# Error Parser Users' Manual


**Daniel Prévost**

**Error Parser Users' Manual**

by Daniel Prévost

# Table of Contents

# Chapter 1. Introduction

Error Parser is a small tool for application developers. It simplifies the management of error codes, error messages and documentation for both small and large projects.

This is accomplished by creating a master XML file containing all the necessary information on the errors (value, error message and documentation). Error Parser uses this file to generate the appropriate output files for different programming languages.

Control of the code generation is done by an option file written in XML. The format for both XML files is explained below.

# Chapter 2. New in this release

This release is a minor update - support for Java was added.

# Chapter 3. User Guide

## 3.1. Introduction

The program errorParser is very easy to use. From the command line just enter:

```
errorParser -o option_file error_file
```

where `option_file` and `error_file` are the names of two xml files specifying the options and all the data associated with the errors. The real complexity is of course in the xml files themselves. There are examples of these files in the distribution directory and their syntax is explained below.

The program can also be called with a single option:

- -h, -? or --help

    will give you the list of options and the syntax of the program.

- -v or --version

    will tell you which version you are using (new since version 1.0).

## 3.2. The xml error file

This section documents version 1.3 of the DTD.

The xml file starts with the usual xml header (version of xml, encoding of the file, the location of the DTD, etc.). If you're not familiar with xml and want more information on this, you might want to read additional documentation, for example the web site of the  W3 Schools (http://www.w3schools.com/xml/default.asp).

Each of the xml tag specific to Error Parser is explained in the list below. Unless specified, each tag can only appear once (for each of its parent) and in the order listed here. Example: the <years> tag must be the first sub-tag of <copyright> and must be unique - however, if you have more than one <copyright> tag, each will have its <years> sub-tag.

- <errorlist version="0.3">

This tag starts the document itself (the root element). The version attribute is a string of your choosing and is there to help you synchronize your work. You can use it to make sure that the output files match the version of your software, for example. This attribute is optional.

- <copyright_group>

This is an optional tag. You use it if you want to generate copyright information and the terms of the license in the output files. This tag has only one sub-tag, <copyright>, which can be repeated multiple times (if you have multiple authors with possibly different licenses).

- <copyright>

Allows you to enter copyright information in the output files. It has multiple sub-tags. As mentioned previously, multiple <copyright> tags are allowed.

  - <years>

    The interval of years for the current copyright notice.

  - <authors>

    The name(s) of the owner(s) of the copyright.

  - <license_para>

    This tag contains a single paragraph of the terms of your license. Use multiple <license_para> if your license has multiple paragraphs. You must have at least one <license_para> for each <copyright>.

- <errgroup>

<errgroup> is used to organize your errors in groups, if you so desire. For example, you could split your errors in a group of i/o errors, a group of network errors, etc. You can have multiple <errgroup> but you must have at least one.

If you don't want to use this feature, use a single group and do not include the <errgroup_ident> sub-tag.

- <errgroup_ident>

This sub-tag of <errgroup> is optional. The sub-tags associated with this tag are used to generate comments in your output header file (and will be used to generate the DocBook documentation eventually).

You can have multiple <errgroup_ident> - one for each language supported by your application. This is done with an optional attribute, xml:lang. If the xml:lang attribute is missing, the default attribute will be used. The default attribute is 'en'. You can modify it by editing the shipped DTD.

Note: the parser will use the first <errgroup_ident> matching the language selected in the option file. If the selected language is not found, the first <errgroup_ident> is used.

- <errgroup_name>

  A name for this group of errors. For example, Network errors.

- <errgroup_desc>

  Description of this group of errors. You must have at least one <errgroup_desc> or more as needed. Each will be mapped to a paragraph.

- <error>

  This tag is also a sub-tag of <errgroup>. You need at least one <error> for each <errgroup> but you can have as many as needed.

  - <errnumber>

    The numerical value of the error code.

  - <errname>

    The generic name of the error. This name is going to be used to generate either an enum entry or a #define in the output header file.

    Note: if you add a prefix to your errors (to avoid namespace conflicts), it is recommended to leave the prefix out. For example, PSO_OBJECT_IS_DELETED would become <errname> OBJECT_IS_DELETED </errname> - the prefix can be added by specifying it in the option file.

  - <message_group>

This tag is used to group the error messages and the documentation for the error - which are encapsulated in the <message> tag. Multiple <message> tags are supported (you must have at least one), one for each supported language.

- <message>

Container for both the error message and the documentation for this error. This tag has an optional attribute, xml:lang. If the xml:lang attribute is missing the default attribute will be used. The default attribute is 'en'. You can modify it by editing the shipped DTD.

You must have at least one <message>, multiple ones are supported (one per language used in your application).

Note: the parser will use the first <message> matching the language selected in the option file. If the selected language is not found, the first <message> is used.

- <errormsg>

This tag identifies the text of the error message that will be retrieved by the function *yourprefix_ErrorMessage()*.

Note: the inclusion of some characters ('%' for example) or chain of characters (escape sequences) might be problematic. You can control how Error Parser will handle these occurrences by setting the appropriate options in the option file.

- <errordoc>

Each <errordoc> is a paragraph of the documentation. Therefore, you can have multiple <errordoc> but you must have at least one.

# 3.3. The xml option file

This section documents version 2.1 of the DTD. This new release of the parser will understand version 2.0 of the DTD. Current option files based on version 1.* must be upgraded.

- <options version="2.2">

This tag starts the document itself (the root element). The attribute "version" (the version of the DTD) is required; it is used by the parser to support multiple versions of the DTD (starting at version 2.0), as needed.

- <general_options>

  Options with a global impact or that will influence the processing of more than one output file.

  - <prefix_error_no_namespace>

    The prefix to be used for the error codes (enum or #define). For example, if your error is OBJECT_IS_DELETED (using the <errname> tag) and your prefix is PSO, the error code becomes PSO_OBJECT_IS_DELETED.

    The prefix is currently used in the error-message files and in the main header file. It is not used in C# or Python since these languages support namespaces (prefixes like this are used to avoid namespace clashes in languages without a namespace, for example C).

  - <selected_lang>

    Optional parameter to select the language that will be used for the documentation and error messages. If absent, the first <message> tag will be used.

    For each error, if there are no <message> for the selected language, the first <message> tag will be used.

    The chosen language is specified using the xml:lang attribute. For example:

    ```
    <selected_lang xml:lang="en"/>
    ```

- <header_file>

  Optional. Options for the main header file.

  - <header_enum>

    Optional. The name of the enum. If absent, #define will be generated instead.

  - <header_dirname>

Optional. The name of the directory where the error header file is to be created. If absent, the current directory or the directory specified by <header_name> will be used.

Alternatively, you can put the directory name (or part of the directory name) directly in the <header_name> tag. But if you do, the "guard"[1] will include that directory name.

Example: the output file name is *my_dir/my_sub_dir/my_sub_project/errors.h* and you want to include *my_sub_project* in the guard. The option file would look like:

```
<header_dirname>my_dir/my_sub_dir</header_dirname>
<header_name>my_sub_project/errors.h</header_name>
```

The generated guard would look like this:

```
#ifndef MY_SUB_PROJECT_ERRORS_H
#define MY_SUB_PROJECT_ERRORS_H
...
#endif /* MY_SUB_PROJECT_ERRORS_H */
```

- <header_name>

  The name of the header file for errors. See the item <header_dirname> for a discussion on names of files, directories and the generated header guard.

- <errmsg_files>

  Optional. Options for the error-message files (.c and .h).

  - <errmsg_dirname>

    Optional. The name of the directory where the header file will be created. If absent, the current directory or the directory specified by <errmsg_header_name> will be used.

    See the item <header_dirname> for a discussion on names of files, directories and the generated header guard.

  - <errmsg_header_name>

The name of the header file for the code to retrieve the error messages. See the item <header_dirname> for a discussion on names of files, directories and the generated header guard.

- <errmsg_c_fullname>

The full name of the "C" file for the code to retrieve the error messages (note: the path can be relative to the current working directory).

- <errmsg_options>

Optional. This tag has no content and a single attribute, build_dll. The attribute is an enumeration with two values, yes and no. The default is no. Set it to yes if you are building a DLL on Microsoft Windows and you want to access the error-message function from outside the DLL.

If you are setting the attribute to yes on other platforms it will generate an empty macro, in other words no side effects. However, if you are building a self-contained EXE on Windows, Visual Studio will warn you about this.

- <errmsg_prefix_var>

The prefix to be used for all variables in the code to retrieve the error messages.

- <errmsg_msg>

This tag has no content but its attributes are important - they define the allowed characteristics (and the transformations, if needed) of the error messages defined in the xml error file (<errormsg>).

The problem: the generated C strings will be used, eventually, by standard C/C++ libraries and some sequence of characters could have negative side effects. The most problematic character is likely the % symbol - it is interpreted differently by libc (printf()) and the iostream library (cout).

The attributes:

- allow_escapes

An enumeration with two values, yes and no. The default is no; escape sequences are not permitted.

Note: if you want to allow quotes in your messages, do *NOT* escape them if allow_escapes is set to no (they will be escaped by Error Parser).

- allow_quotes

  An enumeration with two values, yes and no. The default is yes; quotes (' and ") are allowed and will be escaped as needed.

- percent

  The text that will be used to replace the % symbol. This attribute is mandatory and has no default value. The textual "percent" is recommended.

- &lt;csharp&gt;

  Optional. If present, code for C# will be generated (in the form of an enum). Additional sub-tags, described below, are used to provide the required parameters.

  - &lt;cs_filename&gt;

    Full or relative path to the generated C# code.

  - &lt;cs_enum_name&gt;

    The name of the C# enum.

  - &lt;cs_namespace&gt;

    Optional. The name of the C# namespace. If absent, the code is generated without a namespace.

- &lt;ext_py&gt;

  Optional. If present, a C header file, to be used by a Python extension module, will be generated. Additional sub-tags,described below, are used to provide the required parameters.

  The code generates a C function to create 2 dict (associated arrays) objects. The key for the first dict is the error symbolic name and the value is the error number. The key for the second dict is the error number and the value is the symbolic name.

  Example of use in Python (taken from the tests of Photon):

```
try:
    s.create_object( ", pso.BaseDef(pso.FOLDER, 0) )
except pso.error, (msg, errcode):
    if errcode != pso.errs['INVALID_LENGTH']:
        print 'error = ', pso.errnames[errcode]
        print msg
        raise
```

In this example, pso.errs and pso.errnames are the two dict used by the Photon extension module. See the section "*Python extension module*" for some tips on how to use the generated file in your C code.

• <ext_py_dirname>

  Optional. The name of the directory where the code should be generated. If absent, the current directory or the directory specified by <ext_py_filename> will be used.

  See the item <header_dirname> for a discussion on names of files, directories and the generated header guard.

• <ext_py_filename>

  The name of the header file. See the item <header_dirname> for a discussion on names of files, directories and the generated header guard.

• <ext_py_function>

  Optional. The name of the generated function. If absent, the code will use AddErrors().

• <pure_python>

  Optional. If present, a Python file will be generated. Additional sub-tags, described below, are used to provide the required parameters.

  The code generates a Python function to create 2 dict (associated arrays) objects. The key for the first dict is the error symbolic name and the value is the error number. The key for the second dict is the error number and the value is the symbolic name.

  • <pure_py_filename>

    The name of the Python generated file.

- <pure_py_function>

  Optional. The name of the generated function. If absent, the code will use add_errors().

- <java>

  Optional. If present, a Java file will be generated. Additional sub-tags, described below, are used to provide the required parameters.

  The code generates a Java enum class. Java 5.0 or greater is required.

  - <java_filename>

    The name of the Java generated file.

  - <java_enum_name>

    The name of the Java enum.

  - <java_package>

    Optional. The name of the Java package.

# 3.4. Example: the Photon software

Photon is a framework: it includes a library with interfaces in C, C++, C# and Python and a few standalone executables. Error Parser is used for the library and generates output files for all the different interfaces. It is also used for the Photon server, Quasar, but in a more limited fashion.

Photon uses the GNU toolchain on Linux/Unix. As such, this example might not be very useful if you use a different approach (Visual Studio only, Eclipse, etc.).

It should be noted that the Error Parser program is not needed to build Photon. The output files generated by Error Parser are included in the source code distribution. However, Error Parser is required for developers and maintainers of the framework.

## 3.4.1. configure.ac

The autoconf program is used by the photon framework to determine multiple aspects of the current system: the operating system, the hardware, the availability of some C functions, of some programs, etc. Error Parser is one of these programs.

To tell autoconf to look for Error Parser, the following lines must be added to the file configure.ac:

```
# Tell our makefiles that we can use errorParser to regenerate the error
# handling code, as needed.
AC_CHECK_PROG([ERRORPARSER_IS_PRESENT], [errorParser], [yes], [no])
AM_CONDITIONAL([COND_USE_ERRORPARSER], [test "$ERRORPARSER_IS_PRESENT" = yes])
```

If Error Parser is not present on the system, the automake conditional COND_USE_ERRORPARSER will be false and the Makefile will not attempt to regenerate the error output files even if the xml error file itself is modified.

## 3.4.2. Makefile.am

Including the rules for the parser in a Makefile is relatively simple. Here is a code snippet from the Makefile.am of Quasar; the syntax for ordinary makefiles should be similar.

```
# The main target, quasar
bin_PROGRAMS = quasar

# The list of files generated by errorParser.
OUTPUT_FILES =                 \
        quasarErrors.h         \
        quasarErrorHandler.c \
        quasarErrorHandler.h

# This is "true" if the autoconf program found errorParser. Otherwise
# we will not attempt to redo the generated files.
if COND_USE_ERRORPARSER
$(OUTPUT_FILES): quasarOptions.xml quasarErrors.xml
 errorParser --options quasarOptions.xml quasarErrors.xml
endif

# The source code needed to build quasar. We list the errorParser
# generated files first since they must be "refreshed" first since they
# might be used in the rest of the code (certainly true for quasarErrors.h).
quasar_SOURCES =               \
        $(OUTPUT_FILES) \
        Acceptor.cpp    \
        ...
```

## 3.4.3. Option files

Two examples are shown here. The file option.xml is used for the library of Photon. It uses most of the features of Error Parser, including the generation of code for C# and Python extension module. In contrast, quasarOptions.xml is based on a smaller subset of available features.

### 3.4.3.1. options.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE options PUBLIC
    "-//Error Parser project//DTD Error Parser Options XML V2.2//EN"
    "http://errorparser.sourceforge.net/xml/errorParserOptions22.dtd">

<!--
   This xml file provides the options needed by errorParser.
   It must be used in conjonction with photon.xml.

   Generated output:
     - header file photon/psoErrors.h.
     - the code to retrieve the error messages
     - a C# enum
     - a header file for the pso extension module (python).
-->

<options version="2.2">

  <general_options>
    <!-- Prefix to be used for the error codes. -->
    <prefix_error_no_namespace> PSO </prefix_error_no_namespace>
    <selected_lang xml:lang="en" />
  </general_options>

  <header_file>
    <!-- The name of the enum. -->
    <header_enum> psoErrors </header_enum>
    <!-- Name of the output dir for the error header file. -->
    <header_dirname> ../include/photon </header_dirname>
    <!-- The file name for the .h file which will hold the errors. -->
    <header_name> psoErrors.h </header_name>
  </header_file>

  <errmsg_files>
    <!-- Name of the output dir for the code to retrieve the error messages. -->
    <errmsg_dirname> ../Nucleus </errmsg_dirname>
    <!-- The file name for the .h file for the code to retrieve the error
         messages. -->
    <errmsg_header_name> psoErrorHandler.h </errmsg_header_name>
```

```
    <!-- The file name for the .c file for the code to retrieve the error
         messages. -->
    <errmsg_c_fullname> ../Nucleus/psoErrorHandler.c </errmsg_c_fullname>
    <errmsg_options build_dll="yes"/>
    <!-- Prefix to be used for all variables in the code to retrieve the
         error messages -->
    <errmsg_prefix_var> pson </errmsg_prefix_var>
    <errmsg_msg percent="percent"/>
  </errmsg_files>

  <!-- Optional section for creating an enum for C# -->
  <csharp>
    <!-- Full or relative path to the generated C# code -->
    <cs_filename> ../CS/PhotonLib/Errors.cs </cs_filename>
    <cs_enum_name> PhotonErrors </cs_enum_name>
    <!-- The namespace is optional -->
    <cs_namespace> Photon </cs_namespace>
  </csharp>

  <!-- Optional section for Python (extension module) -->
  <ext_python>
    <ext_py_dirname> ../Python/src </ext_py_dirname>
    <ext_py_filename> errors.h </ext_py_filename>
  </ext_python>

</options>
```

## 3.4.3.2. quasarOptions.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE options SYSTEM
             "/usr/local/share/xml/errorParser/errorParserOptions22.dtd">

<!-- This xml file provides the options to the errorParser program. -->

<options version="2.2">

  <!-- Section for all general options. -->
  <general_options>
    <!-- Prefix to be used for the error codes (enum or #define). -->
    <prefix_error_no_namespace> PSOQ </prefix_error_no_namespace>
    <selected_lang xml:lang="en" />
  </general_options>

  <!-- Section for the main header file. -->
  <header_file>
    <!-- The name of the enum. -->
    <header_enum> psoqErrors</header_enum>
    <!-- The file name for the .h file which will hold the errors. -->
    <header_name> quasarErrors.h </header_name>
```

```
    </header_file>

    <!-- Section for the error message files. -->
    <errmsg_files>
      <!-- The file name for the .h file for the code to retrieve the error
           messages. -->
      <errmsg_header_name> quasarErrorHandler.h </errmsg_header_name>
      <!-- The file name for the .c file for the code to retrieve the error
           messages. -->
      <errmsg_c_fullname> quasarErrorHandler.c </errmsg_c_fullname>
      <!-- Options for the error message -->
      <errmsg_options build_dll="no" />
      <!-- Prefix to be used for all variables in the code to retrieve the
           error messages -->
      <errmsg_prefix_var> psoq </errmsg_prefix_var>
      <errmsg_msg percent="percent" />
    <errmsg_files>

</options>
```

## 3.4.4. The Error File

This is a short version of the error file used by Quasar, the Photon server:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE errorlist PUBLIC "-//Error Parser project//DTD Error Parser XML V1.3//EN"
            "http://errorparser.sourceforge.net/xml/errorParser13.dtd">
<!--
    This file contains the error codes specific to the server of
    Photon, quasar.

    You can replace the DOCTYPE with this, if you prefer:
<!DOCTYPE errorlist SYSTEM
            "/usr/local/share/xml/errorParser/DTD/errorParser13.dtd">

-->

<!-- Photon next version is 0.4 -->
<errorlist version="0.4">
  <!-- Copyright information and any additional info that will appear at
       the top of the generated files. This is optional. -->
  <copyright_group>
    <copyright>
      <years>2006-2008</years>
      <authors>Daniel Prevost</authors>
      <!-- Each <license_para> is mapped to a paragraph (easier to read) -->
      <license_para>
        This file is part of the Photon framework.
```

```
      </license_para>
      <!-- The GPL license is of course for Photon, not for Error Parser... -->
      <license_para>
        This file may be distributed and/or modified under the terms of the
        GNU General Public License version 2 as published by the Free Software
        Foundation and appearing in the file COPYING included in the
        packaging of this library.
      </license_para>
      <license_para>
        This library is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
      </license_para>
    </copyright>
  </copyright_group>
  <errgroup>
    <error>
      <errnumber> 0 </errnumber>
      <errname> OK </errname>
      <message_group>
        <message xml:lang="en">
          <errormsg>No error...</errormsg>
          <errordoc>No error...</errordoc>
        </message>
      </message_group>
    </error>

    <error>
      <errnumber> 1 </errnumber>
      <errname> NOT_ENOUGH_HEAP_MEMORY </errname>
      <message_group>
        <message xml:lang="en">
          <errormsg>
            Not enough heap memory (non shared memory).
          </errormsg>
          <errordoc>
            Not enough heap memory (non shared memory).
          </errordoc>
        </message>
      </message_group>
    </error>

    [the remaining errors were left out for brevity]
  </errgroup>

</errorlist>
```

## 3.4.5. Python extension module

Adding the generated code to an extension module is easy. It requires two steps:

• The fist step is the creation of the the two dict objects. using the function in the generated header file. This should be done in the function used to initialize your module or in a function called by the init() function.

• The second step is to add these two new objects to your module using PyModule_AddObject().

Here's a code snippet taken for the pso module; you will have to modify it to suit your needs:

```c
# include "errors.h" /* Error Parser generated header file */

PyMODINIT_FUNC
initpso(void)
{
   PyObject * m = NULL, * tup = NULL, * errs = NULL, * errNames = NULL;
   int rc;

   m = Py_InitModule3( "pso",
                       pso_methods,
                       "Photon module.");
   if (m == NULL) return;

   /*
    * AddErrors() is defined in errors.h - it creates the two dict objects
    * needed to access the error codes (values and/or symbolic names) and
    * returns them, as a tuple.
    */
   tup = AddErrors();
   if ( tup == NULL ) return;

   if ( ! PyArg_ParseTuple(tup, "OO", &errs, &errNames) ) return;

   /*
    * errs allows you to do your_module.errs['INVALID_NAME'] in Python to
    * test for specific errcode without having to hardcode the error number.
    *
    * [example: I use it in tests to make sure that functions in my
    * module return the expected error/exception (on invalid parameters, etc.).
    *
    * Note: You can choose which ever names you want for the 2 dict instead
    *       of the names "errs" and "err_names".
    */
   rc = PyModule_AddObject( m, "errs", errs );
   if ( rc != 0 ) return;
   /*
    * err_names allows you to retrieve the symbolic name of an error
    * from a returned error number (your_module.err_names[err_number]).
```

```
 * In some cases, it could be useful (possibly to generate a message
 * in your exception handler?).
 */
rc = PyModule_AddObject( m, "err_names", errNames );
if ( rc != 0 ) return;


...
```

# Notes

1. guard: a small chunk of preprocessor code used to avoid including the same header file twice.