NAME

w_error_t − Shore error-handling

SYNOPSIS

```
#include <w.h>
#include <w_error.h>
class w_rc_t;
class w_error_t;
class w_error_info_t;
```

DESCRIPTION

These classes are used internally by

```
class w_rc_t;
```

This manual page is of interest only to writers of value-added servers, and then only those who need to generate their own sets of error codes.

ERROR CODES

Error codes are unsigned integers, returned from Shore methods in the form of a *w_rc_t* (see **rc(fc)**). Each error code has associated metadata, which consists of a descriptive string and a name (either by way of an enumeration, or by a C-preprocessor-defined name) (which could be considered meta-metadata, perhaps).

The integer values associated with error code names, the descriptive strings, the enumerations, and the *#define*s are generated by a Perl script.

Error codes are grouped into modules, so that all the error codes for a software module and their metadata are kept together. Each module is given a mask, which is folded into the values assigned to the errorcodes. This keeps the error codes for different software modules distinct.

The software that manages error codes keeps a (global) list of all the modules of error codes. Each software layer that uses the error codes must invoke a method to 'install' its module in the global list. The method is generated by the Perl script.

The data structure *w_error_info_t* stores the error codes and their associated metadata. This data structure is generated by the Perl script.

The data structure *w_error_t* holds an error code, line number and file name, and its instances can be linked to form stack traces. The application programmer does not directly manipulate them, however; the application programming interface for returning and interpreting errors is the class *w_rc_t*.

GENERATING SETS OF ERROR CODES

The Perl script in the Shore source tree, generates error codes from an input file that is best described with an example. The following example is taken from the Shore storage manager.

The script takes one of two mutually exclusive options, and a file name. One or the other of the options (-d, -e) is required:

```
$(SHORE_SOURCES)/tools/errors.pl -d <input-file>
// or
$(SHORE_SOURCES)/tools/errors.pl -e <input-file>
```

In the first case (-d) the named constants are generated as C preprocessor defined constants. The prefix of the generated names is capitalized and separated from the rest of the name by an underscore character (in concert with Paleozoic convention).

In the second case (-e) the named constants are generated as members of an anonymous enumeration. The prefix of the generated names is taken, case unchanged, from the input file.

```
e = 0x00080000 "Storage Manager" smlevel_0 {

ASSERT          Assertion failed
USERABORT       User initiated abort
}
```

The input is parsed as follows. On the first line:

e — A prefix used to generate the names of the constants for the error codes for this module. This prefix must not conflict with prefixes for other modules.

= — Separates the name prefix from the mask.

0x00080000 — This mask is added into each named constant generated for this module.

Storage Manager — The name of the module.

smlevel_0 — The name of a C++ class. If a class name is present, certain generated data structures and methods will be members of this class. If no class name appears here, the generated data structures will have global names.

{ — Begins the set of error codes descriptions for this module.

Blank lines may appear anywhere. Lines beginning with '#' are comments.

The next three lines define error codes:

ASSERT — This causes the named constant **eASSERT** to appear in an anonymous enumeration type. The value associated with eASSERT will contain the mask **0x00080000.**

```
enum {
       eASSERT               = 0x80000,
       ...
};
```

Assertion failed — This is the descriptive string associated with **eASSERT.** The Perl script generates an array of data structures that associates these descriptive strings with their corresponding integers.

} — Ends the set of error code descriptions for the module. More than one module may be described in a single input file.

GENERATED FILES

The Perl script generates a set of files. The names of the files have the prefix given on the first line of a module's input. In the above example, the output files are:

e_einfo.i — Contains two members of *smlevel_0.* First is the list of error codes with associated descriptive strings:

```
w_error_info_t smlevel_0 ::error_info[] = {
 { eASSERT            , "Assertion failed" },
 ...
};
```

 If a class name did not appear in the input file, the name of the data structure will be '_error_info' prepended with the name of the module. In this case, it would be

```
w_error_info_t e_error_info[] = {
 { eASSERT            , "Assertion failed" },
 ...
};
```

Second is the full definition of a method

```
void smlevel_0 ::init_errorcodes()
```

that your code can call to 'install' the module of error codes in a global list of all the error codes in the running program. This method is generated only if the input for the module contains a class name.

e_error.h — Contains two anonymous enumerations. The first contains all the constants for the error codes. The second contains the minimum and maximum error codes for this

module. Using the above example, we get:

```
enum {
        eASSERT                 = 0x80000,
        eUSERABORT              = 0x80001,
        ...
        eLOGICALIDOVERFLOW      = 0x80041,
        eTRANSITTIMEOUT         = 0x80042,
};

enum {
        eERRMIN = 0x80000,
        eERRMAX = 0x80042
};
```

e_error_def.h

An alternative to If the *-d* option were used on the Perl script, this output file would be generated and would look like this:

```
#define E_OK                    0
#define E_ASSERT                0x80000
#define E_USERABORT             0x80001
#define E_LOGICALIDOVERFLOW     0x80041
#define E_TRANSITTIMEOUT        0x80042
#define E_ERRMIN                 0x80000
#define E_ERRMAX                 0x80042
```

e_error.i This is an ancillary file that might not be needed. It is generated in case the set of descriptive strings is needed by software that is not privy to the class *smlevel_0.* It contains a simple array of the strings, an a constant indicating the length of the array:

```
static char *e_errmsg[] = {
        /* eASSERT              */ "Assertion failed",
        /* eUSERABORT           */ "User initiated abort",
        ...
};

const e_msg_size = 66;
```

Of course, the strings are in the proper order so the array can be indexed by the error code (after subtracting the module's mask).

INSTALLING SETS OF ERROR CODES

Your program must 'install' your modules by calling **init_errorcodes()** or, if you did not use a class name for your module, by explicitly installing the set:

```
    w_rc_t rc = w_error_t::insert(
                    "Storage Manager",   // name of the module
                    e_error_info,               // the error_info list
                    eERRMAX - eERRMIN + 1 // number of items in the
                                                  // error_info list.
        );
        ...
```

USING SETS OF ERROR CODES

If you want to write code that returns one of your error codes, you use the manifest constants to build *w_rc_t* structures, which can be printed directly. See **rc(fc)** for details.

SEE ALSO

rc(fc) and **intro(fc).**