XML Parser

Introduction

XML (eXtensible Markup Language) is a data format for structured document interchange on the Web. It is a standard defined by The World Wide Web consortium (W3C). Information about XML and related technologies can be found at » http://www.w3.org/XML/ .

This PHP extension implements support for James Clark's expat in PHP. This toolkit lets you parse, but not validate, XML documents. It supports three source character encodings also provided by PHP: *US-ASCII*, *ISO-8859-1* and *UTF-8*. *UTF-16* is not supported.

This extension lets you create XML parsers and then define *handlers* for different XML events. Each XML parser also has a few parameters you can adjust.

Installing/Configuring

Requirements

This extension uses an expat compat layer by default. It can use also expat, which can be found at » http://www.jclark.com/xml/expat.html. The Makefile that comes with expat does not build a library by default, you can use this make rule for that:

```
libexpat.a: $(OBJS)
   ar -rc $@ $(OBJS)
   ranlib $@
```

A source RPM package of expat can be found at » http://sourceforge.net/projects/expat/.

Installation

These functions are enabled by default, using the bundled expat library. You can disable XML support with *--disable-xml*. If you compile PHP as a module for Apache 1.3.9 or later, PHP will automatically use the bundled expat library from Apache. In order you don't want to use the bundled expat library configure PHP *--with-expat-dir=DIR*, where DIR should point to the base installation directory of expat.

The Windows version of PHP has built-in support for this extension. You do not need to load any additional extensions in order to use these functions.

Runtime Configuration

This extension has no configuration directives defined in *php.ini*.

Resource Types

The *xml* resource as returned by xml_parser_create() and xml_parser_create_ns() references an xml parser instance to be used with the functions provided by this extension.

Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

XML_ERROR_NONE (integer)

XML_ERROR_NO_MEMORY (integer)

XML_ERROR_SYNTAX (integer)

XML_ERROR_NO_ELEMENTS (integer)

XML_ERROR_INVALID_TOKEN (integer)

XML_ERROR_UNCLOSED_TOKEN (integer)

XML_ERROR_PARTIAL_CHAR (integer)

XML_ERROR_TAG_MISMATCH (integer)

XML_ERROR_DUPLICATE_ATTRIBUTE (integer)

XML_ERROR_JUNK_AFTER_DOC_ELEMENT (integer)

XML_ERROR_PARAM_ENTITY_REF (integer)

XML_ERROR_UNDEFINED_ENTITY (integer)

XML_ERROR_RECURSIVE_ENTITY_REF (integer)

XML_ERROR_ASYNC_ENTITY (integer)

XML_ERROR_BAD_CHAR_REF (integer)

XML_ERROR_BINARY_ENTITY_REF (integer)

XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (integer)

XML_ERROR_MISPLACED_XML_PI (integer)

XML_ERROR_UNKNOWN_ENCODING (integer)

XML_ERROR_INCORRECT_ENCODING (integer)

XML_ERROR_UNCLOSED_CDATA_SECTION (integer)

XML_ERROR_EXTERNAL_ENTITY_HANDLING (integer)

XML_OPTION_CASE_FOLDING (integer)

XML_OPTION_TARGET_ENCODING (integer)

XML_OPTION_SKIP_TAGSTART (integer)

XML_OPTION_SKIP_WHITE (integer)

Event Handlers

The XML event handlers defined are:

Supported XML handlers

PHP function to set handler	Event description
xml_set_element_handler()	Element events are issued whenever the XML parser encounters start or end tags. There are separate handlers for start tags and end tags.
xml_set_character_data_handler()	Character data is roughly all the non-markup contents of XML documents, including whitespace between tags. Note that the XML parser does not add or remove any whitespace, it is up to the application (you) to decide whether whitespace is significant.
xml_set_processing_instruction_handler()	PHP programmers should be familiar with processing instructions (PIs) already. <?php ?> is a processing instruction, where *php* is called the "PI target". The handling of these are application-specific, except that all PI targets starting with "XML" are reserved.
xml_set_default_handler()	What goes not to another handler goes to the default handler. You will get things like the XML and document type declarations in the default handler.
xml_set_unparsed_entity_decl_handler()	This handler will be called for declaration of an unparsed (NDATA) entity.
xml_set_notation_decl_handler()	This handler is called for declaration of a notation.
xml_set_external_entity_ref_handler()	This handler is called when the XML parser finds a reference to an external parsed general entity. This can be a reference to a file or URL, for example. See the external entity example for a demonstration.

Case Folding

The element handler functions may get their element names case-folded. Case-folding is defined by the XML standard as "a process applied to a sequence of characters, in which those identified as non-uppercase are replaced by their uppercase equivalents". In other words, when it comes to XML, case-folding simply means uppercasing.

By default, all the element names that are passed to the handler functions are case-folded. This behaviour can be queried and controlled per XML parser with the xml_parser_get_option() and xml_parser_set_option() functions, respectively.

Error Codes

The following constants are defined for XML error codes (as returned by xml_parse()):

- XML_ERROR_NONE
- XML_ERROR_NO_MEMORY
- XML_ERROR_SYNTAX
- XML_ERROR_NO_ELEMENTS
- XML_ERROR_INVALID_TOKEN
- XML_ERROR_UNCLOSED_TOKEN
- XML_ERROR_PARTIAL_CHAR
- XML_ERROR_TAG_MISMATCH
- XML_ERROR_DUPLICATE_ATTRIBUTE
- XML_ERROR_JUNK_AFTER_DOC_ELEMENT
- XML_ERROR_PARAM_ENTITY_REF
- XML_ERROR_UNDEFINED_ENTITY
- XML_ERROR_RECURSIVE_ENTITY_REF
- XML_ERROR_ASYNC_ENTITY
- XML_ERROR_BAD_CHAR_REF
- XML_ERROR_BINARY_ENTITY_REF
- XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF
- XML_ERROR_MISPLACED_XML_PI
- XML_ERROR_UNKNOWN_ENCODING
- XML_ERROR_INCORRECT_ENCODING
- XML_ERROR_UNCLOSED_CDATA_SECTION
- XML_ERROR_EXTERNAL_ENTITY_HANDLING

Character Encoding

PHP's XML extension supports the » Unicode character set through different character encoding s. There are two types of character encodings, source encoding and target encoding. PHP's internal representation of the document is always encoded with *UTF-8*.

Source encoding is done when an XML document is parsed. Upon creating an XML parser , a source encoding can be specified (this encoding can not be changed later in the XML parser's lifetime). The supported source encodings are *ISO-8859-1*, *US-ASCII* and *UTF-8*. The former two are single-byte encodings, which means that each character is represented by a single byte. *UTF-8* can encode characters composed by a variable number of bits (up to 21) in one to four bytes. The default source encoding used by PHP is *ISO-8859-1*.

Target encoding is done when PHP passes data to XML handler functions. When an XML parser is created, the target encoding is set to the same as the source encoding, but this may be changed at any point. The target encoding will affect character data as well as tag names and processing instruction targets.

If the XML parser encounters characters outside the range that its source encoding is capable of representing, it will return an error.

If PHP encounters characters in the parsed XML document that can not be represented in the chosen target encoding, the problem characters will be "demoted". Currently, this means that such characters are replaced by a question mark.

Examples

XML Element Structure Example

This first example displays the structure of the start elements in a document with indentation.

Example #1 - Show XML Element Structure

```
<?php
$file = "data.xml";
$depth = array();

function startElement($parser, $name, $attrs)
{
   global $depth;
   for ($i = 0; $i < $depth[$parser]; $i++) {
       echo "  ";
   }
   echo "$name\n";
   $depth[$parser]++;
}

function endElement($parser, $name)
{
   global $depth;
   $depth[$parser]--;
}

$xml_parser = xml_parser_create();
xml_set_element_handler($xml_parser, "startElement", "endElement");
if (!($fp = fopen($file, "r"))) {
   die("could not open XML input");
}

while ($data = fread($fp, 4096)) {
   if (!xml_parse($xml_parser, $data, feof($fp))) {
       die(sprintf("XML error: %s at line %d",
                   xml_error_string(xml_get_error_code($xml_parser)),
                   xml_get_current_line_number($xml_parser)));
   }
}
xml_parser_free($xml_parser);
?>
```

XML Tag Mapping Example

Example #2 - Map XML to HTML

This example maps tags in an XML document directly to HTML tags. Elements not found in the "map array" are ignored. Of course, this example will only work with a specific XML document type.

```
<?php
$file = "data.xml";
$map_array = array(
   "BOLD"     => "B",
   "EMPHASIS" => "I",
   "LITERAL"  => "TT"
);

function startElement($parser, $name, $attrs)
{
   global $map_array;
   if (isset($map_array[$name])) {
       echo "<$map_array[$name]>";
   }
}

function endElement($parser, $name)
{
   global $map_array;
   if (isset($map_array[$name])) {
       echo "</$map_array[$name]>";
   }
}

function characterData($parser, $data)
{
   echo $data;
}

$xml_parser = xml_parser_create();
// use case-folding so we are sure to find the tag in $map_array
xml_parser_set_option($xml_parser, XML_OPTION_CASE_FOLDING, true);
xml_set_element_handler($xml_parser, "startElement", "endElement");
xml_set_character_data_handler($xml_parser, "characterData");
if (!($fp = fopen($file, "r"))) {
   die("could not open XML input");
}

while ($data = fread($fp, 4096)) {
   if (!xml_parse($xml_parser, $data, feof($fp))) {
       die(sprintf("XML error: %s at line %d",
                   xml_error_string(xml_get_error_code($xml_parser)),
                   xml_get_current_line_number($xml_parser)));
   }
}
xml_parser_free($xml_parser);
?>
```

XML External Entity Example

This example highlights XML code. It illustrates how to use an external entity reference handler to include and parse other documents, as well as how PIs can be processed, and a way of determining "trust" for PIs containing code.

XML documents that can be used for this example are found below the example (*xmltest.xml* and *xmltest2.xml*.)

Example #3 - External Entity Example

```
<?php
$file = "xmltest.xml";

function trustedFile($file)
{
   // only trust local files owned by ourselves
   if (!eregi("^([a-z]+)://", $file)
       && fileowner($file) == getmyuid()) {
           return true;
   }
   return false;
}

function startElement($parser, $name, $attribs)
{
   echo "&lt;<font color=\"#0000cc\">$name</font>";
   if (count($attribs)) {
       foreach ($attribs as $k => $v) {
           echo " <font color=\"#009900\">$k</font>=\"<font
                  color=\"#990000\">$v</font>\"";
       }
   }
   echo "&gt;";
}

function endElement($parser, $name)
{
   echo "&lt;/<font color=\"#0000cc\">$name</font>&gt;";
}

function characterData($parser, $data)
{
   echo "<b>$data</b>";
}

function PIHandler($parser, $target, $data)
{
   switch (strtolower($target)) {
       case "php":
           global $parser_file;
           // If the parsed document is "trusted", we say it is safe
           // to execute PHP code inside it.  If not, display the code
           // instead.
           if (trustedFile($parser_file[$parser])) {
               eval($data);
```

```
            } else {
                printf("Untrusted PHP code: <i>%s</i>",
                        htmlspecialchars($data));
            }
            break;
    }
}

function defaultHandler($parser, $data)
{
    if (substr($data, 0, 1) == "&" && substr($data, -1, 1) == ";") {
        printf('<font color="#aa00aa">%s</font>',
                htmlspecialchars($data));
    } else {
        printf('<font size="-1">%s</font>',
                htmlspecialchars($data));
    }
}

function externalEntityRefHandler($parser, $openEntityNames, $base,
$systemId,
                                  $publicId) {
    if ($systemId) {
        if (!list($parser, $fp) = new_xml_parser($systemId)) {
            printf("Could not open entity %s at %s\n", $openEntityNames,
                   $systemId);
            return false;
        }
        while ($data = fread($fp, 4096)) {
            if (!xml_parse($parser, $data, feof($fp))) {
                printf("XML error: %s at line %d while parsing entity %s\n",
                       xml_error_string(xml_get_error_code($parser)),
                       xml_get_current_line_number($parser),
$openEntityNames);
                xml_parser_free($parser);
                return false;
            }
        }
        xml_parser_free($parser);
        return true;
    }
    return false;
}

function new_xml_parser($file)
{
    global $parser_file;

    $xml_parser = xml_parser_create();
    xml_parser_set_option($xml_parser, XML_OPTION_CASE_FOLDING, 1);
    xml_set_element_handler($xml_parser, "startElement", "endElement");
    xml_set_character_data_handler($xml_parser, "characterData");
    xml_set_processing_instruction_handler($xml_parser, "PIHandler");
    xml_set_default_handler($xml_parser, "defaultHandler");
    xml_set_external_entity_ref_handler($xml_parser,
"externalEntityRefHandler");

    if (!($fp = @fopen($file, "r"))) {
        return false;
    }
```

```
   if (!is_array($parser_file)) {
       settype($parser_file, "array");
   }
   $parser_file[$xml_parser] = $file;
   return array($xml_parser, $fp);
}

if (!(list($xml_parser, $fp) = new_xml_parser($file))) {
   die("could not open XML input");
}

echo "<pre>";
while ($data = fread($fp, 4096)) {
   if (!xml_parse($xml_parser, $data, feof($fp))) {
       die(sprintf("XML error: %s at line %d\n",
                   xml_error_string(xml_get_error_code($xml_parser)),
                   xml_get_current_line_number($xml_parser)));
   }
}
echo "</pre>";
echo "parse complete\n";
xml_parser_free($xml_parser);

?>
```

Example #4 - xmltest.xml

```
<?xml version='1.0'?>
<!DOCTYPE chapter SYSTEM "/just/a/test.dtd" [
<!ENTITY plainEntity "FOO entity">
<!ENTITY systemEntity SYSTEM "xmltest2.xml">
]>
<chapter>
<TITLE>Title &plainEntity;</TITLE>
<para>
 <informaltable>
  <tgroup cols="3">
   <tbody>
    <row><entry>a1</entry><entry
morerows="1">b1</entry><entry>c1</entry></row>
    <row><entry>a2</entry><entry>c2</entry></row>
    <row><entry>a3</entry><entry>b3</entry><entry>c3</entry></row>
   </tbody>
  </tgroup>
 </informaltable>
</para>
&systemEntity;
<section id="about">
 <title>About this Document</title>
 <para>
  <!-- this is a comment -->
  <?php echo 'Hi!  This is PHP version ' . phpversion(); ?>
 </para>
</section>
</chapter>
```

This file is included from *xmltest.xml*:

Example #5 - xmltest2.xml

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY testEnt "test entity">
]>
<foo>
  <element attrib="value"/>
  &testEnt;
  <?php echo "This is some more PHP code being executed."; ?>
</foo>
```

XML Parser Functions

utf8_decode

utf8_decode -- Converts a string with ISO-8859-1 characters encoded with UTF-8 to single-byte ISO-8859-1

Description

string **utf8_decode** (string `$data`)

This function decodes `data`, assumed to be *UTF-8* encoded, to *ISO-8859-1*.

Parameters

`data`
: An UTF-8 encoded string.

Return Values

Returns the ISO-8859-1 translation of `data`.

See Also

- utf8_encode() for an explanation of UTF-8 encoding

utf8_encode

utf8_encode -- Encodes an ISO-8859-1 string to UTF-8

Description

string **utf8_encode** (string `$data`)

This function encodes the string *`data`* to *UTF-8*, and returns the encoded version. *UTF-8* is a standard mechanism used by Unicode for encoding wide character values into a byte stream. *UTF-8* is transparent to plain ASCII characters, is self-synchronized (meaning it is possible for a program to figure out where in the bytestream characters start) and can be used with normal string comparison functions for sorting and such. PHP encodes *UTF-8* characters in up to four bytes, like this:

UTF-8 encoding

bytes	bits	representation
1	7	0bbbbbbb
2	11	110bbbbb 10bbbbbb
3	16	1110bbbb 10bbbbbb 10bbbbbb
4	21	11110bbb 10bbbbbb 10bbbbbb 10bbbbbb

Each *b* represents a bit that can be used to store character data.

Parameters

`data`
An ISO-8859-1 string.

Return Values

Returns the UTF-8 translation of *`data`*.

xml_error_string

xml_error_string -- Get XML parser error string

Description

string **xml_error_string** (int `$code`)

Gets the XML parser error string associated with the given *code*.

Parameters

code
An error code from xml_get_error_code().

Return Values

Returns a string with a textual description of the error *code*, or **FALSE** if no description was found.

See Also

- xml_get_error_code()

xml_get_current_byte_index

xml_get_current_byte_index -- Get current byte index for an XML parser

Description

int **xml_get_current_byte_index** (resource `$parser`)

Gets the current byte index of the given XML parser.

Parameters

`parser`
A reference to the XML parser to get byte index from.

Return Values

This function returns **FALSE** if *`parser`* does not refer to a valid parser, or else it returns which byte index the parser is currently at in its data buffer (starting at 0).

Notes

Warning
This function returns byte index according to UTF-8 encoded text disregarding if input is in another encoding.

See Also

- **xml_get_current_column_index()**
- **xml_get_current_line_index()**

xml_get_current_column_number

xml_get_current_column_number -- Get current column number for an XML parser

Description

int **xml_get_current_column_number** (resource `$parser`)

Gets the current column number of the given XML parser.

Parameters

`parser`
A reference to the XML parser to get column number from.

Return Values

This function returns **FALSE** if *`parser`* does not refer to a valid parser, or else it returns which column on the current line (as given by xml_get_current_line_number()) the parser is currently at.

See Also

- xml_get_current_byte_index()
- **xml_get_current_line_index()**

xml_get_current_line_number

xml_get_current_line_number -- Get current line number for an XML parser

Description

int **xml_get_current_line_number** (resource `$parser`)

Gets the current line number for the given XML parser.

Parameters

`parser`
A reference to the XML parser to get line number from.

Return Values

This function returns **FALSE** if *`parser`* does not refer to a valid parser, or else it returns which line the parser is currently at in its data buffer.

See Also

- **xml_get_current_column_index()**
- xml_get_current_byte_index()

xml_get_error_code

xml_get_error_code -- Get XML parser error code

Description

int **xml_get_error_code** (resource `$parser`)

Gets the XML parser error code.

Parameters

`parser`
A reference to the XML parser to get error code from.

Return Values

This function returns **FALSE** if *`parser`* does not refer to a valid parser, or else it returns one of the error codes listed in the error codes section.

See Also

- xml_error_string()

xml_parse_into_struct

xml_parse_into_struct -- Parse XML data into an array structure

Description

int **xml_parse_into_struct** (resource `$parser`, string `$data`, array `&$values` [, array `&$index`])

This function parses an XML file into 2 parallel array structures, one (*`index`*) containing pointers to the location of the appropriate values in the *`values`* array. These last two parameters must be passed by reference.

Parameters

`parser`

`data`

`values`

`index`

Return Values

xml_parse_into_struct() returns 0 for failure and 1 for success. This is not the same as **FALSE** and **TRUE**, be careful with operators such as ===.

Examples

Below is an example that illustrates the internal structure of the arrays being generated by the function. We use a simple *note* tag embedded inside a *para* tag, and then we parse this and print out the structures generated:

Example #6 - xml_parse_into_struct() **example**

```
<?php
$simple = "<para><note>simple note</note></para>";
$p = xml_parser_create();
xml_parse_into_struct($p, $simple, $vals, $index);
xml_parser_free($p);
echo "Index array\n";
print_r($index);
echo "\nVals array\n";
```

```
print_r($vals);
?>
```

When we run that code, the output will be:

```
Index array
Array
(
   [PARA] => Array
       (
           [0] => 0
           [1] => 2
       )

   [NOTE] => Array
       (
           [0] => 1
       )

)

Vals array
Array
(
   [0] => Array
       (
           [tag] => PARA
           [type] => open
           [level] => 1
       )

   [1] => Array
       (
           [tag] => NOTE
           [type] => complete
           [level] => 2
           [value] => simple note
       )

   [2] => Array
       (
           [tag] => PARA
           [type] => close
           [level] => 1
       )

)
```

Event-driven parsing (based on the expat library) can get complicated when you have an XML document that is complex. This function does not produce a DOM style object, but it generates structures amenable of being transversed in a tree fashion. Thus, we can create objects representing the data in the XML file easily. Let's consider the following XML file representing a small database of aminoacids information:

Example #7 - moldb.xml - small database of molecular information

```
<?xml version="1.0"?>
<moldb>

 <molecule>
     <name>Alanine</name>
     <symbol>ala</symbol>
     <code>A</code>
     <type>hydrophobic</type>
 </molecule>

 <molecule>
     <name>Lysine</name>
     <symbol>lys</symbol>
     <code>K</code>
     <type>charged</type>
 </molecule>

</moldb>
```

And some code to parse the document and generate the appropriate objects:

Example #8 - parsemoldb.php - parses moldb.xml into an array of molecular objects

```
<?php

class AminoAcid {
   var $name;  // aa name
   var $symbol;    // three letter symbol
   var $code;  // one letter code
   var $type;  // hydrophobic, charged or neutral

   function AminoAcid ($aa)
   {
       foreach ($aa as $k=>$v)
           $this->$k = $aa[$k];
   }
}

function readDatabase($filename)
{
   // read the XML database of aminoacids
   $data = implode("", file($filename));
   $parser = xml_parser_create();
   xml_parser_set_option($parser, XML_OPTION_CASE_FOLDING, 0);
   xml_parser_set_option($parser, XML_OPTION_SKIP_WHITE, 1);
   xml_parse_into_struct($parser, $data, $values, $tags);
   xml_parser_free($parser);

   // loop through the structures
   foreach ($tags as $key=>$val) {
       if ($key == "molecule") {
           $molranges = $val;
           // each contiguous pair of array entries are the
           // lower and upper range for each molecule definition
           for ($i=0; $i < count($molranges); $i+=2) {
```

```
                $offset = $molranges[$i] + 1;
                $len = $molranges[$i + 1] - $offset;
                $tdb[] = parseMol(array_slice($values, $offset, $len));
            }
        } else {
            continue;
        }
    }
    return $tdb;
}

function parseMol($mvalues)
{
    for ($i=0; $i < count($mvalues); $i++) {
        $mol[$mvalues[$i]["tag"]] = $mvalues[$i]["value"];
    }
    return new AminoAcid($mol);
}

$db = readDatabase("moldb.xml");
echo "** Database of AminoAcid objects:\n";
print_r($db);

?>
```

After executing *parsemoldb.php*, the variable *$db* contains an array of AminoAcid objects, and the output of the script confirms that:

```
** Database of AminoAcid objects:
Array
(
    [0] => aminoacid Object
        (
            [name] => Alanine
            [symbol] => ala
            [code] => A
            [type] => hydrophobic
        )

    [1] => aminoacid Object
        (
            [name] => Lysine
            [symbol] => lys
            [code] => K
            [type] => charged
        )

)
```

xml_parse

xml_parse -- Start parsing an XML document

Description

int **xml_parse** (resource `$parser`, string `$data` [, bool `$is_final`])

xml_parse() parses an XML document. The handlers for the configured events are called as many times as necessary.

Parameters

`parser`
A reference to the XML parser to use.

`data`
Chunk of data to parse. A document may be parsed piece-wise by calling xml_parse() several times with new data, as long as the *`is_final`* parameter is set and **TRUE** when the last data is parsed.

`is_final`
If set and **TRUE**, *`data`* is the last piece of data sent in this parse.

Return Values

Returns 1 on success or 0 on failure.

For unsuccessful parses, error information can be retrieved with xml_get_error_code(), xml_error_string(), xml_get_current_line_number(), xml_get_current_column_number() and xml_get_current_byte_index().

Note
Entity errors are reported at the end of the data thus only if *`is_final`* is set and **TRUE**.

xml_parser_create_ns

xml_parser_create_ns -- Create an XML parser with namespace support

Description

resource **xml_parser_create_ns** ([string `$encoding` [, string `$separator`]])

xml_parser_create_ns() creates a new XML parser with XML namespace support and returns a resource handle referencing it to be used by the other XML functions.

Parameters

`encoding`
: The optional *`encoding`* specifies the character encoding for the input/output in PHP 4. Starting from PHP 5, the input encoding is automatically detected, so that the *`encoding`* parameter specifies only the output encoding. In PHP 4, the default output encoding is the same as the input charset. In PHP 5.0.0 and 5.0.1, the default output charset is ISO-8859-1, while in PHP 5.0.2 and upper is UTF-8. The supported encodings are *ISO-8859-1*, *UTF-8* and *US-ASCII*.

`separator`
: With a namespace aware parser tag parameters passed to the various handler functions will consist of namespace and tag name separated by the string specified in *`seperator`* or *':'* by default.

Return Values

Returns a resource handle for the new XML parser.

See Also

- xml_parser_create()
- xml_parser_free()

xml_parser_create

xml_parser_create -- Create an XML parser

Description

resource **xml_parser_create** ([string `$encoding`])

xml_parser_create() creates a new XML parser and returns a resource handle referencing it to be used by the other XML functions.

Parameters

`encoding`

The optional *`encoding`* specifies the character encoding for the input/output in PHP 4. Starting from PHP 5, the input encoding is automatically detected, so that the *`encoding`* parameter specifies only the output encoding. In PHP 4, the default output encoding is the same as the input charset. If empty string is passed, the parser attempts to identify which encoding the document is encoded in by looking at the heading 3 or 4 bytes. In PHP 5.0.0 and 5.0.1, the default output charset is ISO-8859-1, while in PHP 5.0.2 and upper is UTF-8. The supported encodings are *ISO-8859-1*, *UTF-8* and *US-ASCII*.

Return Values

Returns a resource handle for the new XML parser.

See Also

- xml_parser_create_ns()
- xml_parser_free()

xml_parser_free

xml_parser_free -- Free an XML parser

Description

bool **xml_parser_free** (resource `$parser`)

Frees the given XML *`parser`*.

Parameters

`parser`
A reference to the XML parser to free.

Return Values

This function returns **FALSE** if *`parser`* does not refer to a valid parser, or else it frees the parser and returns **TRUE**.

xml_parser_get_option

xml_parser_get_option -- Get options from an XML parser

Description

mixed **xml_parser_get_option** (resource `$parser`, int `$option`)

Gets an option value from an XML parser.

Parameters

`parser`
A reference to the XML parser to get an option from.

`option`
Which option to fetch. **XML_OPTION_CASE_FOLDING** and **XML_OPTION_TARGET_ENCODING** are available. See xml_parser_set_option() for their description.

Return Values

This function returns **FALSE** if *`parser`* does not refer to a valid parser or if *`option`* isn't valid (generates also a **E_WARNING**). Else the option's value is returned.

xml_parser_set_option

xml_parser_set_option -- Set options in an XML parser

Description

bool **xml_parser_set_option** (resource `$parser`, int `$option`, mixed `$value`)

Sets an option in an XML parser.

Parameters

`parser`
A reference to the XML parser to set an option in.

`option`
Which option to set. See below. The following options are available:

XML parser options

Option constant	Data type	Description
XML_OPTION_CASE_FOLDING	integer	Controls whether case-folding is enabled for this XML parser. Enabled by default.
XML_OPTION_SKIP_TAGSTART	integer	Specify how many characters should be skipped in the beginning of a tag name.
XML_OPTION_SKIP_WHITE	integer	Whether to skip values consisting of whitespace characters.
XML_OPTION_TARGET_ENCODING	string	Sets which target encoding to use in this XML parser.By default, it is set to the same as the source encoding used by xml_parser_create(). Supported target encodings are *ISO-8859-1*, *US-ASCII* and *UTF-8*.

`value`

The option's new value.

Return Values

This function returns **FALSE** if *`parser`* does not refer to a valid parser, or if the option could not be set. Else the option is set and **TRUE** is returned.

xml_set_character_data_handler

xml_set_character_data_handler -- Set up character data handler

Description

bool **xml_set_character_data_handler** (resource `$parser`, callback `$handler`)

Sets the character data handler function for the XML parser *`parser`*.

Parameters

`parser`

`handler`

`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept two parameters:
handler (resource `$parser`, string `$data`)

`parser`

The first parameter, *parser*, is a reference to the XML parser calling the handler.

`data`

The second parameter, *`data`*, contains the character data as a string.
Character data handler is called for every piece of a text in the XML document. It can be called multiple times inside each fragment (e.g. for non-ASCII strings). If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

xml_set_default_handler

xml_set_default_handler -- Set up default handler

Description

bool **xml_set_default_handler** (resource `$parser`, callback `$handler`)

Sets the default handler function for the XML parser *`parser`*.

Parameters

`parser`

`handler`

`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept two parameters: *handler* (resource `$parser`, string `$data`)

`parser`

The first parameter, *parser*, is a reference to the XML parser calling the handler.

`data`

The second parameter, *`data`*, contains the character data.This may be the XML declaration, document type declaration, entities or other data for which no other handler exists.

If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

xml_set_element_handler

xml_set_element_handler -- Set up start and end element handlers

Description

bool **xml_set_element_handler** (resource `$parser`, callback `$start_element_handler`, callback `$end_element_handler`)

Sets the element handler functions for the XML *`parser`*. *`start_element_handler`* and *`end_element_handler`* are strings containing the names of functions that must exist when xml_parse() is called for *`parser`*.

Parameters

`parser`

`start_element_handler`

The function named by *`start_element_handler`* must accept three parameters:
start_element_handler (resource `$parser`, string `$name`, array `$attribs`)

`parser`

The first parameter, *parser*, is a reference to the XML parser calling the handler.

`name`

The second parameter, *`name`*, contains the name of the element for which this handler is called.If case-folding is in effect for this parser, the element name will be in uppercase letters.

`attribs`

The third parameter, *`attribs`*, contains an associative array with the element's attributes (if any).The keys of this array are the attribute names, the values are the attribute values.Attribute names are case-folded on the same criteria as element names.Attribute values are *not* case-folded. The original order of the attributes can be retrieved by walking through *`attribs`* the normal way, using each().The first key in the array was the first attribute, and so on.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

`end_element_handler`

The function named by *`end_element_handler`* must accept two parameters:

end_element_handler (resource `$parser`, string `$name`)

`parser`
The first parameter, *parser*, is a reference to the XML parser calling the handler.

`name`
The second parameter, *`name`*, contains the name of the element for which this handler is called.If case-folding is in effect for this parser, the element name will be in uppercase letters.

If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

xml_set_end_namespace_decl_handler

xml_set_end_namespace_decl_handler -- Set up end namespace declaration handler

Description

bool **xml_set_end_namespace_decl_handler** (resource `$parser`, callback `$handler`)

Set a handler to be called when leaving the scope of a namespace declaration. This will be called, for each namespace declaration, after the handler for the end tag of the element in which the namespace was declared.

Parameters

`parser`
A reference to the XML parser.

`handler`
`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept three parameters, and should return an integer value. If the value returned from the handler is **FALSE** (which it will be if no value is returned), the XML parser will stop parsing and xml_get_error_code() will return **XML_ERROR_EXTERNAL_ENTITY_HANDLING**.
handler (resource `$parser`, string `$user_data`, string `$prefix`)

`parser`
The first parameter, *parser*, is a reference to the XML parser calling the handler.

`user_data`

`prefix`

If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

See Also

- xml_set_start_namespace_decl_handler()

xml_set_external_entity_ref_handler

xml_set_external_entity_ref_handler -- Set up external entity reference handler

Description

bool **xml_set_external_entity_ref_handler** (resource `$parser`, callback `$handler`)

Sets the external entity reference handler function for the XML parser *`parser`*.

Parameters

`parser`

`handler`

`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept five parameters, and should return an integer value.If the value returned from the handler is **FALSE** (which it will be if no value is returned), the XML parser will stop parsing and xml_get_error_code() will return **XML_ERROR_EXTERNAL_ENTITY_HANDLING**.
handler (resource `$parser`, string `$open_entity_names`, string `$base`, string `$system_id`, string `$public_id`)

`parser`
The first parameter, *parser*, is a reference to the XML parser calling the handler.

`open_entity_names`
The second parameter, *`open_entity_names`*, is a space-separated list of the names of the entities that are open for the parse of this entity (including the name of the referenced entity).

`base`
This is the base for resolving the system identifier (*`system_id`*) of the external entity.Currently this parameter will always be set to an empty string.

`system_id`
The fourth parameter, *`system_id`*, is the system identifier as specified in the entity declaration.

`public_id`
The fifth parameter, *`public_id`*, is the public identifier as specified in the entity declaration, or an empty string if none was specified; the whitespace in the public identifier will have been normalized as required by the XML spec.
If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

xml_set_notation_decl_handler

xml_set_notation_decl_handler -- Set up notation declaration handler

Description

bool **xml_set_notation_decl_handler** (resource `$parser`, callback `$handler`)

Sets the notation declaration handler function for the XML parser *`parser`*.

A notation declaration is part of the document's DTD and has the following format:

```
<!NOTATION <parameter>name</parameter>
{ <parameter>systemId</parameter> | <parameter>publicId</parameter>?>
```

See » section 4.7 of the XML 1.0 spec for the definition of notation declarations.

Parameters

`parser`

`handler`

`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept five parameters:
handler (resource `$parser`, string `$notation_name`, string `$base`, string `$system_id`, string `$public_id`)

`parser`

The first parameter, *parser*, is a reference to the XML parser calling the handler.

`notation_name`

This is the notation's *`name`*, as per the notation format described above.

`base`

This is the base for resolving the system identifier (*`system_id`*) of the notation declaration. Currently this parameter will always be set to an empty string.

`system_id`

System identifier of the external notation declaration.

`public_id`

Public identifier of the external notation declaration.

If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

xml_set_object

xml_set_object -- Use XML Parser within an object

Description

bool **xml_set_object** (resource `$parser`, object `&$object`)

This function allows to use *`parser`* inside *`object`*. All callback functions could be set with xml_set_element_handler() etc and assumed to be methods of *`object`*.

Parameters

`parser`

`object`

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #9 - xml_set_object() example

```
<?php
class xml  {
   var $parser;

   function xml()
   {
       $this->parser = xml_parser_create();

       xml_set_object($this->parser, $this);
       xml_set_element_handler($this->parser, "tag_open", "tag_close");
       xml_set_character_data_handler($this->parser, "cdata");
   }

   function parse($data)
   {
       xml_parse($this->parser, $data);
   }

   function tag_open($parser, $tag, $attributes)
   {
```

```
        var_dump($parser, $tag, $attributes);
    }

    function cdata($parser, $cdata)
    {
        var_dump($parser, $cdata);
    }

    function tag_close($parser, $tag)
    {
        var_dump($parser, $tag);
    }

} // end of class xml

$xml_parser = new xml();
$xml_parser->parse("<A ID='hallo'>PHP</A>");
?>
```

xml_set_processing_instruction_handler

xml_set_processing_instruction_handler -- Set up processing instruction (PI) handler

Description

bool **xml_set_processing_instruction_handler** (resource `$parser`, callback `$handler`)

Sets the processing instruction (PI) handler function for the XML parser *`parser`*.

A processing instruction has the following format:

<?*targetdata*?>

You can put PHP code into such a tag, but be aware of one limitation: in an XML PI, the PI end tag (*?*>) can not be quoted, so this character sequence should not appear in the PHP code you embed with PIs in XML documents.If it does, the rest of the PHP code, as well as the "real" PI end tag, will be treated as character data.

Parameters

`parser`

`handler`

`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept three parameters: *handler* (resource `$parser`, string `$target`, string `$data`)

`parser`

The first parameter, *parser*, is a reference to the XML parser calling the handler.

`target`

The second parameter, *`target`*, contains the PI target.

`data`

The third parameter, *`data`*, contains the PI data.

If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

xml_set_start_namespace_decl_handler

xml_set_start_namespace_decl_handler -- Set up start namespace declaration handler

Description

bool **xml_set_start_namespace_decl_handler** (resource `$parser`, callback `$handler`)

Set a handler to be called when a namespace is declared. Namespace declarations occur inside start tags. But the namespace declaration start handler is called before the start tag handler for each namespace declared in that start tag.

Parameters

`parser`
A reference to the XML parser.

`handler`
`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept four parameters, and should return an integer value. If the value returned from the handler is **FALSE** (which it will be if no value is returned), the XML parser will stop parsing and xml_get_error_code() will return **XML_ERROR_EXTERNAL_ENTITY_HANDLING**.
handler (resource `$parser`, string `$user_data`, string `$prefix`, string `$uri`)

`parser`
The first parameter, *parser*, is a reference to the XML parser calling the handler.

`user_data`

`prefix`

`uri`

If a handler function is set to an empty string, or **FALSE**, the handler in question is disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

See Also

- xml_set_end_namespace_decl_handler()

xml_set_unparsed_entity_decl_handler

xml_set_unparsed_entity_decl_handler -- Set up unparsed entity declaration handler

Description

bool **xml_set_unparsed_entity_decl_handler** (resource `$parser`, callback `$handler`)

Sets the unparsed entity declaration handler function for the XML parser *`parser`*.

The *`handler`* will be called if the XML parser encounters an external entity declaration with an NDATA declaration, like the following:

```
<!ENTITY <parameter>name</parameter> {<parameter>publicId</parameter> |
<parameter>systemId</parameter>}
       NDATA <parameter>notationName</parameter>
```

See » section 4.2.2 of the XML 1.0 spec for the definition of notation declared external entities.

Parameters

`parser`

`handler`

`handler` is a string containing the name of a function that must exist when xml_parse() is called for *`parser`*. The function named by *`handler`* must accept six parameters:
handler (resource `$parser`, string `$entity_name`, string `$base`, string `$system_id`, string `$public_id`, string `$notation_name`)

`parser`
The first parameter, *parser*, is a reference to the XML parser calling the handler.

`entity_name`
The name of the entity that is about to be defined.

`base`
This is the base for resolving the system identifier (*`systemId`*) of the external entity.Currently this parameter will always be set to an empty string.

`system_id`
System identifier for the external entity.

`public_id`
Public identifier for the external entity.

`notation_name`
Name of the notation of this entity (see xml_set_notation_decl_handler()).
If a handler function is set to an empty string, or **FALSE**, the handler in question is

disabled.

Note
Instead of a function name, an array containing an object reference and a method name can also be supplied.

Return Values

Returns **TRUE** on success or **FALSE** on failure.