Exceptions Package

by Ross Williams
April 1993

Contents

Table of Contents

---

1 Overview

This overview section provides general information about this package.

Processing this file using FunnelWeb will result in the production of the specification and implementation files defined below. In addition a test program for testing this package will be generated. This FunnelWeb source file and the files that it generates are Copyright (C) Ross Williams 1993. See the configuration header notice below.

except.h[1] ≡
  {*Specification file*[18]
  }
This macro is attached to an output file.

except.c[2] ≡
  {*Implementation file*[31]
  }
This macro is attached to an output file.

ex_test.c[3] ≡
  {*Test program file*[74]
  }
This macro is attached to an output file.

The remainder of this overview section contains only descriptions. It does not contain any code.

1.1 Configuration Header Notice

The following notice applies to this FunnelWeb source file as well as to the FunnelWeb output files to which it is written.

Configuration header notice[4]**M** ≡
  {
```
/******************************************************************/
/* Package name  : Except.                                        */
/* Version       : 1.0                                            */
/* Completed     : April 1993.                                    */
/* Released      : 29 September 1993.                             */
/* First created : This package was first created in April 1993.  */
/* Summary       : This package provides Ada-like exceptions for C.  */
/* Components    : except.fw  - FunnelWeb source file.            */
/*                 except.h   - Exported header file.             */
/*                 except.c   - Implementation file.              */
/*                 ex_test.c  - C test program.                   */
/* Requires      : style.h, as.h, as.c.                           */
/* Author        : Ross N. Williams (ross@guest.adelaide.edu.au)  */
/*                 Rocksoft^tm Pty Ltd                            */
/*                 16 Lerwick Avenue, Hazelwood Park 5066, Australia. */
/* FTP Archive   : This file can be found in                      */
/*                 "ftp.adelaide.edu.au/pub/funnelweb/examples/"  */
/* Disclaimer    : This program is distributed WITHOUT ANY WARRANTY; */
/*                 without even the implied warranty of MERCHANTABILITY */
/*                 or FITNESS FOR A PARTICULAR PURPOSE.           */
/* Copyright     : Copyright (C) Ross Williams 1993.              */
```

```
/*                 However, permission is granted for anyone to copy,    */
/*                 modify, and distribute this work for any purpose,     */
/*                 commercial or non-commercial, so long as this notice  */
/*                 is included verbatim, and so long as all              */
/*                 modifications are recorded in the change log below.   */
/* Changes       : Please log any changes to this software either in the */
/*                 originating FunnelWeb source file, or, if you must,   */
/*                 in the C source files produced from the FunnelWeb     */
/*                 file.                                                 */
/* --<Start of Change Log>--                                             */
/* ??-Apr-93: RNW: Created this package.                                 */
/* 29-Sep-93: RNW: Released this package.                                */
/* --<End of Change Log>--                                               */
/*************************************************************************/
}
```

This macro is invoked in definitions 18, 31, and 74.

1.2 Abstract

This package provides an *exceptions* facility for C that is very similar to the exceptions facility in the Ada programming language. An exception is a control-flow event that causes control to be transferred up the stack (with the stack unwinding as it goes) until a *handler* for the particular exception invoked is found. The exception facility is totally under the client's control. The client defines a set of exceptions, and *raises* an exception event by calling `EX_RAISE`. Other functions defined by the client can catch the exceptions by name. The exception facility is useful for error handling and package interface definition.

1.3 Motivation

The motivation for creating this package was to provide a simpler, safer, and more sophisticated non-local jump facility than is provided by the standard ANSI `setjmp` (ANSI 7.6.1.1) and `longjmp` (7.6.2.1) primitives. In particular, the biggest problem with the ANSI functions is that they require that the thrower and catcher of the control event be sufficiently coordinated to share a single global variable containing the catcher's context. The requirement for this link is unacceptable because it precludes the use of non-local jumps as an element of the interface of packages having an abstract client (unless `jmpbuf`s are passed as parameters - yuk!). This lack of abstraction results in the following disadvantages:

* A global variable of type `jmpbuf` must be agreed upon between thrower and catcher for each potential throw. The result is a proliferation of global context variables, whose validity at any particular instant is often unclear.

* If the programmer wishes to set up a nest of handlers for a particular condition, the global `jmpbuf` buffer for the condition must be explicitly and carefully saved and restored by each function in the nest.

* The interface to `setjmp` and `longjmp` is a bit raw, requiring fiddling with integer values and embedding calls in conditional statements.

A better solution to the problem of non-local jumps is a system of exceptions.

1.4 What is an Exception?

An exception is essentially a non-local `goto` to the nearest dynamically enclosing target label in the stack of suspended functions. To execute an exception `goto`, one *raises* the exception, for example with a `raise` statement `raise sloth_exception;`. Control is then immediately transferred to the nearest dynamically enclosing handler (label) for `sloth_exception`.

In fact handlers are not simply labels, but block-structured constructs that define an execution zone within which the set of exceptions that they handle will be caught. For example, the following exception handling anonymous block might appear in a program written in the Ada programming language:

Example of an Ada exception handling block[5]**Z** ≡

```
{
 declare
    n : integer := 2*x+y;
 begin
    sloth_action(n);
    walrus_action(n*n);
 exception
    when sloth_exception    => PUT("The sloth    exception went off.");
    when aardvark_exception => PUT("The aardvark exception went off.");
 end
 }
```

This macro is NEVER invoked.

In the above example, the code between `declare` and `exception` is the code within which exceptions will be caught. The code between `exception` and `end` contains two exception handlers that will catch the exceptions `sloth_exception` and `aardvark_exception` should they be raised by the execution of the declarations or `sloth_action` or `walrus_action`. Any other exception that is raised within the controlled zone will simply propagate further up the call chain to the nearest handler that can handle it.

A most important aspect of the exceptions provided by Ada and this package is that an exception propagates to the nearest *dynamically* enclosing exception handler. This means that, apart from the shared knowledge of the definition of the exception, there is not necessarily any static relationship between the function that raises an exception and the function that catches it. This makes exceptions both dangerous and powerful. Dangerous because a programmer can code a `raise` statement with no knowledge of the function catching the exception. Powerful, because this very abstraction allows packages to raise exceptions that can be caught by abstract clients.

1.5 Exceptions as Active Statuses

Over the years, exceptions have received a lot of bad press and have generally been considered to be dangerous things. For example, Tony Hoare, in his Turing Award lecture of 1980, when covering the programming language Ada, remarked:

Gradually these objectives have been sacrificed in favour of power, supposedly achieved by a plethora of features and notational conventions, many of them unncessary and some of them, like exception handling, even dangerous. — Charles Antony Richard Hoare, Turing Award Speech, 1980.

Certainly exceptions can be dangerous things. However, I believe that, on balance, exceptions enhance reliability rather than deteriorating it. The reason for this belief is that exceptions *actively* indicate problems, whereas their alternative in this capacity, statuses, *passively* indicate problems.

To see this, consider the simple example case of a function that is supposed to write its argument character to its argument device. This is a straightforward operation, unless the device fails in some way, in which case this fact must be indicated to the caller. The traditional way of defining such a function is to have it return a status indicating whether the call succeeded.

Example of a function returning a status[6]**Z** ≡
```
{
EXPORT bool writdev P_((dev_t,char));
/* Returns TRUE upon success, FALSE if the write failed. */
}
```
This macro is NEVER invoked.

This results in calling code that should look something like this:

Example of call to function returning a status[7]**Z** ≡
```
{
if (!writdev(slothdev,'x'))
  bomb("Sloth device failed - aborting.\n");
}
```
This macro is NEVER invoked.

but often looks like this:

Example of call without checking to a function returning a status[8]**Z** ≡
```
{
writdev(slothdev,'x');
}
```
This macro is NEVER invoked.

Code such as this is extremely dangerous because a program written in this way might operate correctly for years, but then suddenly start behaving in odd ways when undetected errors occur. Unfortunately, it is extremely easy to write code such as this, as is easy to forget that a particular function even returns a status! Code such as this is quite common.

Now consider how `writdev` looks if it is defined using an exception to take care of the error case:

Example of a function that uses an exception[9]**Z** ≡
```
{
EXPORT void writdev P_((dev_t,char));
/* Raises writedev_ex exception if the write fails. */
}
```
This macro is NEVER invoked.

By casting the error condition in terms of an exception, the function *by default* ensures that something will happen if an error arises. In fact, one of two things must happen if the exception is ever raised: either the programmer codes a catcher for the exception and deals with it (in which case, it will probably be dealt with properly) or the exception propagates up out of `main()` where it will cause the exceptions package to initiate a controlled program bomb describing the unhandled exception. Both of these alternatives ensure that the condition is explicitly handled which is the important thing.

By using exceptions, the programmer can ensure that *by default* any error condition that arises will cause a program bomb. This is such a strong safety net that it actually *legitimizes* the no-error checking style of programming so criticized earlier. If you are in a hurry to write a program and do not wish to bother with error checking, the use of functions that raise exceptions upon errors ensures that if the code terminates normally, then it has terminated error-free — even though the programmer did not code any checking of return conditions.

Another advantage of exceptions is that they allow whole classes of errors that might arise in a span of code to be handled at one point. For example:

Example of single-point error handling[10]**Z** ≡

```
{
 EX_BEGIN
    f_file_t file1, file2;
    char line[1000];
    f_open(file1,read,"sloth.dat");
    f_open(file2,write,"walrus.dat");
    while (!f_eof(file1))
      {
       f_readline (file1,line,1000);
       f_writeline(file2,line,1000);
      }
    fclose(file1);
    fclose(file2);
 EX_FORGET
    EX_WHEN(f_open_ex ) goto f_handle;
    EX_WHEN(f_read_ex ) goto f_handle;
    EX_WHEN(f_write_ex) goto f_handle;
    EX_WHEN(f_close_ex)
    f_handle:
       f_delete(file2);
       printf("Error performing copy.");
 EX_END}
```

This macro is NEVER invoked.

In the above code, a single exception handler handles all the error conditions that might arise during all the file operations. The resultant code is far cleaner than the code that would be necessary if the status of each call had to be checked independently. Using exceptions, the normal and the exceptional cases can each be expressed clearly.

Summary: There is no doubt that, when misused, exceptions can be an extremely dangerous language construct. However, they also have the outstanding quality of enabling functions to be written that *actively* indicate errors, thus ensuring either that the client code does *something* about the error, or that the program will bomb with an unhandled exception error. In this capacity exceptions act as *active statuses*. In addition, exceptions allow error handling to be separated from ordinary execution. These two properties of exceptions mean that, in my opinion, they improve reliability on balance, and are a worthy programming tool.

1.6 How to Use This Package

Using this package is easy - just follow the following steps!

STEP 1: Define the exception. Before raising or catching an exception, you have to define it and make it visible to every entity that needs to refer to the exception. To define it, you need to think of an identifier that will be used to represent the exception, and you need to think of a short (one-line) description for the exception. Having done this, if you want the exception to be visible only within the current program file, insert a call to the macro `EX_LOCAL` in the variable section. For example:

Example exception definition[11]**Z** ≡

```
{
 EX_LOCAL(buf_full,"buf_full: Buffer is full exception.");
 }
```

This macro is NEVER invoked.

The first argument is the *exception identifier* and the second argument is the *exception description*. The `EX_LOCAL` macro translates into a string constant declaration having the name `buf_full` and the value "buf_full: Buff...". It is important that the exception identifer be unique within its scope as it is implemented by an ordinary constant. It is also very important that the description of the exception be unique within the set of exceptions in the program. If each exception has a unique description, then unhandled exceptions (see later) can be easily identified.

If you want the exception to be visible within all clients of the module declaring it, replace EX_LOCAL above with EX_EXPORT and insert a call to EX_EXCEPT in the header file for the module. EX_EXCEPT requires just the variable name.

Example exception declaration[12]**Z** ≡

```
{
 EX_EXCEPT(buf_full);
 }
```

This macro is NEVER invoked.

STEP 2: Raise the exception. Having completed step 1, you can now raise the exception in any module that can see the exception (i.e. the .c file with the EX_LOCAL or EX_EXPORT, and any .c file that includes the header file containing the EX_EXCEPT) simply be calling EX_RAISE. Note: EX_RAISE is syntactically equivalent to a compound statement.

Example of raising an exception[13]**Z** ≡

```
{
 /* If the buffer is full, raise the full buffer exception. */
 if (bufsiz > BUF_MAX)
    EX_RAISE(buf_ful);
 }
```

This macro is NEVER invoked.

The effect of a call to EX_RAISE is that control will leave the current function, and the stack will unwind until a function is encountered whose context contains an active exception handler construct that specifies a catcher for the exception that has been raised (in this case buf_ful. If no such handler exists, the exception package will bomb the program, printing out an error message stating that an exception has not been handled, and giving the description of the exception.

STEP 3: Catch the exception. Catching exceptions is a little tricker than raising them. To catch an exception, you need to specify so in advance by enclosing the code that might raise the exception within an *exception block* construct provided by the EX_ macros EX_BEGIN, EX_FORGET, EX_WHEN and EX_END. Here is an abstract example:

Exception block syntax[14]**Z** ≡

```
{
 EX_BEGIN
    <Code that might raise an exception>
 EX_FORGET
    EX_WHEN(<name of exception 1>)
       <Code to handle exception 1>
    EX_WHEN(<name of exception 2>)
       <Code to handle exception 2>
    EX_WHEN(<name of exception 3>)
       <Code to handle exception 3>
 EX_END
 }
```

This macro is NEVER invoked.

The entire context (from EX_BEGIN to EX_END) is syntactically equivalent to a compound statement ({}). Its semantics are that the code that might raise an exception is executed, and, if no exceptions are raised, terminates. If an exception is raised, the code aborts to the corresponding handler, or is propagated upwards if no matching handler is present. For a more detailed description of the semantics, see the specification section.

2 Static Parameters of this Package

This package has a few static (compile-time) parameters which can be used to modify its behavior.

2.1 Inline Switch Parameter

Exceptions are tricky constructs, particularly in a programming language like C, and it makes sense to incorporate lots of checking into a package such as this one. At the same time, exceptions are a fundamental language construct that are likely to be used extensively, and it should be possible to configure them to be as efficient as possible.

To satisfy these twin needs, this package provides a boolean parameter which can be used to select two different kinds of implementation. Set `_EX_FAST` to `FALSE` if you want the exported macros to generate function calls and to perform lots of checks. Set `_EX_FAST` to `TRUE` if you want the exported macros to generate inline code that runs as fast as possible with no checking at all. It is strongly recommended that this symbol be set to `FALSE` unless performance requirements absolutely demand that it be set to `TRUE`.

Symbol _EX_FAST[15] ≡

```
{
#define _EX_FAST FALSE
/* Set _EX_FAST to TRUE to turn on inlining and turn off checking. */
}
```

This macro is invoked in definition 32.

Note: Some other packages (such as the `link` package) allow parameters like this to be set by the client package, thus enabling different client modules to specify different safety/efficiency trade-offs. However, this level of parameterization granularity could not be achieved in this package as some of the checking is magic number checking which must be employed globally or not at all. The problem is that a module with checks on might complain about incorrect magic numbers that were not set by another module with checks off.

2.2 Threading Parameter

The parameter `_EX_THRD` tells the package whether it is operating within a multi-threaded environment (`TRUE`) (thus prohibiting single-instance global variables). If you set this to `TRUE`, you will also need to modify the macros for global variables in the next section.

Symbol _EX_THRD[16] ≡

```
{
#define _EX_THRD FALSE
/* To configure this package to work under a multi-threaded environment, */
/* set _EX_THRD to TRUE and modify the k_var calls later on in the file. */
}
```

This macro is invoked in definition 32.

2.3 Macros for Global Variables

This package uses three global variables — `_ex_curr`, `_ex_id`, and `_ex_info` — which root the exception context stack, and store information about the most recently raised exception. Unfortunately, global variables such as these cause problems on multi-threaded systems which require that separate instances of all global variables be maintained for each thread.

The solution to this problem is to define the global variables as usual, but to refer to them only through a set of lvalue-yielding macros. These macros can then be configured to point either directly to a single set of global variables (non-threaded system), or to thread-instance variables (threaded system). This is the approach that has been taken in this package. Accordingly the macros have been defined in this package static parameter section. Both unthreaded and threaded versions appear below.

Macros for global variables[17] ≡

```
{
 #if !_EX_THRD
 /* Non-threaded system macros simply point to variables. */
 #define _EX_CURR _ex_curr
 #define  EX_ID   ((p_ex_t) _ex_id)
 #define _EX_ID   _ex_id
 #define  EX_INFO _ex_info
 #else
 /* Threaded system macros translate to kernel calls.          */
 /* To make this package work in a multi-threaded environment, */
 /* set _EX_THRD to TRUE and modify these definitions.         */
 /* Note: The kernel should initialize _EX_CURR to NULL.       */
 #define _EX_CURR           (* ((_ex_cx_t *) k_var(51)))
 #define  EX_ID   ((p_ex_t) (* ((p_ex_t   *) k_var(52))))
 #define _EX_ID             (* ((p_ex_t   *) k_var(52)))
 #define  EX_INFO           (* ((ptrint   *) k_var(53)))
 #endif
 }
```

This macro is invoked in definition 32.

(The 51, 52 and so on are just example variable numbers. By the way, each of the three global variables will fit into a `ptrint`.).

The global variable `_ex_curr` is not supposed to be used by the client so its symbol starts with _. `EX_ID` and `EX_INFO` are intended to be visible to the client. However, the client is not allowed to write to `EX_ID` and so two forms have been provided, one hidden read/write form (`_EX_ID`) and one read-only form (`EX_ID`). The type cast in `EX_ID` is there solely to convert the value from an lvalue to an rvalue (ANSI 6.3.4 (footnote 44)) so that the client can't write to it.

If these macros are changed to point to thread instance variables, be sure to initialize the `_EX_CURR` variable to `NULL` (before performing any exception actions) if you want unhandled exceptions to be correctly detected. The other two variables don't need to be initialized.

Because these macros can translate to function calls, an attempt has been made in the rest of the package to minimize references to them (e.g. assign their values to temporary variables and manipulate them rather than using the macro itself multiple times).

3 Specification

Because this package exports lots of macros, its specification (`.h`) file contains lots of stuff that should really be in the implementation file out of sight of the client. Luckily, FunnelWeb allows most of these implementation details to be located in the implementation section of this FunnelWeb file, leaving this section to describe, as cleanly as possible, only those components in the specification file intended to be presented to the client.

In a nutshell, the package exports an exception type, and macros to raise and catch exceptions.

Specification file[18] ≡

```
{
 Configuration header notice[4]
 #ifndef _EX_DONE
 #define _EX_DONE 1
 Specification includes[19]
 Exception type[20]
 Hidden definitions[32]
 Macro EX_RAISE[22]
 Macros for constructing exception blocks[23]
 Function spec ex_str[30]
 Hidden function specs[33]
 #endif
 }
```

This macro is invoked in definition 1.

3.1 Specification Includes

The specification file requires just the following includes. The `style.h` include file contains basic C style stuff. The ANSI standard `setjmp.h` include file (ANSI Standard Section 7.6) is required because the `setjmp` and `longjmp` functions/macros are used to implement the package.

Specification includes[19] ≡

```
{
 #include "style.h"
 #include <setjmp.h>
 }
```

This macro is invoked in definition 18.

3.2 Exception Type

Under this package, each exception is represented by an explicit object that resides in memory. However, the package does not export this object type. Like many C packages that export abstract types, this package exports only a pointer-to-object type, the intent being that the client should only ever manipulate the abstract objects through pointers. Thus, this package exports a pointer-to-exception type `p_ex_t`, but does not export a corresponding exception type `ex_t`.

Most packages that export abstract objects by pointer type also export functions to dynamically create and destroy the abstract objects. However, in the case of exceptions there is no need for this. In fact there is a positive need to create exceptions at compile time and bind them statically (Dynamic creation would mean that an exception would have to be created before being raised, which would be awkward to coordinate at run time). So, instead of exporting a function to create an exception, this package exports macros that allow the client to statically create (i.e. at compile time) a new (hidden) exception object and, simultaneously, define a constant pointer (of type `p_ex_t`) that points to the new exception object, and to which an identifier of the client programmer's choosing can be bound. Here are the macros that do this.

Exception type[20] ≡
```
{
 Type p_ex_t[40]
 #define EX_EXCEPT(NAME)       Implementation of EX_EXCEPT[41]
 #define EX_EXPORT(NAME,DESC)  Implementation of EX_EXPORT[42]
 #define EX_LOCAL(NAME,DESC)   Implementation of EX_LOCAL[43]
 }
```
This macro is invoked in definition 18.

The macros `EX_EXPORT` and `EX_LOCAL` each create a brand new exception object. They can be called only at places in a module where variables can be declared. The argument `DESC` must be a one-line constant string that describes the exception; it will be written out if the exception is involved in some kind of error (e.g. if it propagates out of `main()` and has to be described to the user). The `NAME` argument must be a program identifier which will be the name of the pointer constant that will point to the new exception. Macros `EX_EXPORT` and `EX_LOCAL` are identical except that `EX_EXPORT` makes the constant pointer globally visible in the program (i.e. it is present in the symbol table at link time), whereas `EX_LOCAL` does not. If `EX_EXPORT` has been used, the constant pointer to the exception can be made visible in other modules by adding an instance of the `EX_EXCEPT` macro in the header file of the module containing the exception definition.

Here is an example of a global exception declaration:

Example exception declarations[21]**Z** ≡
```
{
 In sloth.h:
    EX_EXCEPT(sloth_exception);

 In sloth.c:
    EX_EXPORT(sloth_exception,"Sloth exception");
 }
```
This macro is NEVER invoked.

3.3 Macro EX_RAISE

The macro `EX_RAISE` raises its argument exception. The argument exception must be an expression evaluating to a pointer to the exception to be raised. See a later section for the precise semantics of an exception raise.

This macro is syntactically equivalent to a compound statement. It is guaranteed that its argument will be evaluated exactly once.

Macro EX_RAISE[22] ≡
```
{
 #if _EX_FAST
 #define EX_RAISE(P_EX) Fast implementation of EX_RAISE(P_EX)[45]
 #else
 #define EX_RAISE(P_EX) Safe implementation of EX_RAISE(P_EX)[44]
 #endif
 }
```
This macro is invoked in definition 18.

3.4 Exception Block Construct

The only way to catch one or more exceptions is to declare a construct that encloses the code that could raise the exceptions, along with the handler code. This construct is called an exception block and is implemented using a small set of preprocessor macros. Here they are:

Macros for constructing exception blocks[23] ≡

```
{
#if _EX_FAST
#define EX_BEGIN    \
Fast implementation of EX_BEGIN[52]
#define EX_FORGET      Fast implementation of EX_FORGET[53]
#define EX_WHEN(P_EX)  Fast implementation of EX_WHEN(P_EX)[54]
#define EX_OTHERS      Fast implementation of EX_OTHERS[55]
#define EX_END         Fast implementation of EX_END[56]
#define EX_POP         Fast implementation of EX_POP[57]
#else
#define EX_BEGIN    \
Safe implementation of EX_BEGIN[46]
#define EX_FORGET      Safe implementation of EX_FORGET[47]
#define EX_WHEN(P_EX)  Safe implementation of EX_WHEN(P_EX)[48]
#define EX_OTHERS      Safe implementation of EX_OTHERS[49]
#define EX_END         Safe implementation of EX_END[50]
#define EX_POP         Safe implementation of EX_POP[51]
#endif
}
```

This macro is invoked in definition 18.

These macros must be used in accordance with the following syntax:

Syntax of exception block[24]**Z** ≡

```
{
compound_statement = other_forms_of_compound_statement | exception_block
                   | EX_POP
exception_block = "EX_BEGIN" c_code "EX_FORGET" { handler } "EX_END"
handler         = "EX_WHEN(" expression ")" c_code
handler         = "EX_OTHERS"
}
```

This macro is NEVER invoked.

The argument to each instantiation of `EX_WHEN` must be an rvalue of type `p_ex_t`. It is guaranteed that this argument will be evaluated *at most* once.

The macro `EX_POP` should be invoked only from within the normal code of an exception block.

3.5 Definition of Semantics

This section contains a detailed English description of the semantics of the exception constructs provided by this package. Normally this sort of information would be laced throughout the package specification. However, in this package, there are so many special cases that it was thought best to collect all the tricky stuff in one place. The reader is advised to study this section thoroughly, as a lack of knowledge about the precise semantics of the exception construct can be very dangerous.

Basic semantics: When control hits an exception block, the code between `EX_BEGIN` and `EX_FORGET` (the *normal code*) is executed. If, during the execution of the normal code, no exceptions propagate up to this construct then, upon completion of the normal code, execution of the entire construct terminates. If, during the execution of the normal code, an exception is raised (in the code or any function it calls) that propagates up to this construct, control exits the normal code at that point and is transferred to the code associated with the handler for the exception raised (the *handler code*). If there is no handler for the exception raised, the exception is propagated to the next dynamically enclosing handler in the dynamic chain. If there is a handler for the target exception, the code associated with that handler is executed and the construct terminates. If an exception propagates out of `main()`, the exception package bombs the program with an error message explaining what has happened and printing out the description of the offending exception.

State of the function: You may be wondering why the middle keyword in the exception construct is `EX_FORGET`. The reason is that the ANSI standard (ANSI 7.6.2.1) states: *All accessible objects have values as of the time longjmp was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding setjmp macro that do not have volatile-qualified type and have been changed between the setjmp invocation and longjmp call are indeterminate.* These are strong words! Because this exception package is implemented using `setjmp` and `longjmp`, these words mean that following the raising of an exception, you cannot rely on the value of any **auto**matic variable that was modified within the normal code of an exception construct before the exception was raised. The `EX_FORGET` syntax has been designed to act as a continual reminder of this danger.

Direct exits from normal code: Because the exception package maintains an explicit stack of exception handling contexts, there are some restrictions about how control can leave such constructs. "Acceptable" ways of leaving the main code of an exception block are 1) reaching the end of the block, and 2) raising an exception (handled or unhandled). However, control can also exit the construct in four other "unacceptable" ways. These are 1) by executing a `goto` out of the block, 2) by executing a `return`, 3) by executing an explicit `longjmp`, 4) by executing `exit` or some other program terminating function (actually this case does not matter as much as the program ends anyway). To ensure that the exception package correctly maintains its stack of live exception blocks, each instance of any of the four unacceptable ways of exiting a block should be prededed by a call to `EX_POP`. Here is an example of a correct explicit exception block termination.

Example of a correct direct exit from an exception block[25]**Z** ≡

```
{
 EX_BEGIN
    add_char(buffer,'x');
    if (is_full(buffer)) {EX_POP; return TRUE;}
 EX_FORGET
    EX_WHEN(buf_ful)
       return TRUE;
 EX_END
 return FALSE;
 }
```

This macro is NEVER invoked.

`EX_POP` can only be called from within the normal code of an exception block.

No restrictions on exiting handler code: By the time control reaches any exception handler code, the current exception context has been popped, so there are no constraints at all on exits from handler code. In the example above, the `return TRUE` from the normal code must be preceeded by a call to `EX_POP`, but no such call is required for the `return TRUE` in the handler code. In fact, such a call would be erroneous. As soon as control hits handler code it is guaranteed that the current context has been popped.

Exception handler code can jump to other exception handlers: Just place a label inside one handler and a goto in another. This is useful where the code in one handler is suitable for many exceptions. For example:

Example of goto in exception handler[26]**Z** ≡

```
{
 EX_BEGIN
```

```
      add_char(buffer,'x');
   EX_FORGET
      EX_WHEN(misc_1) goto doit;
      EX_WHEN(misc_2) goto doit;
      EX_WHEN(misc_3) goto doit;
      EX_WHEN(misc_4) goto doit;
      EX_WHEN(misc_5)
        doit:
           printf("Miscellaneous exception happened.\n");
           return FALSE;
   EX_END
   }
```

This macro is NEVER invoked.

Because there are no restrictions on how handlers can exit, the `gotos` could have pointed outside the construct too.

Local variables: Each normal code and handler code segment is actually framed as a compound statement. This means that you can declare local variables if desired. For example:

Example of local variables in an exception block[27]**Z** ≡

```
  {
   EX_BEGIN
      int result;
      result=add_char(buffer,'x');
      if (result > 10) {EX_POP; return TRUE;}
   EX_FORGET
      EX_WHEN(buf_ful)
         string walrus = "An example string.\n";
         printf(walrus);
         return TRUE;
   EX_END
   return FALSE;
   }
```

This macro is NEVER invoked.

Exception blocks can be nested: Functions can contain many exception blocks. Exception blocks can even be nested. However, be very sure to obey the other rules. Here is an example:

Example of nested exception handlers[28]**Z** ≡

```
  {
   EX_BEGIN
      add_char(buffer,'x');
      EX_BEGIN
         big_func();
         EX_FORGET
            EX_WHEN(buf_ful) printf("big_func raised buf_ful.\n");
      EX_END
   EX_FORGET
      EX_WHEN(buf_ful) printf("add_char raised buf_ful.\n");
      EX_WHEN(sloth)   printf("either add_char or big_func raised sloth.\n");
   EX_END
   }
```

This macro is NEVER invoked.

OTHERS handler: The `EX_OTHERS` handler catches all exceptions.

Handler precedence: The first handler that matches an exception catches the exception. If more than one handler matches the same exception, the first handler will be activated.

Re-raising an exception: To re-raise an exception, simply code the statement `EX_RAISE(EX_ID);`.

3.6 Exception Information

When an exception is raised, two symbols provide information about the exception being raised. These symbols are macros and do not necessarily translate to variable names. However, it is guaranteed that `EX_ID` will be an rvalue (ANSI 6.2.2.1 (footnote 31)) of type `p_ex_t`, and that `EX_INFO` will behave like an modifiable lvalue (ANSI 6.2.2.1) of type `ptrint` (see the header file `style.h` for this type). The declarations of these macros appear in the static parameters section of this document.

Whenever an exception is raised, `EX_ID` contains the ID of the exception being raised. This value persists until the next exception is raised. This variable is only marginally useful in the current version of this package, as each exception handler will always statically "know" the exception that it has caught. In long and complex handlers though, the capacity for self-reference that `EX_ID` provides enables such handlers to be coded without hard-wiring the exception in question whenever it is re-raised. This allows exceptions to be more easily renamed. The macro is also useful because it allows exceptions to be dynamically re-raised from code to which many handlers have been pointed to with `goto`s. For example:

Example of use of EX_ID[29]**Z** ≡

```
{
 EX_BEGIN
    add_char(buffer,'x');
 EX_FORGET
    EX_WHEN(misc_1) goto doit;
    EX_WHEN(misc_2) goto doit;
    EX_WHEN(misc_3) goto doit;
    EX_WHEN(misc_4) goto doit;
    EX_WHEN(misc_5)
      doit:
         printf("Miscellaneous exception happened.\n");
         EX_RAISE(EX_ID);
 EX_END
 }
```

This macro is NEVER invoked.

`EX_ID` has been configured as an rvalue so that it cannot be written. Thus, it is guaranteed always to contain the ID of the exception most recently raised.

The variable `EX_INFO` is more under client control. It exists simply to act as an abstract channel for code raising exceptions to communicate with code catching exceptions. There are no rules for this value — it can be written to, or read from any piece of code at any time; make up your own rules. The type of `EX_INFO` has been set to `ptrint` so as to allow any pointer or integer to be conveyed through the variable. This effectively allows any amount of information to be passed.

3.7 Function ex_str

The function `ex_str` accepts a pointer to an exception and returns a pointer to a constant static string being the one line description of the exception that was provided when the exception was defined with `EX_LOCAL` or `EX_EXPORT`.

Function spec ex_str[30] ≡

```
{EXPORT string ex_str P_((p_ex_t));}
```

This macro is invoked in definition 18.

3.8 A Warning About Names

This package exports a number of compile-time and link-time symbols that pollute the symbol table and link table. This pollution is necessary because this package relies significantly on exported macros that reference exported variables. To reduce naming conflicts, nearly all exported names that are supposed to be visible to the client commence with `ex_` and all exported names that are supposed to be invisible to the client commence with `_ex`. Please be careful to avoid names with these prefixes.

3.9 Glossary

`Abstract client` — A client of this package about which no information is known.

`Catch` — When an exception is raised, it propagates up the run-time context stack until it encounters a corresponding handler at which point it is said to have been caught.

`Client` — Any code that uses this package.

`Exception` — A dynamic non-local control-flow event.

`Exception block` — A special kind of anonymous block (called a compound statement in C) containing exception handlers.

`Exception description` — A single-line string briefly describing the exception. This string is printed out whenever an exception has to be identified to the user.

`Exception identifier` — A program identifier that is bound to a constant whose value is a pointer pointing to an exception object.

`Exception handler` — An exception/code binding that appears at the end of an exception block and catches the specified exception should it propagate to that level.

`Direct exit` — The termination of an exception block in a way that bypasses the exception package (for example, by `return` or explicit `longjmp`). The macro `EX_POP` should be called before all such exits.

`Raise` — An exception is raised by an `EX_RAISE` statement. Once raised, an exception unwinds the stack until it encounters a handler for the specific exception.

4 Implementation

This section describes the implementation of this package. The implementation is short, but is rather messy because of the fast/safe alternatives, and because of the fact that much of the functionality of the package is provided in the `.h` file. Much of that functionality has been deferred using FunnelWeb macros to this section.

The implementation (`.c`) file itself is fairly simply, consisting only of some global variables and a few function definitions.

Implementation file[31] ≡
{
Configuration header notice[4]
`#include "except.h"`
`#include "as.h"`
Global variables (.c)[35]
Function ex_str[58]
Function ex_bomb[67]
Function _exrai[60]
Function _expop[69]
}
This macro is invoked in definition 2.

All the real action happens in the definitions in the specification (`.h`) file that are hidden from the client.

Hidden definitions[32] ≡
{
Symbol _EX_FAST[15]
Symbol _EX_THRD[16]
Exception block context type[34]
Global variables (.h)[36]
Macros for global variables[17]
}
This macro is invoked in definition 18.

Hidden function specs[33] ≡
{
Function spec _exrai[59]
Function spec _expop[68]
}
This macro is invoked in definition 18.

4.1 Overview and Data Structure Description

This section gives an overview of the implementation of this package.

This package operates by maintaining an explicit stack of currently active exception blocks. Whenever an exception block is entered (`EX_BEGIN`), a `setjmp` is executed and the resultant `jmp_buf` is inserted into a stack node and pushed onto the stack. Whenever an exception block is exited normally (`EX_END`), the stack node on the top of the stack is popped. When an exception is raised, the top of the stack is popped off and a `longjmp` is executed to its context. The context then searches its handlers for a handler for the exception raised. If it finds one, it executes it. If it doesn't, it pops and `longjmp`s the top of stack again. This goes on until the exception is caught, or the stack is empty.

The stack is composed of a singly-linked list of records of the following type:

Exception block context type[34] ≡

```
{
 typedef struct _ex_cxt_
   {
 Header magic number field[37]
    jmp_buf          _ex_jmbf;  /* Exception block context.          */
    struct _ex_cxt_ *_ex_prev;  /* Pointer to previous stack record. */
 Trailer magic number field[38]
   } _ex_cx_t;
 Magic number symbols[39]
 }
```

This macro is invoked in definition 32.

The sneaky part is that instead of storing this stack explicitly on the heap, it is stored on the run-time stack itself. This is done by having the `EX_BEGIN` macro create a compound statement containing a local automatic variable of type `_ex_cx_t` which forms a node of the stack. These nodes are threaded into a linked list that is rooted by the global variable `_ex_curr` which points to the node at the top of the stack, or `NULL` if the stack is empty. Two other hidden global variables `_ex_id` and `_ex_info` store the exception id and the exception information.

4.1.1 Global Variables

This package requires three global variables to keep track of everything. The variable `_ex_curr` points to the context record on the top of the stack, or `NULL` if the stack is empty. The variable `_ex_id` contains a pointer to the exception most recently raised. The variable `_ex_info` is provided for the use of the client.

To enable this package (and its client code) to be converted over to a multi-threaded system at a later date, these global variables are referred to only through a set of macros. The macros are described in the static parameters section of this document. They are called `_EX_CURR`, `EX_ID`, `_EX_ID`, `EX_INFO`.

Global variables (.c)[35] ≡

```
{
 #if !_EX_THRD
 GLOVAR _ex_cx_t *_ex_curr = NULL;
 GLOVAR p_ex_t    _ex_id;
 GLOVAR ptrint    _ex_info;
 #endif
 }
```

This macro is invoked in definition 31.

Here is the version of these variables that appears in the header file.

Global variables (.h)[36] ≡

```
{
 #if !_EX_THRD
 extern _ex_cx_t *_ex_curr;
 extern p_ex_t    _ex_id;
 extern ptrint    _ex_info;
 #endif
 }
```

This macro is invoked in definition 32.

4.1.2 Magic Numbers

In order to detect corruptions in exception context records, two magic number fields have been placed at the start and end of the record. In C, there are hundreds of ways for corruptions to occur, but in particular, in this package, if a client exits an exception context without first popping it, it is quite likely that the context will be "run over" by stack frames of functions called later. If the context is then invoked, the program will crash. The following magic number fields greatly improve the chances of detecting such corruptions.

Header magic number field[37] ≡
```
{#if !_EX_FAST
     ulong _ex_mag1;  /* Header magic number. */
 #endif}
```
This macro is invoked in definition 34.

Trailer magic number field[38] ≡
```
{#if !_EX_FAST
     ulong _ex_mag2;  /* Trailer magic number. */
 #endif}
```
This macro is invoked in definition 34.

Magic number symbols[39] ≡
```
{
 #define _EX_MAG1 0xFB8A5D30
 #define _EX_MAG2 0x09F2E7A2}
```
This macro is invoked in definition 34.

4.2 Exception type

Here is the definition of the pointer-to-exception type. Normally, a hidden exception type would be declared to which the pointer type would point, but as exceptions have only a single attribute, their description string, it makes sense simply to declare the pointer type to be `string` type (pointer to char).

Type p_ex_t[40] ≡
```
{typedef string p_ex_t;}
```
This macro is invoked in definition 20.

4.3 Macros for Declaring and Defining Exceptions

The three exception declaration macros make use of the fact that `p_ex_t` is really just a string pointer. There is a slight danger that a too-clever C compiler might overlap two strings for which one is a non-strict suffix of the other. However, this is considered unlikely.

Implementation of EX_EXCEPT[41] ≡
```
{extern const p_ex_t NAME}
```
This macro is invoked in definition 20.

Implementation of EX_EXPORT[42] ≡
```
{       const p_ex_t NAME = DESC}
```
This macro is invoked in definition 20.

Implementation of EX_LOCAL[43] ≡
```
{static const p_ex_t NAME = DESC}
```
This macro is invoked in definition 20.

4.4 Macro for Raising Exceptions

Two forms of the `EX_RAISE` macro are given. The first simply calls `_exrai` which performs lots of checks before raising the argument exception. The fast form does not perform any checking — it just performs the essential operations of setting the ID and executing the `longjmp`.

Safe implementation of EX_RAISE(P_EX)[44] ≡

```
{{_exrai(P_EX);}}
```

This macro is invoked in definition 22.

Fast implementation of EX_RAISE(P_EX)[45] ≡

```
{{_EX_ID = (P_EX); longjmp(_EX_CURR->_ex_jmbf,NON_ZERO);}}
```

This macro is invoked in definition 22.

4.5 Exception Block Macros (Safe)

The exception block macros are messy and deserve some explanation. These macros operate as a coherent whole, so it is probably best to read them through a few times to see how they interact, rather than attempting to understand each one independently. Two versions of the exception block macros are given — the safe version and the fast version. The safe version is described first.

The `EX_BEGIN` macro sets the stage for the entire exception block by starting a compound statement, declaring a local variable for the stack node (`_ex_cxlc`), initializing its magic checking numbers, pushing it onto the stack, and finally saving the current execution context in the `_ex_jmbf` field of the newly created stack node. This work has to be done in this macro as most of the operations require direct access to the local context. Not much checking can be done here either as we have no way of cross checking the value of `_EX_CURR` upon entry. The open `if` statement encloses the entire normal code of the exception block.

Safe implementation of EX_BEGIN[46] ≡

```
{{_ex_cx_t _ex_cxlc;                \
  _ex_cxlc._ex_mag1 = _EX_MAG1;     \
  _ex_cxlc._ex_mag2 = _EX_MAG2;     \
  _ex_cxlc._ex_prev = _EX_CURR;     \
  _EX_CURR          = &_ex_cxlc;    \
  if (!setjmp(_ex_cxlc._ex_jmbf)) { }
```

This macro is invoked in definition 23.

At the point of forgetting, we need to terminate the compound statement for the normal code, pop the current exception context and then test the currently live exception against the candidate handlers. This is most simply performed with a cascaded `if` statement. The `if` statement has to start with a `FALSE` branch so as to establish the syntactic context for the first `EX_WHEN` macro.

An earlier version of this package that used integers to represent exceptions used a `switch` statement instead of an `if` chain. A `switch` statement is actually preferable to an `if` chain because `switch`es prohibit duplicate labels. However, `switch`es had to be abandoned because when the package was changed over so that it used pointers as exception ids, it was noticed that K&R78 section 9.7 says that the argument to `switch` must be of type `int`. K&R88 (section A9.4) and ANSI (section 6.6.4.2) weaken this to include all integral types, but still preclude pointers. So an `if` chain had to be used.

Safe implementation of EX_FORGET[47] ≡

```
{;EX_POP} else {EX_POP; if (FALSE) {}
```

This macro is invoked in definition 23.

The start of each handler in the exception block is marked by a call to the `EX_WHEN` macro. All it does is add an extra clause onto the `if` chain.

Safe implementation of EX_WHEN(P_EX)[48] ≡
```
{} else if ((P_EX) == (EX_ID)) {}
```
This macro is invoked in definition 23.

The `EX_OTHERS` macro is also easily implemented (but, like `EX_FORGET`, could conceivably provoke some inconvenient warnings from `lint`.

Safe implementation of EX_OTHERS[49] ≡
```
{} else if (TRUE) {}
```
This macro is invoked in definition 23.

To end the construct, we terminate the current handler, re-raise the currently live exception if it has not already been caught, and terminate the `else` of `EX_FORGET` and the outer braces of the entire construct.

Safe implementation of EX_END[50] ≡
```
{} else _exrai(EX_ID); }}}
```
This macro is invoked in definition 23.

Popping the current exception context is easily achieved with a function call. Note: The reference to `_ex_cxlc` will simply not compile if there isn't a current context. This correctly precludes the use of this macro outside an exception block. It also guarantees that the argument to `_expop` will be a pointer to the context record of the innermost statically-enclosing exception block.

Safe implementation of EX_POP[51] ≡
```
{{_expop(&_ex_cxlc);}}
```
This macro is invoked in definition 23.

4.6 Fast, Inline Versions of the Exception Block Macros

Here are the fast, inline versions of the exported macros. This version of the macros operates in exactly the same way as the other version except that no checking is performed whatsoever. This version of the macros performs everything on the spot, without calling any support functions at all.

Fast implementation of EX_BEGIN[52] ≡
```
{{_ex_cx_t _ex_cxlc;               \
  _ex_cxlc._ex_prev = _EX_CURR;    \
  _EX_CURR          = &_ex_cxlc;   \
  if (!setjmp(_ex_cxlc._ex_jmbf)) { }
```
This macro is invoked in definition 23.

Fast implementation of EX_FORGET[53] ≡
```
{;EX_POP} else {EX_POP; if (FALSE) {}
```
This macro is invoked in definition 23.

Fast implementation of EX_WHEN(P_EX)[54] ≡
```
{} else if ((P_EX) == (EX_ID)) {}
```
This macro is invoked in definition 23.

Fast implementation of EX_OTHERS[55] ≡
```
{} else if (TRUE) {}
```
This macro is invoked in definition 23.

Fast implementation of EX_END[56] ≡
```
{} else longjmp(_EX_CURR->_ex_jmbf,NON_ZERO); }}}
```
This macro is invoked in definition 23.

Fast implementation of EX_POP[57] ≡
```
{{_EX_CURR = _EX_CURR->_ex_prev;}}
```
This macro is invoked in definition 23.

4.7 Function ex_str

The function `ex_str` is easily implemented. Because exceptions are just pointers to strings, all it has to do is return its argument!

Function ex_str[58] ≡

```
{
 EXPORT string ex_str (p_ex) p_ex_t p_ex; {return p_ex;}}
```

This macro is invoked in definition 31.

4.8 Support Functions

Although most of what this package exports is macros, some of the macros translate into calls to the support functions described in this section. Functions have been used for operations that are particularly long-winded.

4.8.1 Function _exrai

The function `_exrai` raises an exception. This function performs all the grunt work for the `EX_RAISE` macro, which merely translates to a call of this function. The function specification appears in the header file but is not supposed to be seen by the client.

Function spec _exrai[59] ≡

```
{EXPORT void _exrai P_((p_ex_t));}
```

This macro is invoked in definition 33.

Function _exrai[60] ≡

```
{
 EXPORT void _exrai (p_ex)
 p_ex_t p_ex;
 {
 #if _EX_FAST
  as_bomb("_exrai: This function should not be called with _EX_FAST==TRUE.");
 #else
  Take a copy of _EX_CURR in p_curr[61]
  Set the exception ID[62]
  Bomb if the stack is empty[63]
  Bomb if the target context is not on the stack[64]
  Bomb if the target context is corrupted[65]
  Raise the exception[66]
 #endif
 }
 }
```

This macro is invoked in definition 31.

This function refers to the value of `_EX_CURR` lots of times. However, `_EX_CURR` could be a function call. To avoid excessive use of this variable, we take a copy of its value in `p_curr`.

Take a copy of _EX_CURR in p_curr[61]**M** ≡

```
{_ex_cx_t * p_curr = _EX_CURR;}
```

This macro is invoked in definitions 60 and 69.

We set the global exception ID variable `_EX_ID` *before* performing all the sanity checks so that if one of them fails, the current exception can be printed out by `ex_bomb`.

Set the exception ID[62] ≡

```
{_EX_ID = p_ex;}
```

This macro is invoked in definition 60.

If the context stack is empty then there is no handler to catch the exception and we have to bomb the program.

Bomb if the stack is empty[63] ≡

```
{
 if (p_curr == NULL) {as_wl("_exrai: Unhandled exception."); ex_bomb();}}
```

This macro is invoked in definition 60.

If the target context is lower down in memory than the stack frame of *this* function (`_exrai`), then because stacks nearly always grow downwards, it is almost certain that a context was not popped when it should have been. Strictly speaking, this test is non-portable. However, in practice, stacks grow downward in the address space on nearly all machines, and it's such a powerful check, it seems worth the potential portability problems.

Bomb if the target context is not on the stack[64] ≡

```
{
 if (UWIDE(&p_curr) > UWIDE(&p_curr->_ex_jmbf))
   {
    as_wl("_exrai: Target exception context is no longer legitimate.");
    as_wl("        Exception context resides beneath the top of stack.");
    as_wl("        This means that earlier on, control must have left an");
    as_wl("        exception context without first popping its handler.");
    as_wl("        Look for jumps out of exception contexts that are not");
    as_wl("        immediately preceded by calls to EX_POP.");
    ex_bomb();
   }}
```

This macro is invoked in definition 60.

Check the target context for corruptions. There is no point doing a `longjmp` to a corrupted context!

Bomb if the target context is corrupted[65] ≡

```
{
 if ((p_curr->_ex_mag1 != _EX_MAG1) || (p_curr->_ex_mag2 != _EX_MAG2))
   {
    as_wl("_exrai: Target exception context has been corrupted. This could");
    as_wl("        be because an exception context wasn't popped, or it");
    as_wl("        could be just a common garden-variety C corruption :-)");
    ex_bomb();
   }}
```

This macro is invoked in definition 60.

Actually raising the exception is the easy part! All that is required is a `longjmp`.

Raise the exception[66] ≡

```
{longjmp(p_curr->_ex_jmbf,NON_ZERO);}
```

This macro is invoked in definition 60.

4.8.2 Function ex_bomb

The function `ex_bomb` writes out a description of the most recently raised exception and then bombs the program. This function is used by `_exrai`.

Function ex_bomb[67] ≡

```
{
 LOCAL void ex_bomb P_((void));
 LOCAL void ex_bomb ()
 {
  char s[100];
  as_wr("        Exception desc is : \""); as_wr(EX_ID); as_wl("\".");
  as_wr("        Exception id   is : ");
  sprintf(s,"%lu (= %lX)",ULONG(EX_ID),ULONG(EX_ID)); as_wl(s);
  as_wr("        Exception info is : ");
  sprintf(s,"%lu (= %lX)",ULONG(EX_INFO),ULONG(EX_INFO)); as_wl(s);
  as_bomb("Aborting program after exception error.");
 }
}
```

This macro is invoked in definition 31.

4.8.3 Function _expop

The function `_expop` pops the current exception block from the context stack. Before doing this, it performs a few checks. To assist it with its checks, the caller must pass a pointer to the current context record in the first parameter. This can always be done by the caller directly (without referencing `_EX_CURR`) because popping is only supposed to take place within the static context of an enclosing exception block.

Function spec _expop[68] ≡

```
{EXPORT void _expop P_((_ex_cx_t *));}
```

This macro is invoked in definition 33.

Function _expop[69] ≡

```
{
 EXPORT void _expop (p_check)
 _ex_cx_t *p_check;
 {
 #if _EX_FAST
  as_bomb("_exrai: This function should not be called with _EX_FAST==TRUE.");
 #else
  Take a copy of _EX_CURR in p_curr[61]
  Bomb if the current context is not the top of stack context[70]
  Bomb if current context is corrupted[71]
  Zap the current context[72]
  Pop the context[73]
 #endif
 }
}
```

This macro is invoked in definition 31.

The first check test is to make sure that the context about to be popped is the current context — the one statically enclosing the pop call.

Bomb if the current context is not the top of stack context[70] ≡

```
{
 if (p_curr != p_check)
   {
    if (p_curr == NULL)
       as_bomb("_expop: Context stack is empty.");
    else
       as_bomb("_expop: Top of context stack is not the current context.");
   }
 }
```

This macro is invoked in definition 69.

The second check tests the current context (the one about to be popped) to make sure that it isn't corrupted. If it is, then the program should be bombed.

Bomb if current context is corrupted[71] ≡

```
{
 if ((p_curr->_ex_mag1 != _EX_MAG1) || (p_curr->_ex_mag2 != _EX_MAG2))
   {
    as_wl("_ex_rai: Target exception context has been corrupted. This could");
    as_wl("         be because an exception context wasn't popped, or it");
    as_wl("         could be just a common garden-variety C corruption :-)");
    as_bomb("Aborting program after exception error.");
   }
 }
```

This macro is invoked in definition 69.

Before popping the current context, we need to zap its magic numbers so that they don't float around in stack memory where they might cause some future magic number test to succeed when it should have failed.

Zap the current context[72] ≡

```
{
 p_curr->_ex_mag1 = ~_EX_MAG1;
 p_curr->_ex_mag2 = ~_EX_MAG2;
 }
```

This macro is invoked in definition 69.

Having checked and zapped the context, popping it is easy!

Pop the context[73] ≡

```
{_EX_CURR = p_curr->_ex_prev;}
```

This macro is invoked in definition 69.

5 Test Program

This section provides a test program which can be used to test the exceptions package. The program provides nine success tests and nine failure tests. The success tests do not cause the program to terminate, and so can all be performed in a single run. However, each failure test causes the program to bomb with an assertion error, and so the failure tests must be performed separately as nine separate invocations of the test program. At the end of this testing, the user can have a high degree of confidence that the package is working.

Each of the eighteen tests is embodied in a separate function. The global boolean variable **flag** is a miscellaneous temporary variable which is usually used to tell if an exception has fired.

Test program file[74] ≡

```
{
 Configuration header notice[4]

 #include "style.h"
 #include "except.h"
 #include "as.h"

 EX_LOCAL(sloth_ex,"Sloth exception");
 EX_LOCAL(walrus_ex,"Walrus exception");

 GLOVAR bool flag;

 Function sc01[75]
 Function sc02[76]
 Function sc03[77]
 Function sc04[78]
 Function sc05[79]
 Function sc06[80]
 Function sc07[81]
 Function sc08[82]
 Function sc09[83]

 Function fa01[84]
 Function fa02[85]
 Function fa03[86]
 Function fa04[87]
 Function fa05[88]
 Function fa06[89]
 Function fa07[90]
 Function fa08[91]
 Function fa09[92]

 main()
 {
  char ch;

  printf("Test Program for the EXCEPT Package\n");
  printf("===================================\n");
  printf("This test program provides a number of different tests. Because\n");
  printf("some tests provoke the package to bomb on purpose, this test\n");
  printf("program must be run a number of times, once for each test.\n");
  printf("The 0 test performs a group of tests that are not supposed to\n");
```

```
printf("fail. The 1..9 tests each perform a test that is supposed to\n");
printf("bomb the program if the test succeeds. The 0 test should be run\n");
printf("with the package compiled with both _EX_FAST==FALSE and\n");
printf("_EX_FAST==TRUE. The 1..9 tests should use only _EX_FAST==FALSE.\n");
printf("\n");
if (_EX_FAST)
printf("Current value of EX_FAST == TRUE.\n");
else
printf("Current value of EX_FAST == FALSE.\n");
printf("\n");
printf("Enter 0 for success tests, or 1..9 for one of nine fail tests>");
ch=getchar(); printf("\n");
if (ch == '0')
  {
   printf("Success Tests\n");
   printf("-------------\n");
   printf("The following tests (sc01..sc09) test the normal features\n");
   printf("the exceptions package. You should see nine success lines\n");
   printf("appear on the screen. If the package bombs during this test,\n");
   printf("then something is wrong and should be fixed.\n");
   printf("\n");
   sc01();
   sc02();
   sc03();
   sc04();
   sc05();
   sc06();
   sc07();
   sc08();
   sc09();
   printf("\n");
   printf("All of the success tests SUCCEEDED.\n");
  }
else
  {
   if (_EX_FAST)
     {
      as_wl("Error in test configuration. An attempt was made to perform");
      as_wl("a 1..9 test with _EX_FAST==TRUE. This does not make sense as");
      as_wl("these tests test the error checking capability of the package");
      as_wl("and _EX_FAST=TRUE has all error checking turned off!");
      as_bomb("Please recompile with _EX_FAST==FALSE and try again.");
     }
   switch (ch)
    {
     case '1': fa01();
     case '2': fa02();
     case '3': fa03();
     case '4': fa04();
     case '5': fa05();
     case '6': fa06();
     case '7': fa07();
     case '8': fa08();
```

```
        case '9': fa09();
        default:
           printf("\nCharacter entered was not 0..9. Aborting...\n");
       }
      }
  }
  }
```
This macro is invoked in definition 3.

5.1 Function sc01

This test checks to make sure that exception blocks operate correctly when no exceptions are raised.

Function sc01[75] ≡

```
 {
  LOCAL void sc01 P_((void));
  LOCAL void sc01 ()
  {
   printf("Test SC01: Exception block with no exceptions raised.\n");

   /* The NULL exception block should be OK. */
   EX_BEGIN
   EX_FORGET
   EX_END

   /* Now put some code in the normal section. */
   flag=FALSE;
   EX_BEGIN
      flag=TRUE;
   EX_FORGET
   EX_END
   as_cold(flag,"sc01: First flag block test failed.");

   /* Make sure the EX_OTHERS branch only applies to exceptions. */
   flag=FALSE;
   EX_BEGIN
      flag=TRUE;
   EX_FORGET
      EX_OTHERS
         as_bomb("sc01: Raise failed.");
   EX_END
   as_cold(flag,"sc01: Second flag block test failed.");
  }
  }
```
This macro is invoked in definition 74.

5.2 Function sc02

This test tests to make sure that an exception raised within an exception block can be caught by that block.

Function sc02[76] ≡

```
{
 LOCAL void sc02 P_((void));
 LOCAL void sc02 ()
 {
  printf("Test SC02: Exception caught within immediate block.\n");
  flag = FALSE;
  EX_BEGIN
     EX_RAISE(sloth_ex);
     as_bomb("sc02: Raise failed.");
  EX_FORGET
     EX_WHEN(sloth_ex) flag=TRUE;
  EX_END
  as_cold(flag,"sc02: Exception was not caught by local block.");
 }
 }
```

This macro is invoked in definition 74.

5.3 Function sc03

This test tests the `EX_OTHERS` feature.

Function sc03[77] ≡

```
{
 LOCAL void sc03 P_((void));
 LOCAL void sc03 ()
 {
  printf("Test SC03: Exception is caught by OTHERS clause.\n");
  flag = FALSE;
  EX_BEGIN
     EX_RAISE(sloth_ex);
     as_bomb("sc03: Raise failed.");
  EX_FORGET
     EX_OTHERS flag=TRUE;
  EX_END
  as_cold(flag,"sc03: Exception was not caught by EX_OTHERS clause.");
 }
 }
```

This macro is invoked in definition 74.

5.4 Function sc04

This test tests nested exception blocks.

Function sc04[78] ≡

```
{
 LOCAL void sc04 P_((void));
 LOCAL void sc04 ()
 {
```

```
 printf("Test SC04: Exception is caught by nested exception block.\n");

 /* Try a simple nested handler. */
 flag=FALSE;
 EX_BEGIN
    EX_BEGIN
       EX_RAISE(sloth_ex);
       as_bomb("sc04: Raise failed (1).");
    EX_FORGET
       EX_WHEN(walrus_ex)
          as_bomb("sc04: Walrus exception caught (1).");
    EX_END
 EX_FORGET
    EX_WHEN(sloth_ex) flag=TRUE;
    EX_WHEN(walrus_ex) as_bomb("sc04: Walrus exception caught (x).");
 EX_END
 as_cold(flag,"sc04: First nested test failed.");

 /* Test a re-raise. */
 flag=FALSE;
 EX_BEGIN
    EX_BEGIN
       EX_RAISE(sloth_ex);
       as_bomb("sc04: Raise failed (2).");
    EX_FORGET
       EX_WHEN(sloth_ex)
          EX_RAISE(walrus_ex)
          as_bomb("sc04: Raise failed (3).");
       EX_WHEN(walrus_ex)
          as_bomb("sc04: Walrus exception caught (2).");
    EX_END
 EX_FORGET
    EX_WHEN(sloth_ex)
       as_bomb("sc04: Sloth exception escaped.");
    EX_WHEN(walrus_ex)
       flag=TRUE;
 EX_END
 as_cold(flag,"sc04: Second nested test failed.");
}
}
```
This macro is invoked in definition 74.

5.5 Function sc05

This test tests whether `EX_OTHERS` can successfully re-raise an exception.

Function sc05[79] ≡

```
{
LOCAL void sc05 P_((void));
LOCAL void sc05 ()
{
 STAVAR bool flag2;
 printf("Test SC05: Single OTHERS handler reraises current exception.\n");
```

```
 flag=FALSE;
 flag2=FALSE;
 EX_BEGIN
    EX_BEGIN
       EX_RAISE(sloth_ex);
       as_bomb("sc05: Raise failed (2).");
    EX_FORGET
       EX_OTHERS
          flag=TRUE;
          EX_RAISE(EX_ID);
          as_bomb("sc05: Raise failed (3).");
    EX_END
 EX_FORGET
    EX_WHEN(sloth_ex)
       flag2=TRUE;
    EX_OTHERS
       as_bomb("sc05: Exception escaped.");
 EX_END
 as_cold(flag ,"sc05: OTHERS test failed (1).");
 as_cold(flag2,"sc05: OTHERS test failed (2).");
}
}
```

This macro is invoked in definition 74.

5.6 Function sc06

Function sc06[80] ≡

```
{
LOCAL void sc06 P_((void));
LOCAL void sc06 ()
{
 printf("Test SC06: Handlers all GOTO same code.\n");
 flag=FALSE;
 EX_BEGIN
    EX_RAISE(sloth_ex);
    as_bomb("sc06: Raise failed.");
 EX_FORGET
    EX_WHEN(sloth_ex) goto handle;
    EX_WHEN(walrus_ex)
       as_bomb("sc06: Walrus exception went off.");
       handle:
       flag=TRUE;
    EX_OTHERS
       as_bomb("sc06: Exception escaped to OTHERS.");
 EX_END
 as_cold(flag,"sc06: GOTO test failed.");
}
}
```

This macro is invoked in definition 74.

5.7 Function sc07

This test simply checks that EX_INFO can be written to and read from.

Function sc07[81] ≡

```
{
 LOCAL void sc07 P_((void));
 LOCAL void sc07 ()
 {
  printf("Test SC07: Exercise EX_INFO.\n");
  EX_INFO = 3;
  as_cold(EX_INFO == 3,"sc07: EX_INFO test failed.");
 }
 }
```

This macro is invoked in definition 74.

5.8 Function sc08

This test simply checks that the `ex_str` function is working.

Function sc08[82] ≡

```
{
 LOCAL void sc08 P_((void));
 LOCAL void sc08 ()
 {
  printf("Test SC08: Exercise ex_str.\n");
  as_cold(ex_str(sloth_ex) == sloth_ex,"sc08: Failed.");
 }
 }
```

This macro is invoked in definition 74.

5.9 Function sc09

Test the legitimate use of `EX_POP`.

Function sc09[83] ≡

```
{
 LOCAL void sc09 P_((void));
 LOCAL void sc09 ()
 {
  printf("Test SC09: Test EX_POP.\n");
  flag=FALSE;
  EX_BEGIN
     EX_BEGIN
        EX_POP; goto finish;
     EX_FORGET
        EX_WHEN(sloth_ex) as_bomb("sc09: Inner catcher caught the sloth.");
     EX_END
     finish: EX_RAISE(sloth_ex);
  EX_FORGET
     EX_WHEN(sloth_ex) flag=TRUE;
  EX_END
  as_cold(flag,"sc09: Sloth exception failed to work.");
```

```
   as_cold(_EX_CURR == NULL,"sc09: EX_POP failed to pop correctly.");
  }
  }
```

This macro is invoked in definition 74.

5.10 Function fa01

This test tests that an unhandled exception will be properly caught.

Function fa01[84] ≡

```
{
 LOCAL void fa01 P_((void));
 LOCAL void fa01 ()
 {
  printf("Test FA01: Unhandled Exception\n");
  printf("------------------------------\n");
  printf("This test raises the sloth exception but doesn't handle it.\n");
  printf("The result should be an \"unhandled exception\" bomb specifying\n");
  printf("the sloth exception. Here we go...\n");
  printf("\n");
  EX_RAISE(sloth_ex);
  as_wl  ("FA01 FAILED: the exception package did not catch the\n");
  as_bomb("unhandled sloth exception.\n");
 }
 }
```

This macro is invoked in definition 74.

5.11 Function fa02

This test tests two calls to `EX_POP`.

Function fa02[85] ≡

```
{
 LOCAL void fa02 P_((void));
 LOCAL void fa02 ()
 {
  printf("Test FA02: Double EX_POP\n");
  printf("------------------------\n");
  printf("This test executes two calls to EX_POP within the normal code of\n");
  printf("an exception block. The second one should cause a\n");
  printf("\"context stack is empty\" bomb. Here we go...\n");
  printf("\n");
  EX_BEGIN
     EX_POP;
     EX_POP;
     as_bomb("FA02 FAILED: Second EX_POP did not cause bomb (1).");
  EX_FORGET
  EX_END
  as_bomb("FA02 FAILED: Second EX_POP did not cause bomb (2).");
 }
 }
```

This macro is invoked in definition 74.

5.12 Function fa03

This test tests a single call to EX_POP followed by a normal termination of the normal code.

Function fa03[86] ≡

```
{
 LOCAL void fa03 P_((void));
 LOCAL void fa03 ()
 {
  printf("Test FA03: EX_POP Followed by Normal Termination\n");
  printf("------------------------------------------------\n");
  printf("This test executes a call to EX_POP then falls through to the\n");
  printf("end of the normal code of the exception block. The result should\n");
  printf("be a \"context stack is empty\" bomb. Here we go...\n");
  printf("\n");
  EX_BEGIN
     EX_POP;
  EX_FORGET
  EX_END
  as_bomb("FA03 FAILED: Fallthough after EX_POP did not cause bomb.");
 }
 }
```

This macro is invoked in definition 74.

5.13 Function fa04

This test checks to make sure that a context can detect when it is popping the wrong context.

Function fa04[87] ≡

```
{
 LOCAL void fa04 P_((void));
 LOCAL void fa04 ()
 {
  printf("Test FA04: Exit a block without EX_POPing\n");
  printf("-----------------------------------------\n");
  printf("This test exits an exception block using a goto without first\n");
  printf("EX_POPing. The exception block is nested within another, and this\n");
  printf("test is to make sure that the outer block detects that it is\n");
  printf("popping the wrong context. The result should be a\n");
  printf("\"not current context\" bomb. Here we go...\n");
  printf("\n");
  EX_BEGIN
     EX_BEGIN
        goto there;
     EX_FORGET
     EX_END
     there:
  EX_FORGET
  EX_END
  as_bomb("FA04 FAILED: Outer block FAILED to detect stack misalignment.");
 }
 }
```

This macro is invoked in definition 74.

5.14 Function fa05

This test tests for the detection of an unpopped invalid stack context.

Function fa05[88] ≡

```
{
 LOCAL void fa05_x P_((void));
 LOCAL void fa05_x ()
 {
  EX_BEGIN
     return;
  EX_FORGET
  EX_END
 }

 LOCAL void fa05 P_((void));
 LOCAL void fa05 ()
 {
  printf("Test FA05: Target Context Does Not Exist\n");
  printf("----------------------------------------\n");
  printf("This test exits an exception block from within a function using\n");
  printf("a return statement. Thus, the context remains stacked even though\n");
  printf("the enclosing C block has terminated. This test checks to see\n");
  printf("whether this problem is detected next time an exception is raised.\n");
  printf("The result should be an illegitimate context error. Here we go...\n");
  printf("\n");
  EX_BEGIN
     fa05_x();
     EX_RAISE(sloth_ex);
     as_bomb("FA05 FAILED: Failed to raise sloth.");
  EX_FORGET
     EX_WHEN(sloth_ex)
        as_bomb("FA05 FAILED: Sloth exception was raised.");
  EX_END
  as_bomb("FA05 FAILED: Costruct terminated normally - it shouldn't have.");
 }
}
```

This macro is invoked in definition 74.

5.15 Function fa06

This test is identical to the previous one except that the test is made at the point of exit of the outer block, not at the point of raising an exception.

Function fa06[89] ≡

```
{
 LOCAL void fa06 P_((void));
 LOCAL void fa06 ()
 {
  printf("Test FA06: Target Context Does Not Exist (Fallthrough)\n");
  printf("------------------------------------------------------\n");
  printf("This test is identical to test FA05 except that it does not\n");
  printf("raise an exception in the outer block. The result should be a\n");
```

```
    printf("\"not current context\" error. Here we go...\n");
    printf("\n");
    EX_BEGIN
       fa05_x();
    EX_FORGET
    EX_END
    as_bomb("FA06 FAILED: Costruct terminated normally - it shouldn't have.");
   }
   }
```

This macro is invoked in definition 74.

5.16 Function fa07

Function fa07[90] ≡

```
  {
   LOCAL void fa07 P_((void));
   LOCAL void fa07 ()
   {
    as_bomb("Test FA07: Test FA06 is the last test.");
   }
   }
```

This macro is invoked in definition 74.

5.17 Function fa08

Function fa08[91] ≡

```
  {
   LOCAL void fa08 P_((void));
   LOCAL void fa08 ()
   {
    as_bomb("Test FA08: Test FA06 is the last test.");
   }
   }
```

This macro is invoked in definition 74.

5.18 Function fa09

Function fa09[92] ≡

```
  {
   LOCAL void fa09 P_((void));
   LOCAL void fa09 ()
   {
    as_bomb("Test FA09: Test FA06 is the last test.");
   }
   }
```

This macro is invoked in definition 74.

6 Design Notes

This section contains a detailed design discussion of particular aspects of this package. Inserting these notes into the implementation section would have cluttered it up too much.

6.1 Architectural Discussion

The basic engine for exceptions is provided by the ANSI standard `setjmp` and `longjmp` primitives which provide simple non-local jumps. The challenge is organizing them into an exception facility. To see the problems involved in doing this, we need only focus on an executing program at the time that an exception is raised. At this time, the exceptions package must locate the closest handler for the particular exception. From this, two things become immediately clear: 1) that the exceptions package is going to have to perform "work" whenever an exception block is encountered as well as whenever an exception is raised, 2) some sort of stacking of `jmp_buf` contexts is required.

This reasoning suggests two organizations:

1) Maintain a stack for each exception. Whenever an exception block is entered, push all the handlers for that block on their respective stacks. Pop them when exiting the handler. If an exception is raised, simply look on the top of its stack.

2) Maintain a stack of exception blocks. If an exception is raised, search each block in turn for a matching handler.

Of these two organizations, the second is by far the better. Exceptions are almost by definition rare events, and so in assessing the two organizations, the emphasis should be placed on the efficiency of the ordinary case which is the pushing and popping. The first alternative may require multiple pushes and pops per exception block, whereas the second requires just one. There are also a number of more practical reasons why the second alternative is better.

Having decided that a stack of exception blocks should be maintained, the next thing to decide is where it should be stored. The first thing that pops to mind is to organize for the exceptions package to maintain a stack in a linked list growing out of a global variable. While completely feasible, this solution is distasteful as it requires a memory allocation and deallocation per exception block instance. A far better scheme is simply to store the stack on the run-time stack! This can be done by storing each node of the stack in an automatic variable within the stack frame of the function containing the exception block. These automatic variables can then be threaded together with points to form the originally desired explicit stack.

6.2 Exception Identity

One interesting aspect of the design is the identity of exceptions. When the client raises an exception by calling `ex_raise`, an argument must be provided that in some way identifies the exception being raised. A number of possible schemes spring to mind:

Integer/preprocessor constant: Each exception could be mapped onto a unique integer which would represent the exception. These integers could be managed by defining preprocessor symbols for them. To raise an exception, the client would simply pass the preprocessor symbol. This solution is simple and provides compile-time checking (if the client programmer mistypes a number, the problem won't be discovered until the exception is raised and not caught, but if the client mistypes a preprocessor symbol, the compiler will issue an "unknown symbol" error), but it does not provide any way to ensure that two different exceptions in two different modules are not using the same number to represent the two different exceptions. There are organizational solutions to this problem such as having the programmer maintain a global list of exceptions, but these are generally unacceptable in the long run. One other solution is to make exception integers wide (e.g. 32-bits) and have the programmer choose the integer for each exception by tossing a coin 32 times. The probability of a collision would be tiny. To be sure that two exceptions were not using the same number, exceptions could be registered with the exceptions package which would check the uniqueness of each registered exception integer. Registration would also allow the exception package to bind a string name to each exception number.

Integer/variable: The first solution is clean because it uses preprocessor symbols. A slightly different solution is to use global variables instead. Under this scheme, a module exporting an exception exports a global integer variable that represents the exception. When the package starts up, it registers the exception with the exception package. Registration in this case is active, not passive; the exceptions package allocates an exception number which is assigned to the variable. This solution has the advantage of guaranteeing that exception numbers are unique. However, exceptions often happen under strange conditions and it is probably a disadvantage to have an exception represented by a writeable variable.

Pointer to record: An alternative to integers is to represent exceptions by pointers to records describing the exceptions. Each record could be declared statically (thus ensuring that each record has a unique pointer) and could contain a description of the exception and possibly other useful information (e.g. a count of how often the exception has fired). The pointers to the records could be constants.

Pointer to string: Pointers to records are OK, but are rather messy to declare. As it seems that the only information that really needs to be bound to an exception is a string describing the exception, a far better more direct scheme is simply to represent exceptions as pointers to strings. Each exception can be statically declared as a `const` pointer to a string. This solves all the problems: 1) the exception identity is read-only, 2) the exception identity is unique (because each string is stored at a different address), 3) the exception identity is easy to declare, 4) the exceptions package doesn't have to manage a table of exceptions, 5) the exception variable names are subject to compile-time checking. The only disadvantages are that 1) the compiler might overlap two constant strings if one is a non-strict suffix of the other, 2) the exception is represented by an object that is basically a variable which could be accidentally written-to if the program goes haywire before the exception is raised. The `const` attribute helps, but may be ignored by many compilers which will place the variable in writeable memory. These disadvantages are considered minor compared to the advantages.

Of the solutions given above, the simplest and cleanest is the pointer to string solution, and this is the scheme that is used in this package. This is a particularly satisfactory solution because it solves the problem of exception identity and exception description in one go.

6.3 Identifier Names

Because it contains macros that refer to global variables, this package exposes the major proportion of its names to the client. As such, particular care must be taken with names.

To assist portability to brain-dead compilers, all names have generally been restricted to six characters for external identifiers and eight for internal identifiers. Upper case has been used for preprocessor symbols and lower case for other identifiers. Although the use of upper case in the main exception syntax is a little loud, it is also a continuous reminder that this is a dangerous construct.

The menemonic for this package is `ex` and this appears at the beginning of every exported symbol. This will assist to reduce symbol collisions. In addition, any symbol that is not intended to be seen by the client has been prepended with an underscore. This is a little bit naughty as it contravenes the ANSI standard's rule 7.13 on identifiers reserved for libraries ("*All identifiers that begin with an underscore are always reserved for use as identifiers with file scrope in both the ordinary identifier and tag name spaces*"), but as this package is supposed to be a library package too, it seems only a minor breach, and completely in keeping with the style of the ANSI library.

The names of all the exported constructs were chosen very carefully. Because most of the exported items were macros, by universal convention, their identifiers had to be in upper case. This requirement was not a disadvantage though as exception handlers are so important that they ought to be syntactically "loud". Prefixing each symbol with `EX_` is a little verbose, but was worthwhile because it immediately binds the symbol so tightly to this package that generic suffixes such as `BEGIN` can be used as a suffix. Thus `EX_BEGIN` and EX_END strongly convey their identity as the boundaries of an anonymous block, but also strongly indicate that the block is designed to handle exceptions.

As mentioned earlier, the identifier EX_FORGET was chosen so as to continually remind the programmer that, in the event of an exception, all automatic variables defined within the function containing the exception block and which have been written since the EX_BEGIN of the block, will be undefined. This is such an important semantic point that it was considered worthy of being embodied in the syntax too.

The hardest identifier to choose was EX_RAISE. For a long time, this construct was simply to take the form of an exported function with the name ex_rai. However, in the end, it was decided that it would be better to give this facility the same syntactic feel as the other constructs. In addition, the raising of an exception is an event that should probably be syntactically loud. Finally EX_RAISE is more readable than ex_rai.

6.4 Catering for Multi-Threading

An increasing amount of code is being run in a multi-threading environment and it is important to ensure that all library packages (such as this one) are compatible with multi-threading. A key requirement of multi-threading is that there be no global variables, ar at least that all declarations and uses of the global variables that do exist have been encapsulated in macros that can be redefined to work with multi-threading. In this package, the only global variables are _ex_curr, _ex_id, and _ex_info. To simplify possible future conversion for use with a multi-threading package, all uses of these variables have been expressed in terms of the macros _EX_CURR, _EX_ID, EX_ID, and EX_INFO.

6.5 Checking

As mentioned before, exceptions are slippery things that can easily go wrong particularly when the exception facility is hacked together, as it has here, out of macros and longjmps. This section contains a list of likely errors and failures and explains how the package copes with them. The errors listed here are caught only in the checking version of this package. If _EX_FAST then no checking is performed.

Exception has no handler: This simple case is caught by the _exrai function which will detect a NULL pointer in _EX_CURR if there are no more exception blocks to search for handlers.

Client pops the context too often: This case will occur if the client calls EX_POP and then exits the exception block normally, or if the client calls EX_POP more than once within the same exception block. Both these errors are detected by the _expop function which tests to ensure that the context it is being asked to pop is the context enclosing the EX_POP call. (Note: EX_POP won't compile unless it is called from within an exception block, so there is no chance of a context being popped from an external function).

Client exits context without popping it: This is the single biggest headache, as there is no way that this error can be detected when it occurs. There is nothing to stop the client from executing a return or explicit longjmp from within an exception block, and if the client does, the exception block will still be registered as the top context on the stack of exception contexts, and control will be transferred to it if an exception goes off, and if this happens, the program is likely to crash, as the stack frame containing the exception block no longer exists.

This problem cannot be solved, but it can be attenuated somewhat using the three-pronged approach adopted in this package:

1) Whenever a context is popped, a check is performed to ensure that the context being popped is the statically enclosing exception block. If it isn't, the program bombs. This check means that the stack will be properly aligned following each non-illegal termination of each exception block.

2) Whenever an exception is raised, a check is performed to ensure that the address of the context on the top of the context stack is lower than that of the start of the stack frame of the function raising the exception. If it is, then it must be an erroneously unpopped context.

3) Corruptions of context records are detected by having two magic number fields in the record, one at the start of the record and one at the end. If a record is corrupted, the corruption is likely to be detected the next time the record is manipulated.

The last two checks do a good job of covering the two main cases that can arise. When an exception block terminates without its context being popped, control will often leave the function. If it does, then the top of the run-time stack will be higher (in address value for stacks that grow down) than the unpopped context. This is detected by the second test. If, later, the stack grows again, it is likely to overwrite the unpopped context, including the magic numbers it contains. If this happens, then the error will be detected the next time an exception is raised or the stack is popped.

Obviously, there are ways in which a computer program might satisfy all these tests and still be erroneous, but at least these tests provide some sort of safety net.

6.6 Future Improvements

This section gives a list of possible future improvements to this package.

* It might be useful to allow the client to specify a block of code that is executed after an exception is raised, but before the handlers are scanned. This would allow an opportunity for cleaning up.

* Rework the terminology so that it is more consistent.

* Hierarchical exceptions.

That's the end of this package!