Error Parser Manual

Daniel Prévost

Error Parser 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 accomplish 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 programmation 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 major update, the software was partially rewritten and completely reorganized (refactored) to make it easier to both maintain it and to enhance it.

The good news: this update has very little impact on anyone using the program, with one exception: you must update your option file to the new format.

The new format of the option file (version 2.0) adopt the modular approach used in the redesign of the code - it is divided in sections, one section for each type of output file.

Additionally, this new version resolved multiple small bugs and closed all known memory leaks.

Chapter 3. Installation

The installation procedure of the software itself is relatively simple if you use the pre-packaged version of the software (RPM, Inno setup, etc.) or if you build the software from scratch and use make install.

This section will look into some potential installation issues in some details.

3.1. The xml catalog on Linux/Unix

The two DTD (Document Type Definition) for Error Parser can be installed in the global xml catalog (usually /etc/xml/catalog). Is this needed? Depends on your exact situation.

A DTD can be identified using a PUBLIC clause (in the xml file), for example:

```
<!DOCTYPE errorlist PUBLIC
   "-//Error Parser project//DTD Error Parser XML V1.2//EN"
    "http://photonsoftware.org/xml/ep/errorParser12.dtd">
```

In this case the DTD will be downloaded from the Error Parser web site.

You can also use SYSTEM instead:

```
<!DOCTYPE errorlist SYSTEM
   "/usr/local/share/xml/errorParser/errorParser12.dtd">
```

Using SYSTEM is perfectly correct in most cases. The main advantage of the PUBLIC clause is to make sure that multiple developers uses the same DTD (if you modify it).

If you decide to use the PUBLIC clause and do not want to download the DTD on every invocation of Error Parser, adding the Error Parser catalog to your main catalog is the way to go. The program xmlcatalog, included in the libxml2 package, will simplify this task. Furthermore a bash script is provided by Error Parser (possibly installed in /usr/local/share/xml/errorParser/install_catalog.sh if Error Parser is installed in /usr/local). You'll need to edit this script to tailor it for your installation.

Recommendation: use the SYSTEM clause. There is likely no real needs to use the xml catalog (the exception to this: if you decide to modify Error Parser itself, using both SYSTEM and PUBLIC might be handy to distinguish between the production and the development version, although using version identifiers might be a better solution).

3.2. The xml catalog on Windows

The previous section already discussed the use of PUBLIC or SYSTEM clause for identifying a DTD. The biggest difference between Microsoft Windows and for example Linux, is the absence of a centralized way of managing xml DTD and Schemas - each software manages its XML on its own. It makes even more sense to use SYSTEM here.

If you still want to install an xml catalog on Windows, the only thing you need to know is that libxml2 uses the environment variable XML_CATALOG_FILES to find the main xml catalog.

There is an example of a main catalog in the DTD directory (main_catalog_win32.xml). If the main catalog already exists you can use the script DTD\install_catalog.vbs. In both cases you will need to edit the files to tailor them to your installation.

3.3. libxml2 on Windows

You can find binaries for the complete libxml2 package and other associated packages (libxslt, iconv, etc.) here (http://www.zlatkovic.com/libxml.en.html). You will need the full package if you build Error Parser from scratch.

In previous releases, the Error Parser installer gave you the option of installing the run-time environment needed to run errorParser.exe. This is no longer needed - libxml is now included in Error Parser (linked statically).

Chapter 4. Building the Software

The only prerequisite software needed to build and run Error Parser is the library libxml2 (the Gnome project xml parser). If it is not installed on your system, you can download it at xmlsoft.org (http://xmlsoft.org/downloads.html).

To build the documentation, additional software is required as explained in the following sections.

4.1. Building on Linux/Unix from the repository

To retrieve the latest version of Error Parser, use the following command:

```
svn export https://errorparser.svn.sourceforge.net/svnroot/errorparser/trunk my_local_dir
```

Of course, this is a generic command, you can omit the sub-directory `trunk` to retrieve everything (furthermore, if `my_local_dir` is left empty, the current directory will be used).

Warning

The code in the repository is not always stable, be careful (hint: unstable commits will usually have the string "work in progress" included in the commit message).

The Error Parser software can also be downloaded by using git (git is a very fast distributed version control system created for the Linux kernel). Visit git home page (http://git.or.cz/index.html) for more information on this open-source software.

Error Parser uses two external git repositories. They are both kept in sync with the main repository. To retrieve the latest version from github.com, use the following command:

git-clone git@github.com:dprevost/errorparser.git my_repo

or use the following command to get it from gitorious.org

git-clone git@gitorious.org:errorparser/mainline.git my_repo

Warning

git was developed for Linux and will likely runs without problems on most Unix systems. On Microsoft Windows however, things might be different. There are two ports for Windows but I haven't try them (yet).

The next step is to run the shell script autogen.sh (in the trunk directory). This will generate everything you need (the makefiles, the configure script). The script will also run configure for you but you might want to rerun it again if you want to change its default options.

The remaining instructions are identical to the instructions for building the software from a tar file.

4.2. Building on Linux/Unix from the tar file

Step by step instructions to build the package from a tar file:

- Run configure (./configure). Here is a list of options that you might want to use to tailor the package to your needs (you can use ./configure --help to see all the options):
  - --prefix=PREFIX To change the default installation directory (default is /usr/local). Example: --prefix=/usr.
  - --mandir=DIR To change the default installation for man documentation (default is [PREFIX/man]). Example --mandir=/usr/share/man (or /usr/local/share/man).
- make
- make check (to run the test suite - optional)
- make install

4.3. Building the documentation on Linux/Unix

The documentation is provided as part of the source package. If you are only installing and using this software as is, you can skip this section. However, if you decide to modify the documentation (for example if you modify the parser itself and must update the documentation to reflect this), please continue reading.

The documentation is written in DocBook xml. The source for the current document is doc/manual.xml. There is also parser/errorParser.xml used to generate the man page. You can use a proper xml editor to edit these two files but this might be overkill - a simple text editor which understands xml syntax is likely good enough (I use jedit with the XML plugin).

To create or refresh the man page, you will need to install the DocBook DTD itself and the docbook2x package (for the program docbook2x-man which can also be installed as db2x_docbook2man for example on Fedora 9). If present, autoconf (./configure) will detect them and make will refresh the documentation if the source file was modified.

The html and pdf documentation are updated in similar ways. To update the pdf manual, db2pdf must be present on your system. In both cases, you will need a program to transform the xml. The makefile is currently written to use xsltproc.

Note: the Microsoft specific chm file (used for Microsoft help files) can only be updated on Windows.

4.4. Building on Windows from the repository

If you want to retrieve the software directly from the repository, you will need a subversion client for Windows. Tortoise (http://tortoisesvn.tigris.org/) is one possible choice. The repository is:

```
https://errorparser.svn.sourceforge.net/svnroot/errorparser/trunk
```

Warning

The code in the repository is not always stable, be careful (hint: unstable commits will usually have the string "work in progress" included in the commit message).

The remaining instructions are similar to the instructions for building the software from a zip file.

4.5. Building on Windows from a zip file

Very simple, to build the software, just use the solution file parser\parser.snl (VC++ v8, known as VS 2005). It might also work with the newer VS 2008 (untested yet).

One additional note: you will need to add the directory for the header files for libxml2 to the list of include directories in Visual Studio. You will also need to add a directory for the library files (.lib). For VS 2005, you can set this in Options (in the Tools menu).

To run the tests, from a command shell run cscript RunTests.vbs (in the tests directory). You will also need Gnu diff (http://gnuwin32.sourceforge.net/packages/diffutils.htm) for Windows (a copy of it is in the tests directory).

You can also build the software and run the tests from the command line. From the root directory, simply type "nmake -f Makefile.win32" to build the program and "nmake -f Makefile.win32 check" to run the tests (other targets: clean, checkdebug (to run the tests with the debug version of Error Parser) and docs (to build the documentation)). You might need to modify parser/Makefile.win32 if you install libxml2 in a non-standard directory (the hardcoded path for libxml2 is "C:\Program Files\libxml2").

To build the installation package itself, you will need Inno Setup (http://www.jrsoftware.org/isinfo.php). The Inno Setup file is located in installation\errorParser.iss.

4.6. Building the documentation on Windows

The documentation is provided as part of the source package. If you are only installing and using this software as is, you can skip this section. However, if you decide to modify the documentation (for example if you modify the parser itself and must update the documentation to reflect this), please continue reading.

The documentation is written in DocBook xml. The source for the current document is doc/manual.xml. There is also parser/errorParser.xml used to generate the man page. You can use a proper xml editor to edit these two files but this might be overkill - a simple text editor which understands xml syntax is likely good enough (I use jedit with the XML plugin).

As mentioned earlier, the documentation can be updated using "nmake -f Makefile.win32 docs". This will update the Microsoft help-file (chm) version of this manual and the html version. The PDF version and the man page require a Unix/Linux environment (if you use cygwin or MinGW, it might work).

To create the documentation, you will need the DocBook DTD itself, xsltproc and the Microsoft HTML Help Compiler (hhc.exe) which you can get from the HTML Help Workshop from Microsoft (and possibly other similar authoring tools).

Chapter 5. User Guide

5.1. 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 gives you the list of options and -v or --version tells you which version you are using.

5.2. The xml error file

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 (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 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).

Note: from an xml point of view, this tag is unneeded - the <copyright> tag could be used directly without this "dummy" container. However, the code of errorParser is simplify because of it, so...

- <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.

  - <notice>

    The text of the license. More exactly, a paragraph of the license. Use multiple <notice> if your license has multiple paragraphs. You must have at least one <notice> for each <copyright>.

- <errgroup>

  <errgroup> is used to organize your errors in groups, if you so desired (for example, a group of i/o errors, a group of network errors, etc). You can have multiple <errgroup>, you must have at least one.

  If you don’t want to use this feature, only 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 (currently ’en’ but you can modify the shipped DTD to use a different default tailored for your needs).

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> but you can have multiple ones, 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.

    Note: from an xml point of view, this tag is unneeded - the <message> tag could be used directly without this "dummy" container. However, the code of errorParser is simplify because of it, so...

  - <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 (currently 'en' but you can modify the shipped DTD to use a different default tailored for your needs).

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>

  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 to 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>. You must have at least one.

5.3. The xml option file

The number of potential options to the program errorParser was getting quite large and it made more sense to put all these options in a configuration file (written in xml). These options are described below.

This section document version 2.0 of the DTD. This new release of the parser will not understand previous versions of the DTD - you must upgrade it.

The main changes to the DTD: every basic option is part of a group of options. There is one group for general options (mandatory) and optional groups for each of the output format.

- <options version="2.0">

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

- <general_options>

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

  - <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 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" will include that directory name (guard: the small chunk of preprocessor code use to avoid including the same header file twice).

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 (for the code to retrieve the error messages) will be created. If absent, the current directory will be used as the starting point.

    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 use, eventually, by standard C/C++ libraries and some sequence of characters could have some bad side effects (possibly some security issues?). The most problematic issue 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 (a more fine grain approach could be written if the need arise).

    Note: if you want to allow quotes in your messages, do *NOT* escaped 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, even if allow_escapes is set to no).

    - percent

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

- <csharp>

  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.

  - <cs_filename>

    Full or relative path to the generated C# code.

  - <cs_enum_name>

    The name of the C# enum.

  - <cs_namespace>

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

- <ext_py>

  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, the value is the error number. The key for the second dict is the error number, 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
```

Evidently, pso.errs and pso.errnames are the two dict used by the Photon extension module (pso: photonsoftware.org). See the section “*Example: python extended 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 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, the value is the error number. The key for the second dict is the error number, 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().

5.4. Example: Quasar, the connection/recovery server of Photon

This is work in progress for the next iteration of the PSO project (photonsoftware.org). As a matter of fact, while working on this, I've discovered that adding new error codes this way was no more complex than adding error codes to an header file (once you understand the basics).

5.4.1. configure.ac

The autoconf program is used to determine if Error Parser is installed on the system. If it is absent, the Makefile will not attempt to regenerate the error files even if the xml error file itself is modified).

```
# 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])
```

Explanation: to download and build Photon without modifying it, Error Parser is not required (the tar file contains the default files generated by Error Parser). This is the default behaviour.

I do expect that maintainers or anyone else planning to modify the code of Photon (specially the xml error file), will install all the tools required including Error Parser.

5.4.2. Makefile.am

Including the parser in a Makefile is relatively simple. Here are code snippets 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    \
        ...
```

5.4.3. quasarOptions.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE options SYSTEM
             "/usr/local/share/xml/errorParser/errorParserOptions20.dtd">

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

<options version="2.0">

  <!-- 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. If absent, #define will be generated instead. -->
    <header_enum> psoqErrors</header_enum>
    <!-- Name of the output dir for the error header file. If absent, the
         current directory will be used.

         Alternatively, you can put the directory directly in the header_name
         element. But if you do, the "GUARD" will use that directory name. -->
    <!-- <header_dirname> not used... </header_dirname> -->
    <!-- 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>
  <!-- Name of the output dir for the code to retrieve the error messages.
       If absent, the current directory will be used. -->
  <!-- <errmsg_dirname> not used ... </errmsg_dirname> -->

  <!-- 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>
  <!-- How to handle the error message. The problem: the generated C strings
       will be use, eventually, by standard C/C++ libraries and some
       sequence of characters could have some bad side effects, possibly
       including security issues.
       The defaults:
         - escape sequences are not permitted (a more fine grain approach
           could be written if the need arise).
         - quotes (' and ") are allowed (and will be escaped as needed)
         - The % symbol. No default. I recommend the textual "percent"
           (% is the most problematic one, IMHO. %d %p, etc are interpreted
           differently by libc (printf()) and the iostream library (cout) -->
  <errmsg_msg percent="percent" />
<errmsg_files>

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

<!-- Optional section for Python extension module
     Commented out - included for completeness -->
<!-- <ext_py> -->
  <!-- <ext_py_dirname> ../Python </ext_py_dirname> -->
  <!-- <ext_py_filename> errors.h </ext_py_filename> -->
  <!-- The function name is optional -->
  <!-- <ext_py_function> MyAddErrors </ext_py_function> -->
<!-- </ext_py>   -->

<!-- Optional section for Python
```

```
       Commented out - included for completeness -->
  <!-- <pure_python> -->
    <!-- <pure_py_filename> errors.py </pure_py_filename> -->
    <!-- The function name is optional -->
    <!-- <pure_py_function> my_add_errors </pure_py_function> -->
  <!-- </pure_python>   -->

</options>
```

5.4.4. quasarErrors.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE errorlist PUBLIC "-//Error Parser project//DTD Error Parser XML V1.2//EN"
             "http://photonsoftware.org/xml/ep/errorParser12.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/errorParser12.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 <notice> is mapped to a paragraph (easier to read) -->
      <notice>
        This file is part of the Photon framework (photonsoftware.org).
      </notice>
      <!-- The GPL license is of course for Photon, not for Error Parser... -->
      <notice>
        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.
      </notice>
      <notice>
        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.
      </notice>
    </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>

  <error>
    <errnumber> 2 </errnumber>
    <errname> CFG_BCK_LOCATION_TOO_LONG</errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>Backup path is too long.</errormsg>
        <errordoc>
          The path of the backup path is longer
          than PATH_MAX (the maximum length of a filename/pathname).
        </errordoc>
      </message>
    </message_group>
  </error>

  <error>
    <errnumber> 3 </errnumber>
    <errname> NO_VERIFICATION_POSSIBLE</errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>Cannot verify the shared memory.</errormsg>
        <errordoc>
          The shared memory does not exist and therefore cannot be verified.
        </errordoc>
      </message>
    </message_group>
  </error>
```

```
<error>
  <errnumber> 4 </errnumber>
  <errname> MKDIR_FAILURE</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Cannot create the directory for the shared memory.</errormsg>
      <errordoc>
        Cannot create the directory that will be used for the shared memory
        and associated files.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 5 </errnumber>
  <errname>FILE_NOT_ACCESSIBLE</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Cannot access the backstore file for the shared memory.</errormsg>
      <errordoc>
        Cannot access the backstore file for the shared memory. You might want
        to verify the permissions and ownership of the file and/or of the
        parent directory.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 6 </errnumber>
  <errname>COPY_BCK_FAILURE</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>
        Error while creating a copy of the backstore file for the shared memory.
      </errormsg>
      <errordoc>
        Error while creating a copy of the backstore file for the shared memory
        (a backup copy is created before the shared memory is verify for
        inconsistencies).
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 7 </errnumber>
  <errname>CREATE_BACKSTORE_FAILURE</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>
```

```
        Error creating the backstore file for the shared memory.
      </errormsg>
      <errordoc>
        Error creating the backstore file for the shared memory. Possible reasons:
        not enough disk space, invalid file permissions, etc.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 8 </errnumber>
  <errname>OPEN_BACKSTORE_FAILURE</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>
        Error opening the backstore file for the shared memory.
      </errormsg>
      <errordoc>
        Error opening the backstore file for the shared memory.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 9 </errnumber>
  <errname> INCOMPATIBLE_VERSIONS </errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>
        Memory-file version mismatch.
      </errormsg>
      <errordoc>
        The version of the existing memory-file on disk does not match the
        version supported by the Photon server.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 10 </errnumber>
  <errname> BACKSTORE_FILE_MISSING </errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>
        The shared-memory backstore file is missing.
      </errormsg>
      <errordoc>
        The shared-memory backstore file is missing. The existence of this file
        was already verified previously - if it is missing, something
        really weird is going on.
        </errordoc>
      </message>
    </message_group>
  </error>

  <error>
    <errnumber> 11 </errnumber>
    <errname> ERROR_OPENING_MEMORY </errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>
          Generic i/o error when attempting to open the shared memory.
        </errormsg>
        <errordoc>
          Generic i/o error when attempting to open the shared memory.
        </errordoc>
      </message>
    </message_group>
  </error>

</errgroup>

<errgroup>
  <errgroup_ident xml:lang="en">
    <errgroup_name>XML config errors</errgroup_name>
    <errgroup_desc>
      This group of errors described potential errors with the code to read
      and parse the XML configuration file.
    </errgroup_desc>
  </errgroup_ident>

  <error>
    <errnumber> 100 </errnumber>
    <errname> XML_CONFIG_NOT_OPEN</errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>Error opening the XML config file.</errormsg>
        <errordoc>Error opening the XML configuration file.</errordoc>
        <errordoc>
          Most likely reason: the file does not exist but it might also be
          a problem with access permissions (on the file itself or one
          of its parent directories.
        </errordoc>
      </message>
    </message_group>
  </error>

  <error>
    <errnumber> 101 </errnumber>
    <errname> XML_READ_ERROR</errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>Error reading the XML config file.</errormsg>
      <errordoc>Error reading the XML configuration file.</errordoc>
      <errordoc>
        No validation is done at this point. Therefore the error is likely
        something like a missing end-tag or some other non-conformance to
        the XML’s syntax rules.
      </errordoc>
      <errordoc>
        A simple Google search for "well-formed xml" returns many web sites
        that describe the syntax rules for XML. You can also use the
        program xmllint (included in the distribution of libxm2) to
        pinpoint the issue.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 102 </errnumber>
  <errname> XML_NO_ROOT</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>No root element in the XML document.</errormsg>
      <errordoc>No root element in the XML document.</errordoc>
      <errordoc>
        This error is very unlikely - without a root element, the document
        cannot be parsed into a tree and you’ll get the error
        XML_READ_ERROR instead.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 103 </errnumber>
  <errname> XML_INVALID_ROOT</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>The root element is not the expected root.</errormsg>
      <errordoc>
        The root element is not the expected root, <quasar_config>.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 104 </errnumber>
  <errname> XML_NO_SCHEMA_LOCATION</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>The attribute xsi:schemaLocation was not found.</errormsg>
      <errordoc>
        The root element must have an attribute named schemaLocation (in
        the namespace "http://www.w3.org/2001/XMLSchema-instance") to
        point to the schema use for the server configuration file.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 105 </errnumber>
  <errname> XML_MALFORMED_SCHEMA_LOCATION</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>The attribute xsi:schemaLocation is not well formed.</errormsg>
      <errordoc>
        The attribute schemaLocation of the root element
        should contain both the namespace and the location of the
        schema file, separated by a space. You'll get this error if
        the white space is not found.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 106 </errnumber>
  <errname> XML_PARSER_CONTEXT_FAILED</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>The creation of a new schema parser context failed.</errormsg>
      <errordoc>
        The creation of a new schema parser context failed. There might
        be multiple reasons for this, for example, a memory-allocation
        failure in libxml2. However, the most likely reason is that the
        schema file is not at the location indicated by the attribute
        schemaLocation of the root element of the configuration file.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 107 </errnumber>
  <errname> XML_PARSE_SCHEMA_FAILED</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Failure when parsing the XML schema for config.</errormsg>
      <errordoc>
        The parse operation of the schema failed. Most likely, there
        is an error in the schema for the configuration. To debug this
        you can use xmllint (part of the libxml2 package).
      </errordoc>
    </message>
  </message_group>
```

```
</error>

<error>
  <errnumber> 108 </errnumber>
  <errname> XML_VALID_CONTEXT_FAILED</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>The creation of a new XML Schemas validation context failed.</errormsg>
      <errordoc>
        The creation of a new schema validation context failed. There might
        be multiple reasons for this, for example, a memory-allocation
        failure in libxml2.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 109 </errnumber>
  <errname> XML_VALIDATION_FAILED</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Document validation for the configuration file failed.</errormsg>
      <errordoc>
        Document validation for the configuration file failed. To debug this
        problem you can use xmllint (part of the libxml2 package).
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 110 </errnumber>
  <errname> CFG_PSO_LOCATION_TOO_LONG</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Config: mem_location is too long.</errormsg>
      <errordoc>
        The path associated with the tag <mem_location> is longer
        than PATH_MAX (the maximum length of a filename/pathname).
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 111 </errnumber>
  <errname> CFG_PSO_LOCATION_IS_MISSING</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Config: mem_location is missing.</errormsg>
      <errordoc>
        The tag <mem_location> is missing. This would normally
        be caught when the configuration is validated using the schema
        unless of course there is a bug in the code or the schema
        itself was modified.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 112 </errnumber>
  <errname> CFG_MEM_SIZE_IS_MISSING</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Config: mem_size is missing.</errormsg>
      <errordoc>
        The tag <mem_size> is missing. This would normally
        be caught when the configuration is validated using the schema
        unless of course there is a bug in the code or the schema
        itself was modified.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 113 </errnumber>
  <errname> CFG_SIZE_IS_MISSING</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Config: mem_size:size is missing.</errormsg>
      <errordoc>
        The attribute "size" of the tag <mem_size> is missing.
        This would normally be caught when the configuration is
        validated using the schema unless of course there is a bug
        in the code or the schema itself was modified.
      </errordoc>
    </message>
  </message_group>
</error>

<error>
  <errnumber> 114 </errnumber>
  <errname> CFG_UNITS_IS_MISSING</errname>
  <message_group>
    <message xml:lang="en">
      <errormsg>Config: mem_size:units is missing.</errormsg>
      <errordoc>
        The attribute "units" of the tag <mem_size> is missing.
        This would normally be caught when the configuration is
        validated using the schema unless of course there is a bug
        in the code or the schema itself was modified.
      </errordoc>
    </message>
    </message_group>
  </error>

  <error>
    <errnumber> 115 </errnumber>
    <errname> CFG_WATCHDOG_ADDRESS_IS_MISSING</errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>Config: quasar_address is missing.</errormsg>
        <errordoc>
          The tag <quasar_address> is missing. This would normally
          be caught when the configuration is validated using the schema
          unless of course there is a bug in the code or the schema
          itself was modified.
        </errordoc>
      </message>
    </message_group>
  </error>

  <error>
    <errnumber> 116 </errnumber>
    <errname> CFG_FILE_ACCESS_IS_MISSING</errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>Config: file_access is missing.</errormsg>
        <errordoc>
          The tag <file_access> is missing. This would normally
          be caught when the configuration is validated using the schema
          unless of course there is a bug in the code or the schema
          itself was modified.
        </errordoc>
      </message>
    </message_group>
  </error>

  <error>
    <errnumber> 117 </errnumber>
    <errname> CFG_ACCESS_IS_MISSING</errname>
    <message_group>
      <message xml:lang="en">
        <errormsg>Config: file_access:access is missing.</errormsg>
        <errordoc>
          The attribute "access" of the tag <file_access> is missing.
          This would normally be caught when the configuration is
          validated using the schema unless of course there is a bug
          in the code or the schema itself was modified.
        </errordoc>
      </message>
    </message_group>
  </error>

</errgroup>
```

```
</errorlist>
```

5.5. Example: python extended module

Adding the generated header file to a C module is very simple. All you have to do is to create the two dict objects using the provided function (AddErrors() or the name you chose in the option file) in the init function of your module (inityourmodule()).

Here’s a code snippet (you’ll have to modify it, evidently):

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

PyMODINIT_FUNC
inityourmodule(void)
{
   PyObject * m = NULL, * tup = NULL, * errs = NULL, * errNames = NULL;

   m = Py_InitModule3( "yourmodule",
                       yourmodule_methods,
                       "Example 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 yourmodule.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".
    */
   PyModule_AddObject( m, "errs", errs );
   /*
    * err_names allows you to retrieve the symbolic name of an error
    * from a returned error number (a_module.err_names[err_number]).
    * In some cases, it could be useful (possibly to generate a message
 * in your exception handler?).
 */
PyModule_AddObject( m, "err_names", errNames );

...
```