call dword ptr [edx+14h]
call [ebp+arg_0]
call g_myfuncptr
```

In rare cases, the decompiler may fail to detect the actual arguments being passed to the call, especially if optimizer interleaves arguments passed to different calls. In that case, you can give it a hint by adding a cross-reference to the actual function being called (if you know it), or a function of the matching type, for example using the Set callee address[3] feature. You should also check that the stack pointer is properly balanced[4] before and after each call for stack-using calling conventions.

See also:
Igor’s tip of the week #27: Fixing the stack pointer[5]
Decompiler Manual: Failures and troubleshooting[6]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-115-set-callee-address/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-27-fixing-the-stack-pointer/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-27-fixing-the-stack-pointer/
[6] https://www.hex-rays.com/products/decompiler/manual/failures.shtml

#149: Using symbolic constants in the decompiler

21 Jul 2023

https://hex-rays.com/blog/igors-tip-of-the-week-149-using-symbolic-constants-in-the-decompiler/

We've covered the usage of symbolic constants (enums) in the disassembly[1]. but they are also useful in the pseudocode view.

Reusing constants from disassembly

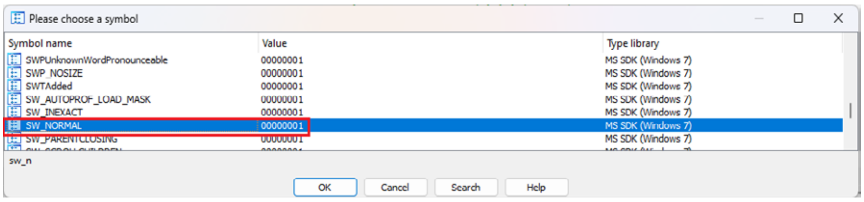
If a number has been converted to a symbolic constant in the disassembly and it is present in unchanged form in pseudocode, the decompiler will use it in the output. For example, consider this call:

```
.text:00405D72   push    1                ; nShowCmd
.text:00405D74   cmovnb  eax, [esp+114h+lpParameters]
.text:00405D79   push    0                ; lpDirectory
.text:00405D7B   push    eax              ; lpParameters
.text:00405D7C   push    offset File      ; “explorer.exe”
.text:00405D81   push    0                ; lpOperation
.text:00405D83   push    0                ; hwnd
.text:00405D85   call    ShellExecuteW
```

Initially, it is decompiled like this:

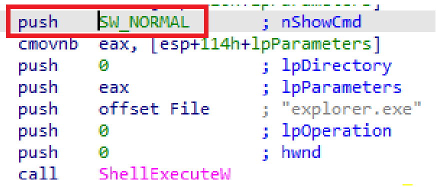
```
ShellExecuteW(0, 0, L”explorer.exe”, v136, 0, 1);
```

However, we can look up[2] that nShowCmd's value 1 corresponds to[3] the constant `SW_NORMAL`, and apply it to the disassembly:


Please choose a symbol

Symbol name	Value	Type library
SWPUnknownWordPronounceable	00000001	MS SDK (Windows 7)
SWP_NOSIZE	00000001	MS SDK (Windows 7)
SWTAdded	00000001	MS SDK (Windows 7)
SW_AUTOPROF_LOAD_MASK	00000001	MS SDK (Windows 7)
SW_INEXACT	00000001	MS SDK (Windows 7)
SW_NORMAL	00000001	MS SDK (Windows 7)
SW_PARENTCLOSING	00000001	MS SDK (Windows 7)

sw_n



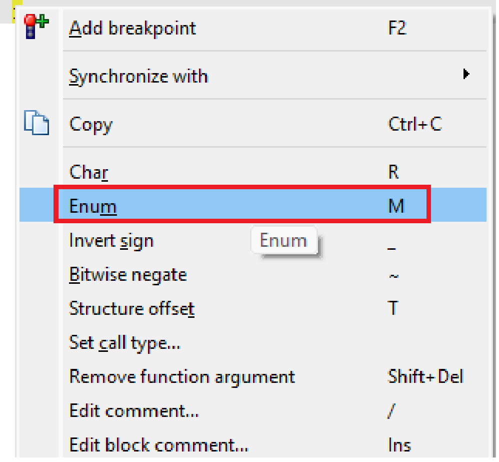
```
push    SW_NORMAL         ; nShowCmd
cmovnb  eax, [esp+114h+lpParameters]
push    0                 ; lpDirectory
push    eax               ; lpParameters
push    offset File       ; "explorer.exe"
push    0                 ; lpOperation
push    0                 ; hwnd
call    ShellExecuteW
```

After refreshing the pseudocode, the constant appears there as well:

```
ShellExecuteW(0, 0, L”explorer.exe”, v136, 0, SW_NORMAL);
```

Applying constants in pseudocode

In fact, you can do the same directly in the pseudocode, using the context menu or the same shortcut (`M`):



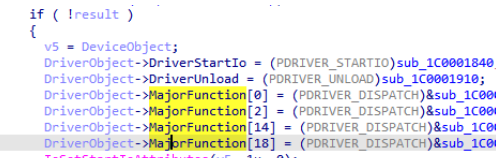
Note that there is no automatic propagation of the constants applied in pseudocode to disassembly. In fact, sometimes it's not possible to map a number you see in the pseudocode to the same number in the disassembly.

Consider this example from a Windows driver's initialization routine (`DriverEntry`):

#149: Using symbolic constants in the decompiler

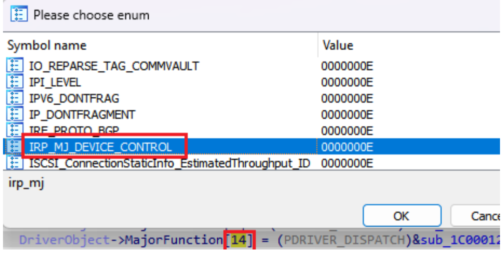
21 Jul 2023

https://hex-rays.com/blog/igors-tip-of-the-week-149-using-symbolic-constants-in-the-decompiler/



```
if ( !result )
{
  v5 = DeviceObject;
  DriverObject->DriverStartIo = (PDRIVER_STARTIO)sub_1C0001840
  DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1C0001910;
  DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)&sub_1C00
  DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)&sub_1C00
  DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)&sub_1C0
  DriverObject->MajorFunction[18] = (PDRIVER_DISPATCH)&sub_1C0
```

We know[4] that indexes into the MajorFunction array correspond to the major IRP codes (`IRP_MJ_xxx`), so we can convert numerical indexes to the corresponding constants:



Symbol name	Value
IO_REPARSE_TAG_COMMVAULT	0000000E
IPI_LEVEL	0000000E
IPV6_DONTFRAG	0000000E
IP_DONTFRAGMENT	0000000E
IRE_PROTO_BGP	0000000E
IRP_MJ_DEVICE_CONTROL	0000000E
ISCSI_ConnectionStaticInfo_EstimatedThroughput_ID	0000000E

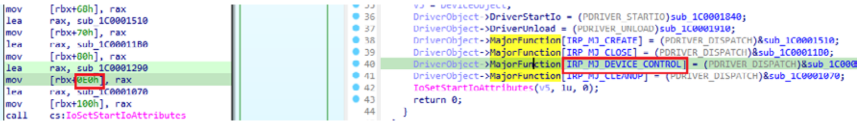
irp_mj

```
DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)&sub_1C0001
```

and the pseudocode becomes:

```
DriverObject->DriverStartIo = (PDRIVER_STARTIO)sub_1C0001840;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1C0001910;
DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)&sub_1C0001510;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)&sub_1C00011B0;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)&sub_1C0001290;
DriverObject->MajorFunction[IRP_MJ_CLEANUP] = (PDRIVER_DISPATCH)&sub_1C0001070;
```

However, if we check the corresponding disassembly (e.g by using `Tab` or synchronizing pseudocode and IDA View), we can see that the array indexes are not present as such in the instruction operands:



```
mov     [rbx+60h], rax
lea     rax, sub_1C0001510
mov     [rbx+70h], rax
lea     rax, sub_1C00011B0
mov     [rbx+80h], rax
lea     rax, sub_1C0001290
mov     [rbx+0E0h], rax
lea     rax, sub_1C0001070
mov     [rbx+100h], rax
call    cs:IoSetStartIoAttributes
```

```
DriverObject->DriverStartIo = (PDRIVER_STARTIO)sub_1C0001840;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1C0001910;
DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)&sub_1C0001510;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)&sub_1C00011B0;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)&sub_1C000
DriverObject->MajorFunction[IRP_MJ_CLEANUP] = (PDRIVER_DISPATCH)&sub_1C0001070;
IoSetStartIoAttributes(v5, 1u, 0);
return 0;
}
```

Another common situation where you can use symbolic constants in pseudocode but not disassembly is swich cases.

See also:
Igor's tip of the week #99: Enums[5]
Decompiler Manual: Hex-Rays interactive operation: Set Number Representation[6]

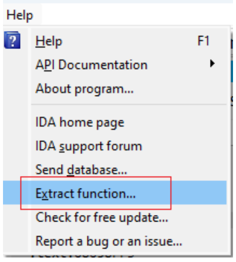
[1] https://hex-rays.com/blog/igors-tip-of-the-week-99-enums/
[2] https://learn.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shellexecutew
[3] https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-showwindow
[4] https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/driverentry-s-required-responsibilities
[5] https://hex-rays.com/blog/igors-tip-of-the-week-99-enums/
[6] https://www.hex-rays.com/products/decompiler/manual/cmd_numform.shtml

#150: Extract function

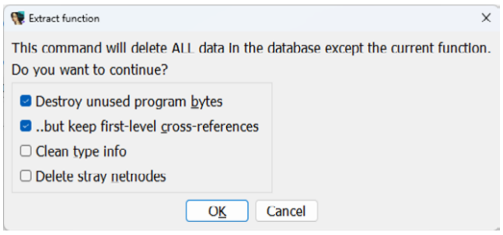
28 Jul 2023

https://hex-rays.com/blog/igors-tip-of-the-week-150-extract-function/

When you open a decompilable file in IDA, you get this somewhat mysterious item in the Help menu:



And if you invoke it, it shows an even more mysterious dialog:



Extract function

This command will delete ALL data in the database except the current function.
Do you want to continue?

☑ Destroy unused program bytes
☑ ..but keep first-level cross-references
☐ Clean type info
☐ Delete stray netnodes

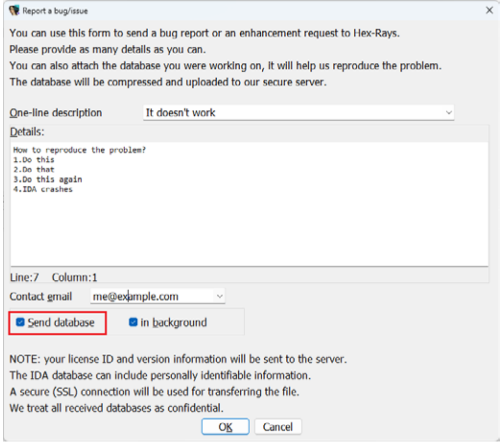
OK Cancel

So, what is it and when it is useful?

Originally this feature was added to the decompiler to make decompiler bug reporting easier: oftentimes. a decompiler issue cannot really be reproduced or debugged without having the original database. However, in some cases sharing the whole database is impractical or impossible:

- Whole database may be very large and difficult to share
- parts of the database may contain private or confidential information
- the rest of the database is not really relevant to the issue and only adds noise

This feature leaves just the current function plus maybe some potentially relevant information in the database. It can then be sent to support for investigation and fixing, either by email or directly from IDA via Help > Report a bug or an issue...



Report a bug/issue

You can use this form to send a bug report or an enhancement request to Hex-Rays.
Please provide as many details as you can.
You can also attach the database you were working on, it will help us reproduce the problem.
The database will be compressed and uploaded to our secure server.

One-line description: It doesn't work

Details:

```
How to reproduce the problem?
1.Do this
2.Do that
3.Do this again
4.IDA crashes
```

Line:7 Column:1

Contact email: me@example.com

☑ Send database ☑ in background

NOTE: your license ID and version information will be sent to the server.
The IDA database can include personally identifiable information.
A secure (SSL) connection will be used for transferring the file.
We treat all received databases as confidential.

OK Cancel

See also:
Igor's tip of the week #39: Export Data[1]
Igor's Tip of the Week #135: Exporting disassembly from IDA[2]
Decompiler Manual: Failures and troubleshooting[3]

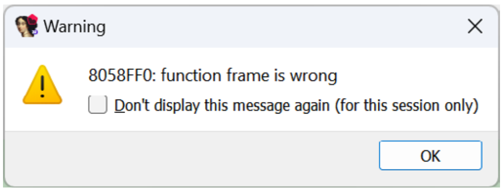
[1] https://hex-rays.com/blog/igors-tip-of-the-week-39-export-data/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-135-exporting-disassembly-from-ida/
[3] https://www.hex-rays.com/products/decompiler/manual/failures.shtml#report

#151: Fixing “function frame is wrong”

04 Aug 2023

https://hex-rays.com/blog/igors-tip-of-the-week-151-fixing-function-frame-is-wrong/

Previously[1], we’ve run into a function which produces a cryptic error if you try to decompile it:



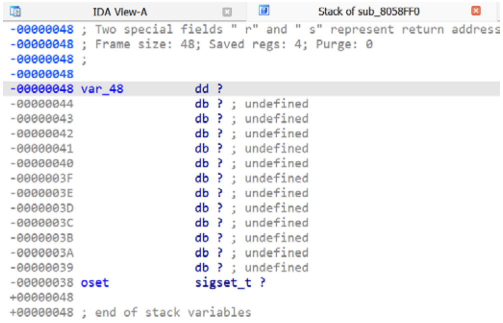
Warning

8058FF0: function frame is wrong

□ Don't display this message again (for this session only)

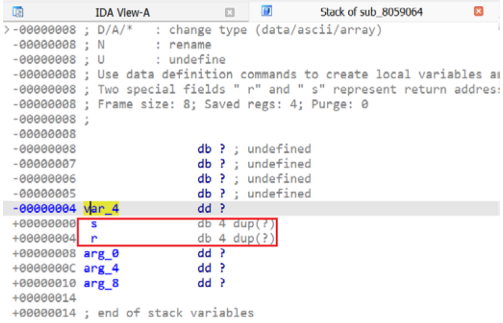
In such situations, you need to go back to disassembly to see what could be wrong. More specifically, check the stack frame layout[2] by double-clicking a stack variable or pressing `Ctrl– K`.

On the first glance it looks normal:



```
-00000048 ; Two special fields " r" and " s" represent return address
-00000048 ; Frame size: 48; Saved regs: 4; Purge: 0
-00000048 ;
-00000048
-00000048 var_48          dd ?
-00000044                 db ? ; undefined
-00000043                 db ? ; undefined
-00000042                 db ? ; undefined
-00000041                 db ? ; undefined
-00000040                 db ? ; undefined
-0000003F                 db ? ; undefined
-0000003E                 db ? ; undefined
-0000003D                 db ? ; undefined
-0000003C                 db ? ; undefined
-0000003B                 db ? ; undefined
-0000003A                 db ? ; undefined
-00000039                 db ? ; undefined
-00000038 oset            sigset_t ?
+00000048
+00000048 ; end of stack variables
```

However, if you compare with another function which decompiles fine, you may notice some notable differences:

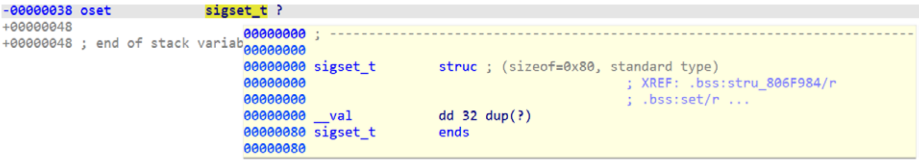


```
-00000008 ; D/A/*   : change type (data/ascii/array)
-00000008 ; N       : rename
-00000008 ; U       : undefine
-00000008 ; Use data definition commands to create local variables ar
-00000008 ; Two special fields " r" and " s" represent return addres:
-00000008 ; Frame size: 8; Saved regs: 4; Purge: 0
-00000008 ;
-00000008
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004 var_4           dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 arg_0           dd ?
+0000000C arg_4           dd ?
+00000010 arg_8           dd ?
+00000014
+00000014 ; end of stack variables
```

This frame has two members which are mentioned in the top comment:

```
Two special fields “ r” and “ s” represent return address and saved registers.
```

They’re absent in the “bad” function, so the whole layout is probably wrong and the function can’t be decompiled reliably. On closer inspection, we can discover that the structure `sigset_t` (type of the variable `oset` in `sub_8058FF0`) is 0x80 bytes, so applying it to the frame overwrote the special members. You can also see that the variable crossed over from the local variable area (negative offsets) to the argument area (positive offsets), which normally should not happen.



```
-00000038 oset            sigset_t ?
+00000048
+00000048 ; end of stack variab
```

```
00000000 ; ---------------------------------------------------------------
00000000
00000000 sigset_t        struc ; (sizeof=0x80, standard type)
00000000                                         ; XREF: .bss:stru_806F984/r
00000000                                         ; .bss:set/r ...
00000000 __val           dd 32 dup(?)
00000080 sigset_t        ends
00000080
```

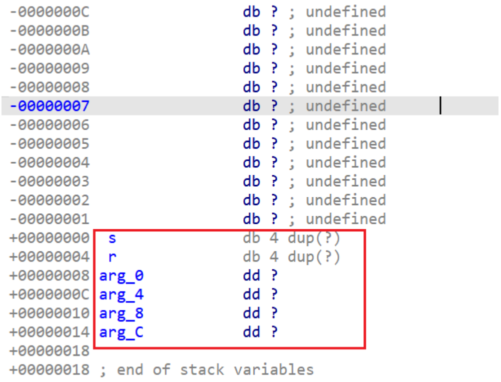
Fixing a bad stack frame

Although you can try to fix the frame layout by rearranging or editing the local variables, this won’t bring back the special variables, so usually the best solution is to recreate the function (and thus its stack frame). This can be done by undefining (`U`) the first instruction, then creating the function (`P`). A quicker and less destructive way is to delete just the function (`Ctrl– P`, `Del`), then recreate it (`P`). Normally this should recreate the default frame then add local variables and stack arguments based on the instructions accessing the stack:

#151: Fixing “function frame is wrong”

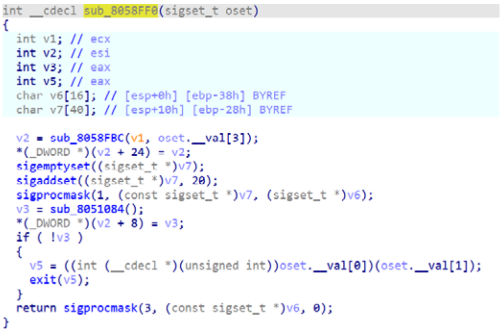
04 Aug 2023

https://hex-rays.com/blog/igors-tip-of-the-week-151-fixing-function-frame-is-wrong/



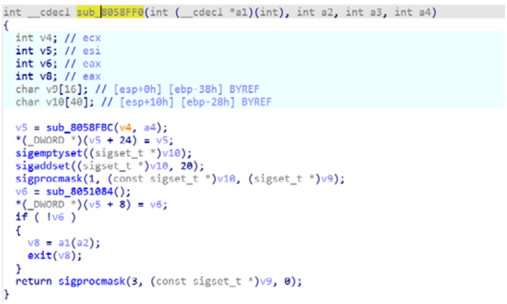
```
-0000000C                  db ? ; undefined
-0000000B                  db ? ; undefined
-0000000A                  db ? ; undefined
-00000009                  db ? ; undefined
-00000008                  db ? ; undefined
-00000007                  db ? ; undefined
-00000006                  db ? ; undefined
-00000005                  db ? ; undefined
-00000004                  db ? ; undefined
-00000003                  db ? ; undefined
-00000002                  db ? ; undefined
-00000001                  db ? ; undefined
+00000000  s               db 4 dup(?)
+00000004  r               db 4 dup(?)
+00000008 arg_0            dd ?
+0000000C arg_4            dd ?
+00000010 arg_8            dd ?
+00000014 arg_C            dd ?
+00000018
+00000018 ; end of stack variables
```

And now the function decompiles fine:



```
int __cdecl sub_8058FF0(sigset_t oset)
{
  int v1; // ecx
  int v2; // esi
  int v3; // eax
  int v5; // eax
  char v6[16]; // [esp+0h] [ebp-38h] BYREF
  char v7[40]; // [esp+10h] [ebp-28h] BYREF

  v2 = sub_8058FBC(v1, oset.__val[3]);
  *(_DWORD *)(v2 + 24) = v2;
  sigemptyset((sigset_t *)v7);
  sigaddset((sigset_t *)v7, 20);
  sigprocmask(1, (const sigset_t *)v7, (sigset_t *)v6);
  v3 = sub_8051084();
  *(_DWORD *)(v2 + 8) = v3;
  if ( !v3 )
  {
    v5 = ((int (__cdecl *)(unsigned int))oset.__val[0])(oset.__val[1]);
    exit(v5);
  }
  return sigprocmask(3, (const sigset_t *)v6, 0);
}
```

Some code is wrong because the function prototype still uses wrongly detected `sigset_t` argument. This is easy to fix – just delete the prototype (Y, Del) to let the decompiler guess the arguments:



```
int __cdecl sub_8058FF0(int (__cdecl *a1)(int), int a2, int a3, int a4)
{
  int v4; // ecx
  int v5; // esi
  int v6; // eax
  int v8; // eax
  char v9[16]; // [esp+0h] [ebp-38h] BYREF
  char v10[40]; // [esp+10h] [ebp-28h] BYREF

  v5 = sub_8058FBC(v4, a4);
  *(_DWORD *)(v5 + 24) = v5;
  sigemptyset((sigset_t *)v10);
  sigaddset((sigset_t *)v10, 20);
  sigprocmask(1, (const sigset_t *)v10, (sigset_t *)v9);
  v6 = sub_8051084();
  *(_DWORD *)(v5 + 8) = v6;
  if ( !v6 )
  {
    v8 = a1(a2);
    exit(v8);
  }
  return sigprocmask(3, (const sigset_t *)v9, 0);
}
```

See also:

Igor’s Tip of the Week #148: Fixing “call analysis failed”[3]
Igor’s tip of the week #65: stack frame view[4]
Decompiler Manual: Failures and troubleshooting[5]

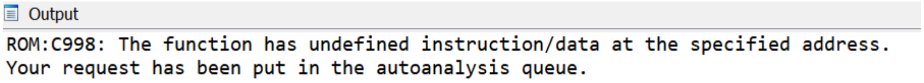
[1] https://hex-rays.com/blog/igors-tip-of-the-week-148-fixing-call-analysis-failed/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-148-fixing-call-analysis-failed/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/
[5] https://www.hex-rays.com/products/decompiler/manual/failures.shtml

#152: Force-creating functions

11 Aug 2023

https://hex-rays.com/blog/igors-tip-of-the-week-152-force-creating-functions/

Occasionally, especially when working with embedded firmware or obfuscated code, you may see an error message when trying to create a function (from context menu or using `P` hotkey):



Output

```
ROM:C998: The function has undefined instruction/data at the specified address.
Your request has been put in the autoanalysis queue.
```

There can be multiple reasons for it, for example:

1. some code has been incorrectly converted to data and the execution flows into it;
2. the function calls a non-returning function[1] which hasn't been marked as such, so IDA thinks that the execution flows into the following data or undefined bytes;
3. the function uses an unrecognized switch pattern[2];
4. the function calls some function which uses embedded data after the call, but IDA tries to decode it as instructions;
5. code has been obfuscated and IDA's autoanalysis went down a wrong path.

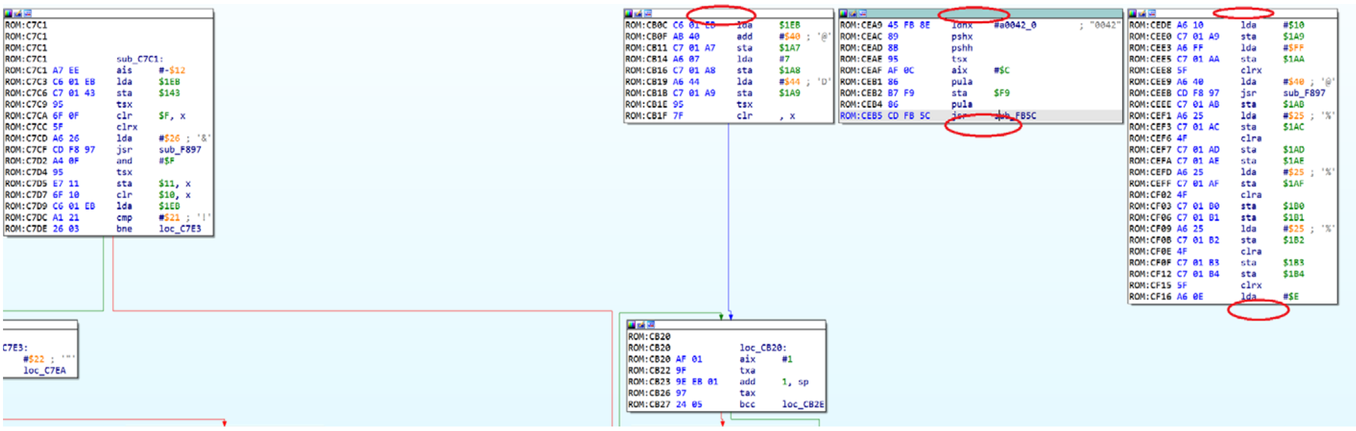
You can double-click the address indicated to jump there and to see if you can identify the issue and try to fix it, but it can take a long time to figure out.

Functions are required to use some of IDA's basic functionality such as graph view[3] or the decompiler[4].

Forcing IDA to create a function

Whatever the reason of the error, you can still create a function manually if you can determine its bounds using your best judgement. For this, the anchor selection[5] is the most simple and convenient way:

1. while staying on the first instruction of the function, use Edit > Begin selection, or press `Alt– L`;
2. navigate down to the function's end (e.g. look for a return instruction or start of the next function);
3. press `P` (Create function)



```
ROM:C7C1
ROM:C7C1
ROM:C7C1
ROM:C7C1                sub_C7C1:
ROM:C7C1 A7 EE          ais     #-$12
ROM:C7C3 C6 01 EB       lda     $1EB
ROM:C7C6 C7 01 43       sta     $143
ROM:C7C9 95             tsx
ROM:C7CA 6F 0F          clr     $F, x
ROM:C7CC 5F             clrx
ROM:C7CD A6 26          lda     #$26 ; '&'
ROM:C7CF CD F8 97       jsr     sub_F897
ROM:C7D2 A4 0F          and     #$F
ROM:C7D4 95             tsx
ROM:C7D5 E7 11          sta     $11, x
ROM:C7D7 6F 10          clr     $10, x
ROM:C7D9 C6 01 EB       lda     $1EB
ROM:C7DC A1 21          cmp     #$21 ; '!'
ROM:C7DE 26 03          bne     loc_C7E3
```

```
C7E3:
      #$22 ; '"'
      loc_C7EA
```

```
ROM:CB0C C6 01 EB       lda     $1EB
ROM:CB0F AB 40          add     #$40 ; '@'
ROM:CB11 C7 01 A7       sta     $1A7
ROM:CB14 A6 07          lda     #7
ROM:CB16 C7 01 A8       sta     $1A8
ROM:CB19 A6 44          lda     #$44 ; 'D'
ROM:CB1B C7 01 A9       sta     $1A9
ROM:CB1E 95             tsx
ROM:CB1F 7F             clr     , x
```

```
ROM:CEA9 45 FB 8E       ldhx    #a0042_0        ; "0042"
ROM:CEAC 89             pshx
ROM:CEAD 8B             pshh
ROM:CEAE 95             tsx
ROM:CEAF AF 0C          aix     #$C
ROM:CEB1 86             pula
ROM:CEB2 B7 F9          sta     $F9
ROM:CEB4 86             pula
ROM:CEB5 CD FB 5C       jsr     sub_FB5C
```

```
ROM:CEDE A6 10          lda     #$10
ROM:CEE0 C7 01 A9       sta     $1A9
ROM:CEE3 A6 FF          lda     #$FF
ROM:CEE5 C7 01 AA       sta     $1AA
ROM:CEE8 5F             clrx
ROM:CEE9 A6 40          lda     #$40 ; '@'
ROM:CEEB CD F8 97       jsr     sub_F897
ROM:CEEE C7 01 AB       sta     $1AB
ROM:CEF1 A6 25          lda     #$25 ; '%'
ROM:CEF3 C7 01 AC       sta     $1AC
ROM:CEF6 4F             clra
ROM:CEF7 C7 01 AD       sta     $1AD
ROM:CEFA C7 01 AE       sta     $1AE
ROM:CEFD A6 25          lda     #$25 ; '%'
ROM:CEFF C7 01 AF       sta     $1AF
ROM:CF02 4F             clra
ROM:CF03 C7 01 B0       sta     $1B0
ROM:CF06 C7 01 B1       sta     $1B1
ROM:CF09 A6 25          lda     #$25 ; '%'
ROM:CF0B C7 01 B2       sta     $1B2
ROM:CF0E 4F             clra
ROM:CF0F C7 01 B3       sta     $1B3
ROM:CF12 C7 01 B4       sta     $1B4
ROM:CF15 5F             clrx
ROM:CF16 A6 0E          lda     #$E
```

```
ROM:CB20
ROM:CB20                loc_CB20:
ROM:CB20 AF 01          aix     #1
ROM:CB22 9F             txa
ROM:CB23 9E EB 01       add     1, sp
ROM:CB26 97             tax
ROM:CB27 24 05          bcc     loc_CB2E
```

Note that the function created this way may have all kinds of issues, e.g. disconnected blocks in the graph view, `JUMPOUT` statements in pseudocode or wrong decompilation, but at least it should allow you to advance in your analysis.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-126-non-returning-functions/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-53-manual-switch-idioms/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-40-decompiler-basics/
[5] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/

#153: Copying pseudocode to disassembly

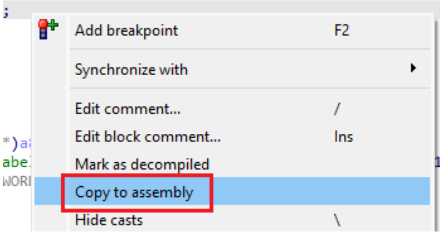
18 Aug 2023

https://hex-rays.com/blog/igors-tip-of-the-week-153-copying-pseudocode-to-disassembly/

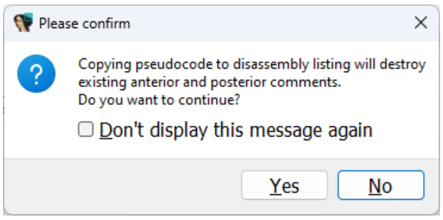
When using the decompiler, you probably spend most of the time in the Pseudocode view[1]. In case you need to consult the corresponding disassembly, it's a quick `Tab` away. However, if you actually prefer the disassembly, there is another option you can try.

Copy to assembly

This action is available in the pseudocode view's context menu when right-clicking outside of the decompiled code:



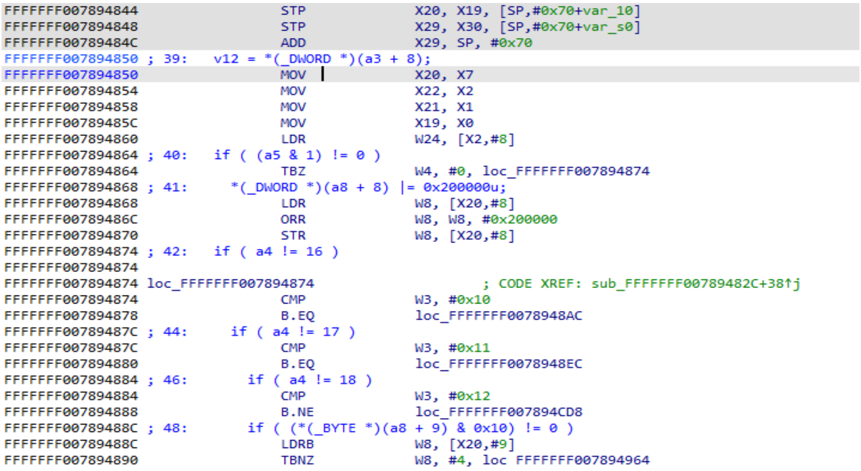
Because the decompiler uses disassembly comments[2] for this feature, it warns you that the action will destroy any existing ones:



Copying pseudocode to disassembly listing will destroy existing anterior and posterior comments.
Do you want to continue?

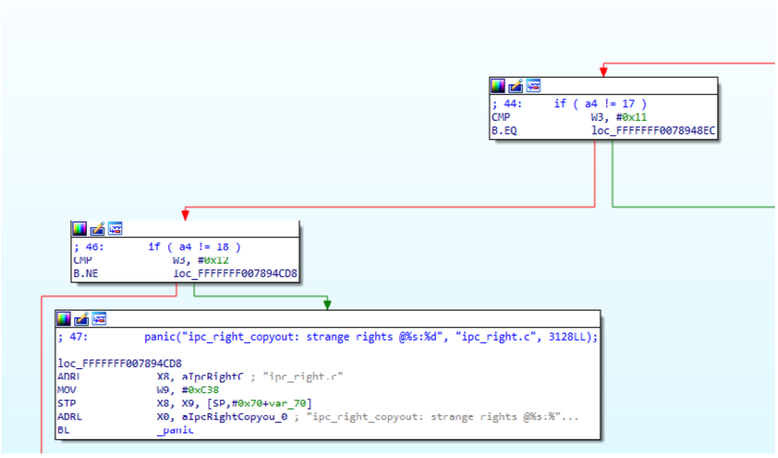
☐ Don't display this message again

After confirmation, comments with pseudocode lines are added to the disassembly:



```
FFFFFFF007894844                STP             X20, X19, [SP,#0x70+var_10]
FFFFFFF007894848                STP             X29, X30, [SP,#0x70+var_s0]
FFFFFFF00789484C                ADD             X29, SP, #0x70
FFFFFFF007894850 ; 39:   v12 = *(_DWORD *)(a3 + 8);
FFFFFFF007894850                MOV             X20, X7
FFFFFFF007894854                MOV             X22, X2
FFFFFFF007894858                MOV             X21, X1
FFFFFFF00789485C                MOV             X19, X0
FFFFFFF007894860                LDR             W24, [X2,#8]
FFFFFFF007894864 ; 40:   if ( (a5 & 1) != 0 )
FFFFFFF007894864                TBZ             W4, #0, loc_FFFFFFF007894874
FFFFFFF007894868 ; 41:     *(_DWORD *)(a8 + 8) |= 0x200000u;
FFFFFFF007894868                LDR             W8, [X20,#8]
FFFFFFF00789486C                ORR             W8, W8, #0x200000
FFFFFFF007894870                STR             W8, [X20,#8]
FFFFFFF007894874 ; 42:   if ( a4 != 16 )
FFFFFFF007894874
FFFFFFF007894874 loc_FFFFFFF007894874                   ; CODE XREF: sub_FFFFFFF00789482C+38↑j
FFFFFFF007894874                CMP             W3, #0x10
FFFFFFF007894878                B.EQ            loc_FFFFFFF0078948AC
FFFFFFF00789487C ; 44:     if ( a4 != 17 )
FFFFFFF00789487C                CMP             W3, #0x11
FFFFFFF007894880                B.EQ            loc_FFFFFFF0078948EC
FFFFFFF007894884 ; 46:       if ( a4 != 18 )
FFFFFFF007894884                CMP             W3, #0x12
FFFFFFF007894888                B.NE            loc_FFFFFFF007894CD8
FFFFFFF00789488C ; 48:       if ( (*(_BYTE *)(a8 + 9) & 0x10) != 0 )
FFFFFFF00789488C                LDRB            W8, [X20,#9]
FFFFFFF007894890                TBNZ            W8, #4, loc FFFFFFF007894964
```

You can see these comments even in the graph view[3]:



```
; 44:     if ( a4 != 17 )
CMP             W3, #0x11
B.EQ            loc_FFFFFFF0078948EC
```

```
; 46:       if ( a4 != 18 )
CMP             W3, #0x12
B.NE            loc_FFFFFFF007894CD8
```

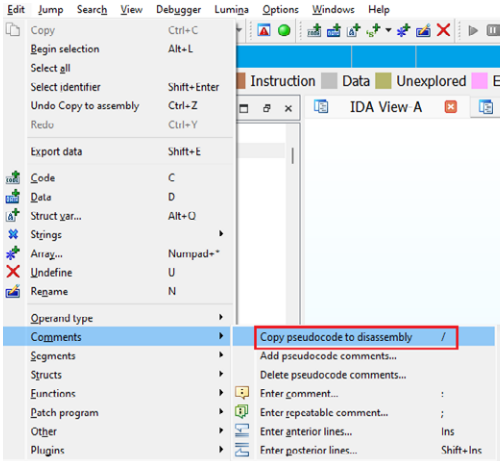
```
; 47:         panic("ipc_right_copyout: strange rights @%s:%d", "ipc_right.c", 3128LL);

loc_FFFFFFF007894CD8
ADRL            X8, aIpcRightC ; "ipc_right.c"
MOV             W9, #0xC38
STP             X8, X9, [SP,#0x70+var_70]
ADRL            X0, aIpcRightCopyou_0 ; "ipc_right_copyout: strange rights @%s:%"...
BL              _panic
```

#153: Copying pseudocode to disassembly

18 Aug 2023

https://hex-rays.com/blog/igors-tip-of-the-week-153-copying-pseudocode-to-disassembly/

In fact, you can make use of this feature even without switching to pseudocode. While in disassembly, use Edit > Comments > Copy pseudocode to disassembly, or the shortcut /


Edit Jump Search View Debugger Lumina Options Windows Help
Copy Ctrl+C
Begin selection Alt+L
Select all
Select identifier Shift+Enter
Undo Copy to assembly Ctrl+Z
Redo Ctrl+Y
Export data Shift+E
Code C
Data D
Struct var... Alt+Q
Strings
Array... Numpad+*
Undefine U
Rename N
Operand type
Comments
Segments
Structs
Functions
Patch program
Other
Plugins
Instruction
Data
Unexplored
E
IDA View-A
Copy pseudocode to disassembly /
Add pseudocode comments...
Delete pseudocode comments...
Enter comment... :
Enter repeatable comment... ;
Enter anterior lines... Ins
Enter posterior lines... Shift+Ins

Note that unlike pseudocode itself, these comments are static and do not change when you make changes in the pseudocode (e.g. rename variables). To update the comments, you need to trigger the action again.

In case you changed your mind and want to clean up the function, use “Delete pseudocode comments” from the same menu.

See also:
Hex-Rays interactive operation: Copy to assembly[4]
Igor’s tip of the week #14: Comments in IDA[5]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-40-decompiler-basics/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/
[4] https://www.hex-rays.com/products/decompiler/manual/cmd_copy.shtml
[5] https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/

#154: Synchronized views

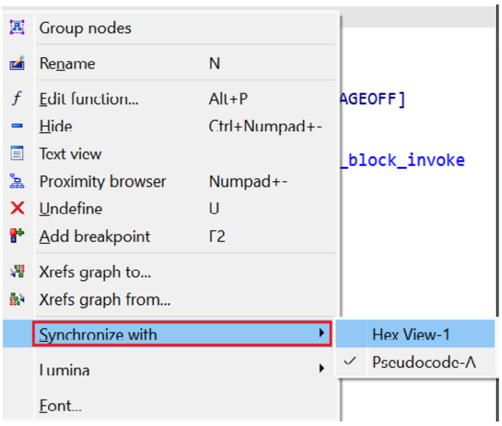
25 Aug 2023

https://hex-rays.com/blog/igors-tip-of-the-week-154-synchronized-views/

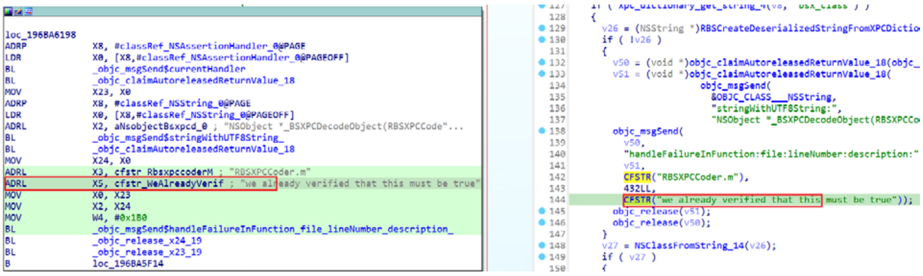
When working with a binary in IDA, most of the time you probably use one of the main views: disassembly (IDA View) or decompilation[1] (Pseudocode). If you need to switch between the two, you can use the `Tab` key – usually it jumps to the the same location in the other view. If you want to consult disassembly and pseudocode at the same time, copying pseudocode to disassembly[2] is one option, however it is of rather limited usefulness. You can dock[3] two view side-by-side and Tab between them, but this can be rather tedious.

Synchronizing views

To ensure that position in one view follows another automatically, select it in the “Synchronize with” context submenu.



Now, if you place disassembly and pseudocode side-by-side, the cursor position will be synchronized automatically when navigating in either window. The matching lines are also helpfully highlighted. Because a single pseudocode line may be represented by several assembly instructions and vice versa, the match is not one-to-one.



```
loc_196BA6198
ADRP            X8, #classRef_NSAssertionHandler_0@PAGE
LDR             X0, [X8,#classRef_NSAssertionHandler_0@PAGEOFF]
BL              _objc_msgSend$currentHandler
BL              _objc_claimAutoreleasedReturnValue_18
MOV             X23, X0
ADRP            X8, #classRef_NSString_0@PAGE
LDR             X0, [X8,#classRef_NSString_0@PAGEOFF]
ADRL            X2, aNsobjectBsxpcd_0 ; "NSObject *_BSXPCDecodeObject(RBSXPCCode"...
BL              _objc_msgSend$stringWithUTF8String_
BL              _objc_claimAutoreleasedReturnValue_18
MOV             X24, X0
ADRL            X3, cfstr_RbsxpccoderM ; "RBSXPCCoder.m"
ADRL            X5, cfstr_WeAlreadyVerif ; "we already verified that this must be true"
MOV             X0, X23
MOV             X2, X24
MOV             W4, #0x1B0
BL              _objc_msgSend$handleFailureInFunction_file_lineNumber_description_
BL              _objc_release_x24_19
BL              _objc_release_x23_19
B               loc_196BA5F14
```

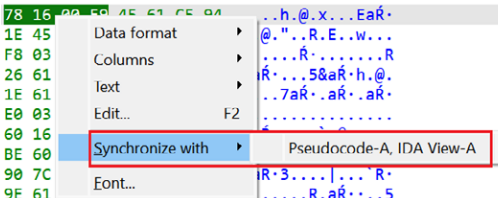
```
  if ( xpc_dictionary_get_string_4(v8, "bsx_class") )
  {
    v26 = (NSString *)RBSCreateDeserializedStringFromXPCDictio
    if ( !v26 )
    {
      v50 = (void *)objc_claimAutoreleasedReturnValue_18(objc_
      v51 = (void *)objc_claimAutoreleasedReturnValue_18(
                      objc_msgSend(
                        &OBJC_CLASS___NSString,
                        "stringWithUTF8String:",
                        "NSObject *_BSXPCDecodeObject(RBSXPCCo
      objc_msgSend(
        v50,
        "handleFailureInFunction:file:lineNumber:description:"
        v51,
        CFSTR("RBSXPCCoder.m"),
        432LL,
        CFSTR("we already verified that this must be true"));
      objc_release(v51);
      objc_release(v50);
    }
    v27 = NSClassFromString_14(v26);
    if ( v27 )
    {
```

Any view which displays information tied to addresses can be synchronized to another. As of IDA 8.3 these include:

1. Disassembly (IDA View)
2. Decompilation (Pseudocode)
3. Hex View[4]

You can even sync more than two views at the same time, although this has to be done in a specific sequence. For example:

1. Synchronize IDA View-A and Pseudocode-A
2. Synchronize Hex View with the other pair



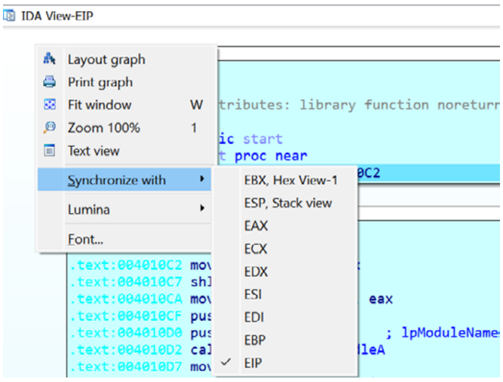
Synchronizing to registers in debugger

During debugging, an additional feature is available: synchronizing a view to a register value. You may have noticed that during debugging the default disassembly view changes name to IDA View-EIP (IDA View-RIP for x64 or IDA View-PC for ARM). This is because cursor follows the current execution address stored in the corresponding processor register.

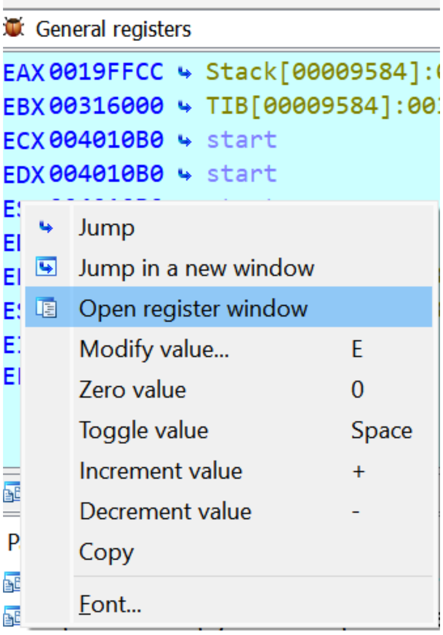
#154: Synchronized views

25 Aug 2023

https://hex-rays.com/blog/igors-tip-of-the-week-154-synchronized-views/

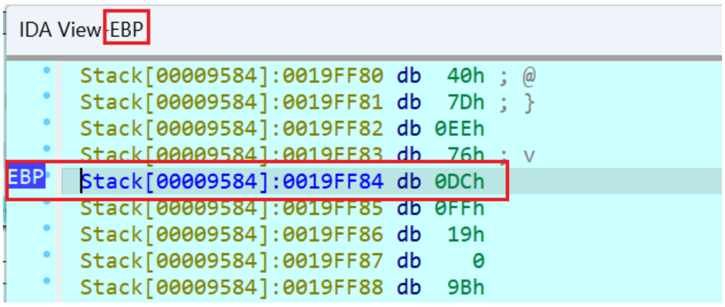


You can also synchronize the default Hex View to a register, or open additional views if you need to follow a specific one. For this, use “Open register window” from the context menu on the register in the registers view.



General registers

```
EAX 0019FFCC ↳ Stack[00009584]:0
EBX 00316000 ↳ TIB[00009584]:00
ECX 004010B0 ↳ start
EDX 004010B0 ↳ start
```



IDA View-EBP

```
Stack[00009584]:0019FF80 db  40h ; @
Stack[00009584]:0019FF81 db  7Dh ; }
Stack[00009584]:0019FF82 db 0EEh
Stack[00009584]:0019FF83 db  76h ; v
EBP Stack[00009584]:0019FF84 db 0DCh
Stack[00009584]:0019FF85 db 0FFh
Stack[00009584]:0019FF86 db  19h
Stack[00009584]:0019FF87 db    0
Stack[00009584]:0019FF88 db  9Bh
```

See also:
Igor’s tip of the week #22: IDA desktop layouts[5]
Igor’s tip of the week #38: Hex view[6]
Igor’s Tip of the Week #153: Copying pseudocode to disassembly[7]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-40-decompiler-basics/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-153-copying-pseudocode-to-disassembly/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-38-hex-view/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/
[6] https://hex-rays.com/blog/igors-tip-of-the-week-38-hex-view/
[7] https://hex-rays.com/blog/igors-tip-of-the-week-153-copying-pseudocode-to-disassembly/

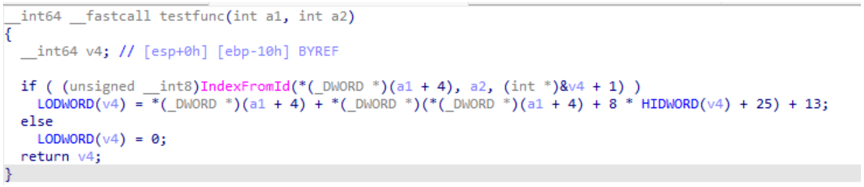
#155: Splitting stack variables in the decompiler

02 Sep 2023

https://hex-rays.com/blog/igors-tip-of-the-week-155-splitting-stack-variables-in-the-decompiler/

We've covered splitting expressions[1] before, but there may be situations where it can't be used.

For example, consider following situation:

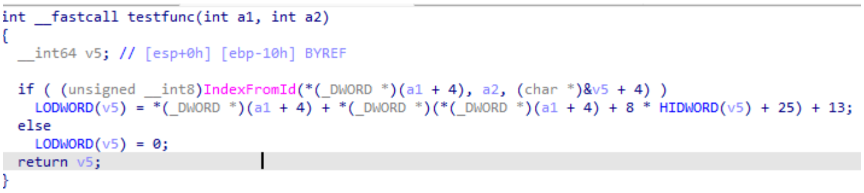


```
__int64 __fastcall testfunc(int a1, int a2)
{
  __int64 v4; // [esp+0h] [ebp-10h] BYREF

  if ( (unsigned __int8)IndexFromId(*(_DWORD *)(a1 + 4), a2, (int *)&v4 + 1) )
    LODWORD(v4) = *(_DWORD *)(a1 + 4) + *(_DWORD *)(*(_DWORD *)(a1 + 4) + 8 * HIDWORD(v4) + 25) + 13;
  else
    LODWORD(v4) = 0;
  return v4;
}
```

The decompiler decided that the function returns a 64-bit integer and allocated a 64-bit stack varible for it. For example, the code may be manipulating a register pair commonly used for 64-bit variables (`eax:edx`) which triggers the heirustics for recovering 64-bit calculations. However, here it seems to be a false positive: we can see separate accesses to the low and high dword of the variable, and the third argument for the IndexFromId call also uses a pointer into the middle of the variable.

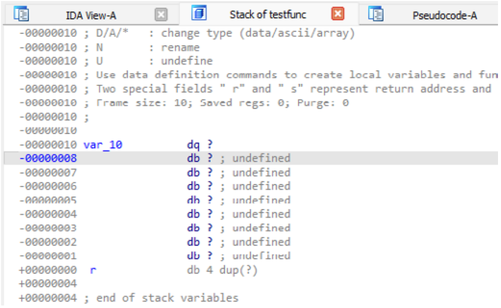
One option is to hint to the decompiler that the function returns a 32-bit integer by editing the function's prototype (use "Set item type" or the `Y` shotrcut on the first line).

Often this fixes the decompilation, but not here:



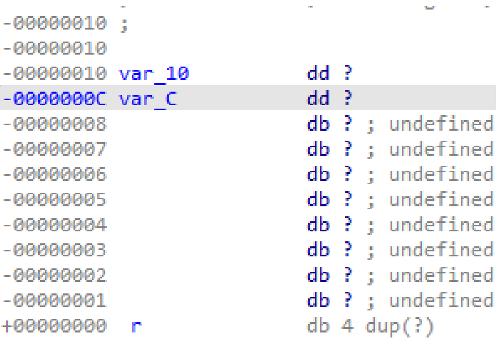
```
int __fastcall testfunc(int a1, int a2)
{
  __int64 v5; // [esp+0h] [ebp-10h] BYREF

  if ( (unsigned __int8)IndexFromId(*(_DWORD *)(a1 + 4), a2, (char *)&v5 + 4) )
    LODWORD(v5) = *(_DWORD *)(a1 + 4) + *(_DWORD *)(*(_DWORD *)(a1 + 4) + 8 * HIDWORD(v5) + 25) + 13;
  else
    LODWORD(v5) = 0;
  return v5;
}
```

We still have a 64-bt variable on the stack at `ebp-10h`, so it's worth inspecting the stack frame[2]. It can be opened by pressing Ctrl-K in disassembly view or double-cliking stack variable in disassembly or pseudocode:



```
-00000010 ; D/A/*   : change type (data/ascii/array)
-00000010 ; N       : rename
-00000010 ; U       : undefine
-00000010 ; Use data definition commands to create local variables and fun
-00000010 ; Two special fields " r" and " s" represent return address and
-00000010 ; Frame size: 10; Saved regs: 0; Purge: 0
-00000010 ;
-00000010
-00000010 var_10          dq ?
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004                 db ? ; undefined
-00000003                 db ? ; undefined
-00000002                 db ? ; undefined
-00000001                 db ? ; undefined
+00000000  r              db 4 dup(?)
+00000004
+00000004 ; end of stack variables
```

We see that there is a quadword (64-bit) variable at offset `-10`. it can be converted to 32-bit(dword) by pressing `D` three times. Another dword can be added in the same manner at offset `-C`:



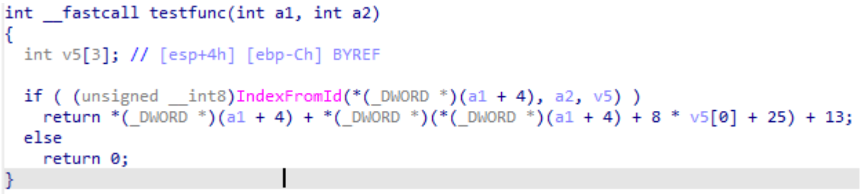
```
-00000010 ;
-00000010
-00000010 var_10          dd ?
-0000000C var_C           dd ?
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004                 db ? ; undefined
-00000003                 db ? ; undefined
-00000002                 db ? ; undefined
-00000001                 db ? ; undefined
+00000000  r              db 4 dup(?)
```

After refreshing pseudocode, we can see improved output:

#155: Splitting stack variables in the decompiler

02 Sep 2023

https://hex-rays.com/blog/igors-tip-of-the-week-155-splitting-stack-variables-in-the-decompiler/

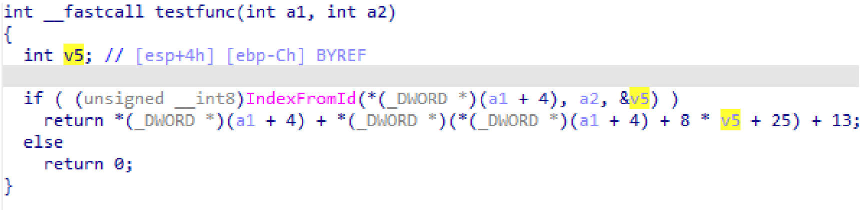


```
int __fastcall testfunc(int a1, int a2)
{
  int v5[3]; // [esp+4h] [ebp-Ch] BYREF

  if ( (unsigned __int8)IndexFromId(*(_DWORD *)(a1 + 4), a2, v5) )
    return *(_DWORD *)(a1 + 4) + *(_DWORD *)(*(_DWORD *)(a1 + 4) + 8 * v5[0] + 25) + 13;
  else
    return 0;
}
```

There's only one small issue: `v5` became an array. This happened bcause passing an array or an address of a single integer produces the same code but there was a gap in the stack frame after `var_C`, so the decompiler decided that it's actually an array. If you're certain that it's a single integer, you have the following options:

1. Edit the stack frame again and define some variables after `var_C` so that there is no space for an array.
2. retype v5 directly from the pseudocode (use Y and enter 'int').

Now the pseudocode looks correct and there is only one variable of correct size:



```
int __fastcall testfunc(int a1, int a2)
{
  int v5; // [esp+4h] [ebp-Ch] BYREF

  if ( (unsigned __int8)IndexFromId(*(_DWORD *)(a1 + 4), a2, &v5) )
    return *(_DWORD *)(a1 + 4) + *(_DWORD *)(*(_DWORD *)(a1 + 4) + 8 * v5 + 25) + 13;
  else
    return 0;
}
```

Note that in some cases a variable passed by address may be really an array, or a structure – in case of doubt inspect the called function to confirm how the argument is being used.

See also:

Igor's tip of the week #65: stack frame view[3]

Igor's tip of the week #42: Renaming and retyping in the decompiler[4]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-69-split-expression/

[2] https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/

[3] https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/

[4] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/

#156: Command-line options for firmware loading

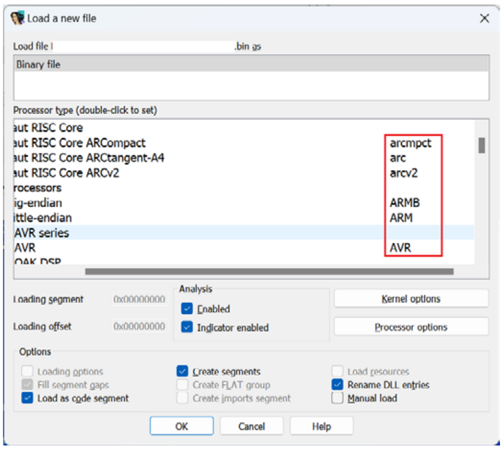
08 Sep 2023

https://hex-rays.com/blog/igors-tip-of-the-week-156-command-line-options-for-firmware-loading/

Firmware binaries often use raw binary file format without any metadata so they have to be loaded manually into IDA. You can do it interactively using the binary file loader[1], but if you have many files to disassemble it can quickly get boring. If you already know some information about the files you're disassembling, you can speed up at least the first steps. For example, if you have a binary for **big endian ARM**, which should be loaded at address **0xFFFF0000**, you can use the following command line:

```
ida -parmb -bFFFF000 firmware.bin
```

The-`p` switch tells IDA which processor module to pre-select. You can see the available names for different processor types in the second column of the processor selector pane in the load dialog:



Load a new file

Load file .bin as

Binary file

Processor type (double-click to set)

aut RISC Core	
aut RISC Core ARCompact	arcmpct
aut RISC Core ARCtangent-A4	arc
aut RISC Core ARCv2	arcv2
rocessors	
ig-endian	ARMB
ittle-endian	ARM
AVR series	
AVR	AVR
OAK DSP	

Loading segment 0x00000000

Loading offset 0x00000000

Analysis

☑ Enabled

☑ Indicator enabled

Options

☐ Loading options

☑ Fill segment gaps

☑ Load as code segment

☑ Create segments

☐ Create FLAT group

☐ Create imports segment

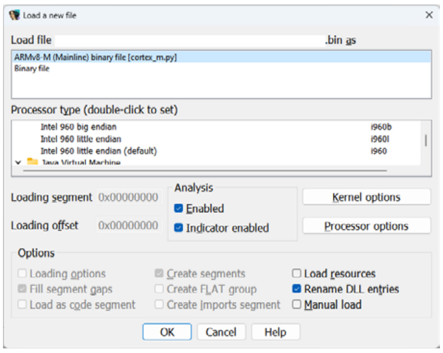
☐ Load resources

☑ Rename DLL entries

☐ Manual load

The `-b` switch specifies the load base to be used, however due to IDA's origins as a DOS program, the value needs to be specified in paragraphs (16-byte units), so we have to omit the last hexadecimal zero.

In case the file is recognized by IDA as some specific format, it will be used instead of the plain binary, but the processor specified will be retained if possible. For example, since IDA 8.3[2] the firmware for Cortex-M processors is usually recognized as such out-of-box:



Load a new file

Load file .bin as

ARMv8-M (Mainline) binary file [cortex_m.py]
Binary file

Processor type (double-click to set)

Intel 960 big endian	i960b
Intel 960 little endian	i960l
Intel 960 little endian (default)	i960
Java Virtual Machine	

Loading segment 0x00000000

Loading offset 0x00000000

Analysis

☑ Enabled

☑ Indicator enabled

Options

☐ Loading options

☑ Fill segment gaps

☐ Load as code segment

☑ Create segments

☐ Create FLAT group

☐ Create imports segment

☐ Load resources

☑ Rename DLL entries

☐ Manual load

If you prefer to have the file loaded as plain binary or another non-default format, you can force it using the -T switch with the unique prefix of the preferred format name:

```
ida -parm -b800400 -Tbinary firmware.bin
```

(`-Tbin` would also work)

See also:
IDA Help: Processor Type[3]
IDA Help: Command line switches[4]
Igor's tip of the week #41: Binary file loader[5]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-41-binary-file-loader/
[2] https://hex-rays.com/products/ida/news/8_3/
[3] https://www.hex-rays.com/products/ida/support/idadoc/618.shtml
[4] https://www.hex-rays.com/products/ida/support/idadoc/417.shtml
[5] https://hex-rays.com/blog/igors-tip-of-the-week-41-binary-file-loader/