IDA Pro 5.6 Appcall user guide

Introduction

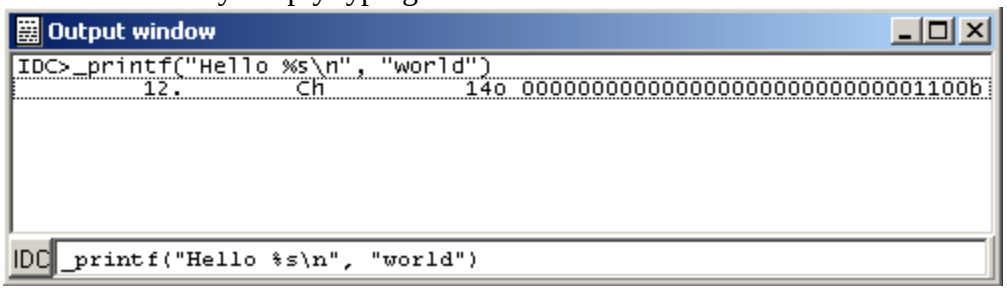
Appcall is a mechanism to call functions inside the debugged program from the debugger or your script as if it were a built-in function.
Such a mechanism can be used for debugging, fuzzing and testing applications.
Appcall mechanism highly depends on the type information of the function to be called. For that reason it is necessary to have a correct function prototype before doing an Appcall, otherwise different or incorrect results may be returned.

Quick start

To start with, we explain the basic concepts of Appcall using the IDC command line:

```
00404A0D ; int printf(const char *format, ...)
00404A0D _printf proc near                       ; CODE XREF: _count_files+92↑p
00404A0D                                         ; _str1_print+17↑p ...
00404A0D
00404A0D retval= dword ptr -1Ch
00404A0D ms_exc= CPPEH_RECORD ptr -18h
00404A0D format= dword ptr  8
00404A0D argptr= byte ptr  0Ch
00404A0D
```

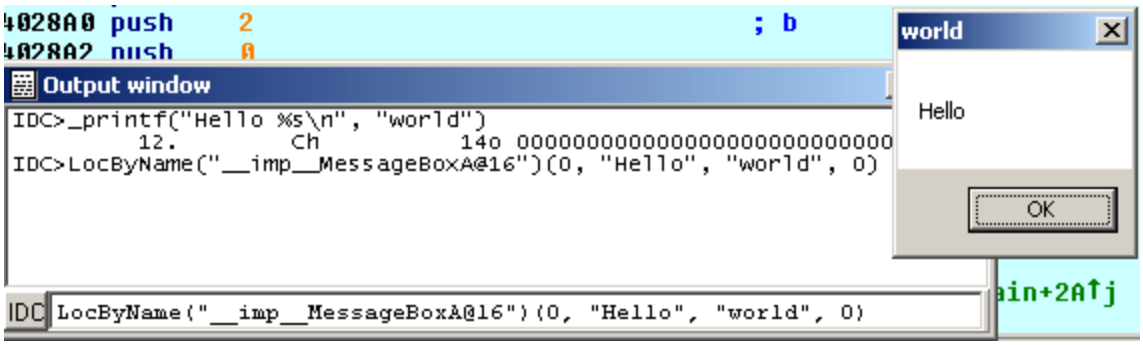
It can be called by simply typing:



As you notice, we invoked an Appcall by simply treating _printf as if it were a built-in IDC function.
If you have a function with a mangled name or with characters that cannot be used as an identifier name in the IDC language:

```
0040F140 ; int __stdcall MessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uTyp
0040F140 __imp__MessageBoxA@16 dd 7529FEC6h      ; CODE XREF: _mb1+34↑p
0040F140                                         ; DATA XREF: _mb1+34↑r ...
```

then issue the Appcall with this syntax:



We use the **LocByName** function to get the address of the function given its name, then using the address (which is callable) we issue the Appcall. In two steps this can be achieved with:

```
auto myfunc = LocByName("_my_func@8");
myfunc("hello", "world");
```

Please note that Appcalls take place in the context of the current thread. If you want to execute in a different thread then switch to the desired thread first.

Appcall and IDC

The Appcall mechanism can be used from IDC through the following function:

```
// Call application function
//      ea - address to call
//      type - type of the function to call. can be specified as:
//             - declaration string. example: "int func(void);"
//             - typeinfo object. example: GetTinfo(ea)
//             - zero: the type will be retrieved from the idb
//      ... - arguments of the function to call
// Returns: the result of the function call
// If the call fails because of an access violation or other exception,
// a runtime error will be generated (it can be caught with try/catch)
// In fact there is rarely any need to call this function explicitly.
// IDC tries to resolve any unknown function name using the application
labels
// and in the case of success, will call the function. For example:
//      _printf("hello\n")
// will call the application function _printf provided that there is
// no IDC function with the same name.

anyvalue Appcall(ea, type, ...);
```

The Appcall IDC function requires you to pass a function address, function type information and the parameters (if any):

```
auto p = LocByName("_printf");
auto ret = Appcall(p, GetTinfo(p), "Hello %s\n", "world");
```

We've seen so far how to call a function if it already have type information, now suppose we have a function that does not:

```
USER32:7524FC3B user32_FindWindowA:
USER32:7524FC3B mov     edi, edi
USER32:7524FC3D push    ebp
```

Before calling this function with Appcall() we need first to get the type information (stored as a typeinfo string) by calling ParseType() and then pass the function ea and type to Appcall():

```
auto p = ParseType("long __stdcall FindWindow(const char *cls, const char
*wndname)", 0);
Appcall(LocByName("user32_FindWindowA"), p, 0, "Untitled - Notepad");
```

Note that we used **ParseType()** function to construct a typeinfo object that we can pass to Appcall(). It is possible to permanently set the prototype of a function, thus:

```
SetType(LocByName("user32_FindWindowA"), "long __stdcall FindWindow(const
char *cls, const char *wndname)");
```

In the following sections, we are going to cover different scenarios such as calling by reference, working with buffers and complex structures, etc...

Passing arguments by reference

To pass function arguments by reference, it suffices to use the **&** symbol as in the C language.

- For example to call this function:

```
void ref1(int *a)
{
  if (a == NULL)
    return;
  int o = *a;
  int n = o + 1;
  *a = n;
  printf("called with %d and returning %d\n", o, n);
}
```

We can use this code from IDC:

```
auto a = 5;
Message("a=%d", a);
ref1(&a);
Message(", after the call=%d\n", a);
```

- To call a C function that takes a string buffer and modifies it:

```
/* C code */
int ref2(char *buf)
{
  if (buf == NULL)
    return -1;

  printf("called with: %s\n", buf);
  char *p = buf + strlen(buf);
  *p++ = '.';
  *p = '\0';
  printf("returned with: %s\n", buf);
  int n=0;
  for (;p!=buf;p--)
    n += *p;
  return n;
}
```

We need to create a buffer and pass it, thus:

```
auto s = strfill('\x00', 20); // create a buffer of 20 characters
```

```
s[0:5] = "hello"; // initialize the buffer
ref2(&s); // call the function and pass the string by reference
if (s[5] != ".")
  Message("not dot\n");
else
  Message("dot\n");
```

__usercall calling convention

It is possible to Appcall functions with non standard calling conventions, such as routines written in assembler that expect parameters in various registers and so on. One way is to describe your function with the **__usercall** calling convention.

Consider this function:

```
/* C code */
// eax = esi - edi
int __declspec(naked) asm1()
{
  __asm
  {
    mov eax, esi
    sub eax, edi
    ret
  }
}
```

And from IDC:

```
auto p = ParseType("int __usercall asm1<eax>(int a<esi>, int b<edi>);", 0);
auto r = Appcall(LocByName("_asm1"), p, 5, 2);
Message("The result is: %d\n", r);
```

Variable argument functions

In C:

```
int va_altsum(int n1, ...)
{
  va_list va;
  va_start(va, n1);

  int r = n1;
  int alt = 1;
  while ( (n1 = va_arg(va, int)) != 0 )
  {
    r += n1*alt;
    alt *= -1;
  }

  va_end(va);
  return r;
}
```

And in IDC:

```
auto result = va_altsum(5, 4, 2, 1, 6, 9, 0);
```

Calling functions that can cause exceptions

Exceptions may occur during an Appcall. To capture them, use the try/catch mechanism:

```
auto e;
try
{
  AppCall(some_func_addr, func_type, arg1, arg2);
  // Or equally:
  // some_func_name(arg1, arg2);
}
catch (e)
{
  // Exception occured .....
}
```

The exception object "e" will be populated with the following fields:

- description: description text generated by the debugger module while it was executing the Appcall
- func: The IDC function name where the exception happened.
- line: The line number in the script
- qerrno: The internal code of last error occured

For example, one could get something like this:

```
  description: "Appcall: The instruction at 0x401F93 referenced memory at
0x5. The memory could not be read"
  file: "<internal>"
  func: "___idc0"
  line: 4
  qerrno: 92
```

In some cases the exception object will contain more information.

Functions that accept or return structures

Appcall mechanism also works with functions that accept or return structure types. Consider this C++ code:

```
struct str1_t
{
  str1 t *next;
  int val;
};

str1_t *str1_make(int n)
{
  if ( n == 0 )
    return NULL;

  str1_t *node = new str1_t(), *head = node;
  node->val = 1;
  for (int i=2;i<=n;i++)
  {
    node->next = new str1_t();
    node = node->next;
    node->val = i;
  }
  node->next = NULL;
```

```
  return head;
}
```

This code will create for us a linked list of structures, if we Appcall str1_make from IDC we expect to have an object as a return value with nested objects until the end of the list:

```
IDC>x = _str1_make(3)
object
  next: object
    next: object
      next:           0.        0h          0o 00000000000000000000000000000000b '....'
      val:          3.        3h         3o 00000000000000000000000000000011b '....'
    val:          2.        2h         2o 00000000000000000000000000000010b '....'
  val:          1.        1h         1o 00000000000000000000000000000001b '....'
```

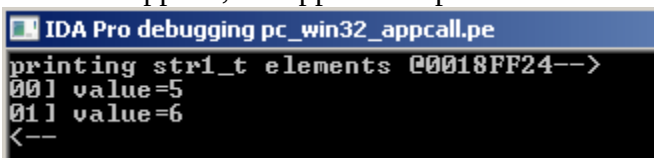
The top level object contains the **val** attribute and the **next** attribute pointing to a nested structure. Finally the third nested object points nowhere (contains a zero).
The Appcall mechanism will automatically create objects for us to represent the passed or returned structures.
Let us take another example where we have a function that accepts a structure:

```
int str1_print(str1_t *node)
{
  if ( node == NULL )
    return 0;

  printf("printing str1_t elements @%p-->\n", node);
  int n;
  for (n = 0; node != NULL; n++)
  {
    printf("%02d] value=%d\n", n, node->val);
    node = node->next;
  }
  printf("<--\n");
  return n;
}
```

To call this function from IDC we need to prepare a structure, populate it and then pass it:

```
auto t;
t = object(); // Create an empty object
t.val = 5;
t.next = object(); // link the nested object
t.next.val = 6;
t.next.next = 0; // end of list

str1_print(t); // Appcall
```

After the Appcall, the application prints:



Few observations:

- Objects are always passed by reference (no need to use the **&**)
- Objects are created on the stack
- Objects are untyped
- Missing object fields are automatically created by IDA and filled with zero

Let us take another example where we call the GetVersionExA API function:

```
kernel32.dll:76AE55F4 ; BOOL __stdcall GetVersionExA(LPOSVERSIONINFOA lpVersionInformatio
kernel32.dll:76AE55F4 GetVersionExA proc near
kernel32.dll:76AE55F4
kernel32.dll:76AE55F4 lpVersionInformation= dword ptr  8
kernel32.dll:76AE55F4
kernel32.dll:76AE55F4 mov     edi, edi
kernel32.dll:76AE55F6 push    ebp
```

Called as:

```
extern ver; // Create "ver" as a global variable
// Create an empty object
ver = object();
// We need to initialize the size of the structure
ver.dwOSVersionInfoSize = sizeof(ParseType("OSVERSIONINFO", 0));
// This is the only field we need to have initialized, the other fields will
be created by IDA and filled with zeroes
// Now issue the Appcall:
GetVersionExA(ver);
```

Now if we dump the **ver** object contents we observe something like this:

```
IDC>ver
object
  dwBuildNumber:        2600.      A28h        5050o 00000000000000000000101000101000b '(...'
  dwMajorVersion:          5.         5h           5o 00000000000000000000000000000101b '....'
  dwMinorVersion:          1.         1h           1o 00000000000000000000000000000001b '....'
  dwOSVersionInfoSize:        148.       94h         224o 00000000000000000000000010010100b '"...'
  dwPlatformId:          2.         2h          2o 00000000000000000000000000000010b '....'
  szCSDVersion: "Service Pack 3\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
```

Working with opaque structures

Opaque structures like FILE, HWND, HANDLE, HINSTANCE, HKEY, etc are not meant to be used as structures by themselves but like pointers.

Let us take for example the FILE structure that is used with fopen():

```
00000000 FILE struc ; (sizeof=0x18, standard type)
00000000 curp dd ?
00000004 buffer dd ?
00000008 level dd ?
0000000C bsize dd ?
00000010 istemp dw ?
00000012 flags dw ?
00000014 hold dw ?
00000016 fd db ?
00000017 token db ?
00000018 FILE ends
```

And the fopen() function:

```
.text:004052BC ; FILE *__cdecl fopen(const char *path, const char *mode)
.text:004052BC _fopen proc near                         ; CODE XREF: sub_4012A6+D
.text:004052BC                                          ; sub_4012A6+89↑p
.text:004052BC
.text:004052BC lpFileName= dword ptr  8
.text:004052BC mode= dword ptr  0Ch
.text:004052BC
.text:004052BC push    ebp
.text:004052BD mov     ebp, esp
```

Let us see how we can get a FILE * and use it as an opaque type and issue an fclose() call properly:

```
IDC>extern fp;
IDC>fp = _fopen("c:\\temp\\1.cpp", "r")
object
  __at__:     4320312.   41EC38h    203660700 00000000010000011110110000111000b '8ìA.'
  bsize:          512.       200h        10000 00000000000000000000001000000000b '....'
  buffer:           0.         0h            0 00000000000000000000000000000000b '....'
  curp: "ääA"
  fd:             3.         3h            30 00000000000000000000000000000011b '....'
  flags:            5.         5h            50 00000000000000000000000000000101b '....'
  hold:             0.         0h            00 00000000000000000000000000000000b '....'
  istemp:           0.         0h            00 00000000000000000000000000000000b '....'
  level:            0.         0h            00 00000000000000000000000000000000b '....'
  token:           56.        38h           700 00000000000000000000000000111000b '8...'
IDC>_fclose(fp.__at__);
```

Nothing special about the fopen/fclose Appcalls except that we see the **__at__** attribute showing up although it does not belong to the FILE structure definition.
This is a special attribute that IDA inserts into all objects, and it contains the memory address from which IDA retrieved the object attribute values. We can use the __at__ to retrieve the C pointer of a given IDC object.

Previously we omitted the **__at__** field from the str1_make() Appcall output, but in reality this is what one expects to see:

```
IDC>x = _str1_make(3)
object
  __at__:    31003208.  1D91248h   1662111100 00000001110110010001001001001
  next: object
    __at__:    31003240.  1D91268h   1662111500 00000001110110010001001001101
    next: object
      __at__:    31003272.  1D91288h   1662112100 00000001110110010001001010100
      next:           0.         0h            00 00000000000000000000000000000000
      val:            3.         3h            30 00000000000000000000000000000011
    val:              2.         2h            20 00000000000000000000000000000010
  val:                1.         1h            10 00000000000000000000000000000001
```

And again we notice that the __at__ holds the address of each linked item in this list. Now if we want to free the list:

```
/* C code */
void str1_free(str1_t *head);
```

We type:

```
/* IDC code */
str1_free(x.__at__); // Free the whole list
// Or to free from the 2nd element and on:
str1_free(x.next.__at__);
```

It is possible to pass as integer (which is a pointer) to a function that expects a pointer to a structure.

FindFirst/FindNext example

```
#include <idc.idc>

// Define global variables
extern p_findfirst, p_findnext, p_findclose;

static init_api()
{
  p_findfirst = LocByName("__imp__FindFirstFileA@8");
  p_findclose = LocByName("__imp__FindClose@4");
  p_findnext  = LocByName("__imp__FindNextFileA@8");
}

static test_ff()
{
  auto fd, h, n;

  fd = object(); // create an object
  h = p_findfirst("*.exe", fd);
  if (h == -1) // INVALID_HANDLE_VALUE
  {
    Message("No files found!\n");
    return -1;
  }
  for (n=1;;n++)
  {
    Message("Found: %s\n", fd.cFileName);
    if ( (n > 5) || (p_findnext(h, fd) == 0) )
      break;
  }
  p_findclose(h);
  return n;
}

static main()
{
  init_api();
  test_ff();
}
```

Using GetProcAddress

```
#include <idc.idc>

extern getmodulehandle, getprocaddr, findwindow;

static init_api()
{
  getmodulehandle = LocByName("kernel32_GetModuleHandleA");
  getprocaddr = LocByName("kernel32_GetProcAddress");

  if (getmodulehandle == BADADDR || getprocaddr == BADADDR)
    return "Failed to locate GetModuleHandle/GetProcAddress";
```

```
  // Let us permanently set the prototypes of these two functions
  SetType(getmodulehandle, "HMODULE __stdcall getmodulehandle(LPCSTR
lpModuleName);");
  SetType(getprocaddr, "FARPROC __stdcall GetProcAddress(HMODULE hModule,
LPCSTR lpProcName);");

  // Resolve address of FindWindow api
  auto t = getmodulehandle("user32.dll");
  if (t == 0)
    return "User32 is not loaded!";

  findwindow = getprocaddr(t, "FindWindowA");
  if (findwindow == 0)
    return "FindWindowA API not found!";

  // Set type
  SetType(findwindow, "HWND __stdcall FindWindowA(LPCSTR lpClassName, LPCSTR
lpWindowName);");

  return "ok";
}

static main()
{
  auto ok = init_api();
  if (ok != "ok")
  {
    Message("Failed to initialize: %s", ok);
    return -1;
  }
  auto ida_hwnd = findwindow("TIdaWindow", 0);
  if (ida_hwnd == 0)
  {
    Message("Failed to locate IDA window!\n");
    return -1;
  }
  Message("IDA hwnd=%x\n", ida_hwnd);
  return 0;
}
```

Retrieving application's command line

```
#include <idc.idc>

extern getcommandline;

static main()
{
  getcommandline = LocByName("kernel32_GetCommandLineA");
  if (getcommandline == BADADDR)
  {
    Message("Failed to resolve GetCommandLineA API address!\n");
    return -1;
  }
  SetType(getcommandline, "const char *__stdcall GetCommandLineA();");
```

```
  Message("This application's command line:<\n%s\n>\n", getcommandline());
  return 0;
}
```

Specifying Appcall options

Appcall can be configured with SetAppcallOptions(), by passing the desired option(s):

- APPCALL_MANUAL: Only set up the appcall, do not run it (you should call CleanupAppcall() when finished). Please Refer to Manual Appcall section for more information.
- APPCALL_DEBEV: If this bit is set, exceptions during appcall will generate IDC exceptions with full information about the exception. Please refer to Capturing exception debug events section for more information.

It is possible to retrieve the Appcall options, change them and then restore them back. To retrieve the options use the **GetAppcallOptions()**.
Please note that Appcall option is saved in the database so if you set it once it will retain its value as you save and load the database.

Manual Appcall

So far we've seen how to issue an Appcall and capture the result from the script, but what if we only want to setup the environment and manually step through a function?
This can be achieved with manual Appcall. The manual Appcall mechanism can be used to save the current execution context, execute another function in another context and then pop back the previous context and continue debugging from that point. Let us directly illustrate manual Appcall with a real life scenario:

1. You are debugging your application
2. You discover a buggy function (foo()) that misbehaves when called with certain arguments: foo(0xdeadbeef)
3. Instead of waiting until the application calls foo() with the desired arguments that can cause foo() to misbehave, you can manually call foo() with the desired arguments, trace the function
4. Finally, one calls CleanupAppcall() to restore the execution context

To illustrate, let us take the ref1 function and call it with an invalid pointer:

1. SetAppcallOptions(APPCALL_MANUAL); // Set manual Appcall mode
2. ref1(6); // call the function with an invalid pointer

Directly after doing that, IDA will switch to the function and from that point on we can debug:

```
00401F80 ; int __cdecl ref1(int *a)
00401F80 _ref1 proc near
00401F80
00401F80 o= dword ptr -8
00401F80 n= dword ptr -4
00401F80 a= dword ptr  8
00401F80
00401F80 push    ebp
00401F81 mov     ebp, esp
00401F83 sub     esp, 8
00401F86 cmp     [ebp+a], 0
00401F8A jnz     short loc_401F90
00401F8C xor     eax, eax
00401F8E jmp     short loc_401FC1
```

When we reach the end of the function:

```
00401FC1
00401FC1 loc_401FC1:
00401FC1 mov     esp, ebp
00401FC3 pop     ebp
00401FC4 retn
00401FC4 _ref1 endp
```

and trace beyond the return instruction, we expect to see something like this:

```
0018FEE4 int     3
0018FEE5
0018FEE5 loc_18FEE5:
0018FEE5 jmp     short loc_18FEE5
```

This is the control code that we use to determine the end of an Appcall. It is at this point that one should call CleanupAppcall() to return to the previous execution context:

```
00404A4B pop     ebx
00404A4C add     eax, ebx
00404A4E push    eax
00404A4F push    1
00404A51 call    __lock_file2
```

Initiating multiple manual Appcalls

It is possible to initiate multiple Appcalls. If manual Appcall is enabled, then issueing an Appcall from an Appcall will stack the current context and switch to the new Appcall context. CleanupAppcall() will pop the contexts one by one (LIFO style). If you happen to be tracing a function then you want to debug another function and return then issue another Appcall!

Manual Appcalls are not designed to be called from a script, nonetheless if you use them from a script:

```
auto i;
printf("Loop started\n");
for (i=0;i<10;i++)
{
  Message("i=%d\n", i);
}
printf("Loop finished\n");
```

We observe the following:

1. First Appcall will be initiated
2. The script will loop and print the results
3. Another Appcall will be initiated
4. The script terminates
5. The execution context will be setup for tracing the last issued Appcall
6. After this Appcall is finished, we observe "Loop finished"
7. We issue CleanupAppcall() and notice that the execution context is back to printf but this time it will print "Loop started"
8. Finally when we call again CleanupAppcall() we resume our initial execution context

Capturing exception debug events

We previously illustrated that we can capture exceptions that occur during an Appcall, but that is not enough if we want to learn more about the nature of the exception from the operating system point of view. It would be better if we could somehow get the last **debug_event_t** that occured inside the debugger module. This is possible if we use the APPCALL_DEBEV option. Let us repeat the previous example but with the APPCALL_DEBEV option enabled:

```
auto e;
try
{
  SetAppcallOptions(APPCALL_DEBEV); // Enable debug event capturing
  ref1(6);
}
catch (e)
{
  // Exception occured ..... this time "e" is populated with debug_event_t
fields (check idd.hpp)
}
```

And in this case, if we dump the exception object's contents, we get these attributes:

```
can_cont: 1
code:  C0000005h
ea:    401F93h
eid:    40h (from idd.hpp: EXCEPTION = 0x00000040 Exception)
file: ""
func: "___idc0"
handled: 1
info: "The instruction at 0x401F93 referenced memory at 0x6. The memory could
not be read"
line: 4h
pid:  123Ch
ref:  6h
tid:  1164h
```

Appcall related functions

There are some functions that can be used while working with Appcalls.

ParseType/GetTinfo/sizeof

The GetTinfo() function is used to retrieve the typeinfo string associated with a given address.

```
// ********************************************
// ** Get type information of function/variable as 'typeinfo' object
//         ea - the address of the object
//    returns: typeinfo object, 0 - failed
// The typeinfo object has 2 attributes: type and fields

typeinfo GetTinfo(long ea);
```

The ParseType() function is used to construct a typeinfo string from a type string. We already used ParseType() to construct a typeinfo string and passed it to Appcall().

```
// ********************************************
// ** Parse one type declaration
//         input -  a C declaration
//         flags -  combination of PT_... constants or 0
//                  PT_FILE should not be specified in flags (it is ignored)
//    returns: typeinfo object or num 0

typeinfo ParseType(string input, long flags);
```

And finally, given a typeinfo string, one can use the sizeof() function to calculate the size of a type:

```
// ********************************************
// Calculate the size of a type
//      type - type to calculate the size of
//             can be specified as a typeinfo object (e.g. the result of
GetTinfo())
//             or a string with C declaration (e.g. "int")
// Returns: size of the type or -1 if error

long sizeof(typeinfo type);
```

Accessing enum members as constants

In IDC, it is possible to access all the defined enumerations as if they were IDC constants:

```
// From IDA enums window:
00000001 ; enum MACRO_PAGE (standard) (bitfield)
00000001 PAGE_NOACCESS  = 1
00000002 PAGE_READONLY  = 2
00000004 PAGE_READWRITE  = 4
00000008 PAGE_WRITECOPY  = 8
00000010 PAGE_EXECUTE  = 10h
00000020 PAGE_EXECUTE_READ  = 20h
00000040 PAGE_EXECUTE_READWRITE  = 40h
```

Then one can type:

```
Message("PAGE_EXECUTE_READWRITE=%x\n", PAGE_EXECUTE_READWRITE);
```

Storing/Retrieving typed elements

It is possible to store/retrieve (aka serialize/deserialize) objects to/from the database. To illustrate, let us consider the following memory contents:

```
0001000C dd 1003219h
00010010 dw 0FFEEh
00010012 dw 0FFEEh
00010014 dd 1
```

And we know that this maps to a given type:

```
struct X
{
  unsigned long a;
  unsigned short b, c;
  unsigned long d;
};
```

To retrieve (deserialize) the memory contents into a nice IDC object:

```
// Create the typeinfo string
auto t = ParseType("struct X { unsigned long a; unsigned short b, c; unsigned
long d;};", 0);
// Create a dummy object
auto o = object();
// Retrieve the contents into the object:
o.retrieve(t, 0x1000C, 0);
```

And now if we dump the contents of **o**:

```
IDC>o
object
  __at__:       65548.    1000Ch      200014o
00000000000000010000000000001100b '....'
  a:    16790041.  1003219h   100031031o 00000001000000000011001000011001b
'.2..'
  b:       65518.     FFEEh      177756o 00000000000000001111111111101110b
'î...'
  c:       65518.     FFEEh      177756o 00000000000000001111111111101110b
'î...'
  d:           1.        1h           1o 00000000000000000000000000000001b
'....'
```

and again we notice the **__at__** which holds the address of the retrieved object. To store (serialize) the object back into memory:

```
o.a++; // modify the field
o.d = 6; // modify another field
o.store(t, o.__at__, 0);
```

And finally to verify, we go to the memory address:

```
0001000C dd 100321Ah
00010010 dw 0FFEEh
00010012 dw 0FFEEh
00010014 dd 6
```

Appcall and Python

The Appcall concept remains the same between IDC and Python, nonetheless Appcall/Python has a different syntax (using references, unicode strings, etc, etc...)

The Appcall mechanism is provided by idaapi module through the Appcall variable. To issue an Appcall:

```
Appcall.printf("Hello world!\n");
```

One can take a reference to an Appcall:

```
printf = Appcall.printf
# ...later...
printf("Hello world!\n");
```

- In case you have a function with a mangled name or with characters that cannot be used as an identifier name in the Python language:

```
findclose     = Appcall["__imp__FindClose@4"]
getlasterror  = Appcall["__imp__GetLastError@0"]
setcurdir     = Appcall["__imp__SetCurrentDirectoryA@4"]
```

- In case you want to redefine the prototype of a given function, then use the Appcall.proto(func_name or func_ea, prototype_string):

```
# pass an address name and Appcall.proto() will resolve it
loadlib = Appcall.proto("__imp__LoadLibraryA@4", "int (__stdcall
*LoadLibraryA)(const char *lpLibFileName);")
# Pass an EA instead of a name
freelib = Appcall.proto( LocByName("__imp__FreeLibrary@4"), "int (__stdcall
*FreeLibrary)(int hLibModule);")
```

- To pass unicode strings you need to use the Appcall.unicode() function:

```
    getmodulehandlew    = Appcall.proto("__imp__GetModuleHandleW@4", "int
(__stdcall *GetModuleHandleW)(LPCWSTR lpModuleName);")
    hmod = getmodulehandlew(Appcall.unicode("kernel32.dll"))
```

- To pass int64 values to a function you need to use the Appcall.int64() function:

```
/* C code */

int64 op_two64(int64 a, int64 b, int op)
{
  if (op == 1)
    return a + b;
  else if (op == 2)
    return a - b;
  else if (op == 3)
    return a * b;
  else if (op == 4)
    return a / b;
  else
    return -1;
}
```

Python Appcall code:

```
r = Appcall.op_two64(Appcall.int64(1), Appcall.int64(2), 1)
print "result=", r.value
```

- To define a prototype and then later assign an address so you can issue an Appcall:

```
# Create a typed object (no address is associated yet)
virtualalloc = Appcall.typedobj("int __stdcall VirtualAlloc(int lpAddress,
SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);")
# Later we have an address, so we pass it:
virtualalloc.ea = LocByName("kernel32_VirtualAlloc")
# Now we can Appcall:
```

```
ptr = virtualalloc(0, Appcall.Consts.MEM_COMMIT, 0x1000,
Appcall.Consts.PAGE_EXECUTE_READWRITE)
```

Passing arguments by reference

- To pass function arguments by reference, one has to use the Appcall.byref():

```
# Create a byref object holding the number 5
i = Appcall.byref(5)
# Call the function
Appcall.ref1(i)
# Retrieve the value
print "Called the function:", i.value
```

- To call a C function that takes a string buffer and modifies it, we need to use the Appcall.buffer(initial_value, [size]) function to create a buffer:

```
buf = Appcall.buffer("test", 100)
Appcall.ref2(buf)
```

- Another real life example is when we want to call the GetCurrentDirectory() API:

```
# Take a reference
getcurdir = Appcall["__imp__GetCurrentDirectoryA@8"]
# make a buffer
buf = Appcall.byref("\x00" * 260)
# get current directory
n = getcurdir(260, buf)
print "curdir=%s" % buf.cstr()
```

- To pass int64 values by reference:

```
/* C code */
int ref4(int64 *a)
{
  if (a == NULL)
  {
    printf("No number passed!");
    return -1;
  }
  printf("Entered with %" FMT_64 "d\n", *a);
  (*a)++;
  return 1;
}
```

We use the following Python code:

```
# Create an int64 value
i = Appcall.int64(5)
# create a reference to it
v = Appcall.byref(i)
# appcall
Appcall.ref4(v)
print "The result is:", i.value
```

- To call a C function that takes an array of integers or an array of a given type:

```
/* C code */
int ref3(int *arr, int sz)
{
  if (arr == NULL)
    return 0;
  int sum = 0;
  for (int i=0;i<sz;i++)
    sum += arr[i];
  return sum;
}
```

First we need to use the Appcall.array() function to create an array type, then we use the Appcall.array().pack() function to encode the Python values into a buffer:

```
# create an array type
arr = a.array("int")
# create a list
L = [x for x in xrange(1, 10)]
# pack the list
p_list = arr.pack(L)
# appcall to compute the total
c_total = a._ref3(p_list, len(L))
# internally compute the total
total = reduce(operator.add, L)
if total != c_total:
    print "Appcall failed!"
```

Functions that accept or return structures

Like in IDC, we can create objects and pass them with at least two methods.

The first method involves using the Appcall.obj() function that takes an arbitrary number of keyword args that will be used to create an object with the arguments as attributes:

```
# object representing the str1_t type
o = Appcall.obj(val=5,next=Appcall.obj(val=-2, next=0))
```

Or with a dictionary:

```
# dictionary representing the str1_t type
# (dictionaries will be converted into IDC objects)
d = {'val':5, 'next': {'val':-2, 'next':0} }
```

And finally, if you happen to have your own object instance then just pass your object. The Python object to IDC object conversion routine will skip attributes starting and ending with "__".

FindFirst/FindNext example

```
# For simplicity:
a = Appcall
getcurdir           = a["__imp__GetCurrentDirectoryA@8"]
getwindir           = a["__imp__GetWindowsDirectoryA@8"]
setcurdir           = a["__imp__SetCurrentDirectoryA@4"]
findfirst           = a["__imp__FindFirstFileA@8"]
findnext            = a["__imp__FindNextFileA@8"]
findclose           = a["__imp__FindClose@4"]
```

```
# Tests changedir/setdir/buffer creation (two methods) and cstr()
def test_enum_files():
    # create a buffer
    savedpath = a.byref("\x00" * 260)
    # get current directory
    n = getcurdir(250, savedpath)
    out = []
    out.append("curdir=%s" % savedpath.value[0:n])

    # get windir
    windir = a.buffer(size=260) # create a buffer using helper function
    n = getwindir(windir, windir.size)
    if n == 0:
        return -1 # could not get current directory

    windir = windir.value[:n]
    out.append("windir=%s" % windir)

    # change to windows folder
    setcurdir(windir)

    # initiate find
    fd = a.obj()
    h = findfirst("*.exe", fd)
    if h == -1:
        return -2 # no files found!

    found = -6
    while True:
        fn = a.cstr(fd.cFileName)
        if "regedit" in fn:
            found = 1
        out.append("fn=%s<" % fn)
        fd = a.obj() # reset the FD object
        ok = findnext(h, fd)
        if not ok:
            break
    findclose(h)

    # restore cur dir
    setcurdir(savedpath.value)

    # verify
    t = a.buffer(size=260)
    n = getcurdir(t.size, t)
    if t.cstr() != savedpath.cstr():
        return -4 # could not restore cur dir

    out.append("curdir=%s<" % t.cstr())
    print "all done!"
    for l in out:
        print l

    return found
```

Using GetProcAddress

```
a = Appcall
loadlib        = a.proto("__imp__LoadLibraryA@4", "int (__stdcall
*LoadLibraryA)(const char *lpLibFileName);")
getprocaddr    = a.proto("__imp__GetProcAddress@8", "int (__stdcall
*GetProcAddress)(int hModule, LPCSTR lpProcName);")

def test_gpa():
    h = loadlib("user32.dll")
    if h == 0:
        print "failed to load library!"
        return -1
    p = getprocaddr(h, "FindWindowA")
    if p == 0:
        print "failed to gpa!"
        return -2
    findwin = a.proto(p, "int FindWindow(LPCTSTR lpClassName, LPCTSTR
lpWindowName);")
    hwnd = findwin("TIdaWindow", 0)
    freelib(h)
    print "%x: ok!->hwnd=%x" % (p, hwnd)
    return 1
```

Setting the Appcall options

In Python, the Appcall options can be set global or locally per Appcall.

- To set the global Appcall setting:

```
old_options = Appcall.set_appcall_options(Appcall.APPCALL_MANUAL)
```

- To set the Appcall setting per Appcall:

```
# take a reference to printf
printf = Appcall._printf
# change the setting for this Appcall
printf.options = Appcall.APPCALL_DEBEV
printf("Hello world!\n")
```

Similarly, retrieving the Appcall options is done by either calling Appcall.get_appcall_options() or by reading the options attribute (for example: printf.options)

To cleanup after a manual Appcall use the Appcall.cleanup_appcall().

Calling functions that can cause exceptions

Like in IDC, it is possible to guard an Appcall against exceptions by using the try/catch mechanism.

- Let us try when the Appcall options does not include the APPCALL_DEBEV flag:

```
try:
```

```
    # causes a DebugBreak()
    Appcall.ex2(1)
except Exception, e:
    if "Software breakpoint" in e.message:
        print "Got an exception!"
```

When the APPCALL_DEBEV flag is absent, any Appcall that generated an exception while executing in the current thread will throw a Python **Exception** object. Now if we use the APPCALL_DEBEV flag, we get an **OSError** exception object with its **args[0]** populated with the last debug_event_t:

```
try:
    Appcall.set_appcall_options(Appcall.APPCALL_DEBEV)
    # causes an OS error and "e" will contain the last debug_event
    # in our case exception, and the code is int_divide_zero = 0xC0000094
    print Appcall.op_two64(2, 0, 4).value
    print "should never reach here"
except OSError, e:
    if idaapi.as_uint32(e.args[0].code) == 0xc0000094:
        print "okay...caught the exception"
except RuntimeError, er:
    print "aha!"
except Exception, e:
    print "Got other exception:", e
    print "should never reach here"
```

Appcall related functions in Python

Storing/Retrieving objects

Storing/Retrieving objects is also supported in Python:

1. Using the IDA SDK (through the idaapi Python module)
2. Using Appcall helper functions

In this example we show how to:

1. Unpack the DOS header at 0x400000 and verify the fields
2. Unpack a string and see if it is unpacked correctly

```
def test_unpack_raw():
    name, tp, flds = idc.ParseType("IMAGE_DOS_HEADER;", 0)
    ok, obj = idaapi.unpack_object_from_idb(idaapi.cvar.idati, tp, flds,
0x400000, 0)
    if obj.e_magic != 23117 and obj.e_cblp != 144:
        return -1

    # Parse the type into a type name, typestring and fields
    name, tp, flds = idc.ParseType("struct abc_t { int a, b;};", 0)
    # Unpack from a byte vector (bv) (aka string)
    ok, obj = idaapi.unpack_object_from_bv(idaapi.cvar.idati, tp, flds,
"\x01\x00\x00\x00\x02\x00\x00\x00", 0)
    if obj.a != 1 and obj.b != 2:
        return -2
```

```
    return 1
```

Now to accomplish similar result using Appcall helper functions:

```
WRITE_AREA = 0x500000 # some writable area
# Packs/Unpacks a structure to the database using appcall facilities
def test_pck_idb():
    print "%x: ..." % WRITE_AREA
    tp = a.typedobj("struct { int a, b; char x[5];};")
    o = a.obj(a=16, b=17,x="zzzhi")
    if tp.store(o, WRITE_AREA) != 0:
        return -1
    idc.RefreshDebuggerMemory()

    ok, r = tp.retrieve(WRITE_AREA)
    if not ok:
        return -2
    if r.a != o.a and r.b != o.b and r.x != o.x:
        return -3
    return 1

# ------------------------------------------------------------------------
# Packs/Unpacks a structure to/from a string
def test_pck_bv():
    tp = a.typedobj("struct { int a, b; char x[5];};")
    o = a.obj(a=16, b=17,x="zzzhi")
    ok, packed = tp.store(o)
    if not ok:
        return -1
    print "packed->", repr(packed)
    ok, r = tp.retrieve(packed)
    if not ok:
        return -2
    if r.a != o.a and r.b != o.b and r.x != o.x:
        return -3
    return 1
```

Accessing enum members as constants

Like in IDC, to access the enums, one can use the Appcall.Consts object:

```
print "PAGE_EXECUTE_READWRITE=%x" % Appcall.Consts.PAGE_EXECUTE_READWRITE
```

If the constant was not defined then zero will be returned. If you desire to specify a default value in case a constant was absent then use the following syntax:

```
print "PAGE_EXECUTE_READWRITE=%x" % Appcall.valueof("PAGE_EXECUTE_READWRITE", 0x40)
```

Please send your comments or questions to support@hex-rays.com