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 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 major update; the software was partially rewritten and completely reorganized (refactored) to make it easier to both maintain it and enhance it.

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) adopts the modular approach used in the redesign of the code - it is divided in sections, with one section for each type of output file.

Additionally, this new version resolves multiple small bugs and memory leaks.

Chapter 3. Installation

The installation procedure of the program 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). Whether this is needed depends on your exact situation.

A DTD can be identified using a PUBLIC clause at the start of an the xml file). For example:

```
<!DOCTYPE errorlist PUBLIC
   "-//Error Parser project//DTD Error Parser XML V1.3//EN"
    "http://photonsoftware.org/xml/ep/errorParser13.dtd">
```

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

Alternatively, the DTD can be identified with a SYSTEM clause:

```
<!DOCTYPE errorlist SYSTEM
   "/usr/local/share/xml/errorParser/errorParser13.dtd">
```

Using SYSTEM is perfectly correct in most cases. The main advantage of the PUBLIC clause is to make sure that multiple developers use the same DTD and not a locally modified version of 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. This script is 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.

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 Linux, for example, is the absence of a centralized method to manage 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.

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).

You can find binaries for libxml2 (built for Microsoft Windows) at this location (http://www.zlatkovic.com/libxml.en.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
```

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

Be careful as the code in the repository is not always stable (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 run without problems on most Unix systems. On Microsoft Windows however, things might be different. There are two ports for Windows but have not been tried as of 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 run it again if you wish 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/localshare/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, without modifying it, you can skip this section.

The documentation is written in DocBook xml. The source for the current document is doc/manual.xml. The file parser/errorParser.xml is used to generate the man page.

To create or refresh the man page, you will need to install the DocBook DTD itself and the docbook2x package.

> **Warning**
>
> The program docbook2x-man is named db2x_docbook2man on some systems, for example on Fedora 9.

If docbook2x-man is present on the system, **autoconf** (./configure) will detect it 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. For both html and pdf, 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**
>
> Be careful as the code in the repository is not always stable (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

To build the software, simply use the solution file parser\parser.snl (VC++ v8, known as VS 2005). Tests with the newer VS 2008 have yet to be conducted.

In addition, you will need to tell Visual Studio the location of the header files and libraries of the libxml2 package. Select the Options window under the Tools menu to add these 2 locations to the default lists used by VS 2005 (exact location: Projects and Solutions\VC++ Directories).

To run the tests, from a command shell run **cscript RunTests.vbs** (in the tests directory). You will need the program 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 by using nmake, the Microsoft version of make:

```
nmake -f Makefile.win32 target
```

The different targets available in Makefile.win32:

- all (the default target)

  Builds Error Parser.

- check

  Runs the test suite.

- checkdebug

  Runs the test suite using the debug version of Error Parser.

- docs

  Builds the documentation.

- clean

  Removes the executable, the object files and all intermediate files.

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 the program 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, without modifying it, you can skip this section.

The documentation is written in DocBook xml. The source for the current document is doc\manual.xml. The file parser/errorParser.xml is used to generate the man page.

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.

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.

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

  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 in version 1.0).

5.2. 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.

5.3. The xml option file

This section documents version 2.0 of the DTD. This new release of the parser will not understand previous versions of the DTD. Current option files must be upgraded.

- <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 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.

- <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 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 (pso: photonsoftware.org). 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().

5.4. 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.

5.4.1. 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.

5.4.2. 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    \
        ...
```

5.4.3. 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.

5.4.3.1. options.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE options PUBLIC
    "-//Error Parser project//DTD Error Parser Options XML V2.0//EN"
    "http://photonsoftware.org/xml/ep/errorParserOptions20.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="2.0">

  <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>
```

5.4.3.2. 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. -->
    <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>
```

5.4.4. 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://photonsoftware.org/xml/ep/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 (photonsoftware.org).
      </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>
```

5.4.5. 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:

```
# 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.