# `CodeGen_PECL` - the PHP extension generator

# `CodeGen_PECL` - the PHP extension generator

# Table of Contents

# List of Examples

# Chapter 1. Introduction

## 1.1. What is it?

`CodeGen_PECL` (formerly known as `PECL_Gen`) is a tool that can automatically create the basic framework for a PHP extension from a rather simple XML specification file. The actual functionality is provided by the script `pecl-gen` that is installed by the `CodeGen_PECL` package.

It also supports the simpler (but less powerful) prototype file format as used by the shell script `ext_skel` that is distributed togehter with the PHP source code.

`pecl-gen`, unlike the older `ext_skel` solution, is a 100% PHP 5 based solution and does not require any external tools like `awk` or `sed`. It only uses PHP functions and PEAR modules that are always enabled in a default build and so it should be usable on any platform that PHP itself runs on.

PHP 5 is only required for the conversion of the XML spec file to C/C++ code. The code generated by `CodeGen_Pecl` is designed to work with both the extension APIs of PHP 5 and PHP 4 as long as only procdural features are used. Object oriented features are only supported for PHP 5 as the OO-specific API changed too much between PHP 4 and 5 to generate code for both.

## 1.2. Features

`CodeGen_PECL` tries to support as many extension writing aspects as possible. This currently includes code and documentation generation for:

- functions
- constants
- `php.ini` configuration directives
- resource types
- per-thread global variables
- test cases

`CodeGen_PECL` also generates `config.m4` configuration files for Unix-like build environments, VisualStudio `*.dsp` project files for Windows and the `package.xml` files needed for PEAR/PECL packaging. Support for the new Windows Scripting Host based build system is currently being worked on, `config.w32` files are already generated but may only work for simple configuration setups for now.

DocBook XML documentation templates are created that can either be used to generate stadalone documentation or that can be integrated into the PHP manual by simply copying over the generated documentation directory.

Test cases are created for each generated function, additional test cases may also be specified.

# 1.3. Installation

`CodeGen_PECL` is available in PEAR, the PHP Extension and Application Repository: `http://pear.php.net/CodeGen_PECL`

## 1.3.1. Online installation

Online installation using the PEAR installer is the easiest way to install `CodeGen_PECL`, just issue the following command:

```
pear install --alldeps CodeGen_PECL
```

The PEAR installer will download and install the package itself and all packages that it depends on.

## 1.3.2. Installing from package files

When installing from package files downloaded from `pear.php.net` you have to resolve dependencies yourself. Currently `CodeGen_PECL` depends on two other PEAR packages: `Console_Getopt`, which is part of the PEAR base installation, and `CodeGen`, the code generator base package. You need to download both `CodeGen` and `CodeGen_PECL` packages for installation. The actual installation is once again performed by the PEAR installer:

```
pear install CogeGen-1.0.1.tgz
pear install CogeGen_PECL-1.0.3.tgz
```

## 1.3.3. Installing from PEAR CVS

You can also install CodeGen_PECL snapshots from PEAR CVS. CVS snapshots may include features not yet available in any release package, but the code in CVS may not be as well tested as the release packages (or even broken at times). Be warned, your milage may vary. Use the following sequence of commands in your PEAR CVS checkout to install the latest `CodeGen_PECL` snapshot:

```
cd pear
cd CodeGen
cvs update
pear install --force package.xml
cd ..
cd CodeGen_PECL
cvs update
pear install --force package.xml
cd ..
```

## 1.4. How to use it

There are three different modes of operation for `pecl-gen`. In its default mode it can create a complete ready-to-compile extension from an XML description (documented in the next chapter). In `ext_skel` compatibility mode it generates the extension from some command line parameters and an optional function prototype file and in immediate mode it just takes a function prototype from command line and writes a C code skeleton for just that function to standard output.

`ext_skel` compatibility and immediate mode are not documented here, please refer to the original `ext_skel` documentation instead.

A more detailed step by step guide on how to invocate `pecl-gen` and how to procede to configure, compile, test and install an extension is given in the "Usage" chapter later in this document.

Below you find a hardcopy of the `pecl-gen --help` output:

```
pecl-gen 1.0.3, Copyright (c) 2003-2005 Hartmut Holzgraefe

Usage:

pecl-gen [-h] [--force] [--experimental] [--version]
  [--extname=name] [--proto=file] [--skel=dir] [--stubs=file]
  [--no-help] [--xml[=file]] [--full-xml] [--function=proto] [specfile.xml]

  -h|--help          this message
  -f|--force         overwrite existing directories
  -d|--dir           output directory (defaults to extension name)
  -l|--lint          check syntax only, don't create output
  --linespecs        generate #line specs
  -x|--experimental  deprecated
  --function         create a function skeleton from a proto right away
  --version          show version info
```

```
the following options are inherited from ext_skel:
--extname=module   module is the name of your extension
--proto=file       file contains prototypes of functions to create
--xml              generate xml documentation to be added to phpdoc-cvs

these wait for functionality to be implemented and are ignored for now ...
--stubs=file       generate only function stubs in file
--no-help          don't try to be nice and create comments in the code
                   and helper functions to test if the module compiled

these are accepted for backwards compatibility reasons but not used ...
--full-xml         generate xml documentation for a self-contained extension
                   (this was also a no-op in ext_skel)
--skel=dir         path to the skeleton directory
                   (skeleton stuff is now self-contained)
```

# Chapter 2. The XML description

## 2.1. Basics

The top level container tag describing an extension is the `<extension>` tag. The name of the extension is given in the `name=...` attribute. The extension name has to be a valid C name as it is used as both the extensions directory name and the base name for several C symbols within the generated C code.

You can specify which CodeGen_PECL version your specification file was written against using the `version=...` attribute. The `pecl-gen` command will not accept specifications written for a newer version of CodeGen_PECL than the one installed. If the requested version is older then the current one then CodeGen_PECL will try to fall back to the older versions behavior for features that have changed in incompatible ways.

> **Note:** So far two such changes have happened: in version 0.9.0 a new prototype parser was introduced that might not be 100% backwards compatible with the older one and with 1.0.0 the names of some of the variables in the generated code changed.

The tags `<summary>` and `<description>` should be added at the very top of your extensions. The summary should be a short one-line description of the extension while the actually description can be as detailed as you like. Both are later used to generate the `package.xml` file and the documentation for your extension. The summary line is also put into the `phpinfo()` output of your extension.

**Example 2-1. Extension basics**

```
<?xml version="1.0" ?>
<!DOCTYPE extension SYSTEM "../extension.dtd">
<extension name="sample" version="0.9.0">
 <summary>A sample PHP extension</summary>
 <description>
  This is a sample extension specification
  showing how to use CodeGen_PECL for
  extension generation.
 </description>
 ...
```

## 2.2. Release information

The release information for your extension should include the extension authors and maintainers, the version number, state and release date, the chosen license and maybe a change log describing previous releases. It is also possible to specify an image file to be used as a product logo with the `phpinfo()` output block for the extension.

The `<maintainers>`, `<release>` and `<changelog>` tags specifications are identical to those in the PEAR `package.xml` specification so please refer to the PEAR documentation (http://pear.php.net/manual/en/developers.packagedef.php) here.

**Example 2-2. Release information**

```
...
  <maintainers>
    <maintainer>
      <user>hholzgra</user>
      <name>Hartmut Holzgraefe</name>
      <email>hartmut@php.net</email>
      <role>lead</role>
    </maintainer>
  </maintainers>

  <release>
    <version>1.0</version>
    <date>2002-07-09</date>
    <state>stable</state>
    <notes>
     The sample extension is now stable
    </notes>
  </release>

  <changelog>
    <release>
      <version>0.5</version>
      <date>2002-07-05</date>
      <state>beta</state>
      <notes>First beta version</notes>
    <release>
    <release>
      <version>0.1</version>
      <date>2002-07-01</date>
      <state>alpha</state>
      <notes>First alpha version</notes>
    <release>
  </changelog>
...
```

The `<license>` tag is a little more restrictive as its `package.xml` counterpart as it is used to decide which license text should actually be written to the `LICENSE`. For now you have to specify either `PHP`, `BSD` or `LGPL`, any other value is taken as '*unknown*'.

> **Warning**
>
> The GPL is not available as a valid license here due to incompatibilities between the GPL and the PHP licese. Please refer to the FSF license comparison page (http://www.fsf.org/licensing/licenses/index_html) for further information on this issue.

**Example 2-3. License**

```
...
  <license>PHP</license>
...
```

A logo to be used within the extensions `phpinfo()` block can be specified using the `<logo>` tag. The actual logo image data may be read from a file specified by the `scr=...` attribute, or it may be included inline in base64 encoded form within the `<logo>` tag. Its MIME type may be specified using the `mimetype=...` attribute. Automatic MIME type detection exists for GIF, PNG and JPEG images.

**Example 2-4. Loading a logo image from a file**

```
...
  <logo src="sample_logo.gif" mimetype="image/gif" />
...
```

**Example 2-5. An inline logo image**

```
...
  <logo>
<![CDATA[
R0lGODdhFQASAOMAAMDEwJCQkLCwsGhoaFhcWLjAuDA0MPj8+FBUUPj4+AAA
AAAAAAAAAAAAAAAAAAAACwAAAAAFQASAAAEehDISWsNQIhBuv+DphWhABhH
qq5GpgHluc5HK24vmiKGwfeIgdAU0wkSyCQSkAhgQhgdQUlNCAKkq0BaVWqh
29S0i8QRtFyyNXQOhA9jshktVq8F7Xe8O3en5WxXTgEcdn2DAwF2hHiCGoQc
HASSQgRukwiZmpucmwYRADs=
]]>
  </logo>
...
```

# 2.3. Dependencies

Dependencies are specified within the `<deps>` environment. Within the `<deps>` section itself it is possible to set the programming language and target platforms using the `language=...` and `platform=...` attributes.

Supported languages are C (`lang="c"`) and C++ (`lang="cpp"`). The language selection does not influence code generation itself (`pecl-gen` always generates C code as the PHP extension API is a pure C API) but the way extensions are compiled and linked. C++ should only be selected to interface to external C++ libraries.

> **Note:** I've been thinking about a Delphi backend, too. But as this would require a substantial ammount of work (the complete code generation would have to become template based) i'm not going to start this unless there's massive interest in something like this or unless someone is willing to financially sponsor this kind of work.

Supported platforms are currently Unix-like systems (`platform="unix"`), Microsoft Windows (`platform="win32"`) or both (`platform="all"`).

`<with>`, `<lib>` and `<header>` tags may be used within the `<deps>` section to add configure switches and library and header file dependencies.

**Example 2-6. Dependencies**

```
...
  <deps language="cpp" platform="win32">
  ...
  </deps>
...
```

## 2.3.1. `--with...`

When building an extension on Unix-like systems or within the Cygwin environment under Windows the `configure` script will try to figure out where external libraries and header files needed by an extension are installed on the build system. Using a `--with-...` option it is possible to specify where to actually look for libraries and headers. This way it is possible to override search paths if things are not installed in the default system paths or to specify the exact version of a package to be used if multiple versions are installed on the target system.

The <with> tag takes three attributes: name=... for the actual name of the --with-... option, testfile for the relative path of a file to check for while running the configure script and a list of default paths to check if no path is given as an argument to the --with-... option in defaults.

Name and defaults are set to the extension base name and /usr:/usr/local if no values are given. The testfile attribute is mandatory.

Textual data enclosed by the <with> is used to describe the "with" option in the output of configure --help calls.

**Example 2-7. --with...**

```
...
    <with defaults='/usr:/usr/local' testfile='include/sample.h'>sample install</with>
...
```

## 2.3.2. Header files

It is possible to specify header files needed by the extension using the <header>. Any headers specified have to exist in the include path set for compiling (see also the section on --with above). #include statements for the specified headers are the last ones to be put into the generated code unless you set the prepend="yes" attribute to have it put in front of the other #includes.

By default header files are searched for in the include subdirectory of the path given in <with>. If a different relative path needs to be used it can be defined using the path attribute.

**Example 2-8. Header file dependencies**

```
...
    <header name="include_me_first.h" prepend="yes" />
    <header name="sample.h" />
    <header name="foobar.h" path="include/foo/bar" />
...
```

## 2.3.3. Libraries

Needed external libraries are specified using the <lib> tag. The name=... attribute is mandatory and takes the library base name. A library dependency by the name "sample" is actually referring to a library file named libsample.a for a static or libsample.so for a dynamic library on Unix-like systems or to sample.DLL on Windows.

It is possible to specify the name of a function symbol expected to be provided by the library using the function=... attribute. This function symbol is being looked for when configure is run for the extension. This way it is possible to verify that the right version of a library was found. With VisualStudio on windows it is not possible to perform this check, in this case the library is just added to the project file.

**Example 2-9. Library dependencies**

```
...
    <lib name="sample_u" platform="unix"  function="sample_v2" />
    <lib name="sample_w" platform="win32" />
    <lib name="sample"   platform="all" />
...
```

## 2.3.4. Dependencies to other extensions

Starting with PHP 5.1 it is possible to define dependendies between extensions. Dependencies determine the order in which extensions are initialized and shut down. They also make sure that extensions like pdo_mysql that require a base extension like pdo are only loaded if that extension is present and that two extensions that would conflict whith each other are not loaded at the same time.

A dependency is defined using an <extension> tag within <deps>. Here the <extension> takes at least a name=....

By default this means that the extension by that name is required by your extension and has to be initialized first. This behavior can also explicitly be requested by setting the type=... attribute to REQUIRED. Other possible values are OPTIONAL if the other extension should be initialized first if present but is not required by your extension and CONFLICTS if your extension and the other one conflict in some way and should not be loaded at the same time.

The extension API and CodeGen_PECL also allow to specify version checks on dependencies, the actual checks are not yet implemented as of PHP 5.1 though. CodeGen_PECL allready supports these anyhow, and as soon as the functionality is implemented within PHP the generated extensions will make use of it.

Version dependencies are defined using the `version=...` attribute and the optional `rel=...` attribute. `version` takes a 'PHP-standardized' version number string (see http://php.net/version_compare) and `rel` defines the operator to be used to compare the actual extension version with the one in `version`. The default value for `rel` is `>=`, so the extensions version number has to be at least as high as specified. Possible `>=` values are `>=`, `>`, `=`, `<` and `<=` or their text alternatives `ge`, `gt`, `eq`, `lt` and `le`.

**Example 2-10. Extension dependencies**

```
<deps>
 <extension name="standard"/>
 <extension name="foobar"      type="OPTIONAL"/>
 <extension name="alternative" type="CONFLICTS"/>
 <extension name="xxx" version="3.2.1" rel="ge"/>
</deps>
```

### 2.3.5. External programs

Not supported yet.

### 2.3.6. Other files

Not supported yet.

## 2.4. Custom code

Custom code may be added to your extension source files using the `<code>` tags. The `role=...` and `position=...` tags specify the actual place in there generated source files where your code should be inserted.

Possible roles are '`code`' (default) for the generated C or C++ code file and '`header`' header file. Possible positions are '`top`' and '`bottom`' (default) for insertion near the beginning or end of the generated file.

## 2.5. Conditional compilation

For some code elements it is possible to define conditional compilation by using the `if=...` attribute. The argument to this attribute is ment to be a valid C preprocessor expression which can be used in an "#if" directive.

To compile in and register a function only if a certain library feature is available you may simply use the `HAVE_...` macros that configure writes into config.h:

**Example 2-11. Conditional compilation**

For a function like this that should only be available if configure has detected either the FOO or BAR feature:

```
<function name="foobar" if="HAVE_FOO || HAVE_BAR">
  ...
</function>
```

so all code generated for this function will put into conditional code blocks like this:

```
#if HAVE_FOO || HAVE_BAR
  ...
#endif
```

So far conditional compilation is only available for procedural functions but it will be added for other code elements, too, in the near future.

## 2.6. Functions

Two different kinds of functions may be defined using the `<function>` tag: public and internal functions. Public functions are functions you want to make available at the PHP code level, internal functions are C functions to be used by the PHP extension API.

Public function names should by convention be prefixed with the extension name followed by an underscore, internal functions are one of `MINIT`, `MSHUTDOWN`, `RINIT`, `RSHUTDOWN` or `MINFO`.

### 2.6.1. Public functions

The definition of a public PHP function requires the attributes `role="public"` and `name=...` and at least the `<proto>` tag to be set.

The function name may be any valid C name. To comply to PHP coding conventions a public function provided by an extension should always be prefixed by the extension name though.

The function prototype specified using the `<proto>` tag is parsed to extract the return type, the function name and the argument list. The function name in the prototype has to match the name attribute given in the `<function>`.

Valid types to be used for arguments and the return type are:

- bool
- int
- float
- string
- array
- object
- mixed
- callback
- resource *[typename]*
- stream

Argument names in prototypes are not prepended by a `$` sign by convention.

Function documentation should be given using the `<summary>` tag for a one line description and the `<description>` tag for a more detailed description. Both are copied to the generated DocBook XML documentation for that function. Within `<description>` DocBook tags may be used. Be aware though that while **pecl-gen** accepts this validating XML parsers may complain when reading/validating an extension specification file.

Skeleton code for parameter parsing and result passing is generated if no `<code>` fragment is specified for a function. A `<code>` section is inserted right after the generated parameter parsing code. Setting a return value is up to the code fragment if any is given, adding a template doesn't make sense in this case.

> **Note:** Maybe some stuff regarding actual coding should be added here?

## 2.6.2. Internal functions

The definition of an internal function requires just the `role="internal"` and `name=...` attributes. The name can only be one of the following:

`MINIT`

The module initialization function. This is called once at startup of a PHP server module or standalone (CLI or CGI) binary.

**Example 2-12. `MINIT()`**

```
...
    <function role="internal" name="MINIT">
      <code>
<![CDATA[
  int dummy = 42;

  dummy = dummy;
]]>
      </code>
    </function>
...
```

`MSHUTDOWN`

The module shutdown function. This is called once when the PHP server module or standalone binary is properly terminated. It may not be called on program crashes or other critical errors.

`RINIT`

The request shutdown function. This is called by PHP server modules before actually executing a PHP script request or once right after `MINIT()` for standalone binaries (CGI or CLI).

`RSHUTDOWN`

The request shutdown function. This is called by PHP server modules after execution of PHP code has been finished or terminated. Is called even if critical PHP errors occurred but you can not rely on it being called on critical errors or crashes on the C level.

`MINFO`

The `phpinfo()` handler for this extension. It will be called whenever `phpinfo()` is invoked or when a standalone PHP binary is called with the `-i` command line option.

The default code generated when no `<code>` section is given includes the extension name, summary line and release version and date, the optional logo image if specified, and the global and actual values of all `php.ini` directives specified.

**Example 2-13. `MINFO()`**

```
...
    <function role='internal' name='MINFO'>
      <code>
<![CDATA[
  php_info_print_table_start();
  php_info_print_table_header(2, "test", "table");
  php_info_print_table_end();
]]>
      </code>
    </function>
...
```

`<code>` sections for the internal functions may be written as if they were C function bodies, including local variable definitions.

# 2.7. Constants

PHP constants are defined using `<constant>` tags within the `<constants>` environment.

The actual constant name, type and value are specified using the `name=...`, `type=...` and `value=...` attributes. The constant name has to be a valid C name. PHP constant names should use uppercase letters only by convention. Possible types are string, int and float, the possible values depend on the type. For int and float you may use either numeric strings or the names of C constants (either true ANSI C/C++ constants or values `#define`d using the C preprocessor. string values are always used "as is", no constants may be used here.

Setting the `define=...` attribute to `yes` defines not only a PHP constant but also adds a C #define with the same name and value to the generated header file so that the constant can be used under the same name in both PHP and C code.

It is sufficient on the other hand to specify a constant `name` only if a C integer constant should be available under the same name in PHP, too.

A descriptive text may be given as content of the `<constant>` tag. This text will be used when generation the DocBook XML documentation.

**Example 2-14. PHP Constants**

```
...
 <constants>
  <constant name="SAMPLE_INT"   type="int"    value="42">
   A sample integer constant.
  </constant>
  <constant name="SAMPLE_FLOAT" type="float"  value="3.14">
   A sample floating point constant.
  </constant>
  <constant name="SAMPLE_FLOAT" type="float"  value="M_PI">
   A sample floating point constant using a #defined constant
  </constant>
  <constant name="SAMPLE_STRING" type="string" value="Hello World!">
   A sample string constant.
  </constant>
  <constant name="SAMPLE_INT2" type="int" value="23>
   This also adds a <literal>#define SAMPLE_INT2 23</literal>
   definition to the generated header file.
  </constant>
  <constant name="MY_CONST">
   A shortcat for already #defined integer constants
  </constant>
 </constants>
...
```

## 2.8. `php.ini` parameters and internal variables

An extension may define variables that are global to either the complete extension or to a specific request. True globals that are global to the complete extensions do not need any registration so they can be defined using C code within the global `<code>` tag.

Module globals that are only global to a single request need to be managed to ensure thread safety and initialization on request initialization. `php.ini` directive values are also stored as module globals but need some additional definitions.

All global definitions have to be put into a `<globals>` environment. Simple module globals are defined using the `<global>` tag. `php.ini` directives are defined using the `<phpini>` tag.

A `<global>` definition requires the `name=...` and `type=...` attributes to be set as valid C names and types. Which C types are allowed depends on what type definitions have been included from header files. The available types are not known when `pecl-gen` parses the XML specification so that types are only

checked for valid name format here. Specifying a type that is not a basic C type or defined in any included file will lead to error messages when compiling the generated extension code later.

Initial values may be specified using the `value=...` attribute. This feature should only be used for simple numeric values, anything more complex should better be initialized within the extensions `RINIT()` function.

`php.ini` directives may be defined using the `<phpini>` within a `<globals>` environment. To define a `php.ini` directive you have to specify its name, type and default value using the `name=...`, `type=...` and `value=...` attributes.

Valid directive names are C variable names. The actual directive name is the extension name followed by a single dot '`.`' and the specified name. Valid directive types are bool, int, float and string.

Directive default values are passed to the engine as strings, so you may not use any C constants or preprocessor macros here. The default value strings are parsed by the `OnUpdate` handler registered for that directive. No value checking takes place during extension code generation or compilation, this is done by the registered `OnUpdate` handler at runtime during request initialization. The `OnUpdate` handler defaults to the appropriate internal `OnUpdate`*type* handler unless you specify a different handler using the `onupdate=...` attribute.

The directive value may be changed at any time unless you specify an `access=...` attribute. Possible values are:

system

: may only be set globally in `php.ini` or the web server configuration

perdir

: may be changed in local `.htaccess` files

user

: may be changed by PHP code

all

: may be changed by anyone at any time

The content data of `<phpini>` tags is used to generate documentation for the defined directive. `<global>` definitions may also include content data but it is for internal documentation only, it is not used in DocBook XML generation (yet).

**Example 2-15. Defining globals and ini entries**

```
...
 <globals>
  <global name="sample_int"    type="int"    value="42" />
  <global name="sample_float"  type="float"  value="3.14" />
  <global name="SAMPLE_STRING" type="char *" />

  <phpini name="my_int" type="int" value="42" onupdate="OnUpdateLong" access="all">
   Definition for directive "sample.my_int"
  </phpini>
 </globals>
...
```

Access to the modul globals and ini parameters is provided in a thread safe manner through the EXTNAME_G() macro (replace EXTNAME with the upper cased name of your extension).

**Example 2-16. Using globals**

```
<extension name="foobar">
 ...
 <globals>
  <global name="sample_int"    type="int"    value="42" />
 </globals>
 ...
 <function ...>
  ...
  <code>
   ...
   int foo = FOOBAR_G(sample_int); // get global value
   ...
   FOOBAR_G(sample_init) = 42; // set global value
   ...
  </code>
 </function>
```

## 2.9. Resources

You may define PHP resource types within a `<resources>` environment. For each `<resource>` you have to specify the `name=...` and `payload=...` attributes. The `name` has to be a valid C name and the `payload` has to be a valid C type specifier. The payload type can only be checked for the correctness of its form as the actual type definitions from included header files are not known to the extension generator when it generates the extension code.

The actual resource data structure carries a pointer to the payload type. You may specify that PHP shall allocate and free the actual payload by setting the `alloc=...` attribute to "`yes`". If the payload is allocated by a library function or by yourself you should set `alloc=...` to "`no`" (the default value).

Resources are destructed when the last variable reference refering to them is unset or at request shutdown. If your resource payload needs to be cleaned up as well you have to add an appropriate C code snippet that takes care of this using the `<destruct>` tag. Within the destructor snippet you may refer to the allocated payload using the `resource` pointer variable.

You don't need to take care of destruction yourself if your resource payload is allocated by PHP (`alloc="yes"`) and needs no further cleanup work besides releasing the allocated memory.

**Example 2-17. Resources**

```
...
  <resources>
    <resource name="sample_resource" payload="float" alloc="yes">
      <description>
        A simple floating point resource
      </description>
      <!-- no <destruct> needed due to the alloc attribute -->
    </resource>

    <resource name="sample_struct" payload="struct foobar" alloc="no">
      <description>
        A foobar resource managed by an external foobar lib.
      </description>
      <destruct>
        foobar_release(resource);
      </destruct>
    </resource>
  </resources>
...
```

## 2.9.1. Resource creation and destruction

The creation of resource instances is not defined within `<resource>`. This is a task to be handled by public PHP functions instead.

**Example 2-18. Resource creation**

```
  <function name="foo_open">
   <proto>resource foo foo_open(string path)</proto>
   <code>
    return_res = foo_open(path);
```

```
  if (!return_res) RETURN_FALSE;
 </code>
</function>
```

Resources are freed using the `FREE_RESOURCE()` macro. The resources destructor function is automaticly called when freeing a resource.

**Example 2-19. Resource destruction**

```
<function name="foo_close">
 <proto>void foo_close(resource foo foores)</proto>
 <code>
   FREE_RESOURCE(foores);
 </code>
</function>
```

## 2.10. OO features

> **Warning**
>
> The OO APIs differ between PHP 4 and 5, CodeGen_PECL only supports the newer PHP 5 API.

### 2.10.1. Classes

Classes are declared using the `class` container. Each class needs to be given a unique class name using the `name=...` attribute.

Using the `extends=...` attribut it is possible to inherit from another internal class. The class inherited from does not necessarily have to be defined in the same package so the given name is not checked for existance. An attempt to extend from a non-existing class will only be caught at PHP runtine.

The optional `abstract=...` and `final=...` attributes can be used to declare a class as abstract (can be inherited from but can't be instanciated) or final (can't be inherited from anymore).

A class can implement one or more interfaces, the interfaces implemented by a class are declared using the `implements` tag and its `interface=...` attribute. The interface needs to be defined in the extension specification (see "Interfaces" later in this section) or by one of the extensions specified in the package dependencies.

## 2.10.2. Methods

Class methods (or member functions) are defined similar to regular functions using the `function` tag. In addition to its global counterpart the `function`.

The `access=...` attribute defines the methods PPP access rights, possible values are `public`, `protected` and `private` (although defining a `private` native method doesn't really make sense).

The `abstract` and `final` attributes declare a method as abstract or final and have the same effect as the PHP keywords of the same name.

The `procedural=...` attribute requests the registration of a procedural alternative for this method. This is a regular function that takes an instance of the class as first argument and allows to call

```
function($class, ...parameters...)
```

instead of

```
$class->method(...parameters...)
```

.

This feature is usefull when creating an extension that implements something as OO classes that used to be a resource type before, see the way `ext/mysqli` reimplements the older `ext/mysql` for example.

The procedural functions name is `classname_methodname` if you just use `procedural='yes'`, any other attribute value will be used as the functions name.

## 2.10.3. Properties

Class properties are defined using the `property` tag. The property name is given using the `name=...` tag and is mandatory.

Access to the property is public by default, it can be set to either `public`, `protected` or `private` using the optional `access=...` attribute.

Property default type and value can be specified using the optional `type=...` and `value=...` attributes. Possible types are `long`, `double`, `string` and `NULL`. The default value and type of a property are `NULL` if not specified using these attributes.

A property can be declared static using the `static=...` attribute. A static attribute is shared by all instances of the class.

### 2.10.4. Constants

Class constants are defined like regular constants, just within a `class` tag. Class constants are always public so there is no need for extra attributes.

### 2.10.5. Interfaces

Interfaces are declared using the `interface` which has two attributes: the mandatory `name=...` attribute defines the name of the interface and the optional `extends=...` attribute can be used to specify that this interface extends another already existing interface. If the interface that is to be extended is defined in another extension then an extension dependency should be declared, too.

Interface methods are declared using the `function` tag. In this context only the `name=...` attribute is supported as interface methods are always public and abstract and never final anyway. Within the `function` a `proto` needs to be defined but there is no support for `code` or `test` tags within interface method declarations.

## 2.11. Streams

Stream filter and wrapper support is experimental and not yet added to the released code base.

## 2.12. config.m4 fragments

Additional configure checks can be added to the generated config.m4 file used by Unix/Cygwin builds using the `<configm4>` tag. Using the 'position' attribute it is possible to specify whether the additional code is to be added near the top or bottom of the config.m4 file.

**Example 2-20. config.m4 additions**

```
<configm4>
  AC_CHECK_PROG(RE2C, re2c, re2c)
```

```
  PHP_SUBST(RE2C)
</configm4>
```

# 2.13. Makefile fragments

Makefile rules may be added using the <makefile> for Unix/Cygwin builds. Using this it is possible to add dependencies or build rules in addition to the default and auto generated rules.

**Example 2-21. Makefile fragments**

```
<makefile>
$(builddir)/scanner.c: $(srcdir)/scanner.re
  $(RE2C) $(srcdir)/scanner.re > $@
</makefile>
```

# 2.14. Tests

Global test cases can be created using the <test> tag. Test cases for functions are automaticly created.

Currently you have to make sure your extension is loaded by php.ini and have to perform the following steps to run the test suite (changing pathes to point to the right files on your system):

```
  TEST_PHP_EXECUTABLE="/usr/local/bin/php"   php path/to/run-tests.php tests
```

Starting with PHP 5.1 it should be possible to test PECL extensions by just typing make test in the extension source dir, the changes needed for this are being reviewed right now and should hopefully be ready in time to be included in the PHP 5.1.0 release.

## 2.14.1. Global test cases

Global test case scripts can be created using the <test> tag. The <test> has a single attribute name. As the test name is used as the test file basename name has to be unique and only characters, digits and '-'

and `'_'` are allowed in test names. A more readable test title may be set using the `<title>` tag within `<test>`.

The actual PHP code to run is specified using a `<code>` section. The expected output is specified using a `<result>` tag, it defaults to `OK`. The PHP test suite supports three different ways to compare test output with the expected result: plain string comparison, comparison using printf style placeholders like %d for numbers and regular expresions (for details see the `README.TESTING*` files in the PHP source). By default the `plain` mode is used, the other two modes can be selected by setting the `mode` attribute of `<result>` to `format` or `regex`.

The `--SKIPIF--` section of the generated tests checks for the generated extension being loaded, the tests will automaticly be skiped if it is not available. Additional skip conditions can be added using the `<skipif>` tag. The content of the tag may either be a PHP expression that evaluates to `true` if the test should be skipped or a complete code snippet that prints `skip` if the test is supposed to be skipped. A string describing the reason for the test being skipped may be added after the `skip` in this case.

Additional php.ini settings to be used for testing may be specified in a `<ini>` section.

**Example 2-22. Minimal test case**

```
<test name="echo">
 <code>echo "OK";</code>
</test>
```

**Example 2-23. Full test case**

```
<test name="full">
 <title>A full test case using all tags</title>
 <skipif>1==0</skipif>
 <ini>max_execution_time=0</ini>
 <code>echo "Random number: ".rand(1,10);</code>
 <result mode='format'>Random number: %d</result>
</test>
```

## 2.14.2. Embedded function test cases

For each function a default test case is created, the name and title for this test are automaticly set to the function name.

Test code and the expected result can be set using a `<test>` section within `<function>`. `<code>`, `<result>`, `<skipif>` and `<ini>` may be used in there in the same way as in a global `<>` section. Use of the `name` attribute to `<test>` or the `<title>` tag are not supported within a function test.

**Example 2-24. Minimal function test case**

```
<function name="foobar">
 ...
 <test><code>echo "OK";</code></test>
</test>
```

**Example 2-25. Full function test case**

```
<function name="foobar">
 ...
 <test>
  <skipif>1==0</skipif>
  <ini>max_execution_time=0</ini>
  <code>echo "Random number: ".rand(1,10);</code>
  <result mode='format'>Random number: %d</result>
 <test>
</function>
```

# Chapter 3. XML input parsing

## 3.1. Includes

The XML parser used by `CodeGen_PECL` supports inclusion of additional source files using three different ways:

- external entities
- a subset of XInclude
- the `source` attribute of `<code>` tags

### 3.1.1. External entities

In SGML and early XML system entites were the only include mechanism availabe. System entities have to be defined in the documents DOCTYPE header, later on in the document the entity can be used to include the specified file:

**Example 3-1. System Entities**

```
<?xml version="1.0" ?>
<!DOCTYPE extension SYSTEM "../extension.dtd" [
<!ENTITY systemEntity SYSTEM "parsing_1.xml">
]>
<extension name="foobar">
...
&systemEntity;
..
</extension>
```

### 3.1.2. XInclude

The `CodeGen` XML parser supports a simple subset of XInclude, it is possible to include additional specification files using the `href=...` attribute of the `<include>` tag:

**Example 3-2. XInclude**

```
<extension name="foobar" xmlns:xi="http://www.w3.org/2001/XInclude">
 ...
 <xi:include href="foobar_2.xml"/>
 ...
</extension>
```

The `parse=...` attirbute is also supported, using `<include parse='text' href='...'/>` it is possible to include arbitrary data without parsing it as XML.

**Example 3-3. Verbatim XInclude**

```
<extension name="foobar" xmlns:xi="http://www.w3.org/2001/XInclude">
 ...
 <description><xi:include href="README" parse="text"/></description>
 ...
</extension>
```

Other `<include>` features and the `<fallback>` tag are not supported yet, and most of them won't make sense in this context anyway.

### 3.1.3. `<code>` tags

In most places the `<code>` tag supports loading of its content using its `src=...` attribute:

**Example 3-4. Using `<code src="...">`**

```
<function name="foobar">
 ...
 <code src="func_foobar.c"/>
</function>
```

## 3.2. Verbatim text data

C code usually contains quite a few >, < and & characters all of which need to be escaped in XML. This can be done by either converting them into entities all over the place or by embedding the code into CDATA sections:

**Example 3-5. CDATA sections**

```
<code>
<![CDATA[
  foo->bar = 42;
]]>
</code>
```

Typing <![CDATA[ can become rather annoying over time (esp. on a german keyboard), so i introduced the <?data processing instruction into the CodeGen XML parser as an alternative to CDATA:

**Example 3-6. <?data processing instruction**

```
<?data
  foo->bar = 42;
?>
```

# Chapter 4. Usage

## 4.1. Invocation

The transformation of a XMP specification file into an extension directory is done by simply calling the `pecl-gen` command with the XML filename as argument:

```
pecl-gen my_extension.xml
```

`pecl-gen` will refuse to overwrite an existing extension directory (as changes made in there may be lost) unless you call it with the `-f` or `--force` option:

```
pecl-gen -f my_extension.xml
```

## 4.2. Configuration

You need to configure an extension for your actual build system before compiling it. Configuring a PECL extension consists of two steps:

First you need to copy some files from your PHP installation into the extension directory and run the autotools to create a `configure`. All this is taken care of by the `phpize` command that is part of your PHP installation:

```
cd my_extension
phpize
```

Next you need to run `configure` to configure your extension for your system installation. Most of the time just running `configure` will be sufficient as appropriate defaults should be picked by the script. If your extension relies on external libraries installed in non-standard places you may want to run `configure` with the appropriate `--with-...` options.

```
configure
```

## 4.3. Compilation

After configuring your extension the actual compilation is done by the `make` command. No further parameters are needed at this point:

```
make
```

## 4.4. Testing

Starting with PHP 5.1 it should be possible to run the generated test cases by simply typing `make test`. For older PHP versions a few more steps are needed:

- you have to specify the PHP binary to be used for testing
- you have to set up a `php.ini` that loags the extension for testing
- you have to manually run the `run-tests.php` that comes with the PHP source

After adding the extension to your `php.ini` a typical test invocation may look like this:

```
TEST_PHP_EXECUTABLE="/usr/local/bin/php"   php ../php-src/run-tests.php tests
```

(your `php` binary and `run-tests.php` script may obviously be in different locations)

## 4.5. Installation

You can copy your newly created extension to your installations default extension directory by simply running `make install`. If you've set a different `extension_dir` in your `php.ini` you have to manually copy the extensions `.so` file to this directory.

Please note that in both cases your regular user permissions may not be sufficient to install the extension file, you may need to run the commands as a different user, e.g. by using the `sudo` command.

```
sudo make install
```

## 4.6. PEAR integration

`pecl-gen` generates a `package.xml` alongside with the other generated files. An extension may be configured, compiled and installed in a single operation using the PEAR installer:

**Example 4-1. Installation using `pear`**

```
cd my_extension
pear install package.xml
```