iksemel programmers manual

A tutorial and API reference for the iksemel library 1.2

Table of Contents

1 Introduction

iksemel is an XML (eXtensible Markup Language) parser library designed for Jabber applications. It is coded in ANSI C for POSIX compatible environments, thus highly portable. It is free software released under the GNU Lesser General Public License.

The purprose of this manual is to tell you how to use the facilities of the iksemel library. Manual is written with the assumption that you are familiar with the C programming language, basic programming concepts, XML and Jabber protocol.

1.1 Compiling the Library

You need to install MinGW (`http://mingw.org`) under Windows to be able to compile iksemel. Although not tested by the author, Cygwin should work equally well.

Library can be built with:

```
./configure
make
```

If you want to make a self test:

```
make test
```

Now you can install it with:

```
make install
```

1.2 Using iksemel in Applications

You need to include '`iksemel.h`' file in your source to access library API. You can do this with:

```
#include "iksemel.h"
```

Now you can use iksemel functions and compile your source. In able to link your compiled object files and generate your executable program, you have to link with iksemel library. This can be done with:

```
gcc -o myprg src1.o src2.o src3.o -liksemel
```

iksemel registers itself with pkg-config while installing, so if you are using autotools in your program, you can simply check the availability of iksemel and configure your build process accordingly with:

```
PKG_CHECK_MODULES(IKSEMEL,iksemel,,exit)
```

This would result in IKSEMEL_LIBS and IKSEMEL_CFLAGS substitution variables set to correct values.

2 Tutorials

2.1 Parsing an XML Document

iksemel parser sequentally processes the XML document. Each encountered XML element (i.e. tags, character data, comments, processing instructions, etc.) is reported to your application by calling the hook functions you have provided. This type of interface is called SAX (serial access) interface.

Parser stores its state in a small structure. This structure is referenced by `iksparser` type, and managed with following functions:

`iksparser*` **iks_sax_new** (`void*` *user_data*, `iksTagHook*` *tagHook*, `iksCDataHook*` *cdataHook*); Function

This function allocates and initializes a parser structure. If allocation fails, NULL value is returned. *user_data* is passed directly to hook functions.

iksTagHook Typedef

int iksTagHook (void* *user_data*, char* *name*, char** *atts*, int *type*);

This function is called when a tag parsed. *name* is the name of the tag. If tag has no attributes *atts* is NULL, otherwise it contains a null terminated list of pointers to tag's attributes and their values. If return value isn't `IKS_OK`, it is passed immediately to the caller of the `iks_parse`.

type is one of the following:

`IKS_OPEN` Opening tag, i.e. <tag attr='value'>

`IKS_CLOSE`
Closing tag, i.e. </tag>

`IKS_SINGLE`
Standalone tag, i.e. <tag attr='value'/>

iksCDataHook Typedef

int iksCDataHook (void* *user_data*, char* *data*, size_t *len*);

data is a pointer to the character data. Encoding is UTF-8 and it isn't terminated with a null character. Size of the data is given with *len* in bytes. This function can be called several times with smaller sized data for a single string. If return value isn't `IKS_OK`, it is passed immediately to the caller of the `iks_parse`.

`int` **iks_parse** (`iksparser*` *prs*, `char *`*data*, `size_t` *len*, `int` *finish*); Function

You give XML document to the parser with this function. *data* is a *len* bytes string. If *len* is zero, data must be a null terminated string.

If *finish* value is zero, parser waits for more data later. If you want to finish parsing without giving data, call it like:

```
iks_parse (my_parser, NULL, 0, 1);
```

You should check the return value for following conditions:

`IKS_OK` There isn't any problem.

`IKS_NOMEM`
Not enough memory.

`IKS_BADXML`
Document is not well-formed.

`IKS_HOOK` Your hook decided that there is an error.

`void` **iks_parser_delete** (`iksparser*` *prs*); Function

This function frees parser structure and associated data.

Now we have learned how to create and use a sax parser. Lets parse a simple XML document. Write following code into a '`test.c`' file.

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

int pr_tag (void *udata, char *name, char **atts, int type)
{
    switch (type) {
        case IKS_OPEN:
            printf ("TAG <%s>\n", name);
            break;
        case IKS_CLOSE:
            printf ("TAG </%s>\n", name);
            break;
        case IKS_SINGLE:
            printf ("TAG <%s/>\n", name);
            break;
    }
    if (atts) {
        int i = 0;
        while (atts[i]) {
            printf ("  ATTRIB %s='%s'\n", atts[i], atts[i+1]);
            i += 2;
        }
    }
    return IKS_OK;
}

enum ikserror pr_cdata (void *udata, char *data, size_t len)
{
    int i;
    printf ("CDATA [");
    for (i = 0; i < len; i++)
        putchar (data[i]);
    printf ("]\n");
    return IKS_OK;
}
```

```
int main (int argc, char *argv[])
{
    iksparser *p;
    p = iks_sax_new (NULL, pr_tag, pr_cdata);
    switch (iks_parse (p, argv[1], 0, 1)) {
        case IKS_OK:
            puts ("OK");
            break;
        case IKS_NOMEM:
            puts ("Not enough memory");
            exit (1);
        case IKS_BADXML:
            puts ("XML document is not well-formed");
            exit (2);
        case IKS_HOOK:
            puts ("Our hooks didn't like something");
            exit (2);
    }
    iks_parser_delete (p);
    return 0;
}
```

Now compile and test it with:

```
gcc -o test test.c -liksemel
./test "<test>Hello<br/>World!</test>"
./test "<lala a='12' b='42'/>"
```

Error Handling

XML standart states that once an error is detected, the processor must not continue normal processing (i.e. it must not pass character data or markup information to the application). So iksemel stops processing immediately when it encounters a syntax error, or one of your hook functions return any one value than `IKS_OK`, and `iks_parse` function returns with the error code.

Since it is useful for debugging, iksemel provides functions to get position of the error. Position is usually at the starting character for syntax errors. Since your hooks are called after whole element (i.e. markup or character data) is passed, position is at the end of the erroneous element for `IKS_HOOK` errors.

`unsigned long` **iks_nr_bytes** (`iksparser*` *prs*); Function

Returns how many number of bytes parsed.

`unsigned long` **iks_nr_lines** (`iksparser*` *prs*); Function

Returns how many number of lines parsed.

If you want to parse another document with your parser again, you should use the following function to reset your parser.

void **iks_parser_reset** (iksparser* *prs*); Function

Resets the parser's internal state.

2.2 Working with XML Trees

SAX interface uses very little memory, but it forces you to access XML documents sequentally. In many cases you want to keep a tree like representation of XML document in memory and want to access and modify its content randomly.

iksemel provides functions for efficiently creating such trees either from documents or programmaticaly. You can access and modify this tree and can easily generate a new XML document from the tree.

This is called DOM (Document Object Model) interface.

2.2.1 Memory Management

Since keeping whole document content uses a lot of memory and requires many calls to OS's memory allocation layer, iksemel uses a simple object stack system for minimizing calls to the `malloc` function and releasing all the memory associated with a tree in a single step.

A parsed XML tree contains following objects:

'`Nodes`' These are basic blocks of document. They can contain a tag, attribute pair of a tag, or character data. Tag nodes can also contain other nodes as children. Node structure has a small fixed size depending on the node type.

'`Names`' Names of tags and attributes. They are utf-8 encoded small strings.

'`Character Data`'

They are similar to names but usually much bigger.

iksemel's object stack has two separate areas for keeping these data objects. Meta chunk contains all the structures and aligned data, while the data chunk contains strings. Each chunk starts with a choosen size memory block, then when necessary more blocks allocated for providing space. Unless there is a big request, each block is double the size of the previous block, thus real memory needs are quickly reached without allocating too many blocks, or wasting memory with too big blocks.

ikstack Typedef

This is a structure defining the object stack. Its fields are private and subject to change with new iksemel releases.

ikstack * **iks_stack_new** (size_t *meta_chunk*, size_t *data_chunk*); Function

Creates an object stack. *meta_chunk* is the initial size of the data block used for structures and aligned data. *data_chunk* is the initial size of the data block used for strings. They are both in byte units.

These two initial chunks and a small object stack structure is allocated in one `malloc` call for optimization purproses.

`void *` **iks_stack_alloc** (`ikstack *` *stack*, `size_t` *size*); Function

Allocates *size* bytes of space from the object stack's meta chunk. Allocated space is aligned on platform's default alignment boundary and isn't initialized. Returns a pointer to the space, or NULL if there isn't enough space available and allocating a new block fails.

`void *` **iks_stack_strdup** (`ikstack *` *stack*, `const char *` *src*, `size_t` *len*); Function

Copies given string *src* into the object stack's data chunk. Returns a pointer to the new string, or NULL if there isn't enough space in the stack. If *len* is zero string must be null terminated.

`void` **iks_stack_delete** (`ikstack *` *stack*); Function

Gives all memory associated with object stack to the system.

Since character data sections are usually parsed in separate blocks, a growable string implementation is necessary for saving memory.

`char *` **iks_stack_strcat** (`ikstack *`*stack*, `char *`*old*, `size_t` *old_len*, `const char *`*src*, `size_t` *src_len*); Function

This function appends the string *src* to the string *old* in the stack's data chunk. If *old* is NULL it behaves like `iks_stack_strdup`. Otherwise *old* has to be a string created with `iks_stack_strdup` or `iks_stack_strcat` functions.

If *old_len* or *src_len* is zero, corresponding string must be null terminated.

Since string can be moved into another block of the data chunk, you must use the returned value for new string, and must not reference to *old* anymore. Return value can be NULL if there isn't enough space in stack, and allocating a new block fails.

2.2.2 Creating a Tree

iks Typedef

This is a structure defining a XML node. Its fields are private and only accessed by following functions.

`iks*` **iks_new** (`const char *`*name*); Function

Creates an object stack and creates a IKS_TAG type of node with given tag name inside the stack. Tag name is also copied into the stack. Returns the node pointer, or NULL if there isn't enough memory.

`iks*` **iks_new_within** (`const char *`*name*, `ikstack*` *stack*); Function

Creates a IKS_TAG type of node with the given tag name. Node and tag name is allocated inside the given object stack. Returns the node pointer, or NULL if there isn't enough memory.

`iks*` **iks_insert** (`iks *`*x*, `const char *`*name*); Function

Creates a IKS_TAG type of node with the given tag name. Node and tag name is allocated inside the *x* node's object stack and linked to *x* as a child node. Returns the node pointer, or NULL if there isn't enough memory.

`iks*` **iks_insert_cdata** (`iks*` *x*, `const char*` *data*, `size_t` *len*); Function

Creates a IKS_CDATA type of node with given character data. Node is allocated inside the *x* node's object stack and linked to *x* as a child node. Data is copied as well. If *len* is zero data must be a null terminated string. Returns the node pointer, or NULL if there isn't enough memory.

`iks*` **iks_insert_attrib** (`iks*` *x*, `const char*` *name*, `const char*` *value*); Function

Creates a IKS_ATTRIBUTE type of node with given attribute name and the value. Node is allocated inside the *x* node's object stack and linked to *x* as an attribute node. Attribute name and value is copied as well. Returns the node pointer, or NULL if there isn't enough memory.

Reinserting another value with same attribute name changes an attribute's value. If *value* is NULL, attribute is removed from the tag.

`iks*` **iks_insert_node** (`iks*` *x*, `iks*` *y*); Function

Links node *y* to node *x* as a child node. Nodes are not copied between object stacks, be careful.

`void` **iks_hide** (`iks *`*x*); Function

Changes the links of the other nodes so that *x* becomes invisible. It stays in the same object stack with neighbour nodes, be careful.

`void` **iks_delete** (`iks *`*x*); Function

Frees the object stack of the node *x*.

Now lets create a tree representation of following XML document:

```
<message type='chat' from='bob@bd.com'>
<subject>song lyric</subject><priority>high</priority>
<body>
<em style='underline'>here is the correct version:</em>
i just don't see why i should even care
it's not dark yet, but it's getting there
</body>
</message>
```

here is the code:

```
iks *x, *y, *z;

x = iks_new ("message");
iks_insert_attrib (x, "type", "chat");
iks_insert_attrib (x, "from", "bob@bd.com");
```

```
    iks_insert_cdata (x, "\n", 1);
    iks_insert_cdata (iks_insert (x, "subject"), "song lyric", 10);
    iks_insert_cdata (iks_insert (x, "priority"), "high", 4);
    iks_insert_cdata (x, "\n", 1);
    y = iks_insert (x, "body");
    iks_insert_cdata (y, "\n", 1);
    z = iks_insert (y, "em");
    iks_insert_attrib (z, "style", "underline");
    iks_insert_cdata (z, "here is the correct version", 0);
    iks_insert_cdata (y, "\n", 1);
    iks_insert_cdata (y, "i just don't see why", 0);
    iks_insert_cdata (y, "i should even care\n", 0);
    iks_insert_cdata (y, "it's not dark yet,", 0);
    iks_insert_cdata (y, "but it's getting there\n", 0);
    iks_insert_cdata (x, "\n", 1);
```

Notice how newlines are inserted for proper formatting of document. They aren't necessary for representing data, but they make it easier to read document for humans.

Also notice how `iks_insert` and `iks_insert_cdata` chained.

There are also functions for duplicating xml trees. They are:

iks * **iks_copy** (iks* *x*); Function

Creates a full copy of the tree in a newly created object stack.

iks * **iks_copy_within** (iks* *x*, ikstack **s*); Function

Creates a full copy of the tree in given object stack.

2.2.3 Accessing a Tree

Basic access functions allow you to move on the tree:

iks* **iks_next** (iks* *x*); Function

iks* **iks_prev** (iks* *x*); Function

iks* **iks_parent** (iks* *x*); Function

iks* **iks_child** (iks* *x*); Function

iks* **iks_attrib** (iks* *x*); Function

These functions return a pointer to the next, previous, parent, first child, and first attribute node of the given node *x*. If that node doesn't exist or *x* is NULL, a NULL value is returned.

iks * **iks_root** (iks **x*); Function

Returns the topmost parent node of the *x*.

iks* **iks_next_tag** (iks* *x*); Function iks* **iks_prev_tag** (iks* *x*); Function

iks* **iks_first_tag** (iks* *x*); Function

These functions return a pointer to the next, previous, first child node of the given node *x*. Only tag nodes are considered, other type of the nodes are skipped. If such a node doesn't exist or *x* is NULL, a NULL value is returned.

Another group of functions allow you to access specific information and content of the nodes:

ikstack* **iks_stack** (iks* *x*); Function

Returns the object stack which node *x* stays.

enum ikstype **iks_type** (iks* *x*); Function

Returns the type of the node.

IKS_TAG Node is a tag and can contain child nodes and attributes.

IKS_CDATA
Node contains character data.

IKS_ATTRIBUTE
Node contains an attribute and its value.

char* **iks_name** (iks* *x*); Function

Returns the name of the tag for nodes with the type *IKS_TAG*.

char* **iks_cdata** (iks* *x*); Function

Returns a pointer to node's character data if available, NULL otherwise.

size_t **iks_cdata_size** (iks* *x*); Function

Returns the size of the node's character data in bytes.

int **iks_has_children** (iks* *x*); Function

Returns a non-zero value if node *x* has a child node.

int **iks_has_attribs** (iks* *x*); Function

Returns a non-zero value if node *x* has attributes.

Last group of the functions simplifies finding and accessing the content of a specific node:

iks* **iks_find** (iks* *x*, const char* *name*); Function

Searches a IKS_TAG type of node with *name* as tag name in child nodes of *x*. Returns a pointer to the node if found, NULL otherwise.

char* **iks_find_cdata** (iks* *x*, const char* *name*); Function

Searches a IKS_TAG type of node with *name* as tag name in child nodes of *x*. Returns a pointer to the character data of the node's first child node if found, NULL otherwise.

`char*` **iks_find_attrib** (`iks*` *x*, `const char*` *name*); Function

Searches an attribute with given name in attributes of the *x*. Returns a pointer to attribute value if found, NULL otherwise.

`iks *` **iks_find_with_attrib** (`iks *`*x*, `const char *`*tagname*, `const char *`*attrname*, `const char *`*value*); Function

Searches for a child tag of *x* which has an attribute with name *attrname* and value *value*. If *tagname* isn't NULL, name of the tag must also match. Returns a pointer to the node if found, NULL otherwise.

Here is an example which demonstrates accessing file names in a fictitious XML playlist file:

```
<playlist>
    <item type='mpg'>
        <name>/home/madcat/download/matrix_rev_trailer.mpg</name>
        <duration>1:17</duration>
    </item>
    <item type='rm'>
        <name>/home/madcat/anim/clementine_ep1.rm</name>
        <duration>22:00</duration>
    </item>
    <item type='avi'>
        <name>/home/madcat/anim/futurama/ep101.avi</name>
        <subtitle>/home/madcat/subs/futurama/ep101.txt</subtitle>
        <duration>30:00</duration>
    </item>
    <repeat/>
    <fullscreen/>
    <noui/>
</playlist>
```

and here is the code:

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

int main (int argc, char *argv[])
{
    iks *x, *y;
    int e;

    if (argc < 2) {
        printf ("usage: %s <playlistfile>", argv[0]);
        return 0;
    }
    e = iks_load (argv[1], &x);
    if (e != IKS_OK) {
     printf ("parse error %d\n", e);
        return 1;
    }
```

```
        if (iks_find (x, "repeat")) puts ("repeat mode enabled");
        y = iks_child (x);
        while (y) {
            if (iks_type (y) == IKS_TAG
                && strcmp (iks_name (y), "item") == 0) {
            printf ("Filename: [%s]\n", iks_find_cdata (y, "name"));
            }
            y = iks_next (y);
         }
        iks_delete (x);
        return 0;
    }
```

2.2.4 Converting a Tree to an XML Document

There is a function for converting given XML tree into a null terminated string.

`char *` **iks_string** (`ikstack*` *stack*, `iks*` *x*); Function

Converts given tree into a string. String is created inside the given object stack. Returns a pointer to the string, or NULL if there isn't enough memory available.

If *stack* is NULL, string is created inside an `iks_malloced` buffer. You can free it later with `iks_free` function.

Here is an example which builds a tree and print it.

```
iks *x;
char *t;

x = iks_new ("test");
iks_insert_cdata (iks_insert (x, "a"), "1234", 4);
iks_insert (x, "br");
iks_insert_cdata (x, "1234", 4);
t = iks_string (iks_stack (x), x);
puts (t);
iks_delete (x);
```

2.2.5 Parsing a Document into a Tree

If you want to automatically convert an XML document into a tree, you can use iksemel's DOM parser. It is created with following function:

`iksparser*` **iks_dom_new** (`iks **`*iksptr*); Function

Creates a DOM parser. A pointer to the created XML tree is put into the variable pointed by *iksptr*. Returns a pointer to the parser, or NULL is there isn't enough memory.

Usage is same as SAX parser. You feed the data with `iks_parse`, and if there isn't an error, you can access to your tree from variable `*iksptr`.

Here is a simple example:

```
iks *x;
iksparser *p;

p = iks_dom_new (&x);
if (IKS_OK != iks_parse (p, "<a>bcd</a>", 9, 1)) {
    puts ("parse error");
}
/* x is useable after that point */

/* this will print 'bcd' */
printf ("%s\n", iks_cdata (iks_child (x)));
```

If you know the size of the file ahead, or you have an approximate idea, you can tell this to the dom parser for choosing a better memory allocation strategy. Here is the function for this.

`void` **iks_set_size_hint** (`iksparser *`*prs*, `size_t` *approx_size*); Function

> Parser *prs* must be a dom type parser. *approx_size* is the expected size of the xml document. Parser chooses its chunk size based on this information. Helps performance while processing big files.

If you already have your XML document in memory, you can simply parse it with:

`iks *` **iks_tree** (`const char *`*xml_str*, `size_t` *len*, `int *`*err*); Function

> This function parses the buffer pointed by *xml_str*. If *len* is zero buffer is considered as a null terminated utf8 string. Returns the parsed tree, or NULL if there is an error. If *err* is not NULL, actual error code (returned by iks_parse) is put there.

Most of the times you want to load your configuration (or similar) files directly into trees. iksemel provides two functions to greatly simplify this:

`int` **iks_load** (`const char *`*fname*, `iks **`*xptr*); Function

> Loads the XML file. Tree is placed into the variable pointed by *xptr*.

`int` **iks_save** (`const char *`*fname*, `iks *`*x*); Function

> Converts tree *x* into a string and saves to the file.

Both functions return same error codes as `iks_parse`. Some additional error codes are defined for indicating file problems. They are:

`IKS_FILE_NOFILE`
: A file with the given name doesn't exist.

`IKS_FILE_NOACCESS`
: Cannot open file. Possibly a permission problem.

`IKS_FILE_RWERR`
: Read or write operation failed.

Here is a simple example which parses a file and saves it into another:

```
iks *x;

if (IKS_OK != iks_load ("file1.xml", &x)) {
    puts ("loading error");
}
if (IKS_OK != iks_save ("file2.xml", x)) {
    puts ("saving error");
}
```

2.3 XML Streams

XML streams function as containers for any XML chunks sent asynchronously between network endpoints. They are used for asyncronously exchanging relatively small payload of structured information between entities.

A stream is initiated by one of hosts connecting to the other, and sending a <stream:stream> tag. Receiving entity replies with a second XML stream back to the initiating entity within the same connection. Each unit of information is send as a direct child tag of the <stream:stream> tag. Stream is closed with </stream:stream>.

XML streams use a subset of XML. Specifically they should not contain processing instructions, non-predefined entities, comments, or DTDs.

Jabber protocol uses XML streams for exchanging messages, presence information, and other information like authorization, search, time and version queries, protocol extensions.

iksemel provides you a stream parser, which automatically handles connection to the server, and calls your hook function with incoming information parsed and converted to an XML tree.

You can create such a parser with:

`iksparser*` **iks_stream_new** (`char*` *name_space*, `void*` *user_data*, `iksStreamHook*` *streamHook*); Function

Allocates and initalizes a stream parser. *name_space* indicates the stream type, jabber clients use "jabber:client" namespace. *user_data* is passed directly to your hook function.

iksStreamHook Typedef

int iksStreamHook (void* *user_data*, int *type*, iks* *node*);

Depending on the value of the *type*, *node* contains:

`IKS_NODE_START`
: Got the <stream:stream> tag, namespace, stream id and other information is contained in the *node*.

`IKS_NODE_NORMAL`
: A first level child of the <stream:stream> tag is received. *node* contains the parsed tag. If you are connected to a jabber server, you can get <message>, <presence>, or <iq> tags.

`IKS_NODE_ERROR`
: Got a <stream:error> tag, details can be accessed from *node*.

`IKS_NODE_STOP`
</stream:stream> tag is received or connection is closed, *node* is `NULL`.

Freeing the node with `iks_delete` is up to you.

You can manually feed this parser with `iks_parse` function, but using iksemel's connection facilities is easier for most of the cases.

This functions return `IKS_OK` for success. Error codes of `iks_parse` are used in same manner. Following additional codes are defined for network related problems:

`IKS_NET_NODNS`
Hostname lookup failed. Possible reasons: hostname is incorrect, you are not online, your dns server isn't accessible.

`IKS_NET_NOSOCK`
Socket cannot created.

`IKS_NET_NOCONN`
Connection attemp failed. Possible reasons: host is not an XML stream server, port number is wrong, server is busy or closed for the moment.

`IKS_NET_RWERR`
`send` or `recv` call is failed when attempting to exchange the data with the server. You should close the connection with `iks_disconnect` after getting this error from data transfer functions.

`int` **iks_connect_tcp** (`iksparser *`*prs*, `const char *`*server*, `int` *port*); Function

This function connects the parser to a server and sends stream header for you. *server* is the host name of the server and *port* is the tcp port number which server is listening to. You can use `IKS_JABBER_PORT` macro for the default jabber client port (5222).

`int` **iks_connect_fd** (`iksparser *`*prs*, `int` *fd*); Function

Attaches parser to an already opened connection. *fd* is the socket descriptor. Note that `iks_disconnect` doesn't close the socket for this kind of connection, opening and closing of the socket is up to your application. Stream header is not sent automatically. You can use `iks_send_header` function for sending it.

`void` **iks_disconnect** (`iksparser *`*prs*); Function

Closes connection to the server, and frees connection resources.

After successfully connecting to a server, you can use following functions for exchanging information with server.

`int` **iks_recv** (`iksparser*` *prs*, `int` *timeout*); Function

If *timeout* is `-1`, waits until some data arrives from server, and process the data. Your stream hook can be called if a complete chunk is arrived.

If *timeout* is a positive integer, `iks_recv` returns if no data arrives for *timeout* seconds.

If *timeout* is zero, `iks_recv` checks if there is any data waiting at the network buffer, and returns without waiting for data.

`int` **iks_fd** (`iksparser*` *prs*); Function

Returns the file descriptor of the connected socket. You can use this in your `select` function or some other input loop to act whenever some data from the server arrives. This value of only valid between a successful `iks_connect_tcp` and `iks_disconnect`.

`int` **iks_send** (`iksparser*` *prs*, `iks*` *x*); Function

Converts the tree given in *x* to a string, and sends to the server. String is created inside the object stack of *x*.

`int` **iks_send_raw** (`iksparser*` *prs*, `char*` *xmlstr*); Function

Sends the string given in *xmlstr* to the server.

`int` **iks_send_header** (`iksparser` `*`*prs*, `char` `*`*to*); Function

Sends the stream header. *to* is the name of the server. Normally `iks_connect_tcp` function calls this for you. This is only useful if you are using `iks_connect_fd`.

Sometimes it is useful to log incoming and outgoing data to your parser for debugging your applications. iksemel provides a logging facility for you.

`void` **iks_set_log_hook** (`iksparser*` *prs*, `iksLogHook*` *logHook*); Function

Sets the log function for your stream parser. You can't use this function on any other type of parser.

iksLogHook Typedef

void iksLogHook (void* *user_data*, const char* *data*, size_t *size*, int *is_incoming*);

user_data is same value which you give with `iks_stream_new`. *data* is *size* bytes of data. Be very careful that this data may be coming from other side of the connection and can contain malicius bytes. It isn't checked by iksemel yet, so you should check it yourself before displaying or passing to other systems in your application or computer. If *is_incoming* is a non-zero value, data is incoming from server, otherwise it is outgoing to the server.

2.4 Writing a Jabber Client

2.4.1 Security

iksemel supports TLS protocol for encrypted communication and SASL protocol for authentication. TLS is handled by gnutls library.

`int` **iks_has_tls** (`void`); Function

If iksemel is compiled with gnutls library, this function returns a non-zero value indicating you can try encrypted connection with the server.

`int` **iks_start_tls** (`iksparser*` *prs*); Function

Starts a TLS handshake over already connected parser. Returns IKS_OK or one of the IKS_NET_ errors. If handshake succeeds you'll get another stream header from server.

int **iks_is_secure** (iksparser* *prs*); Function

Returns a non-zero value if a secure connection is fully established between server.

int **iks_start_sasl** (iksparser* *prs*, enum ikssasltype *type*, char* *username*, char* *pass*); Function

Starts SASL operation.

See tools/iksroster.c for a good example.

2.4.2 Packets

iksemel can parse a jabber XML node and provide you a public packet structure which contains information like node type and subtype, id, namespace, sender's jabber id, etc.

This handles a lot of node parsing for you. Packets are also used in the packet filter subsystem.

ikspak * **iks_packet** (iks **x*); Function

Takes a node from stream and extracts information from it to a packet structure. Structure is allocated inside the node's object stack.

`ikspak` structure has following fields:

`iks *x;` This is a pointer to the node.

`iksid *from;`
Sender's jabber id in parsed form. See below for `iksid` structure.

`iks *query;`
A pointer to the <query> tag for IQ nodes.

`char *ns;` Namespace of the content for IQ nodes.

`char *id;` ID of the node.

`enum ikspaktype type;`
Type of the node. Possible types are:

`IKS_PAK_NONE`
Unknown node.

`IKS_PAK_MESSAGE`
Message node.

`IKS_PAK_PRESENCE`
Presence node with presence publishing operation.

`IKS_PAK_S10N`
Presence node with subscription operation.

`IKS_PAK_IQ`
IQ node.

`enum iksubtype subtype;`
Sub type of the node. Sub types for message nodes:

`IKS_TYPE_NONE`
: A normal message.

`IKS_TYPE_CHAT`
: Private chat message.

`IKS_TYPE_GROUPCHAT`
: Multi user chat message.

`IKS_TYPE_HEADLINE`
: Message from a news source.

`IKS_TYPE_ERROR`
: Message error.

Sub types for IQ nodes:

`IKS_TYPE_GET`
: Asks for some information.

`IKS_TYPE_SET`
: Request for changing information.

`IKS_TYPE_RESULT`
: Reply to get and set requests.

`IKS_TYPE_ERROR`
: IQ error.

Sub types for subscription nodes:

`IKS_TYPE_SUBSCRIBE,`
: Asks for subscribing to the presence.

`IKS_TYPE_SUBSCRIBED,`
: Grants subscription.

`IKS_TYPE_UNSUBSCRIBE,`
: Asks for unsubscribing to the presence.

`IKS_TYPE_UNSUBSCRIBED,`
: Cancels subscription.

`IKS_TYPE_ERROR`
: Presence error.

Sub types for presence nodes:

`IKS_TYPE_PROBE,`
: Asks presence status.

`IKS_TYPE_AVAILABLE,`
: Publishes entity as available. More information can be found in `show` field.

`IKS_TYPE_UNAVAILABLE`
: Publishes entity as unavailable. More information can be found in `show` field.

`enum ikshowtype show;`
: Presence state for the presence nodes.

  `IKS_SHOW_UNAVAILABLE`
  : Entity is unavailable.

  `IKS_SHOW_AVAILABLE`
  : Entity is available.

  `IKS_SHOW_CHAT`
  : Entity is free for chat.

  `IKS_SHOW_AWAY`
  : Entity is away for a short time.

  `IKS_SHOW_XA`
  : Entity is away for a long time.

  `IKS_SHOW_DND`
  : Entity doesn't want to be disturbed.

iksemel has two functions to parse and compare jabber IDs.

`iksid *` **iks_id_new** (`ikstack *`*s*, `const char *`*jid*); Function

Parses a jabber id into its parts. `iksid` structure is created inside the *s* object stack.

`iksid` structure has following fields:

`char *user;`
: User name.

`char *server;`
: Server name.

`char *resource;`
: Resource.

`char *partial;`
: User name and server name.

`char *full;`
: User name, server name and resource.

You can access this fields and read their values. Comparing two parsed jabber ids can be done with:

`int` **iks_id_cmp** (`iksid *`*a*, `iksid *`*b*, `int` *parts*); Function

Compares *parts* of *a* and *b*. Part values are:

```
IKS_ID_USER
IKS_ID_SERVER
IKS_ID_RESOURCE
```

You can combine this values with `or` operator. Some common combinations are predefined for you:

`IKS_ID_PARTIAL`
: `IKS_ID_USER | IKS_ID_SERVER`

`IKS_ID_FULL`
: `IKS_ID_USER | IKS_ID_SERVER | IKS_ID_RESOURCE`

Return value is `0` for equality. If entities are not equal a combination of part values showing different parts is returned.

2.4.3 Packet Filter

Packet filter handles routing incoming packets to related functions.

`iksfilter *` **iks_filter_new** (`void`); Function

Creates a new packet filter.

`void` **iks_filter_packet** (`iksfilter` **f*, `ikspak` **pak*); Function

Feeds the filter with given packet. Packet is compared to registered rules and hook functions of the matching rules are called in most matched to least matched order.

`void` **iks_filter_delete** (`iksfilter` **f*); Function

Frees filter and rules.

Rules are created with following function:

`iksrule *` **iks_filter_add_rule** (`iksfilter` **f*, `iksFilterHook` **filterHook*, `void` **user_data*, `...`); Function

Adds a rule to the filter *f*. *user_data* is passed directly to your hook function *filterHook*.

A rule consist of one or more type and value pairs. Possible types:

`IKS_RULE_ID`
: Compares `char *` value to packet ids.

`IKS_RULE_FROM`
: Compares `char *` value to packet senders.

`IKS_RULE_FROM_PARTIAL`
: Compares `char *` value to packet sender. Ignores resource part of jabber id.

`IKS_RULE_NS`
: Compares `char *` value to namespace of iq packets.

`IKS_RULE_TYPE`
: Compares `int` value to packet types.

`IKS_RULE_SUBTYPE`
: Compares `int` value to packet sub types.

`IKS_RULE_DONE`
: Terminates the rule pairs.

Here is an example which creates a filter and adds three rules:

```
iksfilter *f;

f = iks_filter_new ();
iks_filter_add_rule (f, on_msg, NULL,
                     IKS_RULE_TYPE, IKS_PAK_MESSAGE,
     IKS_RULE_DONE);
iks_filter_add_rule (f, on_auth_result, NULL,
                     IKS_RULE_TYPE, IKS_PAK_IQ,
     IKS_RULE_SUBTYPE, IKS_TYPE_RESULT,
     IKS_RULE_ID, "auth",
     IKS_RULE_DONE);
iks_filter_add_rule (f, on_roster_push, NULL,
                     IKS_RULE_TYPE, IKS_PAK_IQ,
     IKS_RULE_SUBTYPE, IKS_TYPE_SET,
     IKS_RULE_NS, "jabber:iq:roster",
     IKS_RULE_DONE);
```

iksFilterHook Typedef

int iksFilterHook (void *user_data, ikspak *pak);

Your hook is called with your *user_data* and matching packet *pak*. You can return two different values from your hook:

`IKS_FILTER_PASS`
: Packet is forwarded to least matching rules.

`IKS_FILTER_EAT`
: Filtering process for the packet ends.

You can remove the rules with following functions:

`void` **iks_filter_remove_rule** (`iksfilter` **f*, `iksrule` **rule*); Function

Removes the rule from filter.

`void` **iks_filter_remove_hook** (`iksfilter` **f*, `iksFilterHook` **filterHook*); Function

Remove the rules using *filterHook* function from filter.

2.4.4 Creating Common Packets

A usual jabber network traffic contains many similar XML constructs. iksemel provides several utility functions for creating them. They all generate an XML tree, so you can add or modify some parts of the tree, and send to server then.

`iks *` **iks_make_auth** (`iksid` **id*, `const char` **pass*, `const char` **sid*); Function

Creates an authorization packet. *id* is your parsed jabber id, and *pass* is your password.

If stream id *sid* isn't NULL, SHA1 authentication is used, otherwise password is attached in plain text. You can learn stream id from `IKS_STREAM_START` packet in your stream hook like this:

```
char *sid;

if (type == IKS_STREAM_START) {
    sid = iks_find_attrib (node, "id");
}
```

`iks *` **iks_make_msg** (`enum iksubtype` *type*, `const char *`*to*, `const char *`*body*); Function

Creates a message packet. *type* is the message type, *to* is jabber id of the recipient, *body* is the message.

`iks *` **iks_make_s10n** (`enum iksubtype` *type*, `const char *`*to*, `const char *`*msg*); Function

Creates a presence packet for subscription operations. *type* is operation, *to* is jabber id of the recipient, *msg* is a small message for introducing yourself, or explaning the reason of why you are subscribing or unsubscribing.

`iks *` **iks_make_pres** (`enum ikshowtype` *show*, `const char *`*status*); Function

Creates a presence packet for publishing your presence. *show* is your presence state and *status* is a message explaining why you are not available at the moment, or what you are doing now.

`iks *` **iks_make_iq** (`enum iksubtype` *type*, `const char *`*xmlns*); Function

Creates an IQ packet. *type* is operation type and *xmlns* is the namespace of the content. You usually have to add real content to the `<query>` tag before sending this packet.

2.5 Utility Functions

2.5.1 Memory Utilities

`void *` **iks_malloc** (`size_t` *size*); Function

`void` **iks_free** (`void *`*ptr*); Function

These are wrappers around ANSI malloc and free functions used by the iksemel library itself. You can free the output of iks_string (only if you passed it a NULL stack) with iks_free for example. That is important if you are using a malloc debugger in your application but not in iksemel or vice versa.

2.5.2 String Utilities

`char *` **iks_strdup** `(const char *`*src*`);` Function

`int` **iks_strcmp** `(const char *`*a*`, const char *`*b*`);` Function

`int` **iks_strcasecmp** `(const char *`*a*`, const char *`*b*`);` Function

`int` **iks_strncmp** `(const char *`*a*`, const char *`*b*`, size_t` *n*`);` Function

`int` **iks_strncasecmp** `(const char *`*a*`, const char *`*b*`, size_t` *n*`);` Function

`size_t` **iks_strlen** `(const char *`*src*`);` Function

These functions work exactly like their ANSI equivalents except that they allow NULL values for string pointers. If *src* is NULL, iks_strdup and iks_strlen returns zero. If *a* or *b* is NULL in string comparisation functions they return -1.

Their usefulness comes from the fact that they can chained with DOM traversing functions like this:

```
if (iks_strcmp (iks_find_attrib (x, "id"), "x1") == 0) count++;
```

That example works even x doesn't have an 'id' attribute and iks_find_attrib returns NULL. So you don't need to use temporary variables in such situations.

2.5.3 SHA1 Hash

Secure Hash Algorithm (SHA1) is used in the Jabber authentication protocol for encoding your password when sending to the server. This is normally handled by iks_make_auth() function, but if you want to handle it manually, or if you need a good hash function for other purproses you can use these functions.

`iksha*` **iks_sha_new** `(void);` Function

Allocates a structure for keeping calculation values and the state.

`void` **iks_sha_reset** `(iksha *`*sha*`);` Function

Resets the state of the calculation.

`void` **iks_sha_hash** `(iksha *`*sha*`, const unsigned char *`*data*`, int` *len*`, int` *finish*`);` Function

Calculates the hash value of the given data. If *finish* is non zero, applies the last step of the calculation.

`void` **iks_sha_print** `(iksha *`*sha*`, char *`*hash*`);` Function

Prints the result of a finished calculation into the buffer pointed by *hash* in hexadecimal string form. Buffer must be at least 40 bytes long. String is not null terminated.

`void` **iks_sha** `(const char *`*data*`, char *`*hash*`);` Function

Calculates the hash value of *data* and prints into *hash*. This is a helper function for simple hash calculations. It calls other functions for the actual work.

3 Development

This chapter contains information on plan, procedure and standarts of iksemel development.

3.1 Roadmap

There are three main functions iksemel tries to provide to applications:

- A generic XML parser with SAX and DOM interfaces.
- XML stream client and server functionality.
- Utilities for Jabber clients.

Goal of the iksemel is providing these functions while supporting embedded environments, keeping usage simple, and having a robust implementation.

Some decisions are made to reach this goal:

Code is written in ANSI C with a single dependency on C library. Instead of using expat or libxml, a simple built-in parser is used. Similarly glib and gnu only features of glibc (like object stacks) are avoided and built-in memory and string utilities are used. This may seem like code duplication but since they are optimized for iksemel and only a few kb in size, it isn't a big disadvantage.

Code is placed files in a modular fashion, and different modules don't depend on others' internal details. This allows taking unneeded functionality out when building for low resource situations.

It is tried to give functions names which are consistent, clear and short.

API is documented with texinfo for high quality printed output and info file output for fast and simple access during application development. Instead of using an autogenerated system or simply listing function descriptions, a task oriented tutorial approach is used.

3.2 Coding Style

Here is a short list describing preferred coding style for iksemel. Please keep in mind when sending patches.

- Indentation is done with tabs. Aligning is done with spaces.
- Placement of braces is K&R style.
- Function names are put at the start of line.
- Function names are lowercase.
- Words of the function names are separated with underscore character.
- Structure and variable names are lowercase.
- Macro and enumarations names are uppercase.
- Exported library API is contained in the single iksemel.h file.
- Exported function names start with iks_
- Exported structure and type names start with iks
- Exported macro and enumaration names start with IKS_

Here is an example:

```
int
iks_new_func (char *text)
{
    int i;

    i = an_internal_func (text);
    if (IKS_SOME_VALUE == i) {
        iks_some_func (text);
        i++;
    }
    return i;
}
```

3.3 Resources

- RFC 2279, UTF-8 format `http://www.ietf.org/rfc/rfc2279.txt`
- W3C Recommendation, Extensible Markup Language 1.0 `http://www.w3.org/TR/REC-xml`
- Annotated XML Specification `http://www.xml.com/axml/testaxml.htm`
- Jabber Protocol Documents `http://www.jabber.org/protocol/`

Datatype Index

Function Index