# Table of Contents

1. **Introduction** .................................................................................................................. 6  
   1.1 Why This Tutorial? ........................................................................................................ 6  
   1.2 What’s Covered ............................................................................................................. 6  
   1.3 What’s Not Covered ....................................................................................................... 6  
   1.4 Knowledge Required .................................................................................................... 6  
   1.5 Software Required ....................................................................................................... 7  
   1.6 Alternatives to C/C++ .................................................................................................. 7  
   1.7 About This Document ................................................................................................. 7  
   1.8 Credits ......................................................................................................................... 7  
   1.9 Further Reading ........................................................................................................... 7  

2. **The IDA SDK** ................................................................................................................. 8  
   2.1 Installation ..................................................................................................................... 8  
   2.2 Directory Layout .......................................................................................................... 9  
   2.3 Header Files ................................................................................................................ 9  
   2.4 Using the SDK ............................................................................................................. 10  

3. **Setting Up a Build Environment** ................................................................................... 11  
   3.1 Windows, Using Visual Studio ..................................................................................... 11  
   3.2 Windows, Using Dev-C++ With GCC and MinGW ....................................................... 12  
   3.3 Linux, Using GCC ....................................................................................................... 12  
   3.4 A Plug-in Template ..................................................................................................... 13  
   3.5 Configuring and Running Plug-ins ............................................................................. 14  

4. **Fundamentals** ................................................................................................................ 16  
   4.1 Core Types .................................................................................................................. 16  
   4.2 Core Structures and Classes ....................................................................................... 17  
      4.2.1 Meta Information .................................................................................................. 17  
      4.2.2 Areas ................................................................................................................... 17  
      4.2.2.1 The area_t Structure .................................................................................. 18  
      4.2.2.2 The areacb_t Class .................................................................................... 18  
      4.2.3 Segments and Functions ..................................................................................... 19  
      4.2.3.1 Segments ....................................................................................................... 19  
      4.2.3.2 Functions ...................................................................................................... 20  
      4.2.4 Code Representation ............................................................................................ 21  
      4.2.4.1 Operand Types .......................................................................................... 22  
      4.2.4.2 Operands ..................................................................................................... 22  
      4.2.4.3 Mnemonics ................................................................................................. 23  
      4.2.4.4 Instructions ................................................................................................. 23  
      4.2.5 Cross Referencing ............................................................................................... 24  
      4.2.5.1 The xrefblk_t Structure .......................................................................... 24  
      4.2.5.2 Code ........................................................................................................... 25  
      4.2.5.3 Data ............................................................................................................. 25  
   4.3 Byte Flags .................................................................................................................... 26  
   4.4 The Debugger ............................................................................................................. 27  
      4.4.1 The debugger_t Struct ..................................................................................... 28  
      4.4.2 Registers ............................................................................................................. 28  
      4.4.3 Breakpoints ........................................................................................................ 29  
      4.4.4 Tracing ................................................................................................................ 30  
      4.4.5 Processes and Threads ....................................................................................... 32  
   4.5 Event Notifications ..................................................................................................... 32  
      4.5.1 Receiving Notification ....................................................................................... 33  
      4.5.2 UI Event Notifications ....................................................................................... 34  
      4.5.3 Low Level Events ............................................................................................. 35  
      4.5.3.1 High Level Events ...................................................................................... 36  
      4.5.3.2 Function Result Notifications ...................................................................... 37  
   4.6 Strings ........................................................................................................................ 38
5. Functions

5.1 Common Function Replacements

5.2 Messaging

5.3 UI Navigation

5.4 Entry Points

5.5 Areas

5.6 Segments

5.7 Functions

5.8 Instructions

5.9 Cross Referencing

5.10 Names

5.10.1 get_name

5.10.2 get_name_ea

5.10.3 get_name_value
1. Introduction

1.1 Why This Tutorial?

After spending a lot of time going through the header files in the IDA SDK as well as looking at the source to other people's plug-ins, I figured there should be an easier way to get started with writing IDA plug-ins. Although the header file commentary is amazingly thorough, I found it a little difficult navigating and finding things when I needed them without a lot of searching and trial-and-error. I thought that I'd write this tutorial to try and help those getting started as well as hopefully provide a quick reference point for people developing plug-ins. I've also dedicated a section to setting up a development environment which should make the development process quicker to get into.

1.2 What's Covered

This tutorial will get you started with writing IDA plug-ins, beginning with an introduction to the SDK, followed by setting up a development/build environment on various platforms. You'll then gain a good understanding of how various classes and structures are used, followed by usage of some of the more widely used functions exported. Finally, I'll show some examples of using the IDA API for basic things like looping through functions, to hooking into the debugger and manipulating the IDA database (IDB). After reading this, you should be able to apply the knowledge gained to write your own plug-ins and hopefully share them with the community.

1.3 What's Not Covered

I'm focusing on x86 assembly because it's what I have most experience in, although most of the material presented should cover any architecture supported by IDA (which is practically all of them in the Advanced version). Also, if you want a comprehensive reference to all IDA functions, I suggest looking through the header files.

This tutorial is focused more on "read only" functionality within the SDK, rather than functions for adding comments, correcting errors, defining data structures, and so on. These sorts of things are a big part of the SDK, but aren't covered here in an attempt to keep this tutorial at a manageable size.

I have intentionally left out netnodes from this tutorial, as well as many struct/class members because the IDA SDK is massive, and contains a lot of things for specialised purposes – a tutorial cannot cover everything. If there is something you feel really should be in here, drop me a line and I'll probably include it in the next version if it isn't too specialised.

1.4 Knowledge Required

First and foremost, you must know how to use IDA to the point where you can comfortably navigate disassembled binaries and step through the debugger. You should be equipped with a thorough knowledge of the C/C++ language as well as x86 assembly. C++ knowledge is quite important because the SDK is pretty much all C++. If you don't know C++ but know C, you should at least understand general OOP concepts like classes, objects, methods and inheritance.
1.5 Software Required

To write and run IDA plug-ins, you will need the IDA Pro disassembler 4.8 or 4.9, the IDA SDK (which, as a licensed user of IDA, you get for free from http://www.datarescue.com) and a C/C++ compiler with related tools (Visual Studio, GCC toolset, Borland, etc).

Notes have been added throughout the tutorial where things change in 4.9. Also, as of 4.9, the SDK freezes, and so interfaces to 4.9 functions won't change, and plug-ins written for 4.9 (even in binary form) will work with future versions.

1.6 Alternatives to C/C++

If C is not your thing, take a look at IDAPython, which has all the functionality the C++ API offers in the higher-level language of Python. Check out http://d-dome.net/idapython/ for details. There is a tutorial written on using IDAPython by Ero Carrera at http://dkbza.org/idapython_intro.html, which is obviously more more applicable than this text.

There was also an article recently written about using VB6 and C# to write IDA plugins – check it out here: http://www.openrce.org/articles/full_view/13.

1.7 About This Document

If you have any comments, suggestions or if you notice any errors, please contact me, Steve Micallef, at steve@binarypool.com. If you really feel like you've learnt something from this, I'd also appreciate an email, just to make this process worth while :-)

As the SDK continues to grow, this document will be updated gradually over time. You will always be able to obtain the latest copy at http://www.binarypool.com/idapluginwriting/.

1.8 Credits

In no particular order, I'd like to thank the following people for proof reading as well as providing encouragement and feedback for this tutorial.


1.9 Further Reading

At the time of writing, the only other written material on IDA plug-ins is a tutorial on using the universal un-packer plug-in in IDA 4.9, which contains information on how it was written and how it works. It can be found at http://www.datarescue.com/idabase/unpack_pe/unpacking.pdf. If you get stuck while writing a plug-in, you can always ask for help on the Datarescue Bulletin Board (http://www.datarescue.com/cgi-local/ultimatebb.cgi), where even though the SDK is officially unsupported, someone from Datarescue (or one of the many IDA users) is likely to help you out.

Another great resource is http://www.openrce.org/, where you'll find not only some great articles on reverse engineering, but tools, plug-ins and documentation too. There are also a lot of switched-on people on this board, who will most likely be able to help you with almost any IDA or general reverse engineering problem.
2. The IDA SDK

IDA is a fantastic disassembler and more recently comes with a variety of debuggers too. While IDA alone has an amazing amount of functionality, there are always things you'll want to automate or do in some particular way that IDA doesn't support. Thankfully, the guys at Datarescue have released the IDA SDK – a way for you to hook your own desired functionality into IDA.

There are four types of modules you can write for IDA using the IDA SDK, plug-in modules being the subject of this tutorial:

<table>
<thead>
<tr>
<th>Module Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Adding support for different processor architectures. Also known as IDP (IDA Processor) modules.</td>
</tr>
<tr>
<td>Plug-in</td>
<td>Extending functionality in IDA.</td>
</tr>
<tr>
<td>Loader</td>
<td>Adding support for different executable file formats.</td>
</tr>
<tr>
<td>Debugger</td>
<td>Adding support for debugging on different platforms and/or interacting with other debuggers / remote debugging.</td>
</tr>
</tbody>
</table>

From here onwards, the term "plug-in" will be used in place of "plug-in module", unless otherwise stated.

The IDA SDK contains all the header and library files you need to write an IDA plug-in. It supports a number of compilers on both Linux and Windows platforms, and also comes with an example plug-in that illustrates a couple of basic features available.

Whether you're a reverse engineer, vulnerability researcher, malware analyst, or a combination of them, the SDK gives you a tremendous amount of power and flexibility. You could essentially write your own debugger/disassembler using it, and that's just scratching the surface. Here's a tiny sample of some very straight-forward things you could do with the SDK:

➢ Automate the analysis and unpacking of packed binaries.
➢ Automate the process of finding the use of particular functions (for example, LoadLibrary(), strcpy(), and whatever else you can think of.)
➢ Analyse program and/or data flow, looking for things of interest to you.
➢ Binary diff'ing.
➢ Write a de-compiler.
➢ The list goes on..

To see a sample of what some people have written using the IDA SDK, check out the IDA Palace website, at http://home.arcor.de/idapalace/.

2.1 Installation

This is simple. Once you obtain the SDK (which should be in the form of a .zip file), unzip it to a location of your choice. My preference is creating an sdk directory under the IDA installation and putting everything in there, but it doesn't really matter.
2.2 Directory Layout

Rather than go through every directory and file in the SDK, I'm going to go over the directories relevant to writing plug-ins, and what's in them.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Some makefiles for different environments as well as the readme.txt which you should read to get a quick overview of the SDK, in particular anything that might've changed in recent versions.</td>
</tr>
<tr>
<td>include/</td>
<td>Header files, grouped into areas of functionality. I recommend going through every one of these files and jotting down functions that look applicable to your needs once you have gone through this tutorial.</td>
</tr>
<tr>
<td>libbor.wXX/</td>
<td>IDA library to link against when compiling with the Borland C compiler</td>
</tr>
<tr>
<td>libgccXX.lnx/</td>
<td>IDA library to link against when compiling with GCC under Linux</td>
</tr>
<tr>
<td>libgcc.wXX/</td>
<td>IDA library to link against when compiling with GCC under Windows</td>
</tr>
<tr>
<td>libvc.wXX/</td>
<td>IDA library to link against when compiling with Visual C++ under Windows</td>
</tr>
<tr>
<td>plugins/</td>
<td>Sample plug-ins</td>
</tr>
</tbody>
</table>

xx is either 32(bit) or 64(bit), which will depend on the architecture you’re running on.

2.3 Header Files

Of the fifty header files in the include directory, I found the following to be most relevant when writing plug-ins. If you want information on all the headers, look at readme.txt in the SDK root directory, or in the header file itself. This listing is just here to provide a quick reference point when looking for certain functionality – more detail will be revealed in the following sections.

<table>
<thead>
<tr>
<th>File(s)</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>area.hpp</td>
<td>area_t and areacb_t classes, which represent “areas” of code, which will be covered in detail later on</td>
</tr>
<tr>
<td>bytes.hpp</td>
<td>Functions and definitions for dealing with individual bytes within a disassembled file</td>
</tr>
<tr>
<td>dbg.hpp &amp; idd.hpp</td>
<td>Debugger classes and functions</td>
</tr>
<tr>
<td>diskio.hpp &amp; fpro.h</td>
<td>IDA equivalents to fopen(), open(), etc. as well as some misc. file operations (getting free disk space, current working directory, etc.)</td>
</tr>
<tr>
<td>entry.hpp</td>
<td>Functions for getting and manipulating executable entry point information</td>
</tr>
<tr>
<td>frame.hpp</td>
<td>Functions for dealing with the stack, function frames, local variables and labels</td>
</tr>
<tr>
<td>func.s.hpp</td>
<td>func_t class and pretty much everything function related</td>
</tr>
<tr>
<td>ida.hpp</td>
<td>idainfo struct, which holds mostly meta information about the file being disassembled</td>
</tr>
<tr>
<td>File(s)</td>
<td>Contains</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>kernwin.hpp</td>
<td>Functions and classes for interacting with the IDA user interface</td>
</tr>
<tr>
<td>lines.hpp</td>
<td>Functions and definitions that deal with disassembled text, colour</td>
</tr>
<tr>
<td></td>
<td>coding, etc.</td>
</tr>
<tr>
<td>loader.hpp</td>
<td>Mostly functions for loading files into and manipulating the IDB</td>
</tr>
<tr>
<td>name.hpp</td>
<td>Functions and definitions for getting and setting names of bytes</td>
</tr>
<tr>
<td></td>
<td>(variable names, function names, etc.)</td>
</tr>
<tr>
<td>pro.h</td>
<td>Contains a whole range of misc. definitions and functions</td>
</tr>
<tr>
<td>search.hpp</td>
<td>Various functions and definitions for searching the disassembled</td>
</tr>
<tr>
<td></td>
<td>file for text, data, code and more.</td>
</tr>
<tr>
<td>segment.hpp</td>
<td>segment_t class and everything for dealing with binary segments/sections</td>
</tr>
<tr>
<td>strlist.hpp</td>
<td>string_info_t structure and related functions for representing each</td>
</tr>
<tr>
<td></td>
<td>string in IDA's string list.</td>
</tr>
<tr>
<td>ua.hpp</td>
<td>insn_t, op_t and opype_t classes representing instructions,</td>
</tr>
<tr>
<td></td>
<td>operands and operand types respectively as well as functions for</td>
</tr>
<tr>
<td></td>
<td>working with the IDA analyser</td>
</tr>
<tr>
<td>xref.hpp</td>
<td>Functions for dealing with cross referencing code and data references</td>
</tr>
</tbody>
</table>

### 2.4 Using the SDK

Generally speaking, any function within a header file that's prefixed with `ida_export` is available for your use, as well as global variables prefixed with `ida_export_data`. The rule of thumb is to stay away from lower level functions (these are indicated in the header files) and stick to using the higher level interfaces provided. Any defined class, struct and enum is available for your use.
3. Setting Up a Build Environment

**Note for Borland users:** The only compiler supported by the IDA SDK that isn't covered in this section is Borland's. You should read the `install_cb.txt` and `makeenv_br.mak` in the root of the SDK directory to determine the compiler and linker flags necessary.

Before you start coding away it's best to have a proper environment set up to facilitate the development process. The more popular environments have been covered, so apologies if yours isn't. If you're already set up, feel free to skip to the next section.

3.1 Windows, Using Visual Studio

The version of Visual Studio used for this example is Visual Studio.NET 2003, but almost everything should be applicable to later and even some earlier versions.

Once you have Visual Studio running, close any other solutions and/or projects you might have open; we want a totally clean slate.

1. Go to **File→New→Project...** (Ctrl-Shift-N)

2. Expand the **Visual C++ Projects** folder, followed by the Win32 sub-folder, and then select the **Win32 Project** icon. Name the project whatever you like and click **OK**.

3. The Win32 Application Wizard should then appear, click the **Application Settings** tab and make sure **Windows Application** is selected, and then tick the **Empty Project** checkbox. Click **Finish**.

4. In the **Solutions Explorer** on the right hand side, right click on the **Source Files** folder and go to **Add→Add New Item...**

5. Select the **C++ File (.cpp)** icon and name the file appropriately. Click **Open**. Repeat this step for any other files you want to add to the project.

6. Go to **Project→projectname Properties...**

7. Change the following settings (some have been put there to reduce the size of the resulting plug-in, as VS seems to bloat the output file massively):

   - **Configuration Properties→General:** Change **Configuration Type** to Dynamic Library (.dll)
   - **C/C++→General:** Set **Detect 64-bit Portability Issue** checks to **No**
   - **C/C++→General:** Set **Debug Information Format** to **Disabled**
   - **C/C++→General:** Add the SDK include path to the **Additional Include Directories** field. e.g. **C:\IDA\SDK\Include**
   - **C/C++→Preprocessor:** Add **__NT__;__IDP__;** to **Preprocessor Definitions**
   - **C/C++→Code Generation:** Turn off **Buffer Security Check**, and **Basic Runtime Checks**, set **Runtime Library** to **Single Threaded**
   - **C/C++→Advanced:** Calling **Convention** is **stdcall**
   - **Linker→General:** Change **Output File** from a .exe to a .plw in the IDA plugins directory
   - **Linker→General:** Add the path to your **libvc.wXX** to **Additional Library Directories.** e.g. **C:\IDA\SDK\libvc.w32**
   - **Linker→Input:** Add **ida.lib** to **Additional Dependencies**
   - **Linker→Debugging:** No to **Generate Debug Info**
   - **Linker→Command Line:** Add **/EXPORT:PLUGIN**
3.2 Windows, Using Dev-C++ With GCC and MinGW

You can obtain a copy of Dev-C++, GCC and MinGW as one package from http://www.bloodshed.net/dev/devcpp.html. Installing and setting it up is beyond the scope of this tutorial, so from here on, it'll be assumed that it's all in working order.

As before, start up Dev-C++ and ensure no project or other files are open – we want a clean slate.

1. Go to File->New Project, choose Empty Project, make sure C++ Project is selected and give it any name you wish, click OK
2. Choose a directory to save the project file, this can be anywhere you wish.
3. Go to Project->New File, this will hold the source code to your plug-in. Repeat this step for any other files you want to add to the project.
4. Go to Project->Project Options, click on the Parameters tab.

5. Under C++ compiler, add:
   -DWIN32 -D_NT -D_IDP -v -mrtd

6. Under Linker, add:
   ../path/to/your/sdk/libgcc.wXX/ida.a -Wl,--dll -shared
   Just a note here - it's usually best to start with ../, because msys seems to get confused with just /, and tries to reference it from the root of the msys directory.

7. Click on the Directories tab, and Include Directories sub-tab. Add the path to your IDA SDK include directory to the list.
8. Click on the Build Options tab, set the output directory to your IDA plugins directory, and Override the output filename to be a .plw file. Click OK.

9. Move on to section 3.4

3.3 Linux, Using GCC

Unlike Windows plug-ins, which end in .plw, Linux plug-ins need to end in .plx. Also, in this example, there is no GUI IDE, so rather than go through a step-by-step process, I'll just show the Makefile you need to use. The below example probably isn't the cleanest Makefile, but it should work.

In this example, the IDA installation is in /usr/local/idaadv, and the SDK is located under the sdk sub-directory. Put the below Makefile into the same directory where the source to your
plug-in will be. You'll also need to copy the plugin.script file from the sdk/plugins directory into the directory with your source and Makefile.

Set SRC below to the source files that make up your plug-in, and OBJS to the object files they will be compiled to (same filename, just replace the extension with a .o).

SRC=file1.cpp file2.cpp
OBJS=file1.o file2.o
CC=g++
LD=g++
CFLAGS=-D__IDP__ -D_PLUGIN__ -c -D_LINUX__ \
   -I/usr/local/idaadv/sdk/include $(SRC)
LDFLAGS=--shared $(OBJS) -L/usr/local/idaadv -lida \
   --no-undefined -Wl,--version-script=./plugin.script

all:
   $(CC) $(CFLAGS)
   $(LD) $(LDFLAGS) -o myplugin.plx
   cp myplugin.plx /usr/local/idaadv/plugins

To compile your plug-in, make will do the job and copy it into the IDA plugins directory for you.

3.4 A Plug-in Template

The way IDA "hooks in" to your plug-in is via the PLUGIN class, and is typically the only thing exported by your plug-in (so that IDA can use it). Also, the only files you need to #include that are essential for the most basic plug-in are ida.hpp, idp.hpp and loader.hpp.

The below template should serve as a starter for all your plug-in writing needs. If you paste it into a file in your respective development environment, it should compile, and when run in IDA (Edit->Plugins->pluginname, or the shortcut defined), it will insert the text "Hello World" into the IDA Log window.

#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>

int IDAP_init(void)
{
   // Do checks here to ensure your plug-in is being used within
   // an environment it was written for. Return PLUGIN_SKIP if the
   // checks fail, otherwise return PLUGIN_KEEP.
}

void IDAP_term(void)
{
   // Stuff to do when exiting, generally you'd put any sort
   // of clean-up jobs here.
   return;
}

// The plugin can be passed an integer argument from the plugins.cfg
// file. This can be useful when you want the one plug-in to do
// something different depending on the hot-key pressed or menu
// item selected.
void IDAP_run(int arg)
{
    // The "meat" of your plug-in
    msg("Hello world!");
    return;
}

// There isn't much use for these yet, but I set them anyway.
char IDAP_comment[] = "This is my test plug-in";
char IDAP_help[] = "My plugin";

// The name of the plug-in displayed in the Edit->Plugins menu. It can
// be overridden in the user's plugins.cfg file.
char IDAP_name[] = "My plugin";

// The hot-key the user can use to run your plug-in.
char IDAP_hotkey[] = "Alt-X";

// The all-important exported PLUGIN object
plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION, // IDA version plug-in is written for
    0, // Flags (see below)
    IDAP_init, // Initialisation function
    IDAP_term, // Clean-up function
    IDAP_run, // Main plug-in body
    IDAP_comment, // Comment - unused
    IDAP_help, // As above - unused
    IDAP_name, // Plug-in name shown in
               // Edit->Plugins menu
    IDAP_hotkey // Hot key to run the plug-in
};

You can usually get away without setting the flags attribute (second from the top) in the PLUGIN
structure unless it's a debugger module, or you want to do something like hide it from the Edit->Plugins menu. See loader.hpp for more information on the possible flags you can set.

The above template is also available at http://www.binarypool.com/idapluginwriting/template.cpp.

3.5 Configuring and Running Plug-ins

This is the easiest of all – copy the compiled plug-in file (make sure it ends in .plw for Windows
or .plx for Linux) into the IDA plugins directory and IDA will load it automatically at start-up.

Make sure your plug-in can load up all of its DLLs and shared libraries at start-up by ensuring
your environment is set up correctly (LD_LIBRARY_PATH under Linux, for example). You can
start IDA with the -z20 flag, which will enable plug-in debugging. This will usually indicate if there
are errors during the loading process.

If you put code into the IDAP_init() function, it will get executed when IDA is loading the first
file for disassembly, otherwise, if you put code in the IDAP_run() function, it will execute when
the user presses the hot-key combination or goes through the Edit->Plugins menu.
The user can override a few of the PLUGIN settings in the plugins.cfg file (like the name and hot-key), but that's nothing for you to really concern yourself with. The plugins.cfg file can also be used to pass arguments to your plug-in at start-up.
4. Fundamentals

There are quite a few different classes, data structures and types within the IDA SDK, some more widely used than others. The aim of this section is to introduce you to them, as they provide great insight into what IDA knows about a disassembled file, and should get you thinking about the possibilities of what can be done with the SDK.

Some of these classes and structures are quite large, with many member variables and methods/functions. In this section, it's mostly the variables that are covered, whereas the methods are covered in Chapter 5 - Functions. Some of the below code commentary is taken straight from the SDK, some is my commentary, and some is a combination of the two. #defines have, in some cases, been included beneath various members, the same way as it's been done in the SDK. I left these in because it's a good illustration of the valid values a member variable can have.

**Important note about the examples:** Code from any of the examples in this section should be put into the IDAP_run() function from the template in section 3.4, unless otherwise stated.

### 4.1 Core Types

The following types are used all throughout the SDK and this tutorial, so it's important that you are able to recognise what they represent.

All the below types are unsigned long integers, and unsigned long long integers on 64-bit systems. They are defined in pro.h.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ea_t</td>
<td>Stands for 'Effective Address', and represents pretty much any address within IDA (memory, file, limits, etc.)</td>
</tr>
<tr>
<td>sel_t</td>
<td>Segment selectors, as in code, stack and data segment selectors</td>
</tr>
<tr>
<td>uval_t</td>
<td>Used for representing unsigned values</td>
</tr>
<tr>
<td>asize_t</td>
<td>Typically used for representing the size of something, usually a chunk of memory</td>
</tr>
</tbody>
</table>

The following are signed long integers, and signed long long integers on 64-bit systems. They are also defined in pro.h.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sval_t</td>
<td>Used for representing signed values</td>
</tr>
<tr>
<td>adiff_t</td>
<td>Represents the difference between two addresses</td>
</tr>
</tbody>
</table>

Finally, there are a couple of definitions worth noting; one of these is BADADDR, which represents an invalid or non-existent address which you will see used a lot in loops for detecting the end of a readable address range or structure. You will also see MAXSTR used in character buffer definitions, which is 1024.
4.2 Core Structures and Classes

4.2.1 Meta Information

The idainfo struct, which is physically stored in the IDA database (IDB), holds what I refer to as 'meta' information about the initial file loaded for disassembly in IDA. It does not change if more files are loaded, however. Here are some of the more interesting parts of it, as defined in ida.hpp:

```c
struct idainfo {
  char          procName[8];  // Name of processor IDA is running on
  ushort        filetype;     // The input file type. See the
                              // filetype_t enum - could be f_ELF,
                              // f_PE, etc.
  ea_t          startSP;      // [E]SP register value at the start of
                              // program execution
  ea_t          startIP;      // [E]IP register value at the start of
                              // program execution
  ea_t          beginEA;      // Linear address of program entry point,
                              // usually the same as startIP
  ea_t          minEA;        // First linear address within program
  ea_t          maxEA;        // Last linear address within the
                              // program, excluding maxEA
};
```

inf is a globally accessible instance of this structure. You will often see checks done against inf.procName within the initialisation function of a plug-in, checking that the machine architecture is what the plug-in was written to handle.

For example, if you wrote a plug-in to only handle PE and ELF binary formats for the x86 architecture, you could add the following statement to your plug-in's init function (IDAP_init from our plug-in template in section 3.4).

```c
// "metapc" represents x86 architecture
if (strncmp(inf.procName, "metapc", 8) != 0
    || inf.filetype != f_ELF && inf.filetype != f_PE))
{
  error("Only PE and ELF binary type compiled for the x86 "
        "platform is supported, sorry.");
  return PLUGIN_SKIP; // Returning PLUGIN_SKIP means this plug-in
                     // won't be loaded
}
return PLUGIN_KEEP; // Keep this plug-in loaded
```

4.2.2 Areas

Before going into detail on the “higher level” classes for working with segments, functions and instructions, let's have a look at two key concepts; namely areas and area control blocks.
4.2.2.1 The area_t Structure
An area is represented by the area_t struct, as defined in area.hpp. Based on commentary in this file, strictly speaking:

"Areas" consists of separate area_t instances. An area is a non-empty contiguous range of addresses (specified by its start and end addresses, end address is excluded) with characteristics. For example, segments are set of areas.

As you can see from the below excerpt from the area_t definition, it is defined by a start address (startEA) and end address (endEA). There are also a couple of functions to see if an area contains an address, if an area is empty, and to return the size of the area. A segment is an area, but functions are too, which means areas can also encompass other areas.

struct area_t
{
  ...  
  ea_t startEA;
  ea_t endEA;              // endEA address is excluded from
                           // the area
  bool contains(ea_t ea) const { return startEA <= ea && endEA > ea; }
  bool empty(void) const { return startEA >= endEA; }
  asize_t size(void) const { return endEA - startEA; }
  ...  
};

Technically speaking, saying that functions and segments are areas, is to say that the func_t and segment_t classes inherit from the area_t struct. This means that all the variables and functions in the area_t structure are applicable to func_t and segment_t (so for example, segment_t.startEA and func_t.contains() are valid). func_t and segment_t also extend the area_t struct with their own specialized variables and functions. These will be covered later however.

A few other classes that inherit from and extend area_t are as follows:

<table>
<thead>
<tr>
<th>Type (file)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hidden_area_t (bytes.hpp)</td>
<td>Hidden areas where code/data is replaced and summarised by a description that can be expanded to view the hidden information</td>
</tr>
<tr>
<td>regvar_t (frame.hpp)</td>
<td>Register name replacement with user-defined names (register variables)</td>
</tr>
<tr>
<td>memory_info_t (idd.hpp)</td>
<td>A chunk of memory (when using the debugger)</td>
</tr>
<tr>
<td>segreg_t (srarea.hpp)</td>
<td>Segment register (CS, SS, etc. on x86) information</td>
</tr>
</tbody>
</table>

4.2.2.2 The areacb_t Class
An area control block is represented by the areacb_t class, also defined in area.hpp. The commentary for it, shown below, is slightly less descriptive, but doesn't really need to be anyway:

"areacb_t" is a base class used by many parts of IDA

The area control block class is simply a collection of functions that are used to operate on areas. Functions include get_area_qty(), get_next_area() and so on. You probably won't find yourself using any of these methods directly, as when dealing with functions for example, you're
more likely to use func_t’s methods, and the same rule applies to other classes that inherit from area_t.

There are two global instances of the areacb_t class, namely segs (defined in segment.hpp) and funcs (defined in funcs.hpp), which represent all segments and functions, respectively, within the currently disassembled file(s). You can run the following to get the number of segments and functions within the currently disassembled file(s) in IDA:

```c
#include <segment.hpp>
#include <funcs.hpp>

msg("Segments: %d, Functions: %d\n",
    segs.get_area_qty(),
    funcs.get_area_qty());
```

### 4.2.3 Segments and Functions

As mentioned previously, the segment_t and func_t classes both inherit from and extend the area_t struct, which means all the area_t variables and functions are applicable to these classes and they also bring some of their own functionality into the mix.

#### 4.2.3.1 Segments

The segment_t class is defined in segment.hpp. Here are the more interesting parts of it.

```c
class segment_t : public area_t
{
public:
    uchar perm;              // Segment permissions (0-no information). Will
                           // be one or a combination of the below.

#define SEGPERM_EXEC  1 // Execute
#define SEGPERM_WRITE 2 // Write
#define SEGPERM_READ  4 // Read

    uchar type;        // Type of the segment. This will be one of the below.
#define SEG_NORM        0       // Unknown type, no assumptions
#define SEG_XTRN        1       // Segment with 'extern' definitions,
                           // where no instructions are allowed
#define SEG_CODE        2       // Code segment
#define SEG_DATA        3       // Data segment
#define SEG_NULL        7       // Zero-length segment
#define SEG_BSS         9       // Uninitialized segment
...
};
```

SEG_XTRN is a special (i.e. not physically existent) segment type, created by IDA upon disassembly of a file, whereas others represent physical parts of the loaded file. For a typical executable file loaded in IDA for example, the value of type for the .text segment would be SEG_CODE and the value of perm would be SEGPERM_EXEC | SEGPERM_READ.

To iterate through all the segments within a binary, printing the name and address of each one into IDA's Log window, you could do the following:

```c
#include <segment.hpp>

// This will only work in IDA 4.8, because get_seg_name() changed
```
Understanding what the functions above do isn't important at this stage – they'll be explained in more detail under Chapter 5 - Functions.

4.2.3.2 Functions

A function is represented by the `func_t` class, which is defined in `funcs.hpp`, but before going into detail on the `func_t` class, it's probably worth shedding some light on function chunks, parents and tails.

Functions are typically contiguous blocks of code within the binary being analysed, and are usually represented as a single chunk. However, there are times when optimizing compilers move code around, and so functions are broken up into multiple chunks with code from other functions separating them. These loose chunks are known as "tails", and the chunks that reference code (by a JMP or something similar) within the tails are known as "parents". What makes things a little confusing is that all are still of the `func_t` type, and so you need to check the `flags` member of `func_t` to determine if a `func_t` instance is a tail or parent.

Below is highly stripped-down version of the `func_t` class, along with some slightly edited commentary taken from `funcs.hpp`.

```cpp
class func_t : public area_t
{
public:
...
  ushort flags; // flags indicating the type of function
  ... // Some of the flags below:
#define FUNC_NORET   0x00000001L    // function doesn't return
#define FUNC_LIB     0x00000004L    // library function
#define FUNC_HIDDEN  0x00000040L    // a hidden function chunk
#define FUNC_THUNK   0x00000080L    // thunk (jump) function
#define FUNC_TAIL    0x00008000L    // This is a function tail.
  // Other bits should be clear
  // (except FUNC_HIDDEN)
union       // func_t either represents an entry chunk or a tail chunk
{
  struct     // attributes of a function entry chunk
  {
    asize_t argsize; // number of bytes purged from the stack
    // upon returning
    ushort pntqty;  // number of times the ESP register changes
    // throughout the function (due to PUSH, etc.)
    int tailqty;   // number of function tails this function owns
    area_t *tails; // array of tails, sorted by ea
```

// in 4.9. See the Chapter 5 for more information.

// get_segm_qty() returns the number of total segments
// for file(s) loaded.
for (int s = 0; s < get_segm_qty(); s++)
{
  // getnseg() returns a segment_t struct for the segment
  // number supplied
  segment_t *curSeg = getnseg(s);
  // get_segm_name() returns the name of a segment
  // msg() prints a message to IDA's Log window
  msg("%s @ %a
", get_segm_name(curSeg), curSeg->startEA);
}
struct  // attributes of a function tail chunk
{
  ea_t owner;  // the address of the main function
    // possessing this tail
...
};

Because functions are also areas just like segments, iterating through each function is a process almost identical to dealing with segments. The following example lists all functions and their address within a disassembled file, displaying output in IDA's Log window.

```c
#include <funcs.hpp>

// get_func_qty() returns the number of functions in file(s) // loaded.
for (int f = 0; f < get_func_qty(); f++)
{
  // getn_func() returns a func_t struct for the function // number supplied
  func_t *curFunc = getn_func(f);
  char funcName[MAXSTR];

  // get_func_name gets the name of a function, // stored in funcName
  get_func_name(curFunc->startEA, funcName,
                sizeof(funcName)-1);
  msg("%s:	%a
", funcName, curFunc->startEA);
}
```

### 4.2.4 Code Representation

Assembly language instructions consist of, in most cases, mnemonics (PUSH, SHR, CALL, etc.) and operands (EAX, [EBP+0xAh], 0x0Fh, etc.) Some operands can take various forms, and some instructions don't even take operands. All of this is represented very cleanly in the IDA SDK.

You have the `insn_t` type to begin with, which represents a whole instruction, for example "MOV EAX, 0x0A". `insn_t` is made up of, amongst other member variables, up to 6 `op_t`'s (one for each operand supplied to the instruction), and each operand can be a particular `optype_t` (general register, immediate value, etc.)

Let's look at each component from the bottom-up. They are all defined in `ua.hpp`.
4.2.4.1 Operand Types

`optype_t` represents the type of operand that is being supplied to an instruction. Here are the more common operand type values. The descriptions have been taken from the `optype_t` definition in `ua.hpp`.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
<th>Example disassembly (respective operand in bold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_void</td>
<td>No Operand</td>
<td>pusha</td>
</tr>
<tr>
<td>o_reg</td>
<td>General Register</td>
<td>dec eax</td>
</tr>
<tr>
<td>o_mem</td>
<td>Direct Memory Reference</td>
<td>mov eax, ds:1001h</td>
</tr>
<tr>
<td>o_phrase</td>
<td>Memory Ref [Base Reg + Index Reg]</td>
<td>push dword ptr [eax]</td>
</tr>
<tr>
<td>o_displ</td>
<td>Memory Ref [Base Reg + Index Reg + Displacement]</td>
<td>push [esp+8]</td>
</tr>
<tr>
<td>o_imm</td>
<td>Immediate Value</td>
<td>add ebx, 10h</td>
</tr>
<tr>
<td>o_near</td>
<td>Immediate Near Address</td>
<td>call _initterm</td>
</tr>
</tbody>
</table>

4.2.4.2 Operands

`op_t` represents a single operand passed to an instruction. Below is a highly cut-down version of the class.

```cpp
class op_t
{
public:
    char n;                     // number/position of the operand (0,1,2)
    optype_t type;              // type of operand (see previous section)
    ushort reg;                 // register number (if type is o_reg)
    uval_t value;               // operand value (if type is o_imm)
    ea_t addr;                  // virtual address pointed to or used by the operand (if type is o_mem)
    ...
};
```

So, for example, the operand of `[esp+8]` will result in type being `o_displ`, `reg` being 4 (which is the number for the ESP register) and `addr` being 8, because you are accessing 8 bytes from the stack pointer, thereby being a memory reference. You can use the following snippet of code for getting the `op_t` value of the first operand of the instruction your cursor is currently positioned at in IDA:

```cpp
#include <kernwin.hpp>
#include <ua.hpp>

// Disassemble the instruction at the cursor position, store it in // the globally accessible 'cmd' structure.
ua_ana0(get_screen_ea());
// Display information about the first operand
msg("n = %d type = %d reg = %d value = %a addr = %a\n",
    cmd.Operands[0].n,
    cmd.Operands[0].type,
    cmd.Operands[0].reg,
    cmd.Operands[0].value,
    cmd.Operands[0].addr);
```
4.2.4.3 Mnemonics

The mnemonic (PUSH, MOV, etc.) within the instruction is represented by the itype member of the insn_t class (see the next section). This is, however, an integer, and there is currently no textual representation of the instruction available to the user in any data structure – instead, it is obtained through use of the ua_mnem() function, which will be covered in Chapter 5 - Functions.

There is an enum, named instruc_t (allins.hpp) that holds all mnemonic identifiers (prefixed with NN_). If you know what instructions you are after or want to test for, you can utilise it rather than work off a text representation. For example, to test if the first instruction in a binary is a PUSH, you could do the following:

```c
#include <ua.hpp>
#include <allins.hpp>

// Populate 'cmd' with the code at the entry point of the binary
ua_ana0(inf.startIP);
// Test if that instruction is a PUSH
if (cmd.itype == NN_push)
    msg("First instruction is a PUSH");
else
    msg("First instruction isn't a PUSH");
return;
```

4.2.4.4 Instructions

insn_t represents a whole instruction. It contains an op_t array, named Operands, which represents all operands passed to the instruction. Obviously there are instructions that take no operands (like PUSHA, CDQ, etc.), in which case the Operands[0] variable will have an opotype_t of o_void (no operand).

class insn_t
{
public:
    ea_t cs;                      // code segment base (in paragraphs)
    ea_t ip;                      // offset within the segment
    ea_t ea;                      // instruction start addresses
    ushort itype;                 // mnemonic identifier
    ushort size;                  // instruction size in bytes
#define UA_MAXOP  6
    op_t Operands[UA_MAXOP];
#define Op1 Operands[0]        // first operand
#define Op2 Operands[1]        // second operand
#define Op3 Operands[2]        // ...
#define Op4 Operands[3]
#define Op5 Operands[4]
#define Op6 Operands[5]
};

There is a globally accessible instance of insn_t named cmd, which gets populated by the ua_ana0() and ua_code() functions. More on this later, but in the mean time, here's an example to get the instruction at a file's entry point and display its instruction number, address and size in IDA's Log window.

```c
#include <ua.hpp>
```
// ua_ana0() populates the cmd structure with a disassembly of the
// address supplied.
ua_ana0(inf.beginEA); // or inf.startIP
msg("instruction number: %d, at %a is %d bytes in size.\n",
    cmd.itype, cmd.ea, cmd.size);

4.2.5 Cross Referencing

One of the handy features in IDA is the cross-referencing functionality, which will tell you about all
parts of the currently disassembled file that reference another part of that file. For instance, in
IDA, you can highlight a function in the disassembly window, press 'x' and all addresses where
that function is referenced (e.g. calls made to the function) will appear in a window. The same can
be done for data and local variables too.

The SDK provides a simple interface for accessing this information, which is stored internally in a
B-tree data structure, accessed via the xrefblk_t structure. There are other, more manual,
ways to retrieve this sort of information, but they are much slower than the methods outlined
below.

One important thing to remember is that even when an instruction naturally flows onto the next,
IDA can potentially treat the first as referencing the second, but this can be turned off using flags
supplied to some xrefblk_t methods, covered in Chapter 5 - Functions.

4.2.5.1 The xrefblk_t Structure

Central to cross referencing functionality is the xrefblk_t structure, which is defined in
xref.hpp. This structure first needs to be populated using its first_from() or first_to()
methods (depending on whether you want to find references to or from an address), and
subsequently populated using next_from() or next_to() as you traverse through the
references.

The variables within this structure are shown below and commentary is mostly from xref.hpp.
The methods (first_from, first_to, next_from and next_to) have been left out, but will
be covered in Chapter 5 - Functions.

struct xrefblk_t
{
    ea_t from;            // the referencing address
    ea_t to;              // the referenced address
    uchar iscode;         // 1-is code reference; 0-is data reference
    uchar type;           // one of the cref_t or dref_t types (see
                           // section 4.2.5.2 and 4.2.5.3)
...                   
};

As indicated by the iscode variable, xrefblk_t can contain information about a code
reference or a data reference, each of which could be one of a few possible reference types, as
indicated by the type variable. These code and data reference types are explained in the
following two sections.

The below code snippet will give you cross reference information about the address your cursor is
currently positioned at:

#include <kernwin.hpp>
#include <xref.hpp>

xrefblk_t xb;
// Get the address of the cursor position
ea_t addr = get_screen_ea();
// Loop through all cross references
for (bool res = xb.first_to(addr, XREF_FAR); res; res = xb.next_to()) {
    msg("From: %a, To: %a\n", xb.from, xb.to);
    msg("Type: %d, IsCode: %d\n", xb.type, xb.icode);
}

4.2.5.2 Code
Here is the cref_t enum, with some irrelevant items taken out. Depending on the type of reference, the type variable in xrefblk_t will be one of the below if iscode is set to 1. The commentary for the below is taken from xref.hpp.

enum cref_t
{
    ...  
    fl_CF = 16,  // Call Far
                // This xref creates a function at the
                // referenced location
    fl_CN,  // Call Near
               // This xref creates a function at the
               // referenced location
    fl_JF,  // Jump Far
    fl_JN,  // Jump Near
    fl_F,   // Ordinary flow: used to specify execution
            // flow to the next instruction.
    ... 
};

Below is a code cross reference taken from a sample binary file. In this case, 712D9BFE is referenced by 712D9BF6, which is a near jump (fl_JN) code reference type.

.text:712D9BF6     jz      short loc_712D9BFE
...
.text:712D9BFE loc_712D9BFE:
    .text:712D9BF6     lea     ecx, [ebp+var_14]

4.2.5.3 Data
If iscode in xrefblk_t is set to 0, it is a data cross reference. Here are the possible type member values when you're dealing with a data cross reference. The commentary for this enum is also taken from xref.hpp.

enum dref_t
{
    ...  
    dr_O,                   // Offset
                    // The reference uses 'offset' of data
                    // rather than its value
                    // OR
                    // The reference appeared because
                    // the "OFFSET" flag of instruction is set.
                    // The meaning of this type is IDP dependent.
    ... 
};
dr_W,       // Write access
dr_R,       // Read access
...

Keep in mind that when you see the following in a disassembly, you are actually looking at a data cross reference, whereby 712D9BD9 is referencing 712C119C:

.idata:712C119C extrn wsprintfA:dword
...
.text:712D9BD9             call    ds:wsprintfA

In this case, the type member of xrefblk_t would be the typical dr_R, because it's simply doing a read of the address represented by ds:wsprintfA. Another data cross reference is below, where the PUSH instruction at 712EABE2 is referencing a string at 712C255C:

.text:712C255C               aVersion:
.text:712C255C               unicode 0, <Version>,0
...
.text:712EABE2               push    offset aVersion

The type member of xrefblk_t would be dr_O in this case, because it's accessing the data as an offset.

4.3 Byte Flags

For each byte in a disassembled file, IDA records a corresponding four byte (32-bits) value, stored in the id1 file. Of these four bytes, each half-byte (four bits or "nibble") is a flag, which represents an item of information about the byte in the disassembled file. The last byte of the four flag bytes is the actual byte at that address within the disassembled file.

For example, the instruction below takes up a single byte (0x55) in the file being disassembled:

.text:010060FA             push    ebp

The IDA flags for the above address in the file being disassembled are 0x00010755; 0001007 being the flag component and 55 being the byte value at that address in the file. Keep in mind that the address has no bearing on the flags at all, nor is it possible to derive flags from the address or bytes themselves - you need to use getFlags() to get the flags for an address (more on this below).

Obviously, not all instructions are one byte in size; take the below instruction for example, which is three bytes (0x83 0xEC 0x14). The instruction is therefore spread across three addresses; 0x010011DE, 0x010011DF and 0x010011E0:

.text:010011DE             sub     esp, 14h
.text:010011E1 ... 

Here are the corresponding flags for each byte in this instruction:

010011DE: 41010783
010011DF: 001003EC
010011E0: 00100314
Because these three bytes belong to the one instruction, the first byte of the instruction is referred to as the head, and the other two are tail bytes. Once again, notice that the last byte of each flag-set is the corresponding byte of the instruction (0x83, 0xEC, 0x14).

All flags are defined in bytes.hpp, and you can check whether a flag is set by using the flagset returned from getFlags(ea_t ea) as the argument to the appropriate flag-checking wrapper function. Here are some common flags along with their wrapper functions which check for their existence. Some functions are covered in Chapter 5 - Functions, for others you should look in bytes.hpp:

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Flag</th>
<th>Indication</th>
<th>Wrapper function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF_CODE</td>
<td>0x00000600L</td>
<td>Is the byte code?</td>
<td>isCode()</td>
</tr>
<tr>
<td>FF_DATA</td>
<td>0x00000400L</td>
<td>Is the byte data?</td>
<td>isData()</td>
</tr>
<tr>
<td>FF_TAIL</td>
<td>0x00000200L</td>
<td>Is this byte a part (non-head) of an instruction data chunk?</td>
<td>isTail()</td>
</tr>
<tr>
<td>FF_UNK</td>
<td>0x00000000L</td>
<td>Was IDA unable to classify this byte?</td>
<td>isUnknown()</td>
</tr>
<tr>
<td>FF_COMM</td>
<td>0x00000800L</td>
<td>Is the byte commented?</td>
<td>has_cmt()</td>
</tr>
<tr>
<td>FF_REF</td>
<td>0x00001000L</td>
<td>Is the byte referenced elsewhere?</td>
<td>hasRef()</td>
</tr>
<tr>
<td>FF_NAME</td>
<td>0x00004000L</td>
<td>Is the byte named?</td>
<td>has_name()</td>
</tr>
<tr>
<td>FF_FLOW</td>
<td>0x00010000L</td>
<td>Does the previous instruction flow here?</td>
<td>isFlow()</td>
</tr>
</tbody>
</table>

Going back to the first “push ebp” example above, if we were to manually check the flags returned from getFlags(0x010060FA) against a couple of the above flags, we'd get the following results:

0x00010755 & 0x00000600 (FF_CODE) = 0x00000600. We know this is code.
0x00010755 & 0x00000800 (FF_COMM) = 0x00000000. We know this isn't commented.

The above example is purely for illustrative purposes - don't do it this way in your plug-in. As mentioned above, you should always use the helper functions to check whether a flag is set or not. The following will return the flags for the given head address your cursor is positioned at in IDA.

```c
#include <bytes.hpp>
#include <kernwin.hpp>

msg("%08x\n", getFlags(get_screen_ea()));
```

### 4.4 The Debugger

One of the most powerful features of the IDA SDK is the ability to interact with the IDA debugger, and unless you've installed your own custom debugger plug-in, it will be one of the debugger plug-ins that came with IDA. The following debugger plug-ins come with IDA by default, and can be found in your IDA plugins directory:

<table>
<thead>
<tr>
<th>Plugin Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>win32_user.plw</td>
<td>Windows local debugger</td>
</tr>
</tbody>
</table>
These are automatically loaded by IDA and made available at start-up under the **Debugger- >Run** menu. From here on, the term "debugger" will represent whichever you are using (IDA will choose the most appropriate one for you by default).

As mentioned earlier, it is possible to write debugger modules for IDA, but this isn't to be confused with writing plug-in modules that interact with the debugger. The second type of plug-in is what's described below.

Aside from all the functions provided for interacting with the debugger, which will be explored later in *Chapter 5 - Functions*, there are some key data structures and classes that are essential to understand before moving ahead.

### 4.4.1 The debugger_t Struct

The debugger_t struct, defined in idd.hpp and exported as *dbg, represents the currently active debugger plug-in, and is available when the debugger is loaded (i.e. at start-up, not just when you run the debugger).

```c
struct debugger_t {

  ...  
  char *name;       // Short debugger name like 'win32' or 'linux'

#define DEBUGGER_ID_X86_IA32_WIN32_USER  0 // Userland win32 processes
#define DEBUGGER_ID_X86_IA32_LINUX_USER  1 // Userland linux processes

  register_info_t *registers;     // Array of registers
  int             registers_size; // Number of registers

  ...
}
```

As a plug-in module, it's likely that you'll need to access the *name variable, possibly to test what debugger your plug-in is running with. The *registers and registers_size variables are also useful for obtaining a list of registers available (see the following section).

### 4.4.2 Registers

A common task while using the debugger is accessing and manipulating register values. In the IDA SDK, a register is described by the register_info_t struct, and the value held by a register is represented by the regval_t struct. Below is a slightly cut-down register_info_t struct, which is defined in idd.hpp.

```c
struct register_info_t {

    const char *name; // Register full name (EBX, etc.)
    ulong flags; // Register special features,
                  // which can be any combination
                  // of the below.

#define REGISTER_READONLY 0x0001 // the user can't modify
#define REGISTER_WRITEONLY 0x0002 // the current value of this
                               // register
                               // register
```
#define REGISTER_IP       0x0002 // instruction pointer
#define REGISTER_SP       0x0004 // stack pointer
#define REGISTER_FP       0x0008 // frame pointer
#define REGISTER_ADDRESS  0x0010 // Register can contain an address
...

The only instance of this structure is accessible as the array member *registers of *dbg (an instance of debugger_t), therefore it is up to the debugger you're using to populate it with the list of registers available on your system.

To obtain the value for any register, it's obviously essential that the debugger be running. The functions for reading and manipulating register values will be covered in more detail in Chapter 5 - Functions, but for now, all you need to know is to retrieve the value using the ival member of regval_t, or use fval if you're dealing with floating point numbers.

Below is regval_t, which is defined in idd.hpp.

```c
#include <dbg.hpp>

struct regval_t
{
    ulonglong ival;     // Integer value
    ushort fval[6];     // Floating point value in the internal
                        // representation (see ieee.h)
};
```

ival/fval will correspond directly to what is stored in a register, so if EBX contains 0xDEADBEEF, ival (once populated using get_reg_val()), will also contain 0xDEADBEEF.

The following example will loop through all available registers, displaying the value in each. If you run this outside of debug mode, the value will be 0xFFFFFFFF:

```c
#include <dbg.hpp>

// Loop through all registers
for (int i = 0; i < dbg->registers_size; i++) {
    regval_t val;
    // Get the value stored in the register
    get_reg_val((dbg->registers+i)->name, &val);
    msg("%s: %08a\n", (dbg->registers+i)->name, val.ival);
}
```

### 4.4.3 Breakpoints

A fundamental component of debugging is breakpoints, and IDA represents hardware and software breakpoints differently using the bpt_t struct, shown below and defined in dbg.hpp. Hardware breakpoints are created using debug-specific registers on the running CPU (DR0-DR3 on x86), whereas software breakpoints are created by inserting an INT3 instruction at the desired breakpoint address - although this is handled for you by IDA, it's sometimes helpful to know the difference. On x86, the maximum number of hardware breakpoints you can set is four.

```c
#include <dbg.hpp>

// Loop through all registers
for (int i = 0; i < dbg->registers_size; i++) {
    regval_t val;
    // Get the value stored in the register
    get_reg_val((dbg->registers+i)->name, &val);
    msg("%s: %08a\n", (dbg->registers+i)->name, val.ival);
}
```
bpttype_t type; // type of the breakpoint:
// Taken from the bpttype_t const definition in idd.hpp:
// BPT_EXEC  =  0,             // Execute instruction
// BPT_WRITE =  1,             // Write access
// BPT_RDWR  =  3,             // Read/write access
// BPT_SOFT  =  4;             // Software breakpoint
// modifiable characteristics (use update_bpt() to modify):
int pass_count; // how many times does the execution reach
                // this breakpoint? (-1 if undefined)
int flags;
#define BPT_BRK   0x01    // does the debugger stop on this breakpoint?
#define BPT_TRACE 0x02    // does the debugger add trace information
char condition[MAXSTR]; // an IDC expression which will be used as
                        // a breakpoint condition or run when the
                        // breakpoint is hit

Therefore, if the type member of bpt_t is set to 0, 1 or 3, it is a hardware breakpoint, whereas 4 would indicate a software breakpoint.

There are a lot of functions that create, manipulate and read this struct, but for now, I'll provide a simple example that goes through all defined breakpoints and display whether they are a software or hardware breakpoint in IDA's Log window. The functions used will be explained in more detail further on.

#include <dbg.hpp>

// get_bpt_qty() gets the number of breakpoints defined
for (int i = 0; i < get_bpt_qty(); i++) {
    bpt_t brkpnt;
    // getn_bpt fills bpt_t struct with breakpoint information based
    // on the breakpoint number supplied.
    getn_bpt(i, &brkpnt);
    // BPT_SOFT is a software breakpoint
    if (brkpnt.type == BPT_SOFT)
        msg("Software breakpoint found at %a\n", brkpnt.ea);
    else
        msg("Hardware breakpoint found at %a\n", brkpnt.ea);
}

4.4.4 Tracing

In IDA, there are three types of tracing you can enable: Function tracing, Instruction tracing and Breakpoint (otherwise known as read/write/execute) tracing. When writing plug-ins, an additional form of tracing is available; Step tracing. Step tracing is a low level form of tracing that allows you to build your own tracing mechanism on top of it, utilising event notifications (see section 4.5) to inform your plug-in of each instruction that is executed. This is based on CPU tracing functionality, not breakpoints.

A "trace event" is generated and stored in a buffer when a trace occurs, and what triggers the generation of a trace event depends on the type of tracing you have enabled, however it's worth noting that step tracing will not generate trace events, but event notifications instead. The below table lists all the different trace event types along with the corresponding tev_type_t enum value, which is defined in dbg.hpp.
<table>
<thead>
<tr>
<th>Trace Type</th>
<th>Event Type (tev_type_t)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function call and return</td>
<td>tev_call and tev_ret</td>
<td>A function has been called or returned from</td>
</tr>
<tr>
<td>Instruction</td>
<td>tev_insn</td>
<td>An instruction has been executed (this is built on top of step tracing in the IDA kernel)</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>tev_bpt</td>
<td>A breakpoint with tracing enabled has been hit. Also known as a Read/Write/Execute trace</td>
</tr>
</tbody>
</table>

All trace events are stored in a circular buffer, so it never fills up, but old trace events will be overwritten if the buffer is too small. Each trace event is represented by the tev_info_t struct, which is defined in dbg.hpp:

```c
struct tev_info_t
{
    tev_type_t  type; // Trace event type (one of the above or tev_none)
    thread_id_t tid;  // Thread where the event was recorded
    ea_t        ea;   // Address where the event occurred
};
```

Based on the bpt_t struct described in section 4.4.3, a breakpoint trace is the same as a normal breakpoint but has the BPT TRACE flag set on the flags member. Optionally, the condition buffer member could have an IDC command to run at each breakpoint.

Trace information is populated during the execution of a process, but can be accessed even once the process has exited and you are returned to static disassembly mode (unless a plug-in you are using explicitly cleared the buffer on exit). You can use the following code to enumerate all trace events (provided you enabled it during execution):

```c
#include <dbg.hpp>

// Loop through all trace events
for (int i = 0; i < get_tev_qty(); i++) {
    regval_t esp;
    tev_info_t tev;

    // Get the trace event information
    get_tev_info(i, &tev);

    switch (tev.type) {
    case tev_ret:
        msg("Function return at %a\n", tev.ea);
        break;
    case tev_call:
        msg("Function called at %a\n", tev.ea);
        break;
    case tev_insn:
        msg("Instruction executed at %a\n", tev.ea);
        break;
    case tev_bpt:
        msg("Breakpoint with tracing hit at %a\n", tev.ea);
        break;
    }
}
```
It's worth noting at this point that it's not possible for a plug-in to add entries to, or even modify the trace event log.

All of the functions used above will be covered in Chapter 5 - Functions.

### 4.4.5 Processes and Threads

IDA maintains information about the processes and threads currently running under the debugger. Process and Thread IDs are represented by the `process_id_t` and `thread_id_t` types, respectively and both are signed integers. All of these types are defined in `idd.hpp`. The only other type, related to processes, is the `process_info_t` type, which is as follows:

```c
struct process_info_t
{
    process_id_t pid;    // Process ID
    char name[MAXSTR];   // Process Name (executable file name)
};
```

These are only of use when a binary is being executed under IDA (i.e. you can't use them when in static disassembly mode). The following example illustrates a basic example usage of the `process_info_t` structure.

```c
#include <dbg.hpp>

// Get the number of processes available for debugging.
// get_process_qty() also initialises IDA's "process snapshot"
if (get_process_qty() > 0) {
    process_info_t pif;
    get_process_info(0, &pif);
    msg("ID: %d, Name: %s\n", pif.pid, pif.name);
} else {
    msg("No process running!\n");
}
```

The functions that utilise these structures will be discussed under Chapter 5 - Functions.

### 4.5 Event Notifications

Typically, plug-ins are run synchronously, in that they are executed by the user, either via pressing the hot-key or going through the Edit->Plugins menu. A plug-in can, however, run asynchronously, where it responds to event notifications generated by IDA or the user.

During the course of working in IDA, you'd typically click buttons, conduct searches, and so on. All of these actions are "events", and so what IDA does is generate "event notifications" each time these things take place. If your plug-in is setup to receive these notifications (explained below), it can react in any way you program it to. An application for this sort of thing could be recording macros for instance. A plug-in can also generate events, causing IDA to perform various functions.
4.5.1 Receiving Notification

To receive event notifications from IDA, all a plug-in has to do is register a call-back function using hook_to_notification_point(). For generating event notifications, callui() is used, which is covered in more detail in Chapter 5 - Functions.

When registering a call-back function with hook_to_notification_point(), you can specify one of three event types, depending on what notifications you want to receive. These are defined in the hook_type_t enum within loader.hpp:

<table>
<thead>
<tr>
<th>Type</th>
<th>Receive Event Notifications From</th>
<th>Enum of All Event Notification Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>HT_IDP</td>
<td>Processor module</td>
<td>idp_notify (not covered here)</td>
</tr>
<tr>
<td>HT_UI</td>
<td>IDA user interface</td>
<td>ui_notification_t</td>
</tr>
<tr>
<td>HT_DBG</td>
<td>Currently running IDA debugger</td>
<td>dbg_notification_t</td>
</tr>
</tbody>
</table>

Therefore, to receive all event notifications pertaining to the debugger and direct them to your dbg_callback (for example) call-back function, you could put the following inside IDAP_init():

```
hook_to_notification_point(HT_DBG, dbg_callback, NULL);
```

The third argument is typically NULL, unless you want to pass data along to the call-back function when it receives an event (any data structure of your choosing).

The call-back function supplied to hook_to_notification_point() must look something like this:

```
int idaapi mycallback (void *user_data, int notif_code, va_list va)
{
    ...
    return 0;
}
```

When mycallback() is eventually called by IDA to handle an event notification, user_data will point to any data you specified to have passed along to the call-back function (defined in the call to hook_to_notification_point()). notif_code will be the actual event identifier (listed in the following two sections) and va is any data supplied by IDA along with the event, possibly to provide further information.

The call-back function should return 0 if it permits the event notification to be handled by subsequent handlers (the typical scenario), or any other value if it is to be the only/last handler.

Something worth remembering is if you use hook_to_notification_point() in your plug-in, you must also use unhook_from_notification_point(), either once you no longer need to receive notifications, or inside your IDAP_term() function. This will avoid unexpected segmentation faults when exiting IDA. Going by the example above, to unhook the hooked notification point, it would be done like this:

```
unhook_from_notification_point(HT_DBG, dbg_callback, NULL);
```
4.5.2 UI Event Notifications

The following two lists show some of the event notifications that can be received and/or generated by a plug-in. These are only a sub-set of possible event notifications; what's listed are the more general purpose ones.

Although the below can be generated by a plug-in using `callui()`, most have helper functions, which means you don't need to use `callui()` and can just call the helper function instead.

<table>
<thead>
<tr>
<th>Event Notification</th>
<th>Description</th>
<th>Helper Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui_jumpto</code></td>
<td>Moves the cursor to an address</td>
<td><code>jumpto</code></td>
</tr>
<tr>
<td><code>ui_screenea</code></td>
<td>Return the address where the cursor is currently positioned</td>
<td><code>get_screen_ea</code></td>
</tr>
<tr>
<td><code>ui_refresh</code></td>
<td>Refresh all disassembly views</td>
<td><code>refresh_idaviewAnyway</code></td>
</tr>
<tr>
<td><code>ui_mbox</code></td>
<td>Display a message box to the user</td>
<td><code>vwarning</code>, <code>vinfo</code> and more.</td>
</tr>
<tr>
<td><code>ui_msg</code></td>
<td>Print some text in IDA's Log window</td>
<td><code>deb</code>, <code>vmsg</code></td>
</tr>
<tr>
<td><code>ui_askyn</code></td>
<td>Display a message box with Yes and No as options</td>
<td><code>askbuttons_cv</code></td>
</tr>
<tr>
<td><code>ui_askfile</code></td>
<td>Prompt the user for a filename</td>
<td><code>askfile_cv</code></td>
</tr>
<tr>
<td><code>ui_askstr</code></td>
<td>Prompt the user for a single line string</td>
<td><code>vaskstr</code></td>
</tr>
<tr>
<td><code>ui_asktext</code></td>
<td>Prompt the user for some text</td>
<td><code>vasktext</code></td>
</tr>
<tr>
<td><code>ui_form</code></td>
<td>Display a form (very flexible!)</td>
<td><code>AskUsingForm_cv</code></td>
</tr>
<tr>
<td><code>ui_open_url</code></td>
<td>Open a web browser at a particular URL</td>
<td><code>open_url</code></td>
</tr>
<tr>
<td><code>ui_load_plugin</code></td>
<td>Load a plug-in</td>
<td><code>load_plugin</code></td>
</tr>
<tr>
<td><code>ui_run_plugin</code></td>
<td>Run a plug-in</td>
<td><code>run_plugin</code></td>
</tr>
<tr>
<td><code>ui_get_hwnd</code></td>
<td>Get the HWND (Window Handle) for the IDA window</td>
<td><code>none</code></td>
</tr>
<tr>
<td><code>ui_get_curline</code></td>
<td>Get the colour-coded disassembled line</td>
<td><code>get_curline</code></td>
</tr>
<tr>
<td><code>ui_get_cursor</code></td>
<td>Get the X and Y coordinates of the current cursor position</td>
<td><code>get_cursor</code></td>
</tr>
</tbody>
</table>

The following event notifications are received by the plug-in, and would be handled by your callback function.

<table>
<thead>
<tr>
<th>Event Notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui_saving</code></td>
<td>IDA is currently saving and has saved the database, respectively</td>
</tr>
<tr>
<td><code>ui_saved</code></td>
<td>IDA has closed the database</td>
</tr>
</tbody>
</table>
For example, the following code will generate a `ui_screenea` event notification and display the result in an IDA dialog box using an `ui_mbox` event notification.

```c
void IDAP_run(int arg)
{
    ea_t addr;
    va_list va;
    char buf[MAXSTR];

    // Get the current cursor position, store it in addr
    callui(ui_screenea, &addr);
    qsnprintf(buf, sizeof(buf)-1, "Currently at: %a\n", addr);

    // Display an info message box
    callui(ui_mbox, mbox_info, buf, va);

    return;
}
```

In the above case, you would typically use the helper functions, however `callui()` was used for illustrative purposes.

### 4.5.3 Debugger Event Notifications

Debugger event notifications are broken up into Low Level, High Level and Function Result event notifications; the difference between them will be made clear in the following sub-sections. All of the event notifications mentioned below will belong to the `dbg_notification_t` enum, which is defined in `dbg.hpp`. If you supplied `HT_DBG` to `hook_to_notification_point()`, the below event notifications will be passed to your plug-in while a process is being debugged in IDA.

#### 4.5.3.1 Low Level Events

The following events taken from `dbg_notification_t` are all low level event notifications. Low level event notifications are generated by the debugger.

<table>
<thead>
<tr>
<th>Event Notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbg_process_start</code></td>
<td>Process started</td>
</tr>
<tr>
<td><code>dbg_process_exit</code></td>
<td>Process ended</td>
</tr>
<tr>
<td><code>dbg_library_load</code></td>
<td>Library was loaded</td>
</tr>
<tr>
<td><code>dbg_library_unload</code></td>
<td>Library was unloaded</td>
</tr>
<tr>
<td><code>dbg_exception</code></td>
<td>Exception was raised</td>
</tr>
<tr>
<td><code>dbg_breakpoint</code></td>
<td>A non-user defined breakpoint was hit</td>
</tr>
</tbody>
</table>

The `debug_event_t` struct (`idd.hpp`), which you can use to obtain further information about a debugger event notification, is always supplied in the `va` argument to your call-back function (for low level event notifications only). Here is the whole `debug_event_t` struct.

```c
struct debug_event_t
{
    event_id_t   eid;   // Event code (used to decipher 'info' union)
    process_id_t pid;  // Process where the event occurred
    thread_id_t   tid;  // Thread where the event occurred
```

For example, if your call-back function received the dbg_library_load event notification, you could look at debug_event_t’s modinfo member to see what the file loaded was:

```c

// Our callback function to handle HT_DBG event notifications
static int idaapi dbg_callback(void *udata, int event_id, va_list va)
{
    // va contains a debug_event_t pointer
    debug_event_t *evt = va_arg(va, debug_event_t *);

    // If the event is dbg_library_load, we know modinfo will be set
    // and contain the name of the library loaded
    if (event_id == dbg_library_load)
        msg("Loaded library, %s\n", evt->modinfo.name);

    return 0;
}
```

4.5.3.2 High Level Event Notifications

The following events taken from dbg_notification_t are all high level event notifications, which are generated by the IDA kernel:

<table>
<thead>
<tr>
<th>Event Notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbg_bpt</td>
<td>User-defined breakpoint was hit</td>
</tr>
<tr>
<td>dbg_trace</td>
<td>One instruction was executed (needs step tracing enabled)</td>
</tr>
<tr>
<td>dbg_suspend_process</td>
<td>Process has been suspended</td>
</tr>
<tr>
<td>dbg_request_error</td>
<td>An error occurred during a request (see section 5.14)</td>
</tr>
</tbody>
</table>
Each of these event notifications has different arguments supplied along with them in the `va` argument to your call-back function. None have `debug_event_t` supplied, like low level event notifications do.

The `dbg_bpt` event notification comes with both the Thread ID (`thread_id_t`) of the affected thread and the address where the breakpoint was hit in `va`. The below example will display a message in IDA's Log window when a user-defined breakpoint is hit.

```c
int idaapi dbg_callback(void *udata, int event_id, va_list va)
{
    // Only for the dbg_bpt event notification
    if (event_id == dbg_bpt)
    {
        // Get the Thread ID
        thread_id_t tid = va_arg(va, thread_id_t);
        // Get the address of where the breakpoint was hit
        ea_t addr = va_arg(va, ea_t);

        msg("Breakpoint hit at: %a, in Thread: %d\n", addr, tid);

        return 0;
    }

    return 0;
}
```

```c
int IDAP_init(void)
{
    hook_to_notification_point(HT_DBG, dbg_callback, NULL);
    ...
}
```

### 4.5.3.3 Function Result Notifications

In later sections, the concept of Synchronous and Asynchronous debugger functions will be discussed in more detail; until then, all you need to know is that synchronous debugger functions are just like ordinary functions – you call them, they do something and return. Asynchronous debugger functions however, get called and return without having completed the task, effectively having the request put into a queue and run in the background. When the task is completed, an event notification is generated indicating the completion of the original request.

The following are all function result notifications.

<table>
<thead>
<tr>
<th>Event Notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbg_attach_process</code></td>
<td>Debugger attached to a process (IDA 4.8)</td>
</tr>
<tr>
<td><code>dbg_detach_process</code></td>
<td>Debugger detached from a process (IDA 4.8)</td>
</tr>
<tr>
<td><code>dbg_process_attach</code></td>
<td>Debugger attached to a process (IDA 4.9)</td>
</tr>
<tr>
<td><code>dbg_process_detach</code></td>
<td>Debugger detached from a process (IDA 4.9)</td>
</tr>
<tr>
<td><code>dbg_step_into</code></td>
<td>Debugger stepped into a function</td>
</tr>
<tr>
<td><code>dbg_step_over</code></td>
<td>Debugger stepped over a function</td>
</tr>
<tr>
<td><code>dbg_run_to</code></td>
<td>Debugger has run to user's cursor position</td>
</tr>
<tr>
<td><code>dbg_step_until_ret</code></td>
<td>Debugger has run until return to caller was made</td>
</tr>
</tbody>
</table>
For example, the below code in IDAP_run() asks IDA to attach to a process. Once successfully attached, IDA generates the event notification, dbg_attach_process, which is handled by the dbg_callback call-back function.

```c
... int idaapi dbg_callback(void *udata, int event_id, va_list va) {
  // Get the process ID of what was attached to.
  process_id_t pid = va_arg(va, process_id_t);
  // Change dbg_attach_process to dbg_process_attach if you're
  // using IDA 4.9
  if (event_id == dbg_attach_process)
    msg("Successfully attached to PID %d\n", pid);
    return 0;
}

void IDAP_run(int arg) {
  int res;
  // Attach to a process. See Chapter 5 for usage.
  attach_process(NO_PROCESS, res);
  return;
}

int IDAP_init(void) {
  hook_to_notification_point(HT_DBG, dbg_callback, NULL);
  ...}
```

### 4.6 Strings

The Strings window in IDA can be accessed using the SDK, in particular each string within the binary (that is detected when the file is opened) is represented by the string_info_t structure, which is defined in strlist.hpp. Below is a slightly cut-down version of that structure.

```c
struct string_info_t {
  ea_t ea;         // Address of the string
  int length;      // String length
  int type;        // String type (0=C, 1=Pascal, 2=Pascal 2 byte
                   // 3=Unicode, etc.)
...);
```

Keep in mind that the above structure doesn't actually contain the string. To retrieve the string, you need to extract it from the binary file using get_bytes() or get_many_bytes(). To enumerate through the list of strings available, you could do the following:

```c
// Loop through all strings
for (int i = 0; i < get_strlist_qty(); i++) {
  char string[MAXSTR];
  string_info_t si;
  // Get the string item
  get_strlist_item(i, &si);
  if (si.length < sizeof(string)) {
    // Retrieve the string from the binary
    ...}
```
get_many_bytes(si.ea, string, si.length);
if (si.type == 0) // C string
    msg("String %d: %s\n", i, string);
if (si.type == 3) // Unicode
    msg("String %d: %S\n", i, string);
}
}

The above functions will be covered under *Chapter 5 – Functions.*
5. Functions

This section is broken up into different areas that the exported IDA SDK functions mostly fit into. I'll start from the most simple and more frequently used functions to the more complex and "niche" ones. I'll also provide basic examples with each function and the examples under the Examples section should provide more context. Obviously, this isn't a complete reference (refer to the header files in the SDK for that), but more of an overview of the most used and useful functions.

Important note about the examples: All of the functions below can be called from the IDAP_run(), IDAP_init() or IDAP_term() functions, unless otherwise indicated. Any of the examples can be pasted straight into the IDAP_run() function from the plug-in template in section 3.4 and should work. The additional header files required for each function and example will be specified where necessary.

5.1 Common Function Replacements

IDA provides many replacement functions for common C library routines. It is recommended that you use the replacements listed below instead of those provided by your C library. As of IDA 4.9, a lot of the C library routines are no longer available - you must use the IDA equivalent.

<table>
<thead>
<tr>
<th>C Library Functions</th>
<th>IDA Replacements</th>
<th>Defined In</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen, fread, fwrite, fseek, fclose</td>
<td>qfopen, qfread, qfwrite, qfseek, qfclose</td>
<td>fpro.h</td>
</tr>
<tr>
<td>fputc, fgetc, fputs, fgets</td>
<td>qfputc, qfgetc, qfputs, qfgets</td>
<td>fpro.h</td>
</tr>
<tr>
<td>vfprintf, vfscanf, vprintf</td>
<td>qfprintf, qfscanf, qvprintf</td>
<td>fpro.h</td>
</tr>
<tr>
<td>strcpy, strncpy, strcat, strncat</td>
<td>qstrncpy, qstrncat</td>
<td>pro.h</td>
</tr>
<tr>
<td>sprintf, snprintf, wsprintf</td>
<td>qsnprintf</td>
<td>pro.h</td>
</tr>
<tr>
<td>open, close, read, write, seek</td>
<td>qopen, qclose, qread, qwrite, qseek</td>
<td>pro.h</td>
</tr>
<tr>
<td>mkdir, isdir, filesize</td>
<td>qmkdir, qisdir, qfilesize</td>
<td>pro.h</td>
</tr>
<tr>
<td>exit, atexit</td>
<td>qexit, qatexit</td>
<td>pro.h</td>
</tr>
<tr>
<td>malloc, calloc, realloc, strdup, free</td>
<td>qmalloc, qcalloc, qrealloc, qstrdup, qfree</td>
<td>pro.h</td>
</tr>
</tbody>
</table>

It is strongly recommended that you use the above functions, however if you're porting an old plug-in and for some reason need the C library function, you can compile your plug-in with -DUSE_DANGEROUS_FUNCTIONS or -DUSE_STANDARD_FILE_FUNCTIONS.

5.2 Messaging

These are the functions you will probably use the most when writing a plug-in; not because they are the most useful, but simply because they provide a means for simple communication with the user and can be a great help when debugging plug-ins.
As you can probably tell from the definitions, all of these functions are inlined and take printf style arguments. They are all defined in kernwin.hpp.

### 5.2.1 msg

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline int msg(const char *format,...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Display a text message in IDA’s Log window (bottom of the screen during static disassembly, top of the screen during debugging).</td>
</tr>
<tr>
<td>Example</td>
<td>msg(&quot;Starting analysis at: %a\n&quot;, inf.startIP);</td>
</tr>
</tbody>
</table>

### 5.2.2 info

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline int info(const char *format,...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Display a text message in a pop-up dialog box with an ‘info’ style icon.</td>
</tr>
<tr>
<td>Example</td>
<td>info(&quot;My plug-in v1.202 loaded.&quot;);</td>
</tr>
</tbody>
</table>

### 5.2.3 warning

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline int warning(const char *format,...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Display a text message in a pop-up dialog box with an ‘warning’ style icon.</td>
</tr>
<tr>
<td>Example</td>
<td>warning(&quot;Please beware this could crash IDA!\n&quot;);</td>
</tr>
</tbody>
</table>

### 5.2.4 error

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline int error(const char *format,...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Display a text message in a pop-up dialog box with an ‘error’ style icon. Closes IDA (uncleanly) after the user clicks OK.</td>
</tr>
<tr>
<td>Example</td>
<td>error(&quot;There was a critical error, exiting IDA.\n&quot;);</td>
</tr>
</tbody>
</table>
5.3 UI Navigation

The functions below are specifically for interacting with the user and the IDA GUI. Some of them use `callui()` to generate an event to IDA. All are defined in `kernwin.hpp`.

### 5.3.1 get_screen_ea

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline ea_t get_screen_ea(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the address within the current disassembled file(s) that the user's cursor is positioned at.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp>  
msg("Cursor position is %a\n", get_screen_ea()); |

### 5.3.2 jumpto

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline bool jumpto(ea_t ea, int opnum=-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Moves the user's cursor to a position within the current disassembled file(s), represented by ea. opnum is the X coordinate that the cursor will be moved to, or -1 if it isn't to be changed. Returns true if successful, false if it failed.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp>  
// Jump to the binary entry point + 8 bytes, don't move  
// the cursor along the X-axis  
jumpto(inf.startIP + 8); |

### 5.3.3 get_cursor

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline bool get_cursor(int *x, int *y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Fills *x and *y with the X and Y coordinates of the user's cursor position within the current disassembled file(s).</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp>  
int x, y;  
// Store the cursor X coordinate in x, and the Y  
// coordinate in y, display the results in the Log window  
get_cursor(&x, &y);  
msg("X: %d, Y: %d\n", x, y); |
### 5.3.4 get_curline

| Definition | inline char *
| --- | get_curline(void) |
| Synopsis | Return a pointer to the line of text at the user's cursor position. This will return everything on the line – the address, code and comments. It will also be colour-coded, which you would use `tag_remove()` (see section 5.20.1) to clean. |
| Example | #include <kernwin.hpp>
// Display the current line of text in the Log window
msg("%s\n", get_curline()); |

### 5.3.5 read_selection

| Definition | inline bool
| --- | read_selection(ea_t *ea1, ea_t *ea2) |
| Synopsis | Fills *ea1 and *ea2 with the start and end addresses, respectively, of the user's selection. Returns true if there was a selection, false if there wasn't. |
| Example | #include <kernwin.hpp>
e_t saddr, eaddr;
// Get the address range selected, or return false if there was no selection
int selected = read_selection(&saddr, &eaddr);
if (selected) {
    msg("Selected range: %a -> %a\n", saddr, eaddr);
} else {
    msg("No selection.\n");
} |

### 5.3.6 callui

| Definition | idaman callui_t ida_export_data
| --- | (idaapi*callui)(ui_notification_t what,...) |
### Synopsis
The user interface dispatcher function. This enables you to call the events listed in section 4.5.2, and many others within the `ui_notification_t` enum. `callui()` is always passed a `ui_notification_t` type as the first argument (ui_jumpTo, ui_banner, etc.) followed by any arguments required for the respective notification.

### Example
```c
#include <windows.hpp> // For the HWND definition
#include <kernwin.hpp>

HWND hwnd = (HWND)callui(ui_get_hwnd).vptr;
if (hwnd == NULL)
  error("Cannot run in theIDA text version!");
```

---

### 5.3.7 askaddr

#### Definition
```c
inline int
askaddr(ea_t *addr, const char *format,...)
```

#### Synopsis
Presents a dialog box asking the user to supply an address. `*addr` will be the default value to start with, and then filled with the user supplied address upon clicking OK. `*format` is the `printf` style text that goes in the dialog box.

#### Example
```c
#include <kernwin.hpp>
ea_t addr = inf.startIP;
askaddr(&addr, "Please supply an address to jump to.");
jumpto(addr);
```

---

### 5.3.8 AskUsingForm_c

#### Definition
```c
inline int
AskUsingForm_c(const char *form,...)
```

#### Synopsis
Displays a form to the user, and is too flexible to be covered here but is heavily commented in `kernwin.hpp`. It effectively allows you to design your own user form, including buttons, text fields, radio buttons and text as format strings.
### 5.4 Entry Points

The following functions are for working with entry points (where execution begins) in a binary. They can all be found in `entry.hpp`.

#### 5.4.1 `get_entry_qty`

**Definition**

```c
idaman size_t
ida_export get_entry_qty(void)
```

**Synopsis**

Returns the number of entry points in the currently disassembled file(s). This will typically return 1, except for DLLs, which can have many.

**Example**

```c
#include <entry.hpp>
msg("Number of entry points: %d\n", get_entry_qty());
```

#### 5.4.2 `get_entry_ordinal`

**Definition**

```c
idaman uval_t
ida_export get_entry_ordinal(size_t idx)
```

**Synopsis**

Returns the ordinal number of the entry point index number supplied as `idx`. You need the ordinal number because `get_entry()` and `get_entry_name()` use it.
5.4.3 `get_entry`

**Definition**

```c
idaman ea_t
ida_export get_entry(uval_t ord);
```

**Synopsis**

Returns the address of an entry point ordinal number, supplied as the `ord` argument. Use `get_entry_ordinal()` to get the ordinal number of an entry point number, as shown in section 5.4.2.

**Example**

```c
#include <entry.hpp>

// Loop through each entry point.
for (int e = 0; e < get_entry_qty(); e++)
    msg("Entry point found at: %a\n",
        get_entry(get_entry_ordinal(e)));
```

5.4.4 `get_entry_name`

**Definition**

```c
idaman char *
ida_export get_entry_name(uval_t ord)
```

**Synopsis**

Return a pointer to the name of the entry point address (e.g. `start`)

**Example**

```c
#include <entry.hpp>

// Loop through each entry point
for (int e = 0; e < get_entry_qty(); e++) {
    int ord = get_entry_ordinal(e);
    // Display the entry point address and name
    msg("Entry point %a: %s\n",
        get_entry(ord),
        get_entry_name(ord));
}
```

5.5 Areas

The following functions work with areas and area control blocks, as described in section 4.2.2 and 4.2.3 respectively. Unlike all the functions covered so far, they are methods within the `areacb_t` class, and so therefore can only be used on instances of that class. Two instances of `areacb_t` are `funcs` and `segs`, representing all functions and segments within the currently disassembled file(s) in IDA.
Although you should use the segment-specific functions for dealing with segments, and the function-specific functions for dealing with functions, working with areas directly gives you a more abstract way of dealing with functions and segments.

All the below are defined in area.hpp.

### 5.5.1 get_area

<table>
<thead>
<tr>
<th>Definition</th>
<th>area_t * get_area(ea_t ea)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns a pointer to the area_t structure to which ea belongs.</td>
</tr>
</tbody>
</table>

**Example**

```c
#include <kernwin.hpp> // For askaddr() definition
#include <funcs.hpp> // For funcs definition
#include <area.hpp>

ea_t addr;

// Ask the user for an address (see section 5.3.7)
askaddr(&addr, "Find the function owner of address:");

// Get the function that owns that address
// You could use segs.get_area(addr) to get the
// segment that owned to address here too.
area_t *area = funcs.get_area(addr);
msg("Area holding %a starts at %a, ends at %a\n",
    addr,
    area->startEA,
    area->endEA);
```

### 5.5.2 get_area_qty

<table>
<thead>
<tr>
<th>Definition</th>
<th>uint get_area_qty(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Get the number of areas within the current area control block.</td>
</tr>
</tbody>
</table>

**Example**

```c
#include <funcs.hpp> // For funcs definition
#include <segment.hpp> // For segs definition
#include <area.hpp>

msg("%d Functions, and %d Segments",
    funcs.get_area_qty(),
    segs.get_area_qty());
```
### 5.5.3 getn_area

| Definition | area_t *
<table>
<thead>
<tr>
<th></th>
<th>getn_area(unsigned int n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns a pointer to an area_t struct for the area number supplied as n.</td>
</tr>
</tbody>
</table>

| Example | #include <funcs.hpp> // For funcs definition
|         | #include <segment.hpp> // For segs definition
|         | #include <area.hpp>
|         | // funcs represents all functions, so get the first // function area (0).
|         | area_t *firstFunc = funcs.getn_area(0);
|         | msg("First func starts: %a, ends: %a\n", firstFunc->startEA, firstFunc->endEA);
|         | // segs represents all segments, so get the first // segment area (0).
|         | area_t *firstSeg = segs.getn_area(0);
|         | msg("First seg starts: %a, ends: %a\n", firstSeg->startEA, firstSeg->endEA); |

### 5.5.4 get_next_area

| Definition | int
get_next_area(ea_t ea) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the number of the area following the area containing address ea.</td>
</tr>
</tbody>
</table>

| Example | #include <funcs.hpp> // For funcs definition
|         | #include <area.hpp>
|         | // Loop through functions as areas from first to last int i = 0;
|         | for (area_t *func = funcs.getn_area(0); i < funcs.get_area_qty(); i++) |
|         |   |
|         | } |
|         | msg ("Area start: %a, end: %a\n", func->startEA, func->endEA);
|         | int funcNo = funcs.get_next_area(func->startEA);
|         | func = funcs.getn_area(funcNo); |
### 5.5.5 get_prev_area

<table>
<thead>
<tr>
<th>Definition</th>
<th>int get_prev_area(ea_t ea)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the number of the area preceding the area containing address ea.</td>
</tr>
</tbody>
</table>
| Example    | `#include <segment.hpp> // For segs definition
#include <area.hpp>

// Loop through segments as areas from last to first
int i = segs.get_area_qty();
for (area_t *seg = segs.getn_area(0); i > 0; i--) {
    msg("Area start: %a, end: %a\n",
         seg->startEA,
         seg->endEA);
    int segNo = segs.get_next_area(seg->startEA);
    seg = segs.getn_area(segNo);
} |

---

### 5.6 Segments

The following functions work with segments (.text, .idata, etc.) and are defined in `segment.hpp`. A lot of these functions are simply wrappers to `areacb_t` methods for the `segs` variable.

#### 5.6.1 get_segm_qty

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline int get_segm_qty(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the number of segments in the currently disassembled file(s). This simply calls <code>segs.get_area_qty()</code>.</td>
</tr>
</tbody>
</table>
| Example    | `#include <segment.hpp>
msg("%d segments in disassembled file(s).\n", get_segm_qty());` |

---

#### 5.6.2 getnseg

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline segment_t * getnseg(int n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns a pointer to the <code>segment_t</code> struct for the segment number, n, supplied. This is a wrapper to <code>segs.getn_area()</code>.</td>
</tr>
</tbody>
</table>
```c
#include <segment.hpp>

// Get the address of segment 0 (the first segment)
segment_t *firstSeg = getnseg(0);
msg("Address of the first segment is %a\n", firstSeg->startEA);
```

### 5.6.3 get_segm_by_name

| Definition | idaman segment_t *ida_export
get_segm_by_name(const char *name) |
|------------|-----------------------------------------------------------------
| Synopsis   | Returns a pointer to the segment_t struct for the segment with name, *name. Will return NULL if there is no such segment. If there are multiple segments with the same name, the first will be returned. |
| Example    | #include <segment.hpp>

// Get the segment_t structure for the .text segment.
segment_t *textSeg = get_segm_by_name(".text");
msg("Text segment is at %a\n", textSeg->startEA); |

### 5.6.4 getseg

| Definition | inline segment_t *
getseg(ea_t ea) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the segment_t struct for the segment that contains address ea. This function is a wrapper to segs.get_area().</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp> // For get_screen_ea() definition
#include <segment.hpp>

// Get the address of the user's cursor position
// see section 5.2.1 for get_screen_ea()
ea_t addr = get_screen_ea();

// Get the segment that owns that address
area_t *area = segs.get_area(addr);
msg("Segment holding %a starts at %a, ends at %a\n",
    addr,
    area->startEA,
    area->endEA); |
5.6.5 get_segm_name (IDA 4.8)

| Definition | idaman char *ida_export  
get_segm_name(const segment_t *s) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the name (&quot;_text&quot;, &quot;_idata&quot;, etc.) of segment *s.</td>
</tr>
</tbody>
</table>

Example

```
#include <segment.hpp>

// Loop through all segments displaying their names
for (int i = 0; i < get_segm_qty(); i++) {
    segment_t *seg = getnseg(i);
    msg("Segment %d at %a is named %s\n", i, seg->startEA, get_segm_name(seg));
}
```

5.6.6 get_segm_name (IDA 4.9)

| Definition | idaman ssize_t ida_export  
get_segm_name(const segment_t *s, char *buf, size_t bufsize) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Fills *buf, limited by bufsize with the name (&quot;_text&quot;, &quot;_idata&quot;, etc.) of segment *s. Returns the size of the segment name, or -1 if s is NULL.</td>
</tr>
</tbody>
</table>

Example

```
#include <segment.hpp>

// Loop through all segments displaying their names
for (int i = 0; i < get_segm_qty(); i++) {
    char segName[MAXSTR];
    segment_t *seg = getnseg(i);
    get_segm_name(seg, segName, sizeof(segName)-1);
    msg("Segment %d at %a is named %s\n", i, seg->startEA, segName);
}
```

5.7 Functions

The below set of functions are for working with functions within the currently disassembled file(s) in IDA. As with segments, functions are areas, and so some of the below functions are simply wrappers to areacb_t methods, in funcs. All are defined in funcs.hpp.
5.7.1 get_func_qty

Definition

<table>
<thead>
<tr>
<th>idaman size_t ida_export</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_func_qty(void)</td>
</tr>
</tbody>
</table>

Synopsis

Returns the number of functions in the currently disassembled file(s).

Example

```c
#include <functs.hpp>
msg("%d functions in disassembled file(s).\n", get_func_qty());
```

5.7.2 get_func

Definition

<table>
<thead>
<tr>
<th>idaman func_t *ida_export</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_func(ea_t ea)</td>
</tr>
</tbody>
</table>

Synopsis

Returns a pointer to the func_t structure representing the function that "owns" address ea. If ea is not part of a function, NULL is returned. Only function entry chunks are returned (see section 4.2.3.2 for information about chunks and tails).

Example

```c
#include <kernwin.hpp> // For get_screen_ea() definition
#include <functs.hpp>

// Get the address of the user's cursor
ea_t addr = get_screen_ea();
func_t *func = get_func(addr);
if (func != NULL) {
    msg("Current function starts at %a\n", func->startEA);
} else {
    msg("Not inside a function!\n");
}
```

5.7.3 getn_func

Definition

<table>
<thead>
<tr>
<th>idaman func_t *ida_export</th>
</tr>
</thead>
<tbody>
<tr>
<td>getn_func(size_t n)</td>
</tr>
</tbody>
</table>

Synopsis

Returns a pointer to the func_t representing the function number supplied as n. Will return NULL if n is a non-existent function number. It will also only return function entry chunks.

Example

```c
#include <functs.hpp>

// Loop through all functions
for (int i = 0; i < get_func_qty(); i++) {
    func_t *curFunc = getn_func(i);
    msg("Function at: %a\n", curFunc->startEA);
}
```
### 5.7.4 get_func_name

**Definition**

```c
idaman char *ida_export
get_func_name(ea_t ea, char *buf, size_t bufsize)
```

**Synopsis**

Gets the name of the function owning address `ea`, and stores it in `*buf`, limited by the length of `bufsize`. It returns the `*buf` pointer or `NULL` if the function has no name.

**Example**

```c
#include <kernwin.hpp> // For get_screen_ea() definition
#include <funcs.hpp>

// Get the address of the user's cursor
ea_t addr = get_screen_ea();
func_t *func = get_func(addr);
if (func != NULL) {
    // Buffer where the function name will be stored
    char funcName[MAXSTR];
    if (get_func_name(func->startEA, funcName, MAXSTR) != NULL) {
        msg("Current function %a, named %s\n",
             func->startEA,
             funcName);
    }
}
```

### 5.7.5 get_next_func

**Definition**

```c
idaman func_t *
ida_export get_next_func(ea_t ea)
```

**Synopsis**

Returns a pointer to the `func_t` structure representing the function following the one owning `ea`. Returns `NULL` if there is no following function.

**Example**

```c
#include <kernwin.hpp> // For get_screen_ea() definition
#include <funcs.hpp>

ea_t addr = get_screen_ea();
// Get the function after the one containing the address where the user's cursor is positioned
func_t *nextFunc = get_next_func(addr);

if (nextFunc != NULL)
    msg("Next function starts at %a\n",
         nextFunc->startEA);
```
5.7.6 get_prev_func

**Definition**

idaman func_t *
ida_export get_prev_func(ea_t ea)

**Synopsis**

Returns a pointer to the func_t structure representing the function before the one owning ea. Returns NULL if there is no previous function.

**Example**

```c
#include <kernwin.hpp> // For get_screen_ea() definition
#include <funcs.hpp>

ea_t addr = get_screen_ea();
// Get the function before the one containing the
// address where the user's cursor is positioned
func_t *prevFunc = get_prev_func(addr);

if (prevFunc != NULL)
    msg("Previous function starts at %a
        prevFunc->startEA);
```

5.7.7 get_func_comment

**Definition**

inline char *
get_func_comment(func_t *fn, bool repeatable)

**Synopsis**

Return any commentary added by the user or IDA for the function indicated by *fn. If repeatable is true, repeatable comments are included. NULL is returned if there are no comments.

**Example**

```c
#include <funcs.hpp>

// Loop through all functions, displaying their comments
// including repeatable comments.
for (int i = 0; i < get_func_qty(); i++) {
    func_t *curFunc = getn_func(i);
    msg("%a: %s
        curFunc->startEA,
        get_func_comment(curFunc, false));
}
```

5.8 Instructions

The functions below work with instructions within the currently disassembled file(s) in IDA. All are defined in ua.hpp, except for generate_disasm_line(), which is defined in lines.hpp.

5.8.1 generate_disasm_line

**Definition**

idaman bool ida_export
generate_disasm_line(ea_t ea, char *buf, size_t bufsize,
int flags=0)
### 5.8.2 ua_ana0

**Definition**

```c
idaman int  
ida_export ua_ana0(ea_t ea)
```

**Synopsis**

Disassemble `ea`. Returns the length of the instruction in bytes and fills the global `cmd` structure with information about the instruction. If `ea` doesn't contain an instruction, 0 is returned. This is a read-only function and doesn't modify the IDA database.

**Example**

```c
#include <kernwin.hpp> // For get_screen_ea() definition  
#include <ua.hpp>

ea_t ea = get_screen_ea();

if (ua_ana0(ea) > 0)
    msg("Instruction size: %d bytes
\n", cmd.size);
else
    msg("Not at an instruction.
\n");
```

### 5.8.3 ua_code

**Definition**

```c
idaman int  
ida_export ua_code(ea_t ea)
```

**Synopsis**

Disassemble `ea`. Returns the length of the instruction in bytes, fills the global `cmd` structure with information about the instruction and updates the IDA database with the results. If `ea` doesn't contain an instruction, 0 is returned.
```c
#include <kernwin.hpp> // For read_selection() definition
#include <ua.hpp>

ea_t saddr, eaddr;
ea_t addr;

// Get the user selection
int selected = read_selection(&saddr, &eaddr);
if (selected) {
    // Re-analyse the selected address range
    for (addr = saddr; addr <= eaddr; addr++) {
        ua_code(addr);
    }
} else {
    msg("No selection.\n");
}
```

### 5.8.4 ua_outop

**Definition**

```c
idaman bool ida_export
ua_outop(ea_t ea, char *buf, size_t bufsize, int n)
```

**Synopsis**

Fills `*buf`, limited by `bufsize`, with the text representation of operand number `n` to the instruction at `ea` and updates the IDA database with the instruction if it isn't already defined. Returns false if operand `n` doesn't exist.

The text returned in `*buf` is colour coded, so you need to use `tag_remove()` (see section 5.20.1) to get printable text.

```c
#include <ua.hpp>

// Get the entry point address
ea_t addr = inf.startIP;

// Fill cmd with information about the instruction
// at the entry point
ua_ana0(addr);

// Loop through each operand (until one of o_void type // is reached), displaying the operand text.
for (int i = 0; cmd.Operands[i].type != o_void; i++) {
    char op[MAXSTR];
    ua_outop(addr, op, sizeof(op)-1, i);
    msg("Operand %d: %s\n", i, op);
}
```

### 5.8.5 ua_mnem

**Definition**

```c
idaman const char *ida_export
ua_mnem(ea_t ea, char *buf, size_t bufsize)
```
Synopsis

Fills *buf, limited by bufsize, with the mnemonic used in the instruction at ea and updates the IDA database with the instruction if it isn't already defined. Returns the *buf pointer or NULL if there is no instruction at ea.

Example

```c
#include <segment.hpp> // For segment functions
#include <ua.hpp>

// Loop through each executable segment, displaying
// the mnemonic used in each instruction
for (int s = 0; s < get_segm_qty(); s++) {
    segment_t *seg = getnseg(s);
    if (seg->type == SEG_CODE) {
        int bytes = 0;

        // a should always be the address of an
        // instruction, which is why bytes is dynamic
        // depending on the result of ua_mnem()
        for (ea_t a = seg->startEA; a < seg->endEA; a += bytes) {
            char mnem[MAXSTR];
            const char *res;

            // Get the mnemonic at a, store it in mnem
            res = ua_mnem(a, mnem, sizeof(mnem)-1);

            // If this was an instruction, display
            // the mnemonic, set the bytes counter
            // to cmd.size, so that the next address
            // processed by ua_mnem() is the next
            // instruction.
            if (res != NULL) {
                msg("Mnemonic at %a: %s\n", a, mnem);
                bytes = cmd.size;
            } else {
                msg("No code\n");
            }
        }
    }
}
```

5.9 Cross Referencing

The following four functions are a part of the xrefblk_t structure, defined in xref.hpp. They are used to populate and enumerate cross references to or from an address. All functions take flags as an argument, which can be one of the following, as taken from xref.hpp:

```c
#define XREF_ALL        0x00      // return all references
#define XREF_FAR        0x01      // don't return ordinary flow xrefs
#define XREF_DATA       0x02      // return data references only
```
An ordinary flow is when execution normally passes from one instruction to another without the use of a CALL or JMP (or equivalent) instruction. If you are only interested in code cross references (ignoring ordinary flows), then you would use XREF_ALL and check if the isCode member of xrefblk_t is true in each case. Use XREF_DATA if you are only interested in data references.

5.9.1 first_from

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool first_from(ea_t from, int flags)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Populates the xrefblk_t structure with the first cross reference from the from address. flags dictates what cross references you are interested in. Returns false if there are no references from from.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp> // For get_screen_ea() definition
#include <xref.hpp>
ea_t addr = get_screen_ea();
xrefblk_t xb;
if (xb.first_from(addr, XREF_ALL)) {
    // xb is now populated
    msg("First reference FROM %a is %a\n", xb.from, xb.to);
}

5.9.2 first_to

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool first_to(ea_t to, int flags)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Populates the xrefblk_t structure with the first cross reference to the to address. flags dictates what cross references you are interested in. Returns false if there are no references to to</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp> // For get_screen_ea() definition
#include <xref.hpp>
ea_t addr = get_screen_ea();
xrefblk_t xb;
if (xb.first_to(addr, XREF_ALL)) {
    // xb is now populated
    msg("First reference TO %a is %a\n", xb.to, xb.from);
}
### 5.9.3 next_from

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool next_from(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Populates the xrefblk_t structure with the next cross references from the from address. Returns false if there are no more cross references.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp> // For get_screen_ea() definition  
#include <lines.hpp> // For tag_remove() and  
// generate_disasm_line()  
#include <xref.hpp>  
xrefblk_t xb;  
ea_t addr = get_screen_ea();  
// Replicate IDA 'x' keyword functionality  
for (bool res = xb.first_to(addr, XREF_FAR); res;  
    res = xb.next_to()) {  
    char buf[MAXSTR];  
    char clean_buf[MAXSTR];  
    // Get the disassembly text for the referencing addr  
generate_disasm_line(xb.from, buf, sizeof(buf)-1);  
    // Clean out any format or colour codes  
tag_remove(buf, clean_buf, sizeof(clean_buf)-1);  
    msg("%a: %s\n", xb.from, clean_buf);  
} |

### 5.9.4 next_to

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool next_to(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Populates the xrefblk_t structure with the next cross references to the to address. Returns false if there are no more cross references.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp> // For get_screen_ea() definition  
#include <xref.hpp>  
xrefblk_t xb;  
ea_t addr = get_screen_ea();  
// Get the first cross reference to addr  
if (xb.first_to(addr, XREF_FAR)) {  
    if (xb.next_to())  
        msg("There are multiple references to %a\n",  
            addr);  
    else  
        msg("The only reference to %a is at %a\n",  
            addr, xb.from);  
} |
5.10 Names

The following functions deal with function (sub_*), location (loc_*) and variable (arg_*, var_*) names, set by IDA or the user. All are defined in name.hpp. Register names are not recognised by these functions.

5.10.1 get_name

**Definition**

```cpp
idaman char *ida_export
get_name(ea_t from, ea_t ea, char *buf, size_t bufsize)
```

**Synopsis**

Fill *buf, limited by bufsize, with the uncoloured name for ea. The *buf pointer is returned if ea has a name, or NULL if it doesn't. If you are after a name that is local to a function, from should be within the same function, or it won't be seen. If you are not after a local name, from should just be BADADDR.

**Example**

```cpp
#include <name.hpp>
char name[MAXSTR];

// Get the name of the entry point, should be start // in most cases.
char *res = get_name(BADADDR, inf.startIP, // Entry point name,
                      sizeof(name)-1);

if (res != NULL)
  msg("Name: %s\n", name);
else
  msg("No name for %a\n", inf.startIP);
```

5.10.2 get_name_ea

**Definition**

```cpp
idaman ea_t ida_export
get_name_ea(ea_t from, const char *name)
```

**Synopsis**

Return the address of where the name supplied in *name is defined. If you are after a name that is local to a function, from should be within the same function, or it won't be seen. If you are not after a local name, from should just be BADADDR.
#include <kernwin.hpp> // For askstr and get_screen_ea
#include <name.hpp>

// Get the cursor address
ea_t addr = get_screen_ea();

// Ask the user for a string (see kernwin.hpp), which
// will be the name we search for.
char *name = askstr(HIST_IDENT, // History identifier
                     "start", // Default value
                     "Please enter a name"); // Prompt

// Display the address that the name represents. You will
// get FFFFFFFFFF for stack variables and nonexistent
// names.
msg("Address: %a\n", get_name_ea(addr, name));

## 5.10.3 get_name_value

| Definition | idaman int ida_export
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>get_name_value(ea_t from, const char *name, uval_t *value)</td>
<td>Returns the value into *value, represented by the name *name, relative to the address from. *value will contain either a stack offset or linear address. If you are after a name that is local to a function, from should be within the same function, or it won't be seen. If you are not after a local name, from should just be BADADDR. The return value is one of the following, representing the type of name it is. Taken from name.hpp:</td>
</tr>
<tr>
<td>Synopsis</td>
<td></td>
</tr>
<tr>
<td>#define NT_NONE 0 // name doesn't exist or has no value</td>
<td></td>
</tr>
<tr>
<td>#define NT_BYTE 1 // name is byte name (regular name)</td>
<td></td>
</tr>
<tr>
<td>#define NT_LOCAL 2 // name is local label</td>
<td></td>
</tr>
<tr>
<td>#define NT_STKVAR 3 // name is stack variable name</td>
<td></td>
</tr>
<tr>
<td>#define NT_ENUM 4 // name is symbolic constant</td>
<td></td>
</tr>
<tr>
<td>#define NT_ABS 5 // name is absolute symbol</td>
<td></td>
</tr>
<tr>
<td>#define NT_SEG 6 // name is segment or segment register</td>
<td></td>
</tr>
<tr>
<td>#define NT_STROFF 7 // name is structure member</td>
<td></td>
</tr>
<tr>
<td>#define NT_BMASK 8 // name is a bit group mask name</td>
<td></td>
</tr>
</tbody>
</table>
#include <kernwin.hpp> // For get_screen_ea() and askstr()
#include <name.hpp>

uval_t value;
eat addr = get_screen_ea();

// Ask the user for a name
char *name = askstr(HIST_IDENT, "start",
                    "Please enter a name");

// Get the value of that name, relative to addr
int type = get_name_value(addr, name, &value);

// The type will correspond to one of the NT_values
// defined in name.hpp.
// Value will be FFFFFFFF4 for the first local variable
// or 8 for the first argument to a function. It could
// also be the linear address of the strcpy() definition
// for example.
msg("Type: %d, Value: %a\n", type, value);

5.11 Searching

The following functions are used for doing simple searching within the disassembled file(s) in IDA,
and are defined in search.hpp. There are also other search functions for specific search types
(errors, etc.) which can also be found in search.hpp. The search functions take flags, which
dictate how the search is conducted, what is searched for, etc. These flags are, as taken from
search.hpp:

#define SEARCH_UP 0x000 // only one of SEARCH_UP or
                    // SEARCH_DOWN can be specified
#define SEARCH_DOWN 0x001
#define SEARCH_NEXT 0x002 // Search for the next occurrence
#define SEARCH_CASE 0x004 // Make the search case-sensitive
#define SEARCH_REGEX 0x008 // Use the regular expression parser
#define SEARCH_NOBRK 0x010 // don't test ctrl-break
#define SEARCH_NOSHOW 0x020 // don't display the search progress
#define SEARCH_UNICODE 0x040 // treat strings as unicode
#define SEARCH_IDENT 0x080 // search for an identifier
  // it means that the characters before
  // and after the pattern can not be
  // is_visible_char()
#define SEARCH_BRK 0x100 // return BADADDR if break is
   // pressed during find_imm()

Typically, you'd just use SEARCH_DOWN to conduct a case-insensitive search, towards the bottom
of the file(s).

5.11.1 find_text (IDA 4.9 only)

Definition

idaman ea_t ida_export
find_text(eat startEA, int y, int x, const char *ustr,
5.11.2 find_binary

| Definition | idaman ea_t ida_export
find_binary(ea_t startea, ea_t endea, const char *ubinstr, int radix, int sflag) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Searches between startea and endea for the string in *ubinstr. radix is the numeric base (if you're searching for numbers), which can be 8 (octal), 10 (decimal) or 16 (hex). sflag can be any of the previously mentioned flags. Note that this function doesn't search the disassembled text that you see in IDA, but the binary itself. The content of *ubinstr will differ depending on the type of search you are conducting. For strings, the string itself must be wrapped in quotes (&quot;), for single characters, they must be wrapped in single quotes ('). A question-mark (?) can be used to indicate a single wildcard byte.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp> // for askstr() and jumpto()
#include <search.hpp>

// Ask the user for a search string
char *name = askstr(HIST_SRCH, "", "Please enter a string");
char searchstring[MAXSTR];

// Encapsulate the search string in quotes
qsnprintf(searchstring, sizeof(searchstring)-1, "\"%s\"", name);

ea_t res = find_binary(inf.minEA, // Top of the file
inf.maxEA, // Bottom of the file
searchstring,
0, // radix not applicable
SEARCH_DOWN);

if (res != NULL) {
  msg("Match found at %a\n", res);
5.12 IDB

The following functions are for working with IDA database (IDB) files, and can be found in loader.hpp. Although there is no actual definition of the `linput_t` class, you need to call the `open_linput()` (diskio.hpp) function to create an instance of the class, which some functions use as an argument. You can also use `make_linput()` to convert a FILE pointer to a `linput_t` instance; see loader.hpp for more information.

### 5.12.1 open_linput

<table>
<thead>
<tr>
<th>Definition</th>
<th>idaman linput_t *ida_export open_linput(const char *file, bool remote)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Create an instance of the <code>linput_t</code> class for file path <code>*file</code>. If the file is remote, set the <code>remote</code> argument to true. Returns <code>NULL</code> if it failed to open the file.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp> // For askfile_cv definition  
#include <diskio.hpp>  

    // Prompt the user for a file  
    char *file = askfile_cv(0, "", "File to open", NULL);  

    // Open the file  
    linput_t *myfile = open_linput(file, false);  

    if (myfile == NULL)  
        msg("Failed to open or corrupt file.\n");  
    else  
        // Return the size of the opened file.  
        msg("File size: %d\n", qlsize(myfile)); |

### 5.12.2 close_linput

<table>
<thead>
<tr>
<th>Definition</th>
<th>idaman void ida_export close_linput(linput_t *li)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Close the file represented by the <code>linput_t</code> instance, <code>*li</code>, created by <code>open_linput()</code></td>
</tr>
</tbody>
</table>
5.12.3 load_loader_module

| Definition | idaman int ida_export  
|            | load_loader_module(linput_t *li, const char *lname, const char *fname, bool is_remote) |
| Synopsis   | Load a file into the current IDB, either as a linput_t instance, *li, or file path in *fname, using the loader module *lname. If *li is NULL, *fname must be supplied and vise versa. Returns 1 on success, 0 on failure. |
| Example    | #include <kernwin.hpp> // For askfile_cv()  
|            | #include <loader.hpp> |
|            | // Prompt the user for a file to open.  
|            | char *file = askfile_cv(0, "", "DLL file..", NULL); |
|            | // Load it into the IDB using the PE loader module  
|            | int res = load_loader_module(NULL, "pe", file, false) |
|            | if (res < 1)  
|            |   msg("Failed to load %s as a PE file.\n", file); |

5.12.4 load_binary_file

| Definition | idaman bool ida_export  
|            | load_binary_file(const char *filename, linput_t *li, ushort _neflags, long fileoff, ea_t basepara, ea_t binoff, ulong nbytes); |
| Synopsis   | Load a binary file *li, named *filename starting at offset, fileoff. _neflags is any of the NEF_ flags defined in loader.hpp. nbytes specifies the number of bytes to load from the file, or 0 for the whole file. basepara is the paragraph where this new binary will be loaded, and binoff is the offset within that segment. You can safely set basepara to the adress you want the file loaded at, and set binoff to 0. Returns false if the load failed. 
|            | This is not the function you would use for loading a DLL or executable file (a PE file for instance) into the IDB. For that, you would use load_loader_module() above. |
#include <kernwin.hpp> // For askfile_cv()
#include <diskio.hpp> // For open_linput()
#include <loader.hpp>

// Ask the user for a filename
char *file = askfile_cv(0, "", "DLL file..", NULL);

// Create a linput_t instance for that file
linput_t *li = open_linput(file, false);

// Load the file at the end of the currently loaded
// file (inf.maxEA).
bool status = load_binary_file(file,
   li,
   NEF_SEGS,
   0,
   inf.maxEA,
   0,
   0);

if (status)
   msg("Successfully loaded %s at %a\n", file,
        inf.maxEA);
else
   msg("Failed to load file.\n");

5.12.5 gen_file

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>idaman int ida_export</td>
</tr>
<tr>
<td>gen_file(ofile_type_t otype, FILE *fp, ea_t ea1, ea_t ea2, int flags)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate an output file, *fp, based on the currently open IDB file.ea1 and ea2 are the start and end addresses respectively, however these are ignored for some output types.otype must be one of the following, taken from loader.hpp:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>OFILE_MAP = 0, // MAP file</td>
</tr>
<tr>
<td>OFILE_EXE = 1, // Executable file</td>
</tr>
<tr>
<td>OFILE_IDC = 2, // IDC file</td>
</tr>
<tr>
<td>OFILE_LST = 3, // Disassembly listing</td>
</tr>
<tr>
<td>OFILE_ASM = 4, // Assembly</td>
</tr>
<tr>
<td>OFILE_DIF = 5; // Difference</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>flags can be any combination of the following, also taken from loader.hpp:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>#define GENFLG_MAPSEG 0x0001 // map: generate map of segments</td>
</tr>
<tr>
<td>#define GENFLG_MAPNAME 0x0002 // map: include dummy names</td>
</tr>
<tr>
<td>#define GENFLG_MAPDMNG 0x0004 // map: demangle names</td>
</tr>
<tr>
<td>#define GENFLG_MAPLOC 0x0008 // map: include local names</td>
</tr>
<tr>
<td>#define GENFLG_IDCTYPE 0x0008 // idc: gen only information about types</td>
</tr>
</tbody>
</table>
5.12.6 save_database

**Definition**

```c
idaman void ida_export
save_database(const char *outfile, bool delete_unpacked)
```

**Synopsis**

Save the database to the file path, *outfile. If `delete_unpacked` is false, temporary unpacked files are not deleted. As this function doesn't return anything, there is no way to determine if the save was successful, except for testing whether the file exists after the function call is made.

**Example**

```c
#include <loader.hpp>

msg("Saving database...");
char *outfile = "c:\myidb.idb";
save_database(outfile, false);

// There was an error if the filesize is <= 0
if (qfilesize(outfile) <= 0)
  msg("failed.\n");
else
  msg("ok\n");
```

5.13 Flags

The functions below are for checking whether particular flags (see section 4.3) are set for a byte within the currently disassembled file(s). They are all defined in `bytes.hpp`. 
### 5.13.1 getFlags

| Definition         | idaman flags_t ida_export  
<table>
<thead>
<tr>
<th></th>
<th>getFlags(ea_t ea)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synopsis</strong></td>
<td>Returns the flags set for address ea. You will need to run this to obtain the flags for an address to then use with functions like isHead(), isCode(), etc.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| **Example**       | #include <kernwin.hpp> // For get_screen_ea() definition  
|                   | #include <bytes.hpp>     |
|                   | msg("Flags for %a are %08x\n",  
|                   |    get_screen_ea(),  
|                   |    getFlags(get_screen_ea())); |

### 5.13.2 isEnabled

| Definition         | idaman bool ida_export  
<table>
<thead>
<tr>
<th></th>
<th>isEnabled(ea_t ea)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synopsis</strong></td>
<td>Does the address, ea, exist within the currently disassembled file(s)?</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| **Example**       | #include <kernwin.hpp> // For askaddr() definition  
|                   | #include <bytes.hpp>    |
|                   | ea_t addr;              |
|                   | askaddr(&addr, "Address to look for:");     |
|                   | if (isEnabled(addr))    |
|                   |    msg("%a found within the currently opened file(s).",  
|                   |       addr);             |
|                   | else                    |
|                   |    msg("%a was not found.\n");           |

### 5.13.3 isHead

| Definition         | inline bool idaapi  
<table>
<thead>
<tr>
<th></th>
<th>isHead(flags_t F)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synopsis</strong></td>
<td>Does the flagset, F, denote the start of code or data?</td>
</tr>
</tbody>
</table>
```c
#include <kernwin.hpp> // For get_screen_ea() definition
#include <bytes.hpp>

ea_t addr = get_screen_ea();

// Cycle through 20 bytes from the cursor position
// printing a message if the byte is a head byte.
for (int i = 0; i < 20; i++) {
    flags_t flags = getFlags(addr);
    if (isHead(flags))
        msg("%a is a head (flags = %08x).\n",
            addr, flags);
    addr++;
}
```

### 5.13.4 isCode

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline bool idaapi isCode(flags_t F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Does the flagset, F, denote the start of an instruction? This is the same as isHead(), but only returns true for code, not data. Therefore, if used on a code byte that is not a head byte, it will return false.</td>
</tr>
</tbody>
</table>

```c
#include <segment.hpp> // For segment functions
#include <bytes.hpp>

for (int i = 0; i < get_segm_qty(); i++) {
    segment_t *seg = getnseg(i);
    if (seg->type == SEG_CODE) {
        // Look for any bytes in the code segment that aren't code.
        for (ea_t a = seg->startEA; a < seg->endEA; a++) {
            flags_t flags = getFlags(a);
            if (isHead(flags) && !isCode(flags))
                msg("Non-code at %a in segment: %s.\n",
                    a, 
                    get_segm_name(seg));
        }
    }
}
```

### 5.13.5 isData

<table>
<thead>
<tr>
<th>Definition</th>
<th>inline bool idaapi isData(flags_t F)</th>
</tr>
</thead>
</table>
### Synopsis

Does the flagset, \( F \), denote the start of some data? This is the same as \texttt{isHead()}, but only returns true for data, not code. Therefore, if used on a data byte that is not a head byte, it will return false.

#### Example

```cpp
#include <segment.hpp> // For segment functions
#include <bytes.hpp>

for (int i = 0; i < get_segm_qty(); i++) {
    segment_t *seg = getnseg(i);
    if (seg->type == SEG_DATA) {
        // Look for any bytes in the data segment that aren't data (possibly code).
        for (ea_t a = seg->startEA; a < seg->endEA; a++) {
            flags_t flags = getFlags(a);
            if (isHead(flags) && !isData(flags))
                msg("Non-data at %a in segment: %s.\n", a, get_segm_name(seg));
        }
    }
}
```

### 5.13.6 isUnknown

#### Definition

```cpp
inline bool idaapi isUnknown(flags_t F)
```

#### Synopsis

Does the flagset, \( F \), denote a byte that hasn't been successfully analysed by IDA?

#### Example

```cpp
#include <segment.hpp> // For segment functions
#include <bytes.hpp>

// Loop through every segment
for (int i = 0; i < get_segm_qty(); i++) {
    segment_t *seg = getnseg(i);
    // Look for any unexplored bytes in this segment
    for (ea_t a = seg->startEA; a < seg->endEA; a++) {
        flags_t flags = getFlags(a);
        if (isUnknown(flags))
            msg("Unknown bytes at %a in segment: %s.\n", a, get_segm_name(seg));
    }
}
```
5.14 Data

When working with a disassembled file, it can often be very useful to bypass the disassembler and work directly with the bytes in the binary file itself. IDA provides the functionality to do this with the below functions (plus some more). All of the below are defined in bytes.hpp. These functions work with bytes, however there are also functions to work with words, longs and qwords (get_word(), patch_word() and so on), which are also to be found in bytes.hpp. Aside from using these functions to read data from the binary file itself, they can also be used to read process memory while a process is executing under the debugger. More on this under the Debugger functions section.

5.14.1 get_byte

| Definition | idaman uchar ida_export  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>get_byte(ea_t ea)</td>
</tr>
</tbody>
</table>

| Synopsis | Returns the byte at address ea within the disassembled file(s) currently open in IDA. Returns BADADDR if ea doesn't exist. Also available for working with larger chunks is get_word(), get_long() and get_qword(). Use get_many_bytes() for working with multiple byte chunks. |

| Example | 
|---------|---|
| #include <kernwin.hpp> // For get_screen_ea() definition  
| #include <bytes.hpp>  
|  
| // Display the byte value for the current cursor  
| // position. The values returned should correspond  
| // to those in your IDA Hex view.  
| msg("%x\n", get_byte(get_screen_ea()));  |

5.14.2 get_many_bytes

| Definition | idaman bool ida_export  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>get_many_bytes(ea_t ea, void *buf, ssize_t size)</td>
</tr>
</tbody>
</table>

| Synopsis | Fetch size bytes starting at ea, and store them into *buf. |
```
#include <kernwin.hpp>  // For get_screen_ea() definition
#include <bytes.hpp>

char string[MAXSTR];
flags_t flags = getFlags(get_screen_ea());

// Only get a string if we're at actual data.
if (isData(flags)) {
    // Get a string from the binary
    get_many_bytes(get_screen_ea(),
                   string,
                   sizeof(string)-2);
    // NULL terminate the string, if not already
    // terminated in the binary (so strlen doesn't barf)
    string[MAXSTR-1] = '\0';
    msg("String length: %d\n", strlen(string));
}
```

### 5.14.3 patch_byte

#### Definition

`idaman void ida_export
patch_byte(ea_t ea, ulong x)`

#### Synopsis

Replace the byte at `ea` with `x`. The original byte is saved to the IDA database, and can be retrieved using `get_original_byte()` (see `bytes.hpp`). To not save the original byte, use `put_byte(ea_t ea, ulong x)` instead. Also available for working with larger chunks is `put_word()`, `put_long()` and `put_qword()`. Use `put_many_bytes()` for working with multiple byte chunks.

```
#include <kernwin.hpp>  // For get_screen_ea()
#include <bytes.hpp>

// Get the flags for the byte at the cursor position.
flags_t flags = getFlags(get_screen_ea());

// Replace the instruction at the cursor position with
// a NOP instruction (0x90).
// Unless used carefully, your executable will probably
// not work correctly after this :-)
if (isCode(flags))
    patch_byte(get_screen_ea(), 0x90);
```

### 5.14.4 patch_many_bytes

#### Definition

`idaman void ida_export
patch_many_bytes(ea_t ea, const void *buf, size_t size)`
Replace size bytes at ea with the contents of *buf.

Example

```c
#include <kernwin.hpp> // For get_screen_ea() et al
#include <bytes.hpp>

// Prompt the user for an address, then a string
eal t addr = get_screen_ea();
askaddr(&addr, "Address to put string:");
char *string = askstr(0, ",", "Please enter a string");

// Write the user supplied string to the address
// the user specified.
patch_many_bytes(addr, string, strlen(string));
```

5.15 I/O

As mentioned in section 5.1, a lot of standard C library functions for I/O have IDA SDK equivalents, and it's recommended you use them instead of their standard C counterparts. These are all defined in `diskio.hpp`.

5.15.1 fopenWT

**Definition**

```c
idaman FILE *ida_export
fopenWT(const char *file)
```

**Synopsis**

Open the text file, `*file`, in write mode, return a FILE pointer or NULL if opening the file failed. To open the file in read mode, use `fopenRT()`, and for binary files, replace the R with W. For read/write, use `fopenM()`.

**Example**

```c
#include <diskio.hpp>
FILE *fp = fopenWT("c:\temp\txtfile.txt");
if (fp == NULL)
    warning("Failed to open output file.");
```

5.15.2 openR

**Definition**

```c
idaman FILE *ida_export
openR(const char *file)
```

**Synopsis**

Open the binary file, `*file`, in read-only mode, return a FILE pointer or exit (display an error and close IDA) if it fails. To open a text file in read-only mode, exiting on failure, use `fopenRT()`, for read-write use `fopenM()`.
### 5.15.3 ecreate

**Definition**  
=idaman FILE *ida_export  
ecreate(const char *file)

**Synopsis**  
Create the binary file, *file, returning a FILE pointer of the file for write only. Displays an error and exits if it is unable to create the file. To create a text file, use ecreateT().

**Example**  
```c
#include <diskio.hpp>
FILE *fp = ecreate("c:\temp\newbinfile.exe");
```

### 5.15.4 eclose

**Definition**  
=idaman void ida_export  
eclose(FILE *fp)

**Synopsis**  
Closes the file represented by FILE pointer *fp. Displays an error and exits if it is unable to close the file.

**Example**  
```c
#include <diskio.hpp>
// Open the file first.
FILE *fp = openR("c:\temp\binfile.exe");

// Close it  
eclose(fp);
```

### 5.15.5 eread

**Definition**  
=idaman void ida_export  
eread(FILE *fp, void *buf, ssize_t size)

**Synopsis**  
Read size bytes from file represented by FILE pointer *fp, into buffer *buf. If the read is unsuccessful, an error is displayed followed by exiting IDA.
```c
#include <diskio.hpp>
char buf[MAXSTR];

// Open the text file
FILE *fp = openRT("c:\temp\txtfile.txt");

// Read MAXSTR bytes from the start of the file.
eread(fp, buf, MAXSTR-1);
eclose(fp);
```

### 5.15.6 `ewriter`

#### Definition

```c
idaman void ida_export
erwrite(FILE *fp, const void *buf, ssize_t size)
```

#### Synopsis

Write `size` bytes of `*buf` to the file represented by `FILE` pointer `*fp`. If the write operation fails, an error is displayed followed by exiting IDA.

```c
#include <kernwin.hpp> // For read_selection()
#include <bytes.hpp>   // For get_many_bytes()
#include <diskio.hpp>

char buf[MAXSTR];
ea_t saddr, eaddr;

// Create the binary dump file
FILE *fp = ecreate("c:\\bindump");

// Get the address range selected, or return false if there was no selection
if (read_selection(&saddr, &eaddr)) {
    int size = eaddr - saddr;
    // Dump the selected address range to a binary file
    get_many_bytes(saddr, buf, size);
    ewrite(fp, buf, size);
}
eclose(fp);
```

### 5.16 Debugging

Unlike most of the functions covered so far, the next three sections are for working with a binary during execution. This section in particular is for high level operations (like process and thread control) on a binary/process. Debugging and tracing is covered in the following two sections. All functions below are defined in `dbg.hpp` with the exception of `invalidate_dbg_contents()` and `invalidate_dbg_config()`, which are defined in `bytes.hpp`. To get the most out of the examples, you should run them (i.e. invoke your plug-in) whilst a binary is being debugged in IDA.
You will probably notice that all of these functions aren't prefixed with ida_export. They don't need to be because they are all inlined wrappers to callui(), and use event notifications to carry out their respective functionality.

5.16.0 A Note on Requests

Unlike most functions in the SDK, most debugger functions (and some tracing functions too) come in two forms; their normal asynchronous form, for example run_to(), and a synchronous, or request form, like request_run_to(). Both forms of the function will take the same arguments, but it's the way they carry out the respective operation that makes the difference.

The synchronous form of the function (request_) will enter the function into a queue, and eventually be executed by IDA when you call run_requests(). The other, asynchronous form, will run straight away, just like a normal function.

The synchronous form of a function can be very handy when you want to queue a bunch of things to be run by IDA in one hit. 5.17.5 is a good example of this, where deleting a bunch of breakpoints using del_bpt() would fail unless done synchronously, as the ID number of the breakpoints would be re-organised by the time you went to fetch the next one using getn_bpt(). Something important worth noting is that you must use the synchronous form of a function when you are in an debugger event notification handler (see section 4.5, specifically 4.5.3).

All functions in sections 5.16, 5.17 and 5.18 that are also available as requests will have a * following the function name.

5.16.1 run_requests

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi run_requests(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Runs any requests (synchronous functions) that have been queued.</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>// Run to the entry point of the binary request_run_to(inf.startIP);</td>
</tr>
<tr>
<td></td>
<td>// Enable function tracing request_enable_func_trace();</td>
</tr>
<tr>
<td></td>
<td>// Run the above requests run_requests();</td>
</tr>
</tbody>
</table>

5.16.2 get_process_state

| Definition       | int idaapi get_process_state(void) |
### Synopsis

Returns the state of the process currently being debugged. If the process is suspended, -1 is returned, 1 if the process is running or 0 if there is no process running under the debugger.

```c
#include <dbg.hpp>

switch (get_process_state()) {
    case 0:
        msg("No process running.\n");
        break;
    case -1:
        msg("Process is suspended.\n");
        break;
    case 1:
        msg("Process is running.\n");
        break;
    default:
        msg("Unknown status.\n");
}
```

### 5.16.3 get_process_qty

<table>
<thead>
<tr>
<th>Definition</th>
<th>int idaapi get_process_qty(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the number of running processes matching the image of the executable currently open in IDA. This function also needs to be called to initialise the process snapshot, which is used by IDA for populating data structures utilised by other process-related functions.</td>
</tr>
</tbody>
</table>

```c
#include <dbg.hpp>

msg("There are %d processes running.\n",
    get_process_qty());
```

### 5.16.4 get_process_info

<table>
<thead>
<tr>
<th>Definition</th>
<th>process_id_t idaapi get_process_info(int n, process_info_t *process_info);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Populate *process_info with information about process number n (this is not the PID). The process ID of the process number n is returned. If *process_info is NULL, only the PID of the process is returned.</td>
</tr>
</tbody>
</table>
```c
#include <dbg.hpp>

// Only get the info if a process is actually running..
if (get_process_qty() > 0) {
    process_info_t pif;
    // Populate pif
    get_process_info(0, &pif);
    msg("ID: %d, Name: %s\n", pif.pid, pif.name);
} else {
    msg("No process running!\n");
}
```

### 5.16.5 `start_process` *

<table>
<thead>
<tr>
<th>Definition</th>
<th>int idaapi start_process(const char *path = NULL, const char *args = NULL, const char *sdir = NULL);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Start debugging the process *path, with the arguments *args, in the directory *sdir. If any of the arguments are NULL, they are taken from the process options specified under <strong>Debugger-&gt;Process Options</strong>…. This is essentially the same as pressing F9 in IDA.</td>
</tr>
</tbody>
</table>

```c
#include <kernwin.hpp> // For askstr()
#include <dbg.hpp>

// Ask the user for arguments to supply.
char *args = askstr(HIST_IDENT, "", "Arguments");

// Run the process with those arguments
start_process(NULL, args, NULL);
```

### 5.16.6 `continue_process` *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi continue_process(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Continue the execution of a process. Returns false if continuing the process fails. This is equivalent to pressing F9 in IDA when a process is in the suspended state (breakpoint-hit or suspended).</td>
</tr>
</tbody>
</table>

```c
#include <dbg.hpp>

// Continue running the process when the user // invokes this plug-in.
if (continue_process())
    msg("Continuing process..\n");
else
    msg("Failed to continue process execution.\n");
```
### 5.16.7 suspend_process *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi suspend_process(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Suspend the process currently being debugged. Returns false if suspending the process failed. This is the same as pressing the 'Pause Process' button in IDA.</td>
</tr>
</tbody>
</table>
| Example      | #include <dbg.hpp>  
                // Suspend the process being debugged.  
                if (suspend_process())  
                    msg("Suspended process.\n");  
                else  
                    msg("Failed to suspend process.\n"); |

### 5.16.8 attach_process *

<table>
<thead>
<tr>
<th>Definition</th>
<th>int idaapi attach_process(process_id_t pid=NO_PROCESS, int event_id=-1)</th>
</tr>
</thead>
</table>
| Synopsis     | Attach to the process with PID pid. The process being attached to must be the same executable image as the one currently being disassembled in IDA. If the pid argument is NO_PROCESS, the user is prompted with a list of potential processes to attach to. The possible return codes are as follows, which is taken from dbg.hpp:  
                // -2 - impossible to find a compatible process  
                // -1 - impossible to attach to the given process  
                // (process died, privilege needed, not supported by the debugger plugin, ...)  
                // 0 - the user cancelled the attaching to the process  
                // 1 - the debugger properly attached to the process |
| Example      | #include <dbg.hpp>  
                // Present the user with a list of processes to attach to. If there is no executable running that matches what's open in IDA, no dialog box will be presented.  
                int err;  
                if ((err = attach_process(NO_PROCESS)) == 1)  
                    msg("Successfully attached to process.\n");  
                else  
                    msg("Unable to attach, error: %d\n", err); |
### 5.16.9 *detach_process* *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi detach_process(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Detach from the process currently being debugged. This can be a process that was attached to or run through IDA. Returns false if it was unable to detach. Detaching from a process is only supported on Windows XP SP2 and Windows 2003.</td>
</tr>
<tr>
<td>Example</td>
<td></td>
</tr>
</tbody>
</table>

```c
#include <dbg.hpp>

// Detach from the debugged process.
if (detach_process())
    msg("Successfully detached from process.\n");
else
    msg("Failed to detach.\n");
```

### 5.16.10 *exit_process* *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi exit_process(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Terminate the process currently being debugged. Returns false if it was unable to terminate the process.</td>
</tr>
<tr>
<td>Example</td>
<td></td>
</tr>
</tbody>
</table>

```c
#include <dbg.hpp>

// Terminate the debugged process.
if (exit_process())
    msg("Successfully terminated the process.\n");
else
    msg("Failed to terminate the process.\n");
```

### 5.16.11 *get_thread_qty*

<table>
<thead>
<tr>
<th>Definition</th>
<th>int idaapi get_thread_qty(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the number of threads that exist in the debugged process.</td>
</tr>
<tr>
<td>Example</td>
<td></td>
</tr>
</tbody>
</table>

```c
#include <dbg.hpp>

// Only display if there is a process being debugged.
if (get_process_qty() > 0)
    msg("Threads running: %d\n", get_thread_qty());
```
### 5.16.12 `get_reg_val`

| Definition | bool idaapi  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>get_reg_val(const char *regname, regval_t *regval)</code></td>
</tr>
<tr>
<td>Synopsis</td>
<td>Get the value stored in register <code>*regname</code> and store it in <code>*regval</code>. Returns false if it was unable to retrieve the value from the register. The register name is case insensitive.</td>
</tr>
</tbody>
</table>
| Example    | #include <dbg.hpp>  
|            | // Process needs to be suspended for this to work.  
|            | regval_t eax;  
|            | regval_t eax_upper;  
|            | char *regname = "eax";  
|            | char *regname_upper = "EAX";  
|            | // Prooving that the register name is case insensitive  
|            | if (get_reg_val(regname, &eax))  
|            | msg("eax = %08a\n", eax.ival);  
|            | if (get_reg_val(regname_upper, &eax_upper))  
|            | msg("EAX = %08a\n", eax_upper.ival);  

### 5.16.13 `set_reg_val`

| Definition | bool idaapi  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>set_reg_val(const char *regname, const regval_t *regval)</code></td>
</tr>
<tr>
<td>Synopsis</td>
<td>Set the register <code>*regname</code> to value <code>*regval</code> in the current thread. If the write fails, false is returned. Like <code>get_reg_val()</code>, <code>*regname</code> is case insensitive. Unlike other asynchronous functions, this is safe to call from a debug event notification handler.</td>
</tr>
</tbody>
</table>
| Example    | #include <kernwin.hpp>  
|            | // For `get_screen_ea()` definition  
|            | #include <dbg.hpp>  
|            | // Suspend the currently executing process.  
|            | suspend_process();  
|            | // Continue execution from the user's cursor position.  
|            | ea_t addr = get_screen_ea();  
|            | char *regname = "EIP";  
|            | if (set_reg_val(regname, addr)) {  
|            | msg("Continuing execution from %a\n", addr);  
|            | continue_process();  
|            | }  

### 5.16.14 invalidate_dbgmem_contents

<table>
<thead>
<tr>
<th>Definition</th>
<th>Definition of invalidate_dbgmem_contents function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>idaman</strong> void ida_export invalidate_dbgmem_contents(ea_t ea, asize_t size)</td>
<td></td>
</tr>
</tbody>
</table>

#### Synopsis

Invalidate `size` bytes of memory, starting at `ea`. If you want to invalidate the whole of a processes memory, set `ea` to `BADADDR` and `size` to 0.

Invalidating memory contents is essentially flushing the IDA kernel's memory cache for a process, which ensures you are accessing the latest memory contents from a processes memory. You should call this function after a process is suspended, or if you suspect the memory contents have changed.

#### Example

```c
#include <dbg.hpp>
#include <bytes.hpp>

// Process must be suspended for this to work

// Get the address stored in the ESP register
regval_t esp;
get_reg_val("ESP", &esp);

// Get the value at the address stored in the ESP reg.
uchar before = get_byte(esp.ival);

// Invalidate memory contents
invalidate_dbgmem_contents(BADADDR, 0);

// Re-fetch contents of the address stored in ESP
uchar after = get_byte(esp.ival);

msg("%08a: Before: %a, After: %a\n", 
    esp.ival, before, after);
```

### 5.16.15 invalidate_dbgmem_config

<table>
<thead>
<tr>
<th>Definition</th>
<th>Definition of invalidate_dbgmem_config function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>idaman</strong> void ida_export invalidate_dbgmem_config(void)</td>
<td></td>
</tr>
</tbody>
</table>

#### Synopsis

Like `invalidate_dbgmem_contents()`, you use this function to ensure IDA is looking at the latest memory configuration. You need to run this function if the debugged process has allocated or deallocated memory since it was last suspended. This function also flushes the IDA memory cache, however is much slower than `invalidate_dbgmem_contents()`.
5.16.16 run_to *

**Definition**

`bool idaapi run_to(ea_t ea)`

**Synopsis**

Run the process until execution gets to address `ea`. If there is no process running, the currently disassembled file is executed. Returns false if it was unable to execute the process.

**Example**

```
#include <kernwin.hpp> // For get_screen_ea() definition
#include <dbg.hpp>

// Replicate F4 functionality
if (!run_to(get_screen_ea()))
    msg("Failed to run to %a\n", get_screen_ea());
```

5.16.17 step_into *

**Definition**

`bool idaapi step_into(void)`

**Synopsis**

Run one instruction within the current thread of the debugged process. This is the same as F7 in IDA. Returns false if it was unable to step into the instruction.
### 5.16.18 step_over *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi step_over(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Run one instruction within the current thread of the debugged process, but don’t step into functions, treat them as one instruction. This is the same as F8 in IDA. Returns false if it was unable to step over the instruction.</td>
</tr>
</tbody>
</table>
| Example             | `#include <dbg.hpp>`
|                     | `// This can only run when the process is suspended`  
|                     | `// Step over 5 instructions. This needs to be done as`  
|                     | `// a request, otherwise only one step will execute.`  
|                     | `for (int i = 0; i < 5; i ++)`  
|                     | `request_step_over();`  
|                     | `run_requests();` |

### 5.16.19 step_until_ret *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi step_until_ret(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Execute each instruction in the current thread of the debugged process until the current function returns. This is the same as CTRL-F7 in IDA.</td>
</tr>
</tbody>
</table>
5.17 Breakpoints

An essential part of debugging is having the ability to set and manipulate breakpoints, which can be set on any address within a process memory space and be hardware or software breakpoints. The following set of functions work with breakpoints, and are defined in dbg.hpp.

5.17.1 get_bpt_qty

<table>
<thead>
<tr>
<th>Definition</th>
<th>int idaapi get_bpt_qty(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Return the current number of breakpoints that exist (regardless of whether they are enabled or not).</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>msg(&quot;There are currently %d breakpoints set.\n&quot;, get_bpt_qty());</td>
</tr>
</tbody>
</table>

5.17.2 getn_bpt

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi getn_bpt(int n, bpt_t *bpt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Fill *bpt with information about breakpoint number n. Returns false if there is no such breakpoint number.</td>
</tr>
</tbody>
</table>
#include <dbg.hpp>

// Go through all breakpoints, displaying the address
// of where they are set.
for (int i = 0; i < get_bpt_qty(); i++) {
    bpt_t bpt;
    if (getn_bpt(i, &bpt))
        msg("Breakpoint found at %a\n", bpt.ea);
}

5.17.3 get_bpt

Definition

bool idaapi
get_bpt(ea_t ea, bpt_t *bpt)

Synopsis

Fill *bpt with information about the breakpoint set at ea. If no breakpoint is set
at ea, false is returned. If *bpt is NULL, this function simply returns true or
false depending if a breakpoint is set at ea.

Example

#include <kernwin.hpp> // For get_screen_ea() definition
#include <dbg.hpp>
if (get_bpt(get_screen_ea(), NULL))
    msg("Breakpoint is set at %a.\n", get_screen_ea());
else
    msg("No breakpoint set at %a.\n", get_screen_ea());

5.17.4 add_bpt *

Definition

bool idaapi
add_bpt(ea_t ea, asize_t size = 0, bpttype_t type = BPT_SOFT)

Synopsis

Add a breakpoint at ea of type type and size size. Returns false if it was
unable to set the breakpoint. Refer to section 4.4.2 for an explanation of
different breakpoint types. size is irrelevant when setting software
breakpoints.
5.17.5 del_bpt *

**Definition**

`bool idaapi del_bpt(ea_t ea)`

**Synopsis**

Delete the breakpoint defined at `ea`. If there is no breakpoint defined there, returns false.

**Example**

```c
#include <dbg.hpp>

// Go through all breakpoints, deleting each one.
for (int i = 0; i < get_bpt_qty(); i++) {
    bpt_t bpt;
    if (getn_bpt(i, &bpt)) {
        // Because we are performing many delete
        // operations, queue the request, otherwise the
        // getn_bpt call will fail when the id
        // numbers change after the delete operation.
        if (request_del_bpt(bpt.ea))
            msg("Queued deleting breakpoint at %a\n", bpt.ea);
    }
}

// Run through request queue
run_requests();

// Make sure there are no breakpoints left over
if (get_bpt_qty() > 0)
    msg("Failed to delete all breakpoints.\n");
```

5.17.6 update_bpt

**Definition**

`bool idaapi update_bpt(const bpt_t *bpt)`
## Synopsis
Update modifiable elements of the breakpoint represented by *bpt. Returns false if the modification was unsuccessful.

```cpp
#include <dbg.hpp>

// Loop through all breakpoints
for (int i = 0; i < get_bpt_qty(); i++) {
    bpt_t bpt;
    if (getn_bpt(i, &bpt)) {
        // Change the breakpoint to not pause
        // execution when it's hit
        bpt.flags ^= BPT_BRK;

        // Change the breakpoint to a trace breakpoint
        bpt.flags |= BPT_TRACE;

        // Run a little IDC every time it's hit
        qstrncpy(bpt.condition,
                "Message(""Trace hit!"")",
                sizeof(bpt.condition));

        // Update the breakpoint
        if (!update_bpt(&bpt))
            msg("Failed to update breakpoint at %a\n",
                 bpt.ea);
    }    
}
```

### 5.17.7 enable_bpt *

#### Definition
`bool idaapi enable_bpt(ea_t ea, bool enable = true)`

#### Synopsis
Enable or disable the breakpoint set at ea. If no breakpoint is defined at ea, or there was an error enabling/disabling the breakpoint, false is returned. If enable is set to false, the breakpoint is disabled.

```cpp
#include <kernwin.hpp> // For get_screen_ea() definition
#include <dbg.hpp>

bpt_t bpt;

// If a breakpoint exists at the user's cursor, disable // it.
if (get_bpt(get_screen_ea(), &bpt)) {
    if (enable_bpt(get_screen_ea(), false))
        msg("Disabled breakpoint.\n");
}
```
5.18 Tracing

The functions available for tracing mostly revolve around checking whether a certain type of tracing is enabled, enabling or disabling a type of tracing and retrieving trace events. All the below are defined in dbg.hpp.

### 5.18.1 set_trace_size

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi set_trace_size(int size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Set the tracing buffer size to size. Returns false if there was an error allocating size. Setting size to 0 sets an unlimited buffer size (dangerous). If you set size to a value lower than the current number of trace events, size events are deleted.</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>// 1000 trace events allowed</td>
</tr>
<tr>
<td></td>
<td>if (set_trace_size(1000))</td>
</tr>
<tr>
<td></td>
<td>msg(&quot;Successfully set the trace buffer to 1000\n&quot;);</td>
</tr>
</tbody>
</table>

### 5.18.2 clear_trace *

<table>
<thead>
<tr>
<th>Definition</th>
<th>void idaapi clear_trace(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Clear the trace buffer.</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>// Start our plug-in with a clean slate</td>
</tr>
<tr>
<td></td>
<td>clear_trace();</td>
</tr>
</tbody>
</table>

### 5.18.3 is_step_trace_enabled

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi is_step_trace_enabled(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns true if step tracing is currently enabled.</td>
</tr>
</tbody>
</table>
### 5.18.4 enable_step_trace *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi enable_step_trace(int enable = true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Enable step tracing. If <code>enable</code> is set to false, step tracing is disabled.</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>// Toggle step tracing</td>
</tr>
<tr>
<td></td>
<td>if (is_step_trace_enabled())</td>
</tr>
<tr>
<td></td>
<td>msg(&quot;Step tracing is enabled.\n&quot;);</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>enable_step_trace(false);</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>enable_step_trace();</td>
</tr>
</tbody>
</table>

### 5.18.5 is_insn_trace_enabled

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi is_insn_trace_enabled(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns true if instruction tracing is enabled.</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>if (is_insn_trace_enabled())</td>
</tr>
<tr>
<td></td>
<td>msg(&quot;Instruction tracing is enabled.\n&quot;);</td>
</tr>
</tbody>
</table>

### 5.18.6 enable_insn_trace *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi enable_insn_trace(int enable = true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Enable instruction tracing. If <code>enable</code> is set to false, instruction tracing is disabled.</td>
</tr>
</tbody>
</table>

### 5.18.7 is_func_trace_enabled

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi is_func_trace_enabled(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns true if function tracing is enabled.</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>if (is_func_trace_enabled())</td>
</tr>
<tr>
<td></td>
<td>msg(&quot;Function tracing is enabled.\n&quot;);</td>
</tr>
</tbody>
</table>

### 5.18.8 enable_func_trace *

<table>
<thead>
<tr>
<th>Definition</th>
<th>bool idaapi enable_func_trace(int enable = true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Enable function tracing. If <code>enable</code> is set to false, function tracing is disabled.</td>
</tr>
<tr>
<td>Example</td>
<td>#include &lt;dbg.hpp&gt;</td>
</tr>
<tr>
<td></td>
<td>// Toggle function tracing</td>
</tr>
<tr>
<td></td>
<td>if (is_func_trace_enabled())</td>
</tr>
<tr>
<td></td>
<td>enable_func_trace(false);</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>enable_func_trace();</td>
</tr>
</tbody>
</table>

### 5.18.9 get_tev_qty

<table>
<thead>
<tr>
<th>Definition</th>
<th>int idaapi get_tev_qty(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the number of trace events in the trace buffer.</td>
</tr>
</tbody>
</table>
### 5.18.10 get_tev_info

**Definition**

```c
bool idaapi
get_tev_info(int n, tev_info_t *tev_info)
```

**Synopsis**
Fills *tev_info about the trace buffer entry number n. Returns false if there is no such trace event number n.

**Example**

```
#include <dbg.hpp>

// Loop through all trace events
for (int i = 0; i < get_tev_qty(); i++) {
    tev_info_t tev;
    // Get the trace event information
    get_tev_info(i, &tev);

    // Display the address the event took place
    msg("Trace event occurred at %a\n", tev.ea);
}
```

### 5.18.11 get_insn_tev_reg_val

**Definition**

```c
bool idaapi
get_insn_tev_reg_val(int n, const char *regname, regval_t *regval)
```

**Synopsis**
Store the value of register *regname into *regval when instruction trace event number n happened, before execution of the instruction. Returns false if the event wasn't an instruction trace event.

See get_insn_tev_reg_result() for obtaining registers after execution.
```cpp
#include <dbg.hpp>

// Loop through all trace events
for (int i = 0; i < get_tev_qty(); i++) {
    regval_t esp;
    tev_info_t tev;

    // Get the trace event information
    get_tev_info(i, &tev);

    // If it's an instruction trace event...
    if (tev.type == tev_insn) {
        // Get ESP, store into &esp
        if (get_insn_tev_reg_result(i, "ESP", &esp))
            // Display the value of ESP
            msg("TEV #%d after exec: %a\n", i, esp.ival);
        else
            msg("No ESP change for TEV #%d\n", i);
    }
}
```

### 5.18.12 `get_insn_tev_reg_result`

**Definition**
```cpp
bool idaapi
get_insn_tev_reg_result(int n, const char *regname,
regval_t *regval)
```

**Synopsis**
Store the value of register `*regname` into `*regval` when instruction trace event number `n` happened, after execution of the instruction. Returns false if the register wasn't modified or `n` doesn't represent an instruction trace event.

See `get_insn_tev_reg_val()` for obtaining registers before execution.

```cpp
#include <dbg.hpp>

// Loop through all trace events
for (int i = 0; i < get_tev_qty(); i++) {
    regval_t esp;
    tev_info_t tev;

    // Get the trace event information
    get_tev_info(i, &tev);

    // If it's an instruction trace event...
    if (tev.type == tev_insn) {
        // Get ESP, store into &esp
        if (get_insn_tev_reg_result(i, "ESP", &esp))
            // Display the value of ESP
            msg("TEV #%d after exec: %a\n", i, esp.ival);
        else
            msg("No ESP change for TEV #%d\n", i);
    }
}
```
### 5.18.13 get_call_tev_callee

| Definition | `ea_t idaapi
get_call_tev_callee(int n)` |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the address of the function called for function trace event number n. Returns BADADDR if there is no such function trace event number n. The type of the function trace event must be tev_call.</td>
</tr>
</tbody>
</table>
| Example    | #include <dbg.hpp>

// Loop through all trace events
for (int i = 0; i < get_tev_qty(); i++) {
    regval_t esp;
    tev_info_t tev;

    // Get the trace event information
    get_tev_info(i, &tev);

    // If it's an function call trace event...
    if (tev.type == tev_call) {
        ea_t addr;
        // Get ESP, store into &esp
        if ((addr = get_call_tev_callee(i)) != BADADDR)
            msg("Function at %a was called\n", addr);
    }
}

### 5.18.14 get_ret_tev_return

| Definition | `ea_t idaapi
get_ret_tev_return(int n)` |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>Returns the address of the calling function for function trace event number n. Returns BADADDR if there is no such function trace event number n. The type of the function trace event must be tev_ret.</td>
</tr>
</tbody>
</table>
| Example    | #include <dbg.hpp>

// Loop through all trace events
for (int i = 0; i < get_tev_qty(); i++) {
    tev_info_t tev;

    // Get the trace event information
    get_tev_info(i, &tev);

    // If it's an function return trace event...
    if (tev.type == tev_ret) {
ea_t addr;
if ((addr = get_ret_tev_return(i)) != BADADDR)
    msg("Function returned to %a\n", addr);
}
}

5.18.15 get_bpt_tev_ea

| Definition          | ea_t idaapi
                   | get_bpt_tev_ea(int n) |
|---------------------|---------------------|
| Synopsis            | Returns the address of the read/write/execution trace number n. Returns false if the trace event wasn't that of a read/write/execution trace. |
| Example             | #include <dbg.hpp>
                   | // Loop through all trace events
                   |   for (int i = 0; i < get_tev_qty(); i++) {
                   |       tev_info_t tev;
                   |       // Get the trace event information
                   |       get_tev_info(i, &tev);
                   |       // If it's an breakpoint trace event...
                   |       if (tev.type == tev_bpt) {
                   |           ea_t addr;
                   |           if ((addr = get_bpt_tev_ea(i)) != BADADDR)
                   |               msg("Breakpoint trace hit at %a\n", addr);
                   |       }
                   |   }

5.19 Strings

The following functions are used for reading the list of strings in IDA's Strings window, which is derived from strings found in the currently disassembled file(s). The below functions are defined in strlist.hpp.

5.19.1 refresh_strlist

| Definition          | idaman void ida_export
                   | refresh_strlist(ea_t ea1, ea_t ea2) |
|---------------------|---------------------|
| Synopsis            | Refresh the list of strings in IDA's Strings window. Search between ea1 and ea2 in the currently disassembled file(s) for these strings. |
5.19.2 get_strlist_qty

**Definition**

```c
idaman size_t ida_export
get_strlist_qty(void)
```

**Synopsis**

Returns the number of strings found in the currently disassembled file(s).

**Example**

```c
#include <strlist.hpp>
msg("%d strings were found in the currently open file(s)\n",
    get_strlist_qty());
```

5.19.3 get_strlist_item

**Definition**

```c
idaman bool ida_export
get_strlist_item(int n, string_info_t *si)
```

**Synopsis**

Fills `*si` with information about string number `n`. Returns false if there is no such string number `n`.

**Example**

```c
#include <strlist.hpp>
int largest = 0;

// Loop through all strings, finding the largest one.
for (int i = 0; i < get_strlist_qty(); i++) {
    string_info_t si;
    get_strlist_item(i, &si);
    if (si.length > largest)
        largest = si.length;
}

msg("Largest string is %d characters long.\n", largest);
```

5.20 Miscellaneous

These are functions that don’t really fit into any particular category. The headers they are defined in are mentioned in each case.
### 5.20.1 tag_remove

**Definition**

```c
idaman int ida_export
tag_remove(const char *instr, char *buf, int bufsize)
```

**Synopsis**

Remove any colour tags from `*instr`, and store the result in `*buf`, limited by `bufsize`. Supplying the same pointer for `*instr` and `*buf` is also supported, in which case `bufsize` is 0. This function is defined in `lines.hpp`.

**Example**

```c
#include <ua.hpp>   // For ua_ functions
#include <lines.hpp>

// Get the entry point address
ea_t addr = inf.startIP;

// Fill cmd with information about the instruction
// at the entry point
ua_ana0(addr);

// Loop through each operand (until one of o_void type
// is reached), displaying the operand text.
for (int i = 0; cmd.Operands[i].type != o_void; i++) {
    char op[MAXSTR];
    ua_outop(addr, op, sizeof(op)-1, i);
    // Strip the colour tags off
    tag_remove(op, op, 0);
    msg("Operand %d: %s\n", i, op);
}
```

### 5.20.2 open_url

**Definition**

```c
inline void
open_url(const char *url)
```

**Synopsis**

Opens `*url` in the system default web browser. This function is defined in `kernwin.hpp`.

**Example**

```c
#include <kernwin.hpp>

open_url("http://www.binarypool.com/idapluginwriting/");
```

### 5.20.3 call_system

**Definition**

```c
idaman int ida_export
call_system(const char *command)
```

**Synopsis**

Runs the command, `*command`, from a system shell. This function is defined in `diskio.hpp`. 
5.20.4 idadir

**Definition**

```c
idaman const char *ida_export
idadir(const char *subdir)
```

**Synopsis**

Returns the IDA path if `*subdir` is NULL. If `*subdir` is not NULL, the IDA sub-directory path is returned. These are the possible sub-directories, as taken from `diskio.hpp`:

```c
#define CFG_SUBDIR "cfg"
#define IDC_SUBDIR "idc"
#define IDS_SUBDIR "ids"
#define IDP_SUBDIR "procs"
#define LDR_SUBDIR "loaders"
#define SIG_SUBDIR "sig"
#define TIL_SUBDIR "til"
```

This function is defined in `diskio.hpp`.

**Example**

```c
#include <diskio.hpp>
msg("IDA directory is %s
", idadir(NULL));
```

5.20.5 getdspace

**Definition**

```c
idaman ulonglong ida_export
getdspace(const char *path)
```

**Synopsis**

Returns the amount of disk space available on the disk hosting `*path`. This function can be found in `diskio.hpp`.

**Example**

```c
#include <diskio.hpp>
// Get the disk space on the disk with IDA installed on // it.
if (getdspace(idadir(NULL)) < 100*1024*1024)
    msg("You need at least 100 MB free to run this.");
```
### 5.20.6 str2ea

**Definition**

<table>
<thead>
<tr>
<th>idaman bool  ida_export</th>
</tr>
</thead>
<tbody>
<tr>
<td>str2ea(const char *p, ea_t *ea, ea_t screenEA)</td>
</tr>
</tbody>
</table>

**Synopsis**

Convert the string *p to an address stored in *ea if it exists within the currently disassembled file(s), return true on success. This function is defined in kernwin.hpp.

**Example**

```c
#include <kernwin.hpp>

// Just some random address
char *addr_s = "010100F0";
ea_t addr;

// If 010100F0 is in the binary, print the address
if (str2ea(addr_s, &addr, 0))
    msg("Address: %a\n", addr);
```

### 5.20.7 ea2str

**Definition**

<table>
<thead>
<tr>
<th>idaman char *ida_export</th>
</tr>
</thead>
<tbody>
<tr>
<td>ea2str(ea_t ea, char *buf, int bufsize)</td>
</tr>
</tbody>
</table>

**Synopsis**

Convert the address, ea, to string, stored in *buf, limited by bufsize. The format of the string produced is segmentname:address, so for example, supplying the 0100102A address from the .text segment would produce .text:0100102A. This function is defined in kernwin.hpp.

**Example**

```c
#include <kernwin.hpp>
ea_t addr = get_screen_ea();
char addr_s[MAXSTR];

// Convert addr into addr_s
ea2str(addr, addr_s, sizeof(addr_s)-1);
msg("Address: %s\n", addr_s);
```

### 5.20.8 get_nice_colored_name

**Definition**

<table>
<thead>
<tr>
<th>idaman ssize_t ida_export</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_nice_colored_name(ea_t ea, char *buf, size_t bufsize, int flags=0);</td>
</tr>
</tbody>
</table>

**Synopsis**

Get the formatted name of ea, store it in *buf limited by bufsize. If flags is set to GNCN_NOCOLOR, no colour codes will be included in the name. If ea doesn't have a name, its address will be returned in a "human readable" form, like start+56 or .text:01002010 for example. This function is defined in name.hpp.
Example

```c
#include <kernwin.hpp> // For get_screen_ea() definition
#include <name.hpp>

char buf[MAXSTR];

// Get the nicely formatted name/address of the
// current cursor position. No colour codes will
// be included.
get_nice_colored_name(get_screen_ea(),
    buf,
    sizeof(buf)-1,
    GNCN_NOCOLOR);

msg("Name at cursor position: %s\n", buf);
```
6 Examples

The below examples have been included to provide a bit of context to the use of the structures and functions covered in this tutorial. All are extensively commented and will compile as-is, i.e. not requiring any modification or inclusion of headers, etc. like previous examples did.

The code for each of the below is also available at [http://www.binarypool.com/idapluginwriting/](http://www.binarypool.com/idapluginwriting/).

6.1 Looking for Calls to `sprintf`, `strcpy`, and `sscanf`

The below example will find “low hanging fruit” when auditing a binary. It does this by finding calls to usually misused functions like `sprintf`, `strcpy` and `sscanf` (feel free to add more of your choosing). It first finds the address of the extern definitions of these functions, then uses IDA’s cross referencing functionality to find all the addresses within the binary that reference those extern definitions.

```cpp
//
// unsafefunc.cpp
//

#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <lines.hpp>
#include <name.hpp>

int IDAP_init(void)
{
    if(inf.filetype != f_ELF && inf.filetype != f_PE)    {
        error("Executable format must be PE or ELF, sorry.");
        return PLUGIN_SKIP;
    }
    return PLUGIN_KEEP;
}

void IDAP_term(void)
{
    return;
}

void IDAP_run(int arg)
{
    // The functions we're interested in.
    char *funcs[] = { "sprintf", "strcpy", "sscanf", 0 };

    // Loop through all segments
    for (int i = 0; i < get_segm_qty(); i++) {
        segment_t *seg = getnseg(i);

        // We are only interested in the pseudo segment created by
        // IDA, which is of type SEG_XTRN. This segment holds all
        // function 'extern' definitions.
        if (seg->type == SEG_XTRN) {
```
// Loop through each of the functions we're interested in.
for (int i = 0; funcs[i] != 0; i++) {
    // Get the address of the function by its name
    ea_t loc = get_name_ea(seg->startEA, funcs[i]);
    // If the function was found, loop through it's
    // referrers.
    if (loc != BADADDR) {
        msg("Finding callers to %s (%a)\n", funcs[i], loc);
        xrefblk_t xb;
        // Loop through all the TO xrefs to our function.
        for (bool ok = xb.first_to(loc, XREF_DATA);
            ok;
            ok = xb.next_to()) {
            // Get the instruction (as text) at that address.
            char instr[MAXSTR];
            char instr_clean[MAXSTR];
            generate_disasm_line(xb.from, instr, sizeof(instr)-1);
            // Remove the colour coding and format characters
            tag_remove(instr, instr_clean, sizeof(instr_clean)-1);
            msg("Caller to %s: %a [%s]\n",
                funcs[i],
                xb.from,
                instr_clean);
        }
    }
}
return;

char IDAP_comment[] = "Insecure Function Finder";
char IDAP_help[] = "Searches for all instances"
    " of strcpy(), sprintf() and sscanf().\n";

char IDAP_name[] = "Insecure Function Finder";
char IDAP_hotkey[] = "Alt-I";

plugin_t PLUGIN =
{     IDP_INTERFACE_VERSION,
0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};
6.2 Listing Functions Containing MOVS et al.

When looking for the use of vulnerable functions like `strcpy` for example, you might need to look deeper than simple uses of the function and identify functions that use instructions in the movs family (movsb, movsd, etc.). This plug-in will go through all the functions, then each of their instructions looking for anything that uses a movs-like mnemonic.

```c
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <allins.hpp>

int IDAP_init(void)
{
    // Only support x86 architecture
    if(strncmp(inf.procName, "metapc", 8) != 0) {
        error("Only x86 binary type supported, sorry.");
        return PLUGIN_SKIP;
    }

    return PLUGIN_KEEP;
}

void IDAP_term(void)
{
    return;
}

void IDAP_run(int arg)
{
    // Instructions we're interested in. NN_movs covers movsd, movsw, etc.
    int movinstrs[] = { NN_movsx, NN_movsd, NN_movs, 0 };

    // Loop through all segments
    for (int s = 0; s < get_segm_qty(); s++) {
        segment_t *seg = getnseg(s);

        // We are only interested in segments containing code.
        if (seg->type == SEG_CODE) {

            // Loop through each function
            for (int x = 0; x < get_func_qty(); x++) {
                func_t *f = getn_func(x);
                char funcName[MAXSTR];

                // Get the function name
                get_func_name(f->startEA, funcName, sizeof(funcName) - 1);

                // Loop through the instructions in each function
                for (ea_t addr = f->startEA; addr < f->endEA; addr++) {
```
// Get the flags for this address
flags_t flags = getFlags(addr);

// Only look at the address if it's a head byte, i.e.
// the start of an instruction and is code.
if (isHead(flags) && isCode(flags)) {
    char mnem[MAXSTR];

    // Fill the cmd structure with the disassembly of
    // the current address and get the mnemonic text.
    ua_mnem(addr, mnem, sizeof(mnem)-1);

    // Check the mnemonic of the address against all
    // mnemonics we're interested in.
    for (int i = 0; movinstrs[i] != 0; i++) {
        if (cmd.itype == movinstrs[i])
            msg("%s: found %s at %a!\n", funcName, mnem, addr);
    }
}
return;

char IDAP_comment[] = "MOVSx Instruction Finder";
char IDAP_help[] =
    "Searches for all MOVS-like instructions.\n" \
    "\n" \
    "This will display a list of all functions along with\n" \
    "the movs instruction used within."
;

char IDAP_name[] = "MOVSx Instruction Finder";
char IDAP_hotkey[] = "Alt-M";

plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};
6.3 Auto-loading DLLs Into the IDA Database

Most binaries will spread their functionality across multiple files (DLLs), loading them at runtime using LoadLibrary. In these cases, it can be useful to have IDA auto-load these DLLs into the one IDB. This plug-in will search through the strings in a binary looking for anything containing .dll. For strings that do, it is assumed they are DLLs intended to be loaded by the binary and will prompt the user for the full path of that DLL and load it into the IDB.

```c
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <strlist.hpp>

// Maximum number of library files to load into the IDB
#define MAXLIBS 5

int IDAP_init(void)
{
    if (inf.filetype != f_PE) {
        error("Only PE executable file format supported.\n");
        return PLUGIN_SKIP;
    }

    return PLUGIN_KEEP;
}

void IDAP_term(void)
{
    return;
}

void IDAP_run(int arg)
{
    char loadLibs[MAXLIBS][MAXSTR];
    int libno = 0, i;

    // Loop through all strings to find any string that contains .dll. This will eventually be our list of DLLs to load.
    for (i = 0; i < get_strlist_qty(); i++) {
        char string[MAXSTR];
        string_info_t si;

        // Get the string item
        get_strlist_item(i, &si);

        if (si.length < sizeof(string)) {
            // Retrieve the string from the binary
            get_many_bytes(si.ea, string, si.length);

            // We're only interested in C strings.
```
if (si.type == 0) {
    // .. and if the string contains .dll
    if (stristr(string, ".dll") && libno < MAXLIBS) {
        // Add the string to the list of DLLs to load later on.
        strncpy(loadLibs[libno++], string, MAXSTR-1);
    }
}

// Now go through the list of libraries found and load them.
msg("Loading the first %d libraries found...
", MAXLIBS);
for (i = 0; i < MAXLIBS; i++) {
    msg("Lib: %s\n", loadLibs[i]);
    // Ask the user for the full path to the DLL (the executable will
    // only have the file name).
    char *file = askfile_cv(0, loadLibs[i], "File path...
", NULL);
    // Load the DLL using the pe loader module.
    if (load_loader_module(NULL, "pe", file, 0)) {
        msg("Sucessfully loaded %s\n", loadLibs[i]);
    } else {
        msg("Failed to load %s\n", loadLibs[i]);
    }
}

char IDAP_comment[] = "DLL Auto-Loader";
char IDAP_help[] = "Loads the first 5 DLLs
    mentioned in a binary file
";
char IDAP_name[] = "DLL Auto-Loader";
char IDAP_hotkey[] = "Alt-D";

plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};
6.4 Bulk Breakpoint Setter & Saver

This single plug-in gives you the ability to save the currently set breakpoints to a file, as well as load a list of addresses from a file and set breakpoints on them. To keep the plug-in simple, it expects the format of the input file to be sane, otherwise it will fail. You will also need to modify your `plugins.cfg` file to be able to use the one plug-in for both functions (setting and saving), as shown below.

```c
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <diskio.hpp>
#include <dbg.hpp>

#define MAX_BPT 100

void read_breakpoints() {
    char c, ea[9];
    int x = 0, b = 0;
    ea_t ea_list[MAX_BPT];

    // Ask the user for the file containing the breakpoints
    char *file = askfile_cv(0, "", "Breakpoint list file...", NULL);

    // Open the file in read-only mode
    FILE *fp = fopenRT(file);
    if (fp == NULL) {
        warning("Unable to open breakpoint list file, %s\n", file);
        return;
    }

    // Grab 8-byte chunks from the file
    while ((c = qfgetc(fp)) != EOF && b < MAX_BPT) {
        if (isalnum(c)) {
            ea[x++] = c;
            if (x == 8) {
                // NULL terminate the string
                ea[x] = 0;
                x = 0;
            }
        }
    }

    // Convert the 8 character string to an address
```
str2ea(ea, &ea_list[b], 0);
msg("Adding breakpoint at %a\n", ea_list[b]);
// Add the breakpoint as a software breakpoint
add_bpt(ea_list[b], 0, BPT_SOFT);
b++;
}
}
}

// Close the file handle
qfclose(fp);

void write_breakpoints() {
char c, ea[9];
int x = 0, b = 0;
ea_t ea_list[MAX_BPT];

// Ask the user for the file to save the breakpoints to
char *file = askstr(0, "", "Breakpoint list file...", NULL);

// Open the file in write-only mode
FILE *fp = ecreateT(file);

for (int i = 0; i < get_bpt_qty(); i++) {
bpt_t bpt;
char buf[MAXSTR];

gtxn_bpt(i, &bpt);
qsnprintf(buf, sizeof(buf)-1, "%08a\n", bpt.ea);
 ewrite(fp, buf, strlen(buf));
}

// Close the file handle
eclose(fp);
}

void IDAP_run(int arg)
{
// Depending on the argument supplied,
// read the breakpoint list from a file and
// apply it, or write the current breakpoints
// to a file.
switch (arg) {
  case 0:
    write_breakpoints();
    break;
  case 1:
    default:
      read_breakpoints();
    break;
}
}

int IDAP_init(void)
{
  return PLUGIN_KEEP;
void IDAP_term(void)
{
    return;
}

// These are irrelevant because they will be overridden by
// plugins.cfg.
char IDAP_comment[] = "Bulk Breakpoint Setter and Recorder";
char IDAP_help[] =
    "Sets breakpoints at a list of addresses in a text file"
    " or saves the current breakpoints to file.\n"
    "The read list must have one address per line.\n";

char IDAP_name[] = "Bulk Breakpoint Setter and Recorder";
char IDAP_hotkey[] = "Alt-B";

plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};

}
6.5 Selective Tracing (Method 1)

This plug-in gives you the ability to turn on instruction tracing only for a specific address range. It does this by running to the start address, turning on instruction tracing, running to the end address, and then turning instruction tracing off. Method 2 demonstrates a more flexible approach, utilising step tracing.

// snaptrace.cpp

#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <dbg.hpp>

int IDAP_init(void)
{
    return PLUGIN_KEEP;
}

void IDAP_term(void)
{
    return;
}

void IDAP_run(int arg)
{
    // Set the default start address to the user cursor position
    ea_t eaddr, saddr = get_screen_ea();

    // Allow the user to specify a start address
    askaddr(&saddr, "Address to start tracing at");

    // Set the end address to the end of the current function
    func_t *func = get_func(saddr);
    eaddr = func->endEA;

    // Allow the user to specify an end address
    askaddr(&eaddr, "Address to end tracing at");

    // Queue the following
    // Run to the start address
    request_run_to(saddr);
    // Then enable tracing
    request_enable_insn_trace();
    // Run to the end address, tracing all stops in between
    request_run_to(eaddr);
    // Turn off tracing once we've hit the end address
    request_disable_insn_trace();
    // Stop the process once we have what we want
    request_exit_process();

    // Run the above queued requests
run_requests();

} // These are actually pointless because we'll be overriding them
// in plugins.cfg
char IDAP_comment[] = "Snap Tracer";
char IDAP_help[] = "Allow tracing only between user "
    "specified addresses\n";

char IDAP_name[] = "Snap Tracer";
char IDAP_hotkey[] = "Alt-T";

plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};
6.6 Selective Tracing (Method 2)

Utilising step tracing, this plug-in sets up a debug event notification handler to handle a trace event (one instruction executed). Within this handler, it checks whether EIP is within the user-defined range, and if is, displays ESP. Obviously there are much more interesting things you can do with this sort of functionality like alerting based on the contents of registers and/or memory.

```cpp
// snaptrace2.cpp

#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <dbg.hpp>

ea_t start_ea = 0;
ea_t end_ea = 0;

// Handler for HT_DBG events
int idaapi trace_handler(void *udata, int dbg_event_id, va_list va)
{
    regval_t esp, eip;
    // Get ESP register value
    get_reg_val("esp", &esp);
    // Get EIP register value
    get_reg_val("eip", &eip);

    // We'll also receive debug events unrelated to tracing,
    // make sure those are filtered out
    if (dbg_event_id == dbg_trace) {
        // Make sure EIP is between the user-specified range
        if (eip.ival > start_ea && eip.ival < end_ea)
            msg("ESP = %a\n", esp.ival);
    }
    return 0;
}

int IDAP_init(void)
{
    // Receive debug event notifications
    hook_to_notification_point(HT_DBG, trace_handler, NULL);
    return PLUGIN_KEEP;
}

void IDAP_term(void)
{
    // Unhook from the notification point on exit
    unhook_from_notification_point(HT_DBG, trace_handler, NULL);
    return;
}

void IDAP_run(int arg)
```
{
    // Ask the user for a start and end address
    askaddr(&start_ea, "Start Address:" );
    askaddr(&end_ea, "End Address:" );

    // Queue the following

    // Run to the binary entry point
    request_run_to(inf.startIP);
    // Enable step tracing
    request_enable_step_trace();

    // Run queued requests
    run_requests();
}

// These are actually pointless because we'll be overriding them
// in plugins.cfg
char IDAP_comment[] = "Snap Tracer 2";
char IDAP_help[] = "Allow tracing only between user "
    "specified addresses\n";

char IDAP_name[] = "Snap Tracer 2";
char IDAP_hotkey[] = "Alt-I";

plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};
### 6.7 Binary Copy & Paste

Seeing there isn’t any binary copy-and-paste functionality in IDA, this plug-in will take care of both copy and paste operations allowing you to take a chunk of binary from one place and overwrite another with it. You need to modify your plugins.cfg file as this is a multi-function plug-in, needing one invocation for copy and another for paste. Obviously it only supports copying and pasting within IDA, however it could probably be extended to go beyond that.

```cpp
// // copypaste.cpp //

#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>

#define MAX_COPYPASTE 1024

// This will hold our copied buffer for pasting
char data[MAX_COPYPASTE];

// Bytes copied into the above buffer
ssize_t filled = 0;

// Insert the following two lines into your plugins.cfg file
// Replace pluginname with the filename of your plugin minus
// the extension.

// Copy_Buffer pluginname Alt-C 0
// Paste_Buffer pluginname Alt-V 1

int IDAP_init(void)
{
  return PLUGIN_KEEP;
}

void IDAP_term(void)
{
  return;
}

void copy_buffer()
{
  ea_t saddr, eaddr;
  ssize_t size;

  // Get the boundaries of the user selection
  if (read_selection(&saddr, &eaddr)) {
    // Work out the size, make sure it doesn't exceed the buffer
    // we have allocated.
    size = eaddr - saddr;
    if (size > MAX_COPYPASTE) {
      warning("You can only copy a max of %d bytes\n", MAX_COPYPASTE);
    }
  }
}```
return;
}

// Get the bytes from the file, store it in our buffer
if (get_many_bytes(saddr, data, size)) {
    filled = size;
    msg("Successfully copied %d bytes from %a into memory.\n", size, saddr);
} else {
    filled = 0;
}
} else {
    warning("No bytes selected!\n");
    return;
}
}

void paste_buffer() {

    // Get the cursor position. This is where we will paste to
    ea_t curpos = get_screen_ea();

    // Make sure the buffer has been filled with a Copy operation first.
    if (filled) {
        // Patch the binary (paste)
        patch_many_bytes(curpos, data, filled);
        msg("Patched %d bytes at %a.\n", filled, curpos);
    } else {
        warning("No data to paste!\n");
        return;
    }
}

void IDAP_run(int arg) {

    // Based on the argument supplied in plugins.cfg,
    // we can use the one plug-in for both the copy
    // and paste operations.
    switch(arg) {
    case 0:
        copy_buffer();
        break;
    case 1:
        paste_buffer();
        break;
    default:
        warning("Invalid usage!\n");
        return;
    }
}

// These are actually pointless because we'll be overriding them
// in plugins.cfg
char IDAP_comment[] = "Binary Copy and Paster";
char IDAP_help[] = "Allows the user to copy and paste binary\n";
char IDAP_name[] = "Binary Copy and Paster";
char IDAP_hotkey[] = "Alt-I";

plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};