The IDA API

IDA ICE communicates between instances, and with external programs, by socket communication facilitated by the ZeroMQ framework.

On each computer running IDA ICE, the IDA message broker runs (see the IDA message broker chapter), and passes messages between registered clients.

Each registered client has a unique (on that computer) id number (which is written in Help/About IDA/API id). This id is used to communicate to that client. The API id is in many cases the same as the Process ID of the client process.

Once a session is started (by calling the client with some function), the session is locked, and no other API sessions can talk to that client until the session has been closed.

IDA Message Broker Service

The IDA Message Broker Service is a service with the name IDAMessageBrokerService. It is installed into the c:\ProgramData\equa\ folder. The service can be removed and installed by executing the IDAMessageBrokerService.exe with the flags /install and /remove from a command prompt with elevated rights. It is installed by default during installation of IDA ICE.

The service can be started and stopped with the help of the usual Windows service management tool, or by calling sc start(or stop) IDAMessageBrokerService in an elevated command prompt.

When started, the ports on which the program is communicating is written to the file ida_ports.txt in c:\ProgramData\equa. Here is also a ida_message_broker_log.txt-file, where some logging is done.

If you want to open the message broker for external communication, please place a file named ida_external.txt into the c:\ProgramData\equa folder and restart the service. The content of this file is irrelevant.

The idaapi2.dll

The idaapi2.dll is a .dll (a dynamic-link library), which opens up the IDA API for external programs. Every program language where you can load a Windows .dll-file, and use the ANSI C/C++ function calls, can use the API. In this document, both the technical parts of using the .dll and the API itself will be discussed.

If you want to move the idaapi.dll from the default location, please also copy the libzmq-v100-mt-4_0_4.dll-file to the same location as the idaapi2.dll. There is some logging of the API calls in the file ida_api_dll_log.txt placed in the Windows Temp-folder (usually c:\users\username\appdata\local\temp\). Please check this if you are experiencing problems.

To find the correct variable/parameter/object names in IDA ICE, the best way is to first go to Options/Preferences/Developer and check the Do not translate names-checkbox. This makes IDA ICE show the true names in the Outline view. In the outline view, however, not everything is actual objects; entries preceded by a : is a collection, which is just shown for convenience, and does not exist as actual object. Instead, objects listed under the collection, are actually place directly under the parent object for the collection.

Functions in the idaapi2.dll

The functions, except for the connect/disconnect, switch… and get_err and getIDAStatus, are put in a results queue, which you then poll to get the actual result when it is done. These functions immediately return true/false, and then when the real results is done, that result in the pollForQueuedResults call. When a function description say that it returns “A list of child nodes”, it means that pollForQueuedResults returns that after a successful call.

In the idaapi2.dll, the following functions are exposed:

Functions for connecting/disconnecting with IDA:

bool connect_to_ida(char *port_c, char *remote_id)

Function that performs the connection to the IDA message broker. For simplicity, it also specifies to which instance of IDA ICE you send all messages. Port should, by default, be "5945".

If the IDA message broker is listening to another port (see IDA message broker chapter), you can specify the correct port here. The remote_id_c can look in two ways:

1) "4076"

2) "192.168.20.234:4076"

The first is the unique id of the local IDA ICE instance you want to connect to (check Help/About IDA, look for API id). The second is a string concatenating the ip-number and the unique id, separated by a colon. In this case, you will try to connect to a remote instance. If you want to communicate to another computer, both that message broker and the one on your computer needs to be enabled for external communication (see IDA Message Broker chapter).

If successful, the result is true, otherwise false. If it fails, please use the get_err function to see what went wrong.

bool switch_remote_connection(const char *new_remote_id)

Function that switches to another IDA ICE instance as default connection.

The remote_id_c can look in two ways:

1) "4076"

2) "192.168.20.234:4076"

The first is the unique id of the local IDA ICE instance you want to connect to (check Help/About IDA, look for UID). The second is a string concatenating the ip-number and the unique id, separated by a colon. In this case, you will try to connect to a remote instance. If you want to communicate to another computer, both that message broker and the one on your computer needs to be enabled for external communication (see IDA Message Broker chapter).

If successful, the result is true, otherwise false. If it fails, please use the get_err function to see what went wrong.

bool switch_api_version(long new_version_number)

The IDA API is a versioned API, so that several versions can coexist and for compatibility reasons. For the idaapi2.dll, the default version is the most recent one at the time of creation.

If, for some reason, you want to use another version of the api, this is the function to call. Once set, that version will be used until you reset, or start a new session.

long call_ida_function(const char *function, const char *parameter_values, char *out, int out_len)

If you have a need to call some function not yet supported by the IDA API, you can use this function to call any IDA function with given parameter values. function specifies the function name, and parameter_values is a string with the given parameter values the function needs, given as a json-list of type, value pairs. An example would be:

call_ida_function("+","[{\"type\": \"integer\", \"value\": 4},{\"type\": \"integer\", \"value\": 5}]", out, out_len)

This should perform the addition 4 + 5, and return the value, as a json-structure: {\"type\": \"integer\", \"value\": 9}

Returns 0 if the out_len is large enough for the reply. If the buffer is too small, the functions returns the size needed to receive the reply, so you can reallocate the out buffer and retry. If the result is -1, something went wrong, please call get_err to see what.

Recommeneded out_len: Hard to say, it all depends on what function you call. Start at 500.

bool ida_disconnect()

Function that unregister the process at IDA message broker, and then terminates the socket connection.

This should be called when finishing a session and before unloading the .dll.

long get_err(char *out, int out_len)

Function that returns the latest error produced in the functions. You need to send in a char buffer and its length. If the buffer is large enough to copy the error message, the result is 0. Otherwise, it is the needed length, which enables the user to reallocate a large enough buffer and retry.

Recommeneded out_len: Hard to say, start at 500 perhaps.

DOM api-functions:

All functions return -1 if something went wrong, 0 if everything went well, and a positive number if the string buffer sent in is too small. In out, the string representation of a json-structure is located. This can be parsed and used: it is either a structure {"type": type, "value": value}, or a list with such structures. All object values are longs. They are translated back to objects in IDA.

long childNodes(long node, char * out, int out_len)

Returns a json list of child nodes to node.

long parentNode(long node, char * out, int out_len)

Returns the parent node of node.

long setParentNode(long node, long parent, char * out, int out_len)

Sets the parent node of node to parent.

Returns the parent node.

long hasChildNodes(long node, char * out, int out_len)

Returns true/false depending on whether node has any child nodes.

long firstChild(long node, char * out, int out_len)

Returns the first child node of node.

long lastChild(long node, char * out, int out_len)

Returns the last child node of node.

long nextSibling(long node, char * out, int out_len)

Returns the next sibling node of node.

long previousSibling(long node, char * out, int out_len)

Returns the previous sibling node of node.

long childNodesLength(long node, char * out, int out_len)

Returns the number of child nodes of node.

long cloneNode(long node, char * out, int out_len)

Clones, and inserts, the node node. Returns the new node.

long createNode(long node, char * value, char * out, int out_len)

Creates a new node as a child to node, according to the l-form given in the value-parameter. Returns the new node.

long contains(long node, long node2, char * out, int out_len)

Checks if node is contained within node2. Returns true/false

long domAncestor(long node, long node2, char * out, int out_len)

Checks if node is ancestor to node2. Returns true/false

long item(long index, long node, char * out, int out_len)

Returns the node with index index in the list of children of node.

long appendChild(long child, long parent, char * out, int out_len)

Appends child as a child node to parent. Returns the new list of children.

long removeChild(long child, long parent, char * out, int out_len)

Removes child from parent. Returns the new list of children.

long replaceChild(long child, long child_rep, long parent, char * out, int out_len)

Replaces child with child_rep. Returns the parent.

long setAttribute(char * attribute, long node, char * value, char * out, int out_len)

Sets the attribute attribute of node node to value given by the json structure submitted (in text form) in the value parameter. Returns the value of the attribute.

long getAttribute(char * attribute, long node, char * out, int out_len)

Returns the value of the attribute attribute of node node.

Special IDA api-functions:

long openDocument(char *path, char *out, int out_len)

Opens the building specified in path. Returns the building object.

long openDocByTypeAndName(const char * name, const char *type, char * out, int out_len)

Returns the building object of type type and name name (without .idm) for an

already opened building.

long openIFCFile(char *path, long mergeWindows, long keepIntersectingSpaces, long ingoreRoof, char *out, int out_len)

Imports the .ifc file specified by path. The three parameters mergeWindows keepIntersectingSpaces and ignoreRoof can be set to 0 (do not use) or 1 (use). The parameters are from the IFC import options in IDA ICE.

Returns the imported building object.

long saveDocument(long building_object, char * new_path, long unpacked, char * out, int out_len)

Saves the building object. If new_path is just an empty string, it saves, otherwise, it tries to Save as... the pull path/name of the new_path.

new_path can e.g. be "d:\\temp\\building47.idm"

unpacked can be set to 1 if you want unpacked cases, or 0 if you want packed (default behaviour) cases.

Returns true/false.

long closeDocument(long building_object, long unpacked, char * out, int out_len)

Closes the building object.

Returns true/false.

long runSimulation(long building_object, long simulation_type, char * out, int out_len)

Acccording to the simulation_type parameter, the building building_object is run as:

simulation_type:

1) Custom simulation

2) Energy simulation

3) Cooling load simulation

4) Heating load simulation

5) Advanced level Run model... simulation

long pollForQueuedResults(char * out, int out_len)

Returns false if no queued results exists.

Returns a list if queued results exist, where first element is true/false depending if calculation is done. If done, the second element in the list contains the result.

bool closeSession(char * out, int out_len)

Closes down the current session, so a new one can connect to the child instance.

bool exitSession(char * out, int out_len)

Closes down the application, for the current session.

long getZones(long node, char * out, int out_len)

If node is a building object, returns a list of the bulding's zones.

Otherwise tries to return a list of the node's building's zones (same as above).

long getWindows(long object, char * out, int out_len)

If object is a building object, a list of all windows in the building is returned. If a zone object, a list of all external windows in the zone.

long getAllSubobjectsOfType(long node, char *type, char *out, int out_len)

Returns a list of all subobjects of type type recursively for an object.

long runIDAScript(long node, char *script, char *out, int out_len)

Executes a general IDA script with node as base object.

long copyObject(long node, char *new_name, char *out, int *out_len)

Makes a copy of node node and names it new_name. Returns new node.

long findObjectsByCriterium(long node, char *criterium, char *out, int *out_len)

Criterium must be an IDA script expression evaluating to true/false. (E.g. "(= (:call name [@]) "Zone name")" "(eq 'SCHEDULE-DATA (:call model-type [@]))"

Returns a list of all subobjects (recursively) of node, and returns a list with structures as: (object ("Object name" "Parent name" "Grandparent name.." etc, down to building name)))

long findUseOfResource(long resource, char *out, int *out_len)

Resource must be a resource object.

Returns a list of all objects where the resource is used.

long printReport(long report, char *out_path, long report_type, char *out, int *out_len)

report must be a report object. Writes a report to out_path.

report_type specifies what type of report you want:

1. .html

2. .pdf

long compareResults(char *building_objects, char *out_path, long report_type, char *out, int *out_len)

building_objects must be a json-list with building objects: "[{\"type\": "\object\", \"value:\": 1}, {\"type\": "\object\", \"value:\": 1}]"

report_type specifies what type of report you want:

1. .html

2. .pdf

long getIDAStatus(char * out, int out_len)

Returns a list with the present status of the simulation/IDA, and possible other information. The first element is always the state.

Possible normal states:

1) Idle

2) Reading in model

3) Creating mathematical model

4) Simulating (including progress)

5) Finished

6) Terminated

7) Saving

Idle state is when nothing is going on.

Reading in model is when the program is reading in a model from a file and creating the mathematical structure in memory. When finished state is again Idle.

Creating mathematical model is when the program creates the actual input file for the numerical solver. When finished state is again Idle.

Simulating is when the program is simulating, in this case a second value representing the percentage done of the simulation is sent. When finished, state is Finished or Terminated.

Finished is when the simulation is successfully run, but not yet saved.

Terminated is when the simulation failed to run successfully, a second value representing the error is sent.

Saving is when the program is saving the simulation results. When finished state is again Idle.

long getChildrenOfType(long node, char * type, char * out, int out_len)

Returns a list of all the child nodes of node with type type.

long findNamedChild(long node, char * name, char * out, int out_len)

Returns one or zero child nodes of node with name name, will try first string version, then symbol version of name.

Long export3dView(long building_object, char * output_file, long view, char * out, int out_len)

Exports an image of the 3D-view to file output_file given building building_object. Several views are available:

0. Default view (as you looked at the case last)
1. Top view
2. Bottom view
3. Front View
4. Back view
5. Left view
6. Right view

long exportDiagram(long diagram_object, char * output_file, long save_type, char * out, int out_len)

Exports an image of the diagram diagram_object to file output_file. Save_type can be:

0. .emf-file
1. .wmf-file
2. .png-file.

long getNamedObjectFromPath(long object, char * path, long save_type, char * out, int out_len)

Returns an object from a potentially deep structure. Object is parent object, and in path, you can specify the path down to an object you want to access. The path should be a text representation of a JSON array, with type, value pairs, like this: "[{\"type\":\"string\",\"value\": \"Zone\"}, {\"type\":\"string\",\"value\": \"GROUP\"}]"

This would return the GROUP object form the "Zone" object in the building, if the building object was passed in the object argument. This avoids nesting calls just to get to an object, if you are not interested in the objects in between the parent object and the deepest one.