MRO Delay Line

An exception-handling system for C software

D. F. Buscher
dfb@mrao.cam.ac.uk

rev 1.4

2002/06/10

Cavendish Laboratory
JJ Thomson Avenue
Cambridge CB3 0HE
UK

1 Introduction

The exception-handling utility functions in the utilLib library are intended to be a general-purpose method of handling exceptions in C code. The idea of exceptions is that a large part of most (good) software is actually not for doing the primary function of the code, but rather for catching errors that may creep in, either due to users entering incorrect parameters, or through programmer error. Any system for simplifying such error-catching code is therefore useful throughout a software system.

Languages such as C++ and Python have a built-in exception-handling system, but in C we need to provide this ourselves. As in most exception-handling systems, the idea is to allow the detection of errors anywhere in the code and for this error to propagate back through the function-call hierarchy to an appropriate level for handling the error. The handling level can then choose the best way to handle the error, e.g. print the message and/or halt the program.

The reason for this separation of error detection and error handling is that the function levels which can detect the error may well be in a general-purpose library which has no idea of the context in which it is being called, e.g. is this program a simple command-line program where a simple printf() of an error will suffice, a server program that needs to send the error back to a client, or a windowing program which has its own ways of displaying errors? Also, can the program recover from the error, or should we call abort()? These decisions can only be made at the higher levels, so propagating this information back to the higher levels is essential.

The way this is handled in the exception library is to put all the exception information into a variable, conventionally called status, which is passed as the last parameter of any function call (in most cases). This is similar to the scheme used by STARLINK (http://star-www.rl.ac.uk) and DRAMA (http://www.aao.gov.au/drama/html/DramaIndex.html) libraries, with one extension. Instead of being a scalar, status is a pointer to structure which contains multiple items of information about the exception, including an optional error message. Thus it combines the simple status system and the error reporting facilities in these systems into a single object.

2 Example use

A typical code fragment would look like

```
#include <exception.h>

main(){
   ExcStatus *status;
   ...

   status = ExcNew();
   ...

   FileProcess("myfile.txt", status);
   result = Func1(3.0, status);

   if (!ExcOK(status)){
       ExcPrint(status, stderr);
       ExcClear(status);
   }
   ...

}
```

```
double FileProcess(char *file, ExcStatus *status)
{
  FILE *fd;

  if (!ExcOK(status)) return;

  fd = fopen(file, "rb");
  if (fd == NULL) {
    ExcSignal(status);
    ExcExplain(status, "Unable to open %s\n", file);
    return;
  }

  ...

  return;
}

double
Func1(double arg1, ExcStatus *status){
   if (!ExcOK(status))return(0);

   return(2.6*Func2(arg1*3.0, status));
}

double
Func2(double arg1, ExcStatus *status){
   if (!ExcOK(status))return(0);

   if (arg1 <= 0.0){
      ExcSignal(status);
      ExcExplain(status, "Cannot take log of negative number %f\n", arg1);
      return(0);
   }
   return(log(arg1));
}
```

There are several points to note:

1. All functions check their status argument on entry to make sure an error did not occur earlier in the function call sequence. Usually if an error status is set, then the function returns without doing anything. This 'inherited status convention' means that a sequence of functions can be called without explicitly checking for an error till the end. All functions using the status variable should follow this convention where at all possible. Obvious exceptions are error-handling routines, which must execute when the status is set.

2. Only the top level needs to initialise the status variable. The preferred method is to use ExcNew() which malloc's the required storage for the status variable and initialises it. In situations where speed is of the essence (e.g. temporary use of status in an otherwise status-free function), it is also possible to declare a status variable (as opposed to a pointer to a status variable) as a local variable and initialise it using ExcInitialise().

3. The ExcSignal call sets a flag to say that an exception has occurred and the ExcExplain call stores a formatted error message (using printf-style formatting) in the status structure for later use.

4. The ExcPrint() function is a convenient way to print the error message to a file or

the terminal. To return the error message as a string for further processing, use the ExcExplanation() macro.

5. As implemented the exception library is thread-safe, as long as each thread has its own status variable. This is in contrast to the STARLINK and DRAMA error message handling systems, which require a lot of faffing about to get a thread-safe context.

3 Detailed function description

Below are descriptions of the most important functions and macros. A more detailed description may come along one day...

NAME

ExcNew – Create new exception structure.

SYNOPSIS

```
#include <exception.h>

ExcStatus *ExcStatusNew(void);
```

DESCRIPTION

Allocates a new exception structure and initialises it to a default (clear) state. Allocates storage for error messages (currently fixed at 1000 characters).

RETURN VALUE

A pointer to a new exception status structure. If memory allocation was unsuccessful, it returns a null pointer.

NAME

ExcInitialise – Initialise existing exception structure.

SYNOPSIS

```
#include <exception.h>

void ExcInitialise(ExcStatus *status, char *buffer, int length);
```

DESCRIPTION

Sets up an existing status variable to default values. If buffer is non-null, it is used as storage for up to length characters of error message. Without this storage, the status system will still operate, but no explanatory messages will be stored.

NOTES

Currently implemented as a macro.

NAME

ExcSignal, ExcClear, ExcExplain – Set and clear exception status

SYNOPSIS

```
#include <exception.h>

void ExcSignal(ExcStatus *status);
void ExcClear(ExcStatus *status);
void ExcExplain(ExcStatus *status, const char *fmt, ...)
```

DESCRIPTION

ExcSignal

Raise an exception. Sets the status variable to indicate an error. If status->debug is set, the abort() function is called to dump core, at which point a debugger can be used to examine the program state.

ExcClear

Clears the exception state and error message.

ExcExplain

Formats a printf(3)-style error message as indicated by fmt. This message is appended to a buffer which is part of the status object. If fmt is an empty string, the error message will contain the line number and file where the exception was raised. This function can be called multiple times to append multiple error messages.

NOTES

The function in which the exception is raised should typically return immediately to its calling function to allow the error message to be processed at a higher level.

ExcSignal is a macro which stores file and line number information as well as setting a flag. ExcClear is also a macro.

NAME

ExcOK, ExcSignalled, ExcPrint – Interrogate exception status

SYNOPSIS

```
#include <exception.h>

int ExcOK(ExcStatus *status);
int ExcSignalled(ExcStatus *status);
void ExcPrint(ExcStatus *status, FILE *file);
char *ExcExplanation(ExcStatus *status);
```

DESCRIPTION

ExcOK

Tests the status variable and returns non-zero if there has not been an exception since ExcClear() was last called.

ExcSignalled

The complement of ExcOK: tests the status variable and returns non-zero if there has been an exception since ExcClear() was last called.

ExcPrint

If there has been no exception, does nothing. Otherwise prints the stored error explanations to the output stream. If no error message storage was allocated, prints information about the file and line number where the error occurred.

ExcExplanation

Returns a null-terminated string containing the error messages which have been stored since ExcClear() was last called. Works even if no storage space has been allocated for error messages.

NOTES

ExcOK, ExcSignalled and ExcExplanation are implemented as macros for speed.