A Portable Exception Handling Mechanism for C++

Mary Fontana
LaMott Oren

Texas Instruments Incorporated
Computer Science Center
Dallas, TX 75265
fontana@csc.ti.com
oren@csc.ti.com

Martin Neath

Texas Instruments Incorporated
Information Technology Group
Austin, TX 78714
neath@itg.ti.com

ABSTRACT

The Texas Instruments C++ Object-Oriented Library (COOL) is a collection of classes, templates, and macros for use by C++ programmers who need to write complex yet portable applications. The COOL exception handling mechanism is one component of this library that substantially improves the development and expressive capabilities available to programmers by allowing them to control the action to be taken by their programs when exceptional or unexpected situations occur. This paper describes the facilities provided by COOL to raise, handle, and proceed from exceptions in a C++ class library or program and provides several examples of its use and flexibility.

1. Introduction

The Texas Instruments C++ Object-Oriented Library (COOL) is a collection of classes, templates, and macros for use by C++ programmers writing complex applications. An important feature of this library is the ability to create and raise exceptions in the library for which a user of this class library can define his/her own exception handler routines to effectively and appropriately handle the exceptions in an application at runtime. This is especially important since current C++ compiler implementations do not contain an exception handling facility [6]. For an overview of the COOL class library, see the paper, *COOL - A C++ Object-Oriented Library* [2]. For complete details, see the reference document, *COOL User's Guide* [8].

The COOL exception handling facility consists of a set of classes and macros to define, create, raise and handle exceptions. It is implemented with the COOL macro facility [3] and symbolic computing capability [4] and is similar to the Common Lisp Condition Handling system [1] in that both implement an exception system with the ability for resumption. Briefly, when a program encounters a particular or unusual situation that is often (but not necessarily) an error, it can (1) represent the situation in an exception object, (2) announce that the situation has occurred by raising the exception, (3) provide ways to deal with the situation by defining and establishing handlers, and (4) continue or proceed from the situation after invoking a handler. One possible action for a handler is to correct some piece of erroneous information and retry the operation.

The COOL exception handling facility represents exceptions as objects derived from an exception class and provides a generic exception handler class, a set of predefined subclasses of the base exception class, and a set of default exception handler functions. Several macros -- **EXCEPTION**, **RAISE**, **STOP**, and **VERIFY** -- provide

a simple interface to the exception handling facility and allow a programmer to easily create and raise exceptions. In addition, the macro **IGNORE_ERRORS** provides a convenient means by which a programmer can explicitly disable the exception handling system while executing a block of statements. Finally, the **DO_WITH_HANDLER** and **HANDLER_CASE** macros offer functionality similar to that described by Koenig and Stroustrup [5] to associate a block of statements with a provision for handling a specific set of possible exceptions.

Exception handlers are also represented as objects derived from an exception handler class. When an exception handler object is instantiated, it is placed at the top of a global exception handler stack. When an exception handler object is destroyed, the handler is removed from the exception handler stack. This stack is maintained in a similar way to that described by Miller [7]. When an exception is raised, a search is performed for an appropriate handler starting at the top of the exception handler stack. Since the most recently defined handler objects are placed at the top of the stack, localized or specialized handlers take precedence over more generic system-wide handlers. When a match against an exception type or group name is found, the exception handler object invokes the handler function. An exception handler function might report the exception to the standard error stream and terminate the program, generate debug information by dumping a core image or stack trace to disk, or attempt to fix the problem and retry the operation again.

2. The Exception Class

Exceptions are represented as objects of an exception class and are used as a means of saving and communicating the state information representing a particular problem or condition to the appropriate exception handler. When an exception can be corrected and resumption is possible, information on the new state is stored in the exception object by the invoked exception handler function and returned to the point at which the exception was raised. The COOL **Exception** class is the base class from which specialized exception classes are derived. It contains data members to store a message string prefix, a format control string, a list of one or more group names or exception aliases for which this exception class is appropriate, and a flag to indicate whether or not an exception is handled. User-derived exception classes can include additional data members for saving the incorrect values detected by a program. These values report the state to the exception handler and are often used by an exception handler when reporting the exception/error message to some interactive stream.

The **Exception** class has several generic member functions for use by all exception objects. The virtual member function **report()** uses the message prefix and format string data members to report an exception message on a specified stream. The virtual member function **raise()** searches for an exception handler and invokes it if found. The **match()** member function indicates if an exception object is included in one of the exception group names (aliases). The output operator is overloaded for the **Exception** class to call the **report()** member function. Member functions to query whether or not an exception has been handled and to set the handled flag are also available. Finally, the virtual **default_handler()** member function is invoked if no exception handler is found.

As mentioned above, data members can be included in a derived exception class as a way for the signaller (the one who raises the exception) to indicate to an exception handler ways of proceeding from the exception. For example, if an exception occurs because a variable has an incorrect value, an exception object of the appropriate type is created and then the exception is raised. The exception object defined for this situation would have a data member with the incorrect value and a data member for the new value. An exception handler could be established to handle this type of situation by supplying a new value (possibly by interactively informing the user about the incorrect value and querying for a new value through an appropriate interface), store this new value in the exception object, and return the exception object to the signaller. The signaller would then assign this new value to the variable in error and execution at the point the exception was raised would resume.

3. Exception Handler Class

When an exception handler is invoked after a successful search of the global exception handler stack, it's handler function is called with the raised exception object as an argument. The handler function may correct the problem, ignore it, or do most anything else appropriate. For the case in which an attempt is made to fix the problem and resumption is possible, the point in the program or library at which the exception is raised can

contain statements to determine the new or changed values and state information, update any local variables accordingly, and resume execution. All the information and processing associated with exception handling is represented by an instance of an exception handler class.

The **Excp_Handler** class has two data members. The first is a list of one or more exception types or group names (aliases) and the second is a pointer to a function to be called to handle a raised exception that matches against a value in the exception type list. These data members are initialized by the argument list of the constructor and cannot be changed once set. The **Excp_Handler** class has a single virtual member function **invoke_handler()** that takes a single argument -- a pointer to the exception object -- and invokes the exception handler function. This function may or may not return, depending upon whether the handler attempts to resume execution or terminate the operation.

4. Exception Group Names (Aliases)

As with most exception handling systems, the COOL exception facility supports the grouping of exceptions by the class hierarchy. However, as mentioned above, the COOL exception handling facility also supports the concept of exception group names or aliases. Grouping of exception names is implemented through the alias/group name data member in each exception object. These group names allow a programmer to raise a single exception but associate that exception with several names or aliases rather than with just one. This means that a single exception class might be handled by one of several different exception handlers appropriate under different situations. The net result for the programmer is that only one exception class needs to be defined instead of several very similar classes. The group names are implemented using the COOL symbolic computing facility [4] for efficiency, but could be implemented using simple character strings to represent each name.

For example, suppose a programmer is implementing a parameterized Vector<Type> class in a generic class library to be used by several other programmers in the company. Some of these other programmers want to have a detailed set of options for dealing with exceptions, including resumption, while others want only a simple fail-safe termination mechanism. The Vector class programmer wishes to provide exception handling in the overloaded Vector<Type>::operator[] member function that satisfies all potential users of the class. To accomplish this, a single exception class **Out_Of_Bounds** is derived from the base class **Exception** with appropriate data members added to contain the old index value and a possible new value. If an index out of bounds error is detected, an exception object is created with one type name provided by the class hierarchy mechanism -- **Out_Of_Bounds** -- and two group names -- **Vector_Error** and **Fatal_If_Not_Handled** -- representing different exception reporting granularity. These three names for one exception type allow three different users of this class to achieve varying levels of sophistication in their exception handlers while requiring the class programmer to only implement one exception class for the parameterized vector class. If an Out_Of_Bounds exception is raised, the first exception handler found on the global exception handler stack that can handle either the exception type or one of the three exception group names supported will be invoked.

5. Predefined Exception Types and Handlers

COOL provides seven predefined exception class and five default exception handlers. The **Exception** class is the base exception class from which all other exception classes are derived. The **Warning**, **Fatal**, **Error**, and **System_Signal** classes are immediately derived from the base class. The **System_Error** and **Verify_Error** classes are derived from the **Error** class. Each of these predefined exception types has a default report member function and a default exception handler member function that is only invoked if no other exception handler is established by the programmer and found on the global exception handler stack when an exception is raised.

For exceptions of type **Error** and **Fatal**, the default exception handler reports the error message on the standard error stream and terminates the program with **exit()** or writes a core image and/or stack trace out to disk with **abort()**. If the exception is of type **Warning**, the warning message is reported on the standard error stream and the program resumes at the point at which the exception was raised. If the exception is of type **System_Error**, the system error message is reported on the standard error stream and the program is terminated. If the exception is of type **System_Signal**, the signal is reported and the program resumes execution after the call of the system function **signal(2)**. In all cases, the default exception handler is merely a place-holder to insure that there is

at least one handler for each type of exception available at all times on the global exception handler stack. Users are expected to provide their own exception handler classes that handle particular exception group names used within the COOL class library in a more appropriate application-specific manner.

6. Exception Handling Macros

The COOL exception handling facility uses the COOL macro facility [3] to create macros for creating, raising, and manipulating exceptions. The **EXCEPTION** macro simplifies the process of creating an instance of a particular type of exception object. The **RAISE** macro allows the programmer to easily construct and raise an exception, then perform a search for an appropriate exception handler and invoke the handler function. The **STOP** macro is similar to **RAISE**, except that it guarantees to end the program if the exception is not handled. The **VERIFY** macro raises an exception if an assertion for some specified expression evaluates to zero. The **IGNORE_ERRORS** macro is a wrapper that can be placed around a body of statements to ignore exceptions raised while executing that body. The **DO_WITH_HANDLER** macro establish an exception handler effective for the duration of a block. Finally, the **HANDLER_CASE** macro establishes an exception handler that transfers control to a series of exception-case clauses similar to the **try**/**catch** concept proposed by Koenig/Stroustrup [5].

6.1. The EXCEPTION Macro

The **EXCEPTION** macro simplifies the process of creating an instance of a particular type of exception object. It provides an interface for the application programmer to create an exception object using the specified arguments to indicate one or more group names, initialize any data members, or generate a format message. **EXCEPTION** is implemented as a COOL macro and has the following syntax:

EXCEPTION (*excp_name* **[,** *group_names*] [, *format_string*] [, *args***])**

where *excp_name* is the COOL symbol representing the exception class type (such as, **Error** or **Warning**), *group_names* are one or more pointers to COOL symbols each of which represents a group or alias name for this exception, *format_string* is a **printf(2)** compatible control string, and *args* are any combination of format arguments and keyword arguments to initialize data members. For example, the following macro invocation might be used to implement the index-bounds exception for the Vector<Type> class discussed above:

```
EXCEPTION (Out_Of_Bounds, SYM(Vector_Error), SYM(Fatal_If_Not_Handled)),
                "Index %d out of bounds for vector of type %s", bad_index = i, #Type);
```

The first argument is the exception type (ie. a new exception class derived from **Exception**), the second and third arguments are entries in the COOL symbol table for two group names (aliases) to be associated with this exception object, the third argument is a message string, and the last two arguments provide values for the fields in the message. The fourth argument also initializes a data member. When expanded, this macro generates:

```
(Exception_g = new Out_Of_Bounds(),
 Exception_g = set_group_names(2, SYM(Vector_Error), SYM(Fatal_If_Not_Handled)),
 Exception_g->format_msg = ERR_MSG (hprintf ("Index %d out of bounds for vector of type %s",
                                             i, #Type)),
 ((Out_Of_Bounds*)Exception_g)->bad_index = i,
 Exception_g);
```

This comma-separated expression creates a new **Out_Of_Bounds** exception object on the heap and assigns it to the pointer **Exception_g**. The two group names are added as symbols using the **set_group_names()** member function. The format message is created by **hprintf()**, which is a variation of **printf(2)** that returns a formated string allocated on the heap. This string is placed in the COOL **ERR_MSG** table that facilitates simple internationalization of text strings in an application [4]. Finally, the data member **bad_index** is initializied with the value **i**.

6.2. The RAISE Macro

The **RAISE** macro simplifies the process of creating and raising an exception. The exception object is constructed using **EXCEPTION** and is raised using the member function **raise()**. If one is found of the appropriate

type, it's handler function is invoked and passed the exception object as an argument. **RAISE** returns the exception object if the exception handler function returns or if no exception handler is found. **RAISE** is implemented as a COOL macro and has the following syntax:

RAISE (*excp_name* **[,** *group_names*] [, *format_string*] [, *args***])**

where *excp_name* is the COOL symbol representing the exception class type (such as, **Error** or **Warning**), *group_names* are one or more pointers to COOL symbols each of which represents a group or alias name for this exception, *format_string* is a **printf(2)** compatible control string, and *args* are any combination of format arguments and keyword arguments to initialize data members.

The **RAISE** macro is used extensively in COOL and is often the most convenient form used by programmers. The following example continues with the index-bounds exception for the Vector<Type> class from above. Suppose an application programmer is working on a tutorial programming environment that provides assistance at runtime to novice programmers. The programmer wishes to set up an exception handler to handle the index-bounds error by prompting for a new value, storing the result in the exception object, and retrying the operation. The class programmer implementing the Vector<Type> could write the code for the member function Vector<Type>::operator[] in the class library and raise an exception and process it as follows:

```
class Out_Of_Bounds : public Error {
public:
    unsigned int bad_index;
    unsigned int new_index;
    char* container;
    char* type;
    Out_of_Bounds() { format_msg = "Index %d out of bounds for %s of type %s"; }
    virtual void report (ostream& os) { os << msg_prefix << form(format_msg, bad_index, container, type);}
};

int Vector<Type>::operator[] (int i) {
    while(TRUE) {
        if (i >= 0 && i < this->number_elements)
            return this->data[i];
        else {
            Error* e = RAISE (Out_of_Bounds, bad_index = i, container = "Vector", type = #Type);
            if (e->exception_handled())
                i = e->new_index;
        }
    }
}
```

The application programmer writing the programming environment defines an exception handler function to prompt for a new index and an instance of a handler object to associate with the function for the **Out_Of_Bounds** exception in the following manner:

```
void Bounds_Index_Handler (Exception& excp) {
    excp.report(cout);
    cout << "New index to use instead: " << flush;
    cin >> excp.new_index;
    excp.handled(TRUE);
}

Excp_Handler vec_eh (Bounds_Index_Handler, SYM(Out_Of_Bounds));
```

The exception handler object **vec_eh** should be declared at what ever scope is necessary for exceptions of this type to be handled in this manner. This is usually, but not always, declared as a global instance so as to insure that the handler is available at all times during the run of an application. When the user of this system runs the

program with an erroneous index, the **Out_Of_Bounds** exception is raised in the class library, the exception handler **Bounds_Index_Handler** written by the application programmer is invoked and prompts for a new index, then the operation retryed. Note that the three programmer's involved in this system (the class programmer, the application programmer, and the novice programmer) had no interaction with each other and no discussion over the design and interface to the exception handling facility. However, the class programmer was able to use a flexible exception handling system that was customized with an application-specific handler to afford a more helpful system for the novice programmer. Finally, note that in this example, proceeding from the raised exception involves no use of the system setjmp/longjmp functions.

6.3. The STOP Macro

The **STOP** macro creates and raises an exception similar to the **RAISE** macro except that it guarantees to terminate program execution if an exception handler returns (ie. attempts to resume) or if no exception handler is found. The exception object is constructed in the same manner using the **EXCEPTION** macro and raised using the member function **stop()**. **STOP** is implemented as a COOL macro and has the following syntax:

STOP (*excp_name* **[,** *group_names*] [, *format_string*] [, *args***])**

where *excp_name* is the COOL symbol representing the exception class type (such as, **Error** or **Warning**), *group_names* are one or more pointers to COOL symbols each of which represents a group or alias name for this exception, *format_string* is a **printf(2)** compatible control string, and *args* are any combination of format arguments and keyword arguments to initialize data members.

6.4. The VERIFY Macro

The **VERIFY** macro asserts that an expression has a non-zero value by raising an exception of the specified type if the expression evaluates to zero. The exception object is constructed with **EXCEPTION** and is raised using the member function **raise()**. The exception type is optional and, if not given, defaults to **Verify_Error**. This exception class is derived from **Error** and contains a data member for storing a string representation of the expression that failed. **VERIFY** is implemented as a COOL macro and has the following syntax:

VERIFY (*test_expression***,** *excp_name* **= Verify_Error** [, *group_names*] [, *format_string*] [, *args***])**

where *test_expression* is the C++ expression to be verified, *excp_name* is the optional argument specifying the exception type, *group_names* are one or more pointers to COOL symbols each of which represents a group or alias name for this exception, *format_string* is a **printf(2)** compatible control string, and *args* are any combination of format arguments and keyword arguments to initialize data members. For example, the following macro invocation might be used to implement an alternate approach for the index-bounds exception discussed above:

```
int Vector<Type>::operator[] (int i) {
    VERIFY ((i >= 0 && i < this->number_elements));
    return this->data[i];
}
```

Since only the expression to assert is passed as an argument, the **VERIFY** macro creates a **Verify_Error** exception object by default and initializes the inherited data members. When expanded, this macro generates:

```
if (!(i >= 0 && i < this->number_elements))
    (Exception_g = new Verify_Error(),
    Exception_g)->raise();
```

6.5. The IGNORE_ERRORS Macro

The **IGNORE_ERRORS** macro ignores an exception that is raised while executing a body of statements. If an exception of a specified type or types is raised while within the scope of this body, the macro insures that the handler for that exception is not invoked, but rather, control is returned to the point immediately following the body. **IGNORE_ERRORS** is implemented as a COOL macro and has the following syntax:

IGNORE_ERRORS (*excp_ptr* [, *excp_class* = **Error**] [, *group_names*]) { *body* }

If an exception of type *excp_class* (or with group *group_names*) is raised while executing *body*, then *excp_ptr* is set to the address of this exception object and program control transfers to the statement following **IGNORE_ERRORS**. If no exception is raised while executing *body*, then *excp_ptr* is set to **NULL**. The exception must have been raised with the **RAISE**, **STOP**, or **VERIFY** macros. If *excp_class* is not specified, the default exception class is **Error**. **IGNORE_ERRORS** is implemented using the system functions, setjmp and longjmp. As a result, the statements within the braces should not require destructors to be invoked because the setjmp/longjmp mechanism does not currently support this capability.

In the following program fragment, **IGNORE_ERRORS** is used to ignore an exception of type **Error** raised while using operator[] of the Vector<Type> class. In this case, the exception object is send to the standard error stream and execution continues. If **IGNORE_ERRORS** were not used and no other exception handler had been defined, the default exception handler for **Error** would terminate the program with a call to **exit(2)**.

```
Vector<int> data[4];
Error* e;
IGNORE_ERRORS (e) {
    int i;
    i = data[5];
}
if (e != NULL)
    cerr << e;
```

6.6. The DO_WITH_HANDLER Macro

The **DO_WITH_HANDLER** macro establishes an exception handler whose scope is restricted to a specified body of statements. An exception handler that matches against one of the exception types or group names in the exception list is established during execution of a series of statements. The specified exception handler will be invoked if an exception of the specified type or group is raised and no other more-recently-established handler matches the type or group names. The exception must have been raised with the **RAISE**, **STOP**, or **VERIFY** macros. **DO_WITH_HANDLER** is implemented as a COOL macro and has the following syntax:

DO_WITH_HANDLER (*exh_func* [, *excp_types*]) { *body* }

where *exh_func* is the name of an exception handler function, *excp_types* is a list of one or more exception types and group names, and *body* specifies one or more C++ statements surrounded by braces. In the following example, a specialized exception handler for the index-bounds problem in the Vector<Type>::operator[] class is established for a small body of code:

```
extern void New_Out_Of_Bounds_Handler(Exception&);
Vector<int> data[4];
DO_WITH_HANDLER (New_Out_Of_Bounds_Handler, SYM(Out_Of_Bounds)) {
    int sum;
    for(int i = 0; i <= data.length(); i++)
        sum += data[i];
}
```

This code fragment contains a typical indexing problem where the programmer has an inaccurate loop-termination test, thus incrementing the index one too many times and causing an exception to be raised. When expanded, this macro generates:

```
extern void New_Out_Of_Bounds_Handler(Exception&);
Vector<int> data[4];
{
    Excp_Handler _do_eh(New_Out_Of_Bounds_Handler, SYM(Out_Of_Bounds));
    int sum;
    for(int i = 0; i <= data.length(); i++)
        sum += data[i];
}
```

6.7. The HANDLER_CASE Macro

The **HANDLER_CASE** macro establishes an exception handler that transfers control to a set of exception-case clauses when an exception is raised while executing a body of statements. **HANDLER_CASE** is implemented as a COOL macro and has the following syntax:

HANDLER_CASE { *body* } { *case_clauses* }
case_clauses ::= **case** ([*excp_spec*]) **:** {*statements*} [*case_clauses*]
excp_spec ::= [*excp_types*] [, *excp_class excp_decl*]
excp_types ::= *excp_class_or_group_type* [, *excp_types*]

If an exception is raised while executing *body*, an exception handler will be invoked to transfer control to *case_clauses*. The statements of the *case_clause* whose *excp_spec* matches the raised exception type or one of its group names is executed. The variable *excp_decl* is bound to the *excp_class* exception object raised and may be referenced by any statement in the matching *case_clause*. The exception must have been raised with **RAISE**, **STOP**, or **VERIFY**. Finally, as in **IGNORE_ERRORS**, the **HANDLER_CASE** macro is implemented using the system functions, setjmp and longjmp. As a result, the statements within the matching *case_clause* should not require destructors to be invoked because the setjmp/longjmp mechanism does not currently support this capability. The following example shows the use of this macro with the index-bounds problem used throughout this paper:

```
Boolean calculate_sum (Vector<int>& v1, int start, int end)
{
    HANDLER_CASE {
        int sum;
        for(int i = start; i <= end; i++)
            sum += v1[i];
        }
       v1.push(sum);
    }
    case (SYM (Out_Of_Bounds), Error eh) : {
        cerr << eh;
        return FALSE;
    }
    case (SYM (Resize_Error), SYM (Static_Error), Error eh) : {
        cerr << eh;
        abort();
    }
    case (Error eh) : {
        cerr << eh;
        exit();
    }
    return TRUE;
}
```

This function takes a reference to a vector object, a starting index, and an ending index to compute the sum of the elements between and including these indexes. If an exception occurs during this operation, control is

transferred to the appropriate case statement clause. If an exception of type **Out_Of_Bounds** is raised, the exception object is printed on the standard error stream and the value FALSE is returned from the function. If the raised exception has a group name of either **Resize_Error** or **Static_Error**, the exception object is printed on the standard error stream and the program aborts. Finally, as a default condition, if any other type of exception is raised, the exception object is printed on the standard error stream and the program exits. If no exceptions are raised, the sum of the elements is pushed onto the end of the vector object and the value TRUE is returned from the function.

7. COOL Exception Handling Comparison to Koenig/Stroustrup Proposal

The COOL exception handling facility implements a portable, compiler independent, object-oriented exception handling capability. Exceptions are classified according to subtype relationships making it easy to test for a group of exceptions. These exception objects have data members through which state information is conveyed from the point at which the exception is raised to the handler. The base exception and handler classes contain generic virtual functions upon which more sophisticated capabilities can be built through inheritance and derivation.

The COOL exception handling macros, **RAISE** and **HANDLER_CASE**, provide the same type of functionality as the **throw** and **try-catch** statements proposed by Koenig and Stroustrup in their paper, *Exception Handling for C++* [5]. Both **throw** and **RAISE** transfer control to the most recently establish handler for a particular type of exception. However, any object may be used as an argument in a **throw** expression, whereas **RAISE** only passes exception objects. In a similar manner, the **try-catch** block and the **HANDLER_CASE** macro establish some handlers while executing a body of statements. The difference here is that the **catch** expression in a **try** block is like a function definition and any data type can be specified in the exception declaration. The case statements in the **HANDLER_CASE** macro, on the other hand, accept only COOL symbol objects. These symbols allow for group names and aliases to be used to handle a single kind of exception in one of several ways.

These differences are minor, however, when compared to the philosophical models each system follows: termination versus resumption. In the one, the **throw** unwinds the stack before the call of the exception handler, thus supporting a termination model for exception handling, while in the second, the **RAISE** macro expands into a function call before the stack is unwound, thus supporting a resumption model for exception handling.

8. Conclusion

The COOL exception handling facility is a set of classes and macros that provide a mechanism to detect and raise an exception and independently establish and invoke an exception handler to deal with the exception in another part of the application. An exception handler may either fix the problem and resume execution, retry the operation, or terminate the program. The virtual member functions **report**, **raise**, **default_handler**, and **match** can be customized in user-defined exception classes derived from the base **Exception** and **Excp_Handler** classes. The macros **EXCEPTION**, **RAISE**, **STOP**, **VERIFY**, **IGNORE_ERRORS**, **DO_WITH_HANDLER**, and **HANDLER_CASE** provide a consistent and convenient way for the programmer to create, raise, and handle exceptions.

There are still a number of problems that need to be addressed in the COOL exception handling system. Of particular concern are how to handle exceptions raised in constructors and destructors, and the destruction of objects when setjmp/longjmp are used. Texas Instruments has been using the exception handling facility internally on several projects in conjunction with the COOL class library. We have found that the use of a flexible exception handling system enables programmer's to customize the error handling to suit a specific application. As a result, class libraries are more useful and reusable, thus significantly increasing the productivity of the programmer and thus enabling applications to be prototyped in a shorter time period than might otherwise be possible. COOL is currently up and running on a Sun SPARCstation 1 (TM) running SunOS (TM) 4.x, a PS/2 (TM) model 70 running SCO XENIX® 2.3, a PS/2 model 70 running OS/2 1.1, and a MIPS running RISC/os 4.0. The

---

SunOS and SPARCstation 1 are trademarks of Sun Microsystems, Inc.
PS/2 is a trademark of International Business Machines Corporation.
XENIX is a registered trademark of Microsoft Corporation.

SPARC and MIPS ports utilize the AT&T C++ translator (cfront) version 2.0 and the XENIX and OS/2 ports utilize the Glockenspiel translator with the Microsoft C compiler.

9. References

[1] Andy Daniels and Kent Pitman, *Common Lisp Condition System Revision #18*, ANSI X3J13 subcommittee on Error Handling, March 1988.

[2] Mary Fontana, Martin Neath and Lamott Oren, *COOL - A C++ Object-Oriented Library*, Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[3] Mary Fontana, Martin Neath and Lamott Oren, *A Portable Implementation of Parameterized Templates Using A Sophisticated C++ Macro Facility*, Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[4] Mary Fontana, Martin Neath and Lamott Oren, *Symbolic Computing for C++*, Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[5] Andrew Koenig and Bjarne Stroustrup, *Exception Handling for C++*, Submitted as document X3J16/90-042 to the ANSI C++ committee, July, 1990.

[6] Stanley Lippman, *C++ Primer*, Addison-Wesley, Reading, MA, 1989.

[7] Mike Miller, *Exception Handling Without Language Extensions*, Proceedings of the USENIX C++ Conference, Denver, CO, October 17-21, 1988, pp. 327-341.

[8] Texas Instruments Incorporated, *COOL User's Guide*, Information Technology Group, Austin, TX, Internal Original Issue January 1990.