

Extending PHP

Wez Furlong

wez@thebrainroom.com

PHP{Con West, Santa Clara, October 2003

Extending PHP

Sooner or later, you will find that PHP doesn't do everything that you need it to do.

- **No bindings for a particular library**
- **Might want to integrate PHP more tightly into your existing systems (or vice versa)**
- **Maybe PHP script is too slow for performance critical parts of your application**

Rather than abandon (the otherwise perfect!) PHP and search for an alternative language, it could easily be worthwhile to extend PHP to make it do what you want.

What we will cover

- **Generating an extension skeleton (ext_skel)**
- **config.m4 and the PHP build system**
- **Extension Anatomy 101**
- **PHP_FUNCTION**
- **Resources**
- **Parameter passing**

We have 3 hours of interactive tutorial time; feel free to ask questions at any point. **Shout** if I don't seem to have noticed you!

To give us focus, I've picked on an asynchronous DNS resolving library and we'll create a PHP extension for it today, right now.

libares

Starting with PHP 4.3.0, it is now possible to write some pretty smart multiplexing networking applications **stream_select()**. The only problem is that the standard DNS functions **gethostbyname()** block the whole script until the domain name resolves.

Wouldn't it be nice to have an alternative method of resolving DNS that won't cause your existing sockets to time out?

We'll be using libares (written by Greg Hudson) as the basis of a PHP extension that provides such an alternative method. libares is released under an MIT license.

ares C API

Before we can write the extension, we have to have some knowledge of the code that we want to expose to our PHP scripts.

Focusing on replacing **gethostbyname()**, ares provides the following interesting functions:

interesting ares API

```
typedef void (*ares_host_callback)(void *arg, int status,
    struct hostent *hostent);

int ares_init(ares_channel *channelptr);

void ares_destroy(ares_channel channel);

void ares_gethostbyname(ares_channel channel, const char *name,
    int family, ares_host_callback callback, void *arg);

int ares_fds(ares_channel channel, fd_set *read_fds, fd_set *write_fds);

void ares_process(ares_channel channel, fd_set *read_fds,
    fd_set *write_fds);
```

Typical usage for these functions (taken from the ahost example code in ares distribution) looks something like this:

example of ares API in action

```
#include <ares.h>

/* gets called when the name has been resolved */
static void callback(void *arg, int status, struct hostent *host)
{
    char *mem, **p;
    struct in_addr addr;

    if (status != ARES_SUCCESS) {
        fprintf(stderr, "%s: %s\n", (char*)arg,
            ares_strerror(status, &mem));
        ares_free_errmem(mem);
    }

    for (p = host->h_addr_list; *p; p++) {
        memcpy(&addr, *p, sizeof(addr));
        printf("%s: %s\n", host->h_name, inet_ntoa(addr));
    }
}

int main(int argc, char **argv)
{
    ares_channel channel;
    char *errmem;
    int status, nfd;
    fd_set read_fds, write_fds;
    struct timeval *tvp, tv;

    /* create/initialize a channel for DNS communications */
```

```

status = ares_init(&channel);
if (status != ARES_SUCCESS) {
    fprintf(stderr, "ares_init: %s\n",
        ares_strerror(status, &errmem));
    ares_free_errmem(errmem);
    return 1;
}

/* for each command line arg */
for (argv++; *argv; argv++) {
    /* look up the domain name */
    ares_gethostbyname(channel, *argv, AF_INET, callback, *argv);
}

/* wait for the queries to complete */
while (1) {
    FD_ZERO(&read_fds);
    FD_ZERO(&write_fds);
    nfd = ares_fds(channel, &read_fds, &write_fds);
    if (nfd == 0)
        break;
    tvp = ares_timeout(channel, NULL, &tvp);
    select(nfd, &read_fds, &write_fds, NULL, tvp);
    /* calls the callback as appropriate */
    ares_process(channel, &read_fds, &write_fds);
}

/* all done */
ares_destroy(channel);
return 0;
}

```

extension functions

Based on the C api, a nice convenient PHP alternative might look like this:

PHP version of ares resolving

```
<?php

function gothost($hostname, $status, $hostent)
{
    if ($status != ARES_SUCCESS) {
        echo "Failed to resolve $hostname: "
            . ares_strerror($status) . "\n";
        return;
    }
    foreach ($hostent['addr_list'] as $ip) {
        echo "$hostent[name] -> $ip\n";
    }
}

/* for each argument, resolve it */
function lookup_hosts()
{
    $args = func_get_args();
    $resolver = ares_init();
    foreach ($args as $arg) {
        ares_gethostbyname($resolver, $arg, 'gothost', $arg);
    }

    // wait up to 2 seconds for hosts to be resolved
    while (ares_process_with_timeout($resolver, 2) > 0) {
        // timed out (2 seconds are up)
        // could do other stuff here while waiting
        echo ".";
    }

    ares_destroy($resolver);
}
?>
```

We have arrived at a list of 5 functions that we need to implement in our PHP extension:

- **ares_init()**
- **ares_gethostbyname()**
- **ares_process_with_timeout()**
- **ares_destroy()**
- **ares_strerror()**

Of these, **ares_process_with_timeout()** is special, as it wraps up a number of bits and pieces that are difficult to bring into user space. Later, we will look at plugging ares into **stream_select()**, but lets get this basic step out of the way first.

ext_skel

Make sure you have these things ready:

- **Latest PHP 4 source release**
- **ares headers and library**
- **Working build environment(!)**

First thing to do is to generate an extension skeleton which will contain all the stuff that PHP needs to be able to see your code.

generating a skeleton

```
% cd php4.3.x/ext
% ./ext_skel --extname=ares
```

ext_skel will produce output like this:

ext_skel output

```
Creating directory ares
Creating basic files: config.m4 .cvsignore ares.c php_ares.h CREDITS EXPERIMENTAL
tests/001.phpt ares.php [done].
```

To use your new extension, you will have to execute the following steps:

1. \$ cd ..
2. \$ vi ext/ares/config.m4
3. \$./buildconf
4. \$./configure --[with|enable]-ares
5. \$ make
6. \$./php -f ext/ares/ares.php
7. \$ vi ext/ares/ares.c
8. \$ make

Repeat steps 3-6 until you are satisfied with ext/ares/config.m4 and step 6 confirms that your module is compiled into PHP. Then, start writing code and repeat the last two steps as often as necessary.

As it suggests, looking in the config.m4 file is the first thing we need to do - we want configure to be able to find libares, so it is an essential step.

config.m4: the PHP build system

PHP has an excellent flexible build system. Each extension defines a config.m4 containing shell script (and m4 macros) to help locate libraries and headers that are required.

ext_skel has conveniently generated a template for us to use; all we need to do is uncomment the parts that apply to our extension.

ares config.m4

```

dnl $Id$
dnl config.m4 for extension ares

PHP_ARG_WITH(ares, for ares support,
[ --with-ares          Include ares support])

if test "$PHP_ARES" != "no"; then
# --with-ares -> check with-path
SEARCH_PATH="/usr/local /usr"
SEARCH_FOR="/include/ares.h"
if test -r $PHP_ARES/; then # path given as parameter
    ARES_DIR=$PHP_ARES
else # search default path list
    AC_MSG_CHECKING([for ares files in default path])
    for i in $SEARCH_PATH ; do
        if test -r $i/$SEARCH_FOR; then
            ARES_DIR=$i
            AC_MSG_RESULT(found in $i)
        fi
    done
fi

if test -z "$ARES_DIR"; then
    AC_MSG_RESULT([not found])
    AC_MSG_ERROR([Please reinstall the ares distribution])
fi

# --with-ares -> add include path
PHP_ADD_INCLUDE($ARES_DIR/include)

# --with-ares -> check for lib and symbol presence
LIBNAME=ares
LIBSYMBOL=ares_init

PHP_CHECK_LIBRARY($LIBNAME,$LIBSYMBOL,
[
    PHP_ADD_LIBRARY_WITH_PATH($LIBNAME, $ARES_DIR/lib, ARES_SHARED_LIBADD)
    AC_DEFINE(HAVE_ARESLIB,1,[ ])
],[
    AC_MSG_ERROR([wrong ares lib version or lib not found])
],[
    -L$ARES_DIR/lib -lm
])

PHP_SUBST(ARES_SHARED_LIBADD)

PHP_NEW_EXTENSION(ares, ares.c, $ext_shared)
fi
```

Extension Anatomy 101

php_ares.h as generated by ext_skel

```
/*
+-----+
| PHP Version 4                                     |
+-----+
| Copyright (c) 1997-2003 The PHP Group           |
+-----+
| This source file is subject to version 2.02 of the PHP license, |
| that is bundled with this package in the file LICENSE, and is  |
| available at through the world-wide-web at          |
| http://www.php.net/license/2_02.txt.              |
| If you did not receive a copy of the PHP license and are unable to |
| obtain it through the world-wide-web, please send a note to   |
| license@php.net so we can mail you a copy immediately.      |
+-----+
| Author:                                           |
+-----+

$Id: header,v 1.10.8.1 2003/07/14 15:59:18 sniper Exp $
*/

#ifndef PHP_ARES_H
#define PHP_ARES_H

extern zend_module_entry ares_module_entry;
#define phpext_ares_ptr &ares_module_entry

#ifdef PHP_WIN32
#define PHP_ARES_API __declspec(dllexport)
#else
#define PHP_ARES_API
#endif

#ifdef ZTS
#include "TSRM.h"
#endif

PHP_MINIT_FUNCTION(ares);
PHP_MSHUTDOWN_FUNCTION(ares);
PHP_RINIT_FUNCTION(ares);
PHP_RSHUTDOWN_FUNCTION(ares);
PHP_MINFO_FUNCTION(ares);

PHP_FUNCTION(confirm_ares_compiled); /* For testing, remove later. */

/*
    Declare any global variables you may need between the BEGIN
    and END macros here:

ZEND_BEGIN_MODULE_GLOBALS(ares)
    long global_value;
    char *global_string;
ZEND_END_MODULE_GLOBALS(ares)
*/

/* In every utility function you add that needs to use variables
in php_ares_globals, call TSRMLS_FETCH(); after declaring other
variables used by that function, or better yet, pass in TSRMLS_CC
after the last function argument and declare your utility function
with TSRMLS_DC after the last declared argument. Always refer to
the globals in your function as ARES_G(variable). You are
encouraged to rename these macros something shorter, see
examples in any other php module directory.
*/
```

```

#ifdef ZTS
#define ARES_G(v) TSRMLS_G(ares_globals_id, zend_ares_globals *, v)
#else
#define ARES_G(v) (ares_globals.v)
#endif

#endif /* PHP_ARES_H */

```

```

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * indent-tabs-mode: t
 * End:
 */

```

ares.c as generated by ext_skel

```

/*
+-----+
| PHP Version 4                                     |
+-----+
| Copyright (c) 1997-2003 The PHP Group           |
+-----+
| This source file is subject to version 2.02 of the PHP license, |
| that is bundled with this package in the file LICENSE, and is  |
| available at through the world-wide-web at          |
| http://www.php.net/license/2_02.txt.              |
| If you did not receive a copy of the PHP license and are unable to |
| obtain it through the world-wide-web, please send a note to   |
| license@php.net so we can mail you a copy immediately.      |
+-----+
| Author:                                           |
+-----+

$Id: header,v 1.10.8.1 2003/07/14 15:59:18 sniper Exp $
*/

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "php_ares.h"

/* If you declare any globals in php_ares.h uncomment this:
ZEND_DECLARE_MODULE_GLOBALS(ares)
*/

/* True global resources - no need for thread safety here */
static int le_ares;

/* {{{ ares_functions[]
 *
 * Every user visible function must have an entry in ares_functions[].
 */
function_entry ares_functions[] = {
    PHP_FE(confirm_ares_compiled, NULL) /* For testing, remove later. */
    {NULL, NULL, NULL} /* Must be the last line in ares_functions[] */
};
/* }}} */

/* {{{ ares_module_entry
 */
zend_module_entry ares_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    "ares",

```

```

    ares_functions,
    PHP_MINIT(ares),
    PHP_MSHUTDOWN(ares),
    PHP_RINIT(ares),      /* Replace with NULL if there's nothing to do at request
start */
    PHP_RSHUTDOWN(ares), /* Replace with NULL if there's nothing to do at request end
*/
    PHP_MINFO(ares),
#if ZEND_MODULE_API_NO >= 20010901
    "0.1", /* Replace with version number for your extension */
#endif
    STANDARD_MODULE_PROPERTIES
};
/* }}} */

#ifdef COMPILE_DL_ARES
ZEND_GET_MODULE(ares)
#endif

/* {{{ PHP_INI
*/
/* Remove comments and fill if you need to have entries in php.ini
PHP_INI_BEGIN()
    STD_PHP_INI_ENTRY("ares.global_value",      "42", PHP_INI_ALL, OnUpdateInt,
global_value, zend_ares_globals, ares_globals)
    STD_PHP_INI_ENTRY("ares.global_string", "foobar", PHP_INI_ALL, OnUpdateString,
global_string, zend_ares_globals, ares_globals)
PHP_INI_END()
*/
/* }}} */

/* {{{ php_ares_init_globals
*/
/* Uncomment this function if you have INI entries
static void php_ares_init_globals(zend_ares_globals *ares_globals)
{
    ares_globals->global_value = 0;
    ares_globals->global_string = NULL;
}
*/
/* }}} */

/* {{{ PHP_MINIT_FUNCTION
*/
PHP_MINIT_FUNCTION(ares)
{
    /* If you have INI entries, uncomment these lines
    ZEND_INIT_MODULE_GLOBALS(ares, php_ares_init_globals, NULL);
    REGISTER_INI_ENTRIES();
    */
    return SUCCESS;
}
/* }}} */

/* {{{ PHP_MSHUTDOWN_FUNCTION
*/
PHP_MSHUTDOWN_FUNCTION(ares)
{
    /* uncomment this line if you have INI entries
    UNREGISTER_INI_ENTRIES();
    */
    return SUCCESS;
}
/* }}} */

/* Remove if there's nothing to do at request start */
/* {{{ PHP_RINIT_FUNCTION
*/
PHP_RINIT_FUNCTION(ares)
{
    return SUCCESS;
}
/* }}} */

/* Remove if there's nothing to do at request end */

```

```

/* {{{ PHP_RSHUTDOWN_FUNCTION
*/
PHP_RSHUTDOWN_FUNCTION(ares)
{
    return SUCCESS;
}
/* }}} */

/* {{{ PHP_MININFO_FUNCTION
*/
PHP_MININFO_FUNCTION(ares)
{
    php_info_print_table_start();
    php_info_print_table_header(2, "ares support", "enabled");
    php_info_print_table_end();

    /* Remove comments if you have entries in php.ini
    DISPLAY_INI_ENTRIES();
    */
}
/* }}} */

/* Remove the following function when you have succesfully modified config.m4
so that your module can be compiled into PHP, it exists only for testing
purposes. */

/* Every user-visible function in PHP should document itself in the source */
/* {{{ proto string confirm_ares_compiled(string arg)
Return a string to confirm that the module is compiled in */
PHP_FUNCTION(confirm_ares_compiled)
{
    char *arg = NULL;
    int arg_len, len;
    char string[256];

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &arg, &arg_len) ==
FAILURE) {
        return;
    }

    len = sprintf(string, "Congratulations! You have successfully modified
ext/%.78s/config.m4. Module %.78s is now compiled into PHP.", "ares", arg);
    RETURN_STRINGL(string, len, 1);
}
/* }}} */
/* The previous line is meant for vim and emacs, so it can correctly fold and
unfold functions in source code. See the corresponding marks just before
function definition, where the functions purpose is also documented. Please
follow this convention for the convenience of others editing your code.
*/

/*
* Local variables:
* tab-width: 4
* c-basic-offset: 4
* End:
* vim600: noet sw=4 ts=4 fdm=marker
* vim<600: noet sw=4 ts=4
*/

```

ares_init - resources

The first function that we are going to implement is `ares_init()`. In the C library, `ares_init` initializes a structure and then all other functions operate on a pointer to that structure. Since we don't have structures and pointers in PHP, we need to declare a resource type.

Resources are implemented in PHP using a global (per request) linked list of resource identifiers. Each individual resource has associated with it:

- A resource "type" id
- A pointer value

In order to return a resource to the script, we will need to register a resource type when our extension is loaded. We need to do this so that the Zend Engine knows how to release the resource when the script terminates - we register the resource with a destructor function.

registering a resource

```
/* True global resources - no need for thread safety here */
static int le_ares_channel;

static void ares_channel_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    /* TODO: rsrc->ptr need to be released correctly */
}

/* {{{ PHP_MSHUTDOWN_FUNCTION
*/
PHP_MINIT_FUNCTION(ares)
{
    le_ares_channel = zend_register_list_destructors_ex(ares_channel_dtor,
        NULL, "ares channel", module_number);

    return SUCCESS;
}
*/
```

Now we need to write the PHP function itself.

implementing ares_init()

```
/* {{{ proto resource ares_init()
   Creates a DNS resolving communications channel */
PHP_FUNCTION(ares_init)
{
    ares_channel channel;
    int status;

    status = ares_init(&channel);

    if (status != ARES_SUCCESS) {
```

```

    char *errmem;

    ares_strerror(status, &errmem);
    php_error_docref(NULL TSRMLS_CC, E_WARNING, "failed to init ares channel: %s",
        errmem);
    ares_free_errmem(errmem);

    RETURN_NULL();
}

ZEND_REGISTER_RESOURCE(return_value, channel, le_ares_channel);
}
/* }}} */

```

- **ares_channel** is typedef'd as a pointer type
- If the channel couldn't be created, a regular php **E_WARNING** is raised, and **NULL** is returned.

For the sake of completeness, lets implement `ares_destroy()` now too. It demonstrates how to accept a resource type as a parameter.

ares_destroy()

```

/* {{{ proto void ares_destroy(resource $channel)
   Destroys a DNS resolving channel */
PHP_FUNCTION(ares_destroy)
{
    zval *r;
    ares_channel channel;

    if (FAILURE == zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
        "r", &r)) {
        return;
    }

    ZEND_FETCH_RESOURCE(channel, ares_channel, &r, -1, "ares channel",
        le_ares_channel);

    zend_list_delete(Z_LVAL_P(r));
}
/* }}} */

```

- **ZEND_FETCH_RESOURCE** will automatically emit an error and return from the function if the `$channel` parameter is not a valid channel resource (eg: stream or image)

Almost done - now that we know how we are creating the channel, we should fill in that dtor so it really does some work.

registering a resource

```

static void ares_channel_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    ares_channel channel = (ares_channel)rsrc->ptr;
    ares_destroy(channel);
}

```

ares_gethostbyname()

Now we can write the real core of the extension - the resolver. We decided that the API for the function would work like this:

function prototype for ares_gethostbyname

```
<?php
ares_gethostbyname($channel, $hostname, $callback, $callbackarg);
?>
```

So we need to write a PHP_FUNCTION that accepts a resource, a string, and two generic zval values as parameters.

ares_gethostbyname() in C

```
/* {{{ proto void ares_gethostbyname(resource $channel, string $hostname, mixed
$callback, mixed $arg)
   Initiate resolution of $hostname; will call $callback with $arg when complete */
PHP_FUNCTION(ares_gethostbyname)
{
    zval *zchannel, *zcallback, *zarg;
    char *hostname;
    long hostname_len;
    ares_channel channel;

    if (FAILURE == zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rszz",
        &zchannel, &hostname, &hostname_len, &zcallback, &zarg)) {
        return;
    }
    ZEND_FETCH_RESOURCE(channel, ares_channel, &zchannel, -1, "ares channel",
        le_ares_channel);

    /* more code to go here ... */

}
/* }}} */
```

Now, we've hit a little bit of a hurdle - we need to call back into the engine when the resolution completes. There is a little bit of magic required, since we can't call a PHP_FUNCTION directly from C (well, we can, but we can't call it directly from this callback). We need to define a structure to hold the function name (or object + method name) of the callback, as well as the argument. We'll pass those to ares and have a little stub C function to act as the real callback:

structure and callback for gethostbyname

```
struct php_ares_callback_struct {
    zval *callback;
    zval *arg;
};

static void php_ares_hostcallback(void *arg, int status, struct hostent *host)
```

```

{
    struct php_ares_callback_struct *cb = (struct php_ares_callback_struct *)arg;

    /* TODO: map hostent to php and call the callback */
}

```

Let's jump back to the `gethostbyname` function and fill in the callback structure:

ares_gethostbyname() continued

```

/* {{{ proto void ares_gethostbyname(resource $channel, string $hostname, mixed
$callback, mixed $arg)
    Initiate resolution of $hostname; will call $callback with $arg when complete */
PHP_FUNCTION(ares_gethostbyname)
{
    zval *zchannel, *zcallback, *zarg;
    char *hostname, *callback_name;
    long hostname_len;
    ares_channel channel;
    struct php_ares_callback_struct *cb;

    if (FAILURE == zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rszz",
        &zchannel, &hostname, &hostname_len, &zcallback, &zarg)) {
        return;
    }
    ZEND_FETCH_RESOURCE(channel, ares_channel, &zchannel, -1, "ares channel",
        le_ares_channel);

    /* verify that the callback will work */
    if (!zend_is_callable(zcallback, 0, &callback_name)) {
        php_error_docref1(NULL TSRMLS_CC, callback_name, E_WARNING,
            "3rd argument is not a valid callback");
        efree(callback_name);
        return;
    }

    /* copy the values into the structure */
    cb = (struct php_ares_callback_struct*)emalloc(sizeof(*cb));

    MAKE_STD_ZVAL(cb->callback);
    *cb->callback = *zcallback;
    zval_copy_ctor(*cb->callback);

    MAKE_STD_ZVAL(cb->arg);
    *cb->arg = *zarg;
    zval_copy_ctor(*cb->arg);

    ares_get_hostbyname(channel, hostname, AF_INET,
        php_ares_hostcallback, cb);
}
/* }}} */

```

We can come back to the callback now - we have another little problem - we can't pass a struct `hostent` directly to a script, so we need to copy the data from it into a `zval` that can be passed. Looking back at our API sample, we chose to use an array with keys named similarly to the C structure:

the sample PHP version of the callback

```

<?php

function gothost($hostname, $status, $hostent)
{

```

```

        if ($status != ARES_SUCCESS) {
            echo "Failed to resolve $hostname: "
              . ares_strerror($status) . "\n";
            return;
        }
        foreach ($hostent['addr_list'] as $ip) {
            echo "$hostent[name] -> $ip\n";
        }
    }
?>

```

So the task is now to create an array and set it up as follows:

layout of the hostent parameter

```

<?php
    $hostent = array();
    $hostent['name'] = $hostname;
    $hostent['addr_list'] = array();
    $hostent['addr_list'][] = $ip1;
?>

```

The C translation of that code looks like this:

building a hostent array in C

```

static void php_ares_hostcallback(void *arg, int status, struct hostent *host)
{
    struct php_ares_callback_struct *cb = (struct php_ares_callback_struct *)arg;
    struct in_addr addr;
    char **p;
    zval *hearray, *addr_list;

    MAKE_STD_ZVAL(hearray);
    array_init(hearray);

    add_assoc_string(hearray, "name", host->h_name, 1);

    MAKE_STD_ZVAL(addr_list);
    array_init(addr_list);

    for (p = host->h_addr_list; *p; p++) {
        memcpy(&addr, *p, sizeof(addr));
        add_next_index_string(addr_list, inet_ntoa(addr), 1);
    }

    add_assoc_zval(hearray, "addr_list", addr_list);
}

```

Now we get to the most important part - calling the callback.

using call_user_function to call the callback

```

static void php_ares_hostcallback(void *arg, int status, struct hostent *host)
{
    zval retval;
    zval *arguments[3];
    struct php_ares_callback_struct *cb = (struct php_ares_callback_struct *)arg;

    /* insert code for mapping the hostent to an array (from above) here */

    arguments[0] = cb->arg;

    MAKE_STD_ZVAL(arguments[1]);
    ZVAL_LONG(arguments[1], status);
}

```

```
arguments[1] = hearray;

if (call_user_function(EG(function_table), NULL,
    cb->callback, &retval, 3, arguments TSRMLS_CC) == SUCCESS) {
    /* release any returned zval - it's not important to us */
    zval_dtor(&retval);
}

FREE_ZVAL(arguments[1]);

/* and clean up the structure */
zval_dtor(cb->callback);
zval_dtor(cb->arg);
zval_dtor(hearray);
efree(cb);
}
```

ares_process_with_timeout

This function provides the easy way to check on the status of the DNS resolution, and call the callbacks if it has completed. For convenience we can specify a timeout so we can "sleep" on it while waiting.

Lets take a look at the fragment of code from the ahost example again:

fragment from ahost

```
int main(int argc, char **argv)
{
    ares_channel channel;
    int status, nfd;
    fd_set read_fds, write_fds;
    struct timeval *tvp, tv;

    /* ... */

    /* wait for the queries to complete */
    while (1) {
        FD_ZERO(&read_fds);
        FD_ZERO(&write_fds);
        nfd = ares_fds(channel, &read_fds, &write_fds);
        if (nfd == 0)
            break;
        tvp = ares_timeout(channel, NULL, &tv);
        select(nfd, &read_fds, &write_fds, NULL, tvp);
        /* calls the callback as appropriate */
        ares_process(channel, &read_fds, &write_fds);
    }

    /* ... */
}
```

We'll do things a little differently - we will be specifying our own timeout.

implementing the function

```
/* {{{ proto long ares_process_with_timeout(resource $channel, long $secs)
   check for completion of DNS resolution and call callbacks. Returns number of
   outstanding requests. */
PHP_FUNCTION(ares_process_with_timeout)
{
    zval *zchannel;
    ares_channel channel;
    long timeout_secs;
    struct timeval tv;
    int status, nfd;
    fd_set read_fds, write_fds;

    if (FAILURE == zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rl",
        &zchannel, &timeout_secs)) {
        return;
    }

    ZEND_FETCH_RESOURCE(channel, ares_channel, &zchannel, -1, "ares channel",
        le_ares_channel);

    FD_ZERO(&read_fds);
```

```
FD_ZERO(&write_fds);
nfds = ares_fds(channel, &read_fds, &write_fds);
if (nfds) {
    tv.tv_secs = timeout_secs;
    tv.tv_usec = 0;
    if (select(nfds, &read_fds, &write_fds, NULL, &tv) > 0) {
        /* calls the callback as appropriate */
        ares_process(channel, &read_fds, &write_fds);
    }
    RETURN_LONG(nfds);
} else {
    RETURN_LONG(0);
}
}
/* }}} */
```

Nice things for error handling

If the DNS resolution fails for some reason, we might want to notify the user of the error message, or we might want to act upon the error code and take some alternative course of action. To do these things we need to have a function to retrieve the error message text, and it would also be nice to have some symbolic constants to describe the error codes.

implementing ares_strerror

```
/* {{{ proto string ares_strerror(long $statuscode)
   returns a string description for an error code */
PHP_FUNCTION(ares_strerror)
{
    long statuscode;
    char *errmem = NULL;

    if (FAILURE == zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l",
        &statuscode)) {
        return;
    }

    ares_strerror(status, &errmem);
    RETVAL_STRING(return_value, errmem, 1);
    ares_free_errmem(errmem);
}
/* }}} */
```

The other nice thing to have is symbolic constants. We can do the equivalent of **define()** when our extension is loaded. We do this in our module init (MINIT) function, just after the point where we register our resource type.

error codes defined by libares

```
#define ARES_SUCCESS          0

/* Server error codes (ARES_ENODATA indicates no relevant answer) */
#define ARES_ENODATA         1
#define ARES_EFORMERR        2
#define ARES_ESERVFAIL       3
#define ARES_ENOTFOUND       4
#define ARES_ENOTIMP         5
#define ARES_EREREFUSED      6

/* Locally generated error codes */
#define ARES_EBADQUERY        7
#define ARES_EBADNAME         8
#define ARES_EBADFAMILY       9
#define ARES_EBADRESP        10
#define ARES_ECONNREFUSED     11
#define ARES_ETIMEOUT         12
#define ARES_EOF              13
#define ARES_EFILE            14
#define ARES_ENOMEM           15
#define ARES_EDESTRUCTION     16
```

To map these into user space:

registering a resource

```
/* {{{ PHP_MSHUTDOWN_FUNCTION
*/
PHP_MINIT_FUNCTION(ares)
{
    le_ares_channel = zend_register_list_destructors_ex(ares_channel_dtor,
        NULL, "ares channel", module_number);

    REGISTER_LONG_CONSTANT("ARES_SUCCESS", ARES_SUCCESS, CONST_CS|CONST_PERSISTENT);
    REGISTER_LONG_CONSTANT("ARES_ENODATA", ARES_ENODATA, CONST_CS|CONST_PERSISTENT);
    REGISTER_LONG_CONSTANT("ARES_EFORMERR", ARES_EFORMERR, CONST_CS|CONST_PERSISTENT);
    /* ... */

    return SUCCESS;
}
*/
```

Summary

We can now make asynchronous DNS resolution requests, and even run them concurrently. Along the way we have learned about the major parts of writing a PHP extension:

- **config.m4**
- **general structure**
- **PHP_FUNCTION**
- **zend_parse_parameters**
- **making callbacks**
- **registering resources**
- **registering constants**
- **returning values to the script**

how to use with stream_select()

```
<?php
$s1 = fsockopen($host1, $port);
$s2 = fsockopen($host2, $port);
$chan = ares_init();

ares_gethostbyname($chan, 'sample.com', 'sample.com');

/* wait for things to come back */
while (true) {
    /* don't wait for anything - return immediately */
    ares_process_with_timeout($chan, 0);
    /* wait 5 seconds for data on the sockets */
    $r = array($s1, $s2);
    $n = stream_select($r, $w = null, $e = null, 5);
    if ($n) {
        /* do something with it */
    }
}
?>
```