Writing Your First Instant Messaging Program

Okay, so assuming you have [libpurple installed](http://libpurple.com/how-to-install-libpurple/), **you can now write your first instant messaging program**. This will be written in C. If you’re not too familiar with C, don’t worry. I’ll guide you through it.

**First things first, you’ll need to open your terminal.** In this case, we’ll be using [bash](http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29) for the shell and [nano](http://en.wikipedia.org/wiki/Nano_%28text_editor%29) (same as [pico](http://en.wikipedia.org/wiki/Pico_%28text_editor%29)) for the text editor. **CD** to the directory where you want to run your program and create a file called **hello.c** (*user* is your username on this machine):

cd /home/*user*

nano hello.c

Now, I am going to jump straight into it and give you a fully working libpurple program. I could tell you what every single line does, but instead, I’m just going to show you which parts are necessary and which parts you can play with. The black parts are necessary, the green parts are suggested, and the blue parts are what you probably want to customize:

/\*

\* Sample libpurple program written by Michael C. Brook (http://libpurple.com/)

\* (Some fragments taken from libpurple nullclient.c example found at http://pidgin.im/)

\*/

#include "purple.h"

#include <glib.h>

#include <signal.h>

#include <string.h>

#include <unistd.h>

#include <stdio.h>

#define CUSTOM\_USER\_DIRECTORY "/dev/null"

#define CUSTOM\_PLUGIN\_PATH ""

#define PLUGIN\_SAVE\_PREF "/purple/user/plugins/saved"

#define UI\_ID "user"

/\*\*

\* The following eventloop functions are used in both pidgin and purple-text. If your

\* application uses glib mainloop, you can safely use this verbatim.

\*/

#define PURPLE\_GLIB\_READ\_COND (G\_IO\_IN | G\_IO\_HUP | G\_IO\_ERR)

#define PURPLE\_GLIB\_WRITE\_COND (G\_IO\_OUT | G\_IO\_HUP | G\_IO\_ERR | G\_IO\_NVAL)

typedef struct \_PurpleGLibIOClosure {

PurpleInputFunction function;

guint result;

gpointer data;

} PurpleGLibIOClosure;

typedef struct

{

PurpleAccountRequestType type;

PurpleAccount \*account;

void \*ui\_handle;

char \*user;

gpointer userdata;

PurpleAccountRequestAuthorizationCb auth\_cb;

PurpleAccountRequestAuthorizationCb deny\_cb;

guint ref;

} PurpleAccountRequestInfo;

static void purple\_glib\_io\_destroy(gpointer data)

{

g\_free(data);

}

static gboolean purple\_glib\_io\_invoke(GIOChannel \*source, GIOCondition condition, gpointer data)

{

PurpleGLibIOClosure \*closure = data;

PurpleInputCondition purple\_cond = 0;

if (condition & PURPLE\_GLIB\_READ\_COND)

purple\_cond |= PURPLE\_INPUT\_READ;

if (condition & PURPLE\_GLIB\_WRITE\_COND)

purple\_cond |= PURPLE\_INPUT\_WRITE;

closure->function(closure->data, g\_io\_channel\_unix\_get\_fd(source),

purple\_cond);

return TRUE;

}

static guint glib\_input\_add(gint fd, PurpleInputCondition condition, PurpleInputFunction function,

gpointer data)

{

PurpleGLibIOClosure \*closure = g\_new0(PurpleGLibIOClosure, 1);

GIOChannel \*channel;

GIOCondition cond = 0;

closure->function = function;

closure->data = data;

if (condition & PURPLE\_INPUT\_READ)

cond |= PURPLE\_GLIB\_READ\_COND;

if (condition & PURPLE\_INPUT\_WRITE)

cond |= PURPLE\_GLIB\_WRITE\_COND;

channel = g\_io\_channel\_unix\_new(fd);

closure->result = g\_io\_add\_watch\_full(channel, G\_PRIORITY\_DEFAULT, cond,

purple\_glib\_io\_invoke, closure, purple\_glib\_io\_destroy);

g\_io\_channel\_unref(channel);

return closure->result;

}

static PurpleEventLoopUiOps glib\_eventloops =

{

g\_timeout\_add,

g\_source\_remove,

glib\_input\_add,

g\_source\_remove,

NULL,

#if GLIB\_CHECK\_VERSION(2,14,0)

g\_timeout\_add\_seconds,

#else

NULL,

#endif

/\* padding \*/

NULL,

NULL,

NULL

};

/\*\*\* End of the eventloop functions. \*\*\*/

static void network\_disconnected(void)

{

printf("This machine has been disconnected from the internet\n");

}

static void report\_disconnect\_reason(PurpleConnection \*gc, PurpleConnectionError reason, const char \*text)

{

PurpleAccount \*account = purple\_connection\_get\_account(gc);

printf("Connection disconnected: \"%s\" (%s)\n >Error: %d\n >Reason: %s\n", purple\_account\_get\_username(account), purple\_account\_get\_protocol\_id(account), reason, text);

}

static PurpleConnectionUiOps connection\_uiops =

{

NULL, /\* connect\_progress \*/

NULL, /\* connected \*/

NULL, /\* disconnected \*/

NULL, /\* notice \*/

NULL, /\* report\_disconnect \*/

NULL, /\* network\_connected \*/

network\_disconnected, /\* network\_disconnected \*/

report\_disconnect\_reason, /\* report\_disconnect\_reason \*/

NULL,

NULL,

NULL

};

static void ui\_init(void)

{

/\*\*

\* This should initialize the UI components for all the modules.

\*/

purple\_connections\_set\_ui\_ops(&connection\_uiops);

}

static PurpleCoreUiOps core\_uiops =

{

NULL,

NULL,

ui\_init,

NULL,

/\* padding \*/

NULL,

NULL,

NULL,

NULL

};

static void init\_libpurple(void)

{

/\* Set a custom user directory (optional) \*/

purple\_util\_set\_user\_dir(CUSTOM\_USER\_DIRECTORY);

/\* We do not want any debugging for now to keep the noise to a minimum. \*/

purple\_debug\_set\_enabled(FALSE);

/\* Set the core-uiops, which is used to

\* - initialize the ui specific preferences.

\* - initialize the debug ui.

\* - initialize the ui components for all the modules.

\* - uninitialize the ui components for all the modules when the core terminates.

\*/

purple\_core\_set\_ui\_ops(&core\_uiops);

/\* Set the uiops for the eventloop. If your client is glib-based, you can safely

\* copy this verbatim. \*/

purple\_eventloop\_set\_ui\_ops(&glib\_eventloops);

/\* Set path to search for plugins. The core (libpurple) takes care of loading the

\* core-plugins, which includes the protocol-plugins. So it is not essential to add

\* any path here, but it might be desired, especially for ui-specific plugins. \*/

purple\_plugins\_add\_search\_path(CUSTOM\_PLUGIN\_PATH);

/\* Now that all the essential stuff has been set, let's try to init the core. It's

\* necessary to provide a non-NULL name for the current ui to the core. This name

\* is used by stuff that depends on this ui, for example the ui-specific plugins. \*/

if (!purple\_core\_init(UI\_ID)) {

/\* Initializing the core failed. Terminate. \*/

fprintf(stderr,

"libpurple initialization failed. Dumping core.\n"

"Please report this!\n");

abort();

}

/\* Create and load the buddylist. \*/

purple\_set\_blist(purple\_blist\_new());

purple\_blist\_load();

/\* Load the preferences. \*/

purple\_prefs\_load();

/\* Load the desired plugins. The client should save the list of loaded plugins in

\* the preferences using purple\_plugins\_save\_loaded(PLUGIN\_SAVE\_PREF) \*/

purple\_plugins\_load\_saved(PLUGIN\_SAVE\_PREF);

/\* Load the pounces. \*/

purple\_pounces\_load();

}

static void signed\_on(PurpleConnection \*gc)

{

PurpleAccount \*account = purple\_connection\_get\_account(gc);

printf("Account connected: \"%s\" (%s)\n", purple\_account\_get\_username(account), purple\_account\_get\_protocol\_id(account));

}

static void received\_im\_msg(PurpleAccount \*account, char \*sender, char \*message,

PurpleConversation \*conv, PurpleMessageFlags flags)

{

if (conv==NULL)

{

conv = purple\_conversation\_new(PURPLE\_CONV\_TYPE\_IM, account, sender);

}

printf("(%s) %s (%s): %s\n", purple\_utf8\_strftime("%H:%M:%S", NULL), sender, purple\_conversation\_get\_name(conv), message);

}

static void connect\_to\_signals(void)

{

static int handle;

purple\_signal\_connect(purple\_connections\_get\_handle(), "signed-on", &handle,

PURPLE\_CALLBACK(signed\_on), NULL);

purple\_signal\_connect(purple\_conversations\_get\_handle(), "received-im-msg", &handle,

PURPLE\_CALLBACK(received\_im\_msg), NULL);

}

int main(int argc, char \*argv[])

{

GMainLoop \*loop = g\_main\_loop\_new(NULL, FALSE);

/\* libpurple's built-in DNS resolution forks processes to perform

\* blocking lookups without blocking the main process. It does not

\* handle SIGCHLD itself, so if the UI does not you quickly get an army

\* of zombie subprocesses marching around.

\*/

signal(SIGCHLD, SIG\_IGN);

init\_libpurple();

printf("libpurple initialized. Running version %s.\n", purple\_core\_get\_version()); //I like to see the version number

connect\_to\_signals();

PurpleAccount \*account = purple\_account\_new("**YOUR\_IM\_ACCOUNTS\_USERNAME\_HERE**", "prpl-**IM\_NETWORK\_HERE**"); //this could be prpl-aim, prpl-yahoo, prpl-msn, prpl-icq, etc.

purple\_account\_set\_password(account, "**YOUR\_IM\_ACCOUNTS\_PASSWORD\_HERE**");

purple\_accounts\_add(account);

purple\_account\_set\_enabled(account, UI\_ID, TRUE);

g\_main\_loop\_run(loop);

return 0;

}

[**Download hello.c**](http://libpurple.com/wp-content/uploads/2010/09/hello.c)

I know this might look like one big scrambled mess, but for the time being, let’s not concern ourselves with what the program actually does. First, let’s just see if it works. **Copy the above program into the text editor in your terminal.** Then, make sure to replace **YOUR\_IM\_ACCOUNTS\_USERNAME\_HERE**, **YOUR\_IM\_ACCOUNTS\_PASSWORD\_HERE**, and **IM\_NETWORK\_HERE** with your IM account’s username, password, and network, respectively. Here’s a list of networks that libpurple supports by default. You will need to insert the “prpl-*whatever*” part in place of the network in your program:

AIM prpl-aim

Bonjour prpl-bonjour

Gadu-Gadu prpl-gg

GroupWise prpl-novell

ICQ prpl-icq

IRC prpl-irc

MSN prpl-msn

MySpaceIM prpl-myspace

QQ prpl-qq

SILC prpl-silc

SIMPLE prpl-simple

Sametime prpl-meanwhile

XMPP prpl-jabber

Yahoo prpl-yahoo

Yahoo JAPAN prpl-yahoojp

Zephyr prpl-zephyr

After you have edited the necessary parts of the program, save and close it (hit **CTRL+X**, **Y**,and **ENTER**). **Now, we need to compile the program.** To do so, enter the following command:

gcc `pkg-config --libs glib-2.0` -I /usr/include/libpurple/ -I /usr/include/glib-2.0/ -I /usr/lib/glib-2.0/include/ -lpurple hello.c -o hello

This is where you will find out if everything is working properly or not. **If you receive no output after running this command, congratulations! Everything is working fine.** However, if you receive any errors, it is likely to be related to your installation of libpurple or glib. If you have already gone through the [installation tutorial](http://libpurple.com/how-to-install-libpurple/) but you are still getting errors, feel free to leave a comment and we’ll see if we can sort it out.

*(Note: the paths used to include libpurple and glib in the command may depend on your operating system and the paths you have chosen to install each library in. This is confirmed to work at least for Ubuntu and Debian Linux using the default installation paths. Also note that using the* ***CC*** *compiler instead of* ***GCC*** *will work as well.)*

**Now, it is time to run your program!** Type in the terminal:

./hello

If all goes well, your account should be signed in through your program, and you should see something similar to:

libpurple initialized. Running version 2.6.2.

Account connected: "username" (prpl-network)

At the time this tutorial was written, version 2.6.2 was the latest stable version. And of course, you will see your own “username” and “prpl-network”. If you would like to exit the program, just hit **CTRL+C**. If your account was connected, it will be signed off.

For the time being, **the program will only print messages when your account is connected**, an **instant message is received**, the **IM account is disconnected because of an error**, or the **machine is disconnected from the internet**. These are just examples of events that you can track and perform any action from. You can actually track events anywhere from an account sign-on, to a buddy request being received, to when a buddy is typing to you, to pretty much anything else you can imagine. This is especially useful when you are working with databases and would like to update record values based on different events. You can also send out buddy requests, instant messages, programmatically show a buddy that you are typing to them, and all of the above. In short, you can programmatically do anything your instant messenger can do and more.

If you decide to go back and make any changes to your program, just remember to compile your program again afterwards, since the actual program is running from the compiled version (it is not automatically modified by simply saving the source code). For PHP developers (like me), this is something you need to remember.

Well, I hope you enjoyed the ride! Again, if you have trouble, don’t hesitate to leave a comment.

In the previous post, [Writing Your First Instant Messaging Program](http://libpurple.com/writing-your-first-instant-messaging-program/), we made an IM program that was only capable of capturing four different events. Those four events were when the **IM account was connected**, an **instant message was received from a buddy**, the **account was disconnected because of an error**, or when the **machine was disconnected from the internet**.

Well, while that’s all fine and dandy, you may want to add more events that your program should listen for, such as when a **buddy signs on or off**, a **buddy’s status changes (and to what status)**, someone sends you a **buddy request**, and maybe when a **buddy starts (or stops) typing to you**. Of course, there are more events that you can track, but these are some basic ones we can start adding to the events we already have. Hopefully, with some examples and some explanation, you will also be able to add more events in the future if need be.

So without further ado, let’s get our hands dirty… hence the picture of dirty hands… which aren’t mine by the way… because if they were, my computer would be slightly dirtier than it already is… ANYWAY…

To start off, here are some things that you should know. For whatever reason, libpurple has two ways of handling what I call “events”. The first is by using what’s called “**UI ops**“, which stands for “**User Interface Operations**“. The second is by using what are called “**Signals**“. **What’s the difference between the two?** Well, in short, UI ops are much more complicated. For this reason, I will mostly concentrate on signals, but I will introduce UI ops later on as well. In some cases, it may be beneficial to use UI ops instead of signals, but it depends on your requirements.

Signals

Going back to the code in my [previous post](http://libpurple.com/writing-your-first-instant-messaging-program/), let’s jump to the part that contains:

static void signed\_on(PurpleConnection \*gc)

{

PurpleAccount \*account = purple\_connection\_get\_account(gc);

printf("Account connected: \"%s\" (%s)\n", purple\_account\_get\_username(account), purple\_account\_get\_protocol\_id(account));

}

static void received\_im\_msg(PurpleAccount \*account, char \*sender, char \*message,

PurpleConversation \*conv, PurpleMessageFlags flags)

{

if (conv==NULL)

{

conv = purple\_conversation\_new(PURPLE\_CONV\_TYPE\_IM, account, sender);

}

printf("(%s) %s (%s): %s\n", purple\_utf8\_strftime("%H:%M:%S", NULL), sender, purple\_conversation\_get\_name(conv), message);

}

static void connect\_to\_signals(void)

{

static int handle;

purple\_signal\_connect(purple\_connections\_get\_handle(), "signed-on", &handle,

PURPLE\_CALLBACK(signed\_on), NULL);

purple\_signal\_connect(purple\_conversations\_get\_handle(), "received-im-msg", &handle,

PURPLE\_CALLBACK(received\_im\_msg), NULL);

}

Lucky for us, signals are pretty straightforward. We can start by looking at **connect\_to\_signals()**. In this function, we have set up two signals: one for **when the account has signed on**, and one for **when an IM is received**. Looking at each signal, you’ll notice that each signal corresponds to a **function**. This function is executed whenever the event occurs, and it brings some arguments along with it. Each type of event (or signal) executes a function with a different set of parameters. To figure out exactly what each function should look like, you can take a look here:

<http://developer.pidgin.im/doxygen/dev/html/pages.html>

After you click on the link for the appropriate type of signal you want, you will see a list of signal names and their functions. In the above example, the signal name goes in the part in **blue**.

Now that you know how signals work, let’s add a bunch more signals.

static void signed\_on(PurpleConnection \*gc)

{

PurpleAccount \*account = purple\_connection\_get\_account(gc);

printf("Account connected: \"%s\" (%s)\n", purple\_account\_get\_username(account), purple\_account\_get\_protocol\_id(account));

}

static void buddy\_signed\_on(PurpleBuddy \*buddy)

{

printf("Buddy \"%s\" (%s) signed on\n", purple\_buddy\_get\_name(buddy), purple\_account\_get\_protocol\_id(purple\_buddy\_get\_account(buddy)));

}

static void buddy\_signed\_off(PurpleBuddy \*buddy)

{

printf("Buddy \"%s\" (%s) signed off\n", purple\_buddy\_get\_name(buddy), purple\_account\_get\_protocol\_id(purple\_buddy\_get\_account(buddy)));

}

static void buddy\_away(PurpleBuddy \*buddy, PurpleStatus \*old\_status, PurpleStatus \*status)

{

printf("Buddy \"%s\" (%s) changed status to %s\n", purple\_buddy\_get\_name(buddy), purple\_account\_get\_protocol\_id(purple\_buddy\_get\_account(buddy)), purple\_status\_get\_id(status));

}

static void buddy\_idle(PurpleBuddy \*buddy, gboolean old\_idle, gboolean idle)

{

printf("Buddy \"%s\" (%s) changed idle state to %s\n", purple\_buddy\_get\_name(buddy), purple\_account\_get\_protocol\_id(purple\_buddy\_get\_account(buddy)), (idle) ? "idle" : "not idle");

}

static void received\_im\_msg(PurpleAccount \*account, char \*sender, char \*message, PurpleConversation \*conv, PurpleMessageFlags flags)

{

if (conv==NULL)

{

conv = purple\_conversation\_new(PURPLE\_CONV\_TYPE\_IM, account, sender);

}

printf("(%s) %s (%s): %s\n", purple\_utf8\_strftime("%H:%M:%S", NULL), sender, purple\_conversation\_get\_name(conv), message);

}

static void buddy\_typing(PurpleAccount \*account, const char \*name)

{

printf("User \"%s\" (%s) is typing...\n", name, purple\_account\_get\_protocol\_id(account));

}

static void buddy\_typed(PurpleAccount \*account, const char \*name) *//not supported on all protocols*

{

printf("User \"%s\" (%s) has typed something...\n", name, purple\_account\_get\_protocol\_id(account));

}

static void buddy\_typing\_stopped(PurpleAccount \*account, const char \*name)

{

printf("User \"%s\" (%s) has stopped typing...\n", name, purple\_account\_get\_protocol\_id(account));

}

static int account\_authorization\_requested(PurpleAccount \*account, const char \*user)

{

printf("User \"%s\" (%s) has sent a buddy request\n", user, purple\_account\_get\_protocol\_id(account));

return 1; *//authorize buddy request automatically (-1 denies it)*

}

static void connect\_to\_signals(void)

{

static int handle;

purple\_signal\_connect(purple\_connections\_get\_handle(), "signed-on", &handle,

PURPLE\_CALLBACK(signed\_on), NULL);

purple\_signal\_connect(purple\_blist\_get\_handle(), "buddy-signed-on", &handle,

PURPLE\_CALLBACK(buddy\_signed\_on), NULL);

purple\_signal\_connect(purple\_blist\_get\_handle(), "buddy-signed-off", &handle,

PURPLE\_CALLBACK(buddy\_signed\_off), NULL);

purple\_signal\_connect(purple\_blist\_get\_handle(), "buddy-status-changed", &handle,

PURPLE\_CALLBACK(buddy\_away), NULL);

purple\_signal\_connect(purple\_blist\_get\_handle(), "buddy-idle-changed", &handle,

PURPLE\_CALLBACK(buddy\_idle), NULL);

purple\_signal\_connect(purple\_conversations\_get\_handle(), "received-im-msg", &handle,

PURPLE\_CALLBACK(received\_im\_msg), NULL);

purple\_signal\_connect(purple\_conversations\_get\_handle(), "buddy-typing", &handle,

PURPLE\_CALLBACK(buddy\_typing), NULL);

purple\_signal\_connect(purple\_conversations\_get\_handle(), "buddy-typed", &handle,

PURPLE\_CALLBACK(buddy\_typed), NULL);

purple\_signal\_connect(purple\_conversations\_get\_handle(), "buddy-typing-stopped", &handle,

PURPLE\_CALLBACK(buddy\_typing\_stopped), NULL);

purple\_signal\_connect(purple\_accounts\_get\_handle(), "account-authorization-requested", &handle,

PURPLE\_CALLBACK(account\_authorization\_requested), NULL);

}

Hopefully that’s enough to get you started for now. Again, if you go to the link I mentioned earlier, there are many more signals you can use as well.

That’s about all there is to signals. Next, I’ll talk about tracking events using the more complicated UI ops.

UI ops (advanced)

Now, I want to jump to another portion of code from my [previous post](http://libpurple.com/writing-your-first-instant-messaging-program/). Note that some of the functionality of UI ops can be done using signals as well, however, that is not always the case.

static void network\_disconnected(void)

{

printf("This machine has been disconnected from the internet\n");

}

static void report\_disconnect\_reason(PurpleConnection \*gc, PurpleConnectionError reason, const char \*text)

{

PurpleAccount \*account = purple\_connection\_get\_account(gc);

printf("Connection disconnected: \"%s\" (%s)\n >Error: %d\n >Reason: %s\n", purple\_account\_get\_username(account), purple\_account\_get\_protocol\_id(account), reason, text);

}

static PurpleConnectionUiOps connection\_uiops =

{

NULL, /\* connect\_progress \*/

NULL, /\* connected \*/

NULL, /\* disconnected \*/

NULL, /\* notice \*/

NULL, /\* report\_disconnect \*/

NULL, /\* network\_connected \*/

network\_disconnected, /\* network\_disconnected \*/

report\_disconnect\_reason, /\* report\_disconnect\_reason \*/

NULL,

NULL,

NULL

};

static void ui\_init(void)

{

/\*\*

\* This should initialize the UI components for all the modules.

\*/

purple\_connections\_set\_ui\_ops(&connection\_uiops);

}

Looking at this code, instead of reading from top to bottom, read from bottom to top. At the bottom, you’ll notice that first **ui\_init()** calls **purple\_connections\_set\_ui\_ops(&connection\_uiops)**, where **connection\_uiops** is the list above **ui\_init()**. **connection\_uiops** is just a variable containing a list of functions. The functions in that list are functions that are executed when events occur. By making an element in that list **NULL**, you are basically ignoring that event. So in this example, we have two functions for two different events. These are **network\_disconnected** and **report\_disconnect\_reason**. The first is executed when you are **disconnected from the internet**. The second is executed when **one of your accounts has been disconnected** (possibly because of an incorrect password, or the account has signed on somewhere else, etc.). Whatever you put in those functions is entirely up to you. Here, they just print messages.

Now that I’ve given you the technical explanation, let’s look at the same example with one more event added to it.

static void \*request\_authorize(PurpleAccount \*account, const char \*remote\_user, const char \*id, const char \*alias, const char \*message, gboolean on\_list, PurpleAccountRequestAuthorizationCb authorize\_cb, PurpleAccountRequestAuthorizationCb deny\_cb, void \*user\_data)

{

printf("Buddy authorization request from \"%s\" (%s): %s\n", remote\_user, purple\_account\_get\_protocol\_id(account), message);

authorize\_cb(user\_data);

*//deny\_cb(user\_data);*

}

static void network\_disconnected(void)

{

printf("This machine has been disconnected from the internet\n");

}

static void report\_disconnect\_reason(PurpleConnection \*gc, PurpleConnectionError reason, const char \*text)

{

PurpleAccount \*account = purple\_connection\_get\_account(gc);

printf("Connection disconnected: \"%s\" (%s)\n >Error: %d\n >Reason: %s\n", purple\_account\_get\_username(account), purple\_account\_get\_protocol\_id(account), reason, text);

}

static PurpleAccountUiOps account\_uiops =

{

NULL, /\* notify\_added \*/

NULL, /\* status\_changed \*/

NULL, /\* request\_add \*/

request\_authorize, /\* request\_authorize \*/

NULL, /\* close\_account\_request \*/

NULL,

NULL,

NULL,

NULL

};

static PurpleConnectionUiOps connection\_uiops =

{

NULL, /\* connect\_progress \*/

NULL, /\* connected \*/

NULL, /\* disconnected \*/

NULL, /\* notice \*/

NULL, /\* report\_disconnect \*/

NULL, /\* network\_connected \*/

network\_disconnected, /\* network\_disconnected \*/

report\_disconnect\_reason, /\* report\_disconnect\_reason \*/

NULL,

NULL,

NULL

};

static void ui\_init(void)

{

/\*\*

\* This should initialize the UI components for all the modules.

\*/

purple\_connections\_set\_ui\_ops(&connection\_uiops);

purple\_accounts\_set\_ui\_ops(&account\_uiops);

}

Notice anything different? I have now added an event for **buddy requests**. First, because buddy requests are part of the **accounts UI ops**, I added **purple\_accounts\_set\_ui\_ops(&account\_uiops)** to **ui\_init()**. Then, I created the **account\_uiops** list, containing **request\_authorize**. And finally, I created the **request\_authorize** function at the top. This function will print a message when a buddy request is received and it will automatically authorize the buddy request. You can very easily make it deny the buddy request by switching the uncommented line with the commented one as well.

**You may be wondering** how I know what the **request\_authorize** function looks like, or how I know where to put the function in the **account\_uiops** list, or how I even knew to call **purple\_accounts\_set\_ui\_ops(&account\_uiops)**. Unfortunately, this is where Pidgin’s nasty documentation comes in. In general, you can find documentation on UI ops here:

<http://developer.pidgin.im/doxygen/dev/html/ui-ops.html>

If you click on the relevant UI op and then follow the first link, you will find a list of functions. You will notice that the list of functions is similar to the list of **UI ops** in the example I used, except many of the functions are set to **NULL**. Keep in mind that functions in the list of **UI ops** must be listed in the same order as they are in the documentation, and you must use **NULL** to fill any empty spaces.

That’s about all there is to capturing events. Hopefully this helps and makes things a little clearer. If you have any questions, please don’t hesitate to ask.