Advanced High Level Control Structures Chapter One

1.1 Chapter Overview

Volume One introduced some basic HLA control structures like the IF and WHILE statements (see “Some Basic HLA Control Structures” on page 29.). This section elaborates on some of those control structures (discussing features that were a little too advanced to present in Volume One) and it introduces the remaining high level language control structures that HLA provides.

This includes a full discussion of HLA’s boolean expressions, the TRY..ENDTRY statement, the RAISE statement, the BEGIN..END/EXIT/EXITIF statements, and the SWITCH/CASE/ENDSWITCH statement the HLA Standard Library provides.

1.2 Conjunction, Disjunction, and Negation in Boolean Expressions

One obvious omission in HLA’s high level control structures is the ability to use conjunction (logical AND), disjunction (logical OR), and negation (logical NOT) in run-time boolean expressions. This omission, however, has been in this text, not in the HLA language. HLA does provide these facilities, this section will describe their use.

HLA uses the “&&” operator to denote logical AND in a run-time boolean expression. This is a dyadic (two-operand) operator and the two operands must be legal run-time boolean expressions. This operator evaluates true if both operands evaluate to true. Example:

```
if( eax > 0 && ch = 'a' ) then

    mov( eax, ebx );
    mov( ' ', ch );

endif;
```

The two MOV statements above execute only if EAX is greater than zero *and* CH is equal to the character ‘a’. If either of these conditions is false, then program execution skips over the two MOV instructions.

Note that the expressions on either side of the “&&” operator may be any expression that is legal in an IF statement, these expressions don’t have to be comparisons using one of the relational operators. For example, the following are all legal expressions:

```
@z && al in 5..10
al in {'a'..'z'} && ebx
boolVar && !eax
!fileio.eof( fileHandle ) && fileio.getc( fileHandle ) <> ' '
```

HLA uses short circuit evaluation when compiling the “&&” operator. If the left-most operand evaluates false, then the code that HLA generates does not bother evaluating the second operand (since the whole expression must be false at that point). Therefore, in the last expression above, the code will not execute the call to *fileio.getc* if the file pointer is currently pointing at the end of the file.

Note that an expression like “eax < 0 && ebx <> eax” is itself a legal boolean expression and, therefore, may appear as the left or right operand of the “&&” operator. Therefore, expressions like the following are perfectly legal:

```
eax < 0 && ebx <> eax    &&    !ecx
```

The “&&” operator is left associative, so the code that HLA generates evaluates the expression above in a left-to-right fashion. Note that if EAX is less than zero, the code will not test either of the remaining expres-

sions. Likewise, if EAX is not less than zero but EBX is equal to EAX, this code will not evaluate the third expression since the whole expression is false regardless of ECX's value.

HLA uses the "||" operator to denote disjunction (logical OR) in a run-time boolean expression. Like the "&&" operator, this operator expects two otherwise legal run-time boolean expressions as operands. This operator evaluates true if either (or both) operands evaluate true. Like the "&&" operator, the disjunction operator uses short-circuit evaluation. If the left operand evaluates true, then the code that HLA generates doesn't bother to test the value of the second operand. Instead, the code will transfer to the location that handles the situation when the boolean expression evaluates true. Examples of legal expressions using the "||" operator:

```
@z || al = 10
al in {'a'..'z'} || ebx
!boolVar || eax
```

As for the "&&" operator, the disjunction operator is left associative so multiple instances of the "||" operator may appear within the same expression. Should this be the case, the code that HLA generates will evaluate the expressions from left to right, e.g.,

```
eax < 0 || ebx <> eax   ||   !ecx
```

The code above executes if either EAX is less than zero, EBX does not equal EAX, or ECX is zero. Note that if the first comparison is true, the code doesn't bother testing the other conditions. Likewise, if the first comparison is false and the second is true, the code doesn't bother checking to see if ECX is zero. The check for ECX equal to zero only occurs if the first two comparisons are false.

If both the conjunction and disjunction operators appear in the same expression then the "&&" operator takes precedence over the "||" operator. Consider the following expression:

```
eax < 0 || ebx <> eax  && !ecx
```

The code HLA generates evaluates this as

```
eax < 0 || (ebx <> eax  && !ecx)
```

If EAX is less than zero, then the code HLA generates does not bother to check the remainder of the expression, the entire expression evaluates true. However, if EAX is not less than zero, then both of the following conditions must evaluate true in order for the overall expression to evaluate true.

HLA allows you to use parentheses to surround subexpressions involving "&&" and "||" if you need to adjust the precedence of the operators. Consider the following expression:

```
(eax < 0 || ebx <> eax)  && !ecx
```

For this expression to evaluate true, ECX must contain zero and either EAX must be less than zero or EBX must not equal EAX. Contrast this to the result obtained without the parentheses.

As you saw in Volume One, HLA uses the "!" operator to denote logical negation. However, the "!" operator may only prefix a register or boolean variable; you may not use it as part of a larger expression (e.g., "!eax < 0"). To achieve logical negative of an existing boolean expression you must surround that expression with parentheses and prefix the parentheses with the "!" operator, e.g.,

```
!( eax < 0 )
```

This expression evaluates true if EAX is not less than zero.

The logical not operator is primarily useful for surrounding complex expressions involving the conjunction and disjunction operators. While it is occasionally useful for short expressions like the one above, it's usually easier (and more readable) to simply state the logic directly rather than convolute it with the logical not operator.

Note that HLA's "|" and "&" operators (compile-time address expressions) are distinct from "||" and "&&" and have completely different meanings. See the chapter on the HLA Run-Time Language in this volume or the chapter on constants in the previous volume for details.

1.3 TRY..ENDTRY

Volume One discusses the TRY..ENDTRY statement, but does not fully discuss all of the features available. This section will complete the discussion of TRY..ENDTRY and discuss some problems that could occur when you use this statement.

As you may recall, the TRY..ENDTRY statement surrounds a block of statements in order to capture any exceptions that occur during the execution of those statements. The system raises exceptions in one of three ways: through a hardware fault (such as a divide by zero error), through an operating system generated exception, or through the execution of the HLA RAISE statement. You can write an exception handler to intercept specific exceptions using the EXCEPTION clause. The following program provides a typical example of the use of this statement:

```
program testBadInput;
#include( "stdlib.hhf" );

static
    u:      uns16;


begin testBadInput;

    try

        stdout.put( "Enter an unsigned integer:" );
        stdin.get( u );
        stdout.put( "You entered: ", u, nl );

      exception( ex.ConversionError )

        stdout.put( "Your input contained illegal characters" nl );

      exception( ex.ValueOutOfRange )

        stdout.put( "The value was too large" nl );

    endtry;


end testBadInput;
```

Program 1.1 TRY..ENDTRY Example

HLA refers to the statements between the TRY clause and the first EXCEPTION clause as the *protected* statements. If an exception occurs within the protected statements, then the program will scan through each of the exceptions and compare the value of the current exception against the value in the parentheses after each of the EXCEPTION clauses[1]. This exception value is simply an *uns32* value. The value in the parentheses after each EXCEPTION clause, therefore, must be an unsigned 32-bit value. The HLA "excepts.hhf" header file predefines several exception constants. Other than it would be an incredibly bad style violation,

1. Note that HLA loads this value into the EAX register. So upon entry into an EXCEPTION clause, EAX contains the exception number.

you could substitute the numeric values for the two EXCEPTION clauses above (see the excepts.hhf header file for the actual values).

1.3.1 Nesting TRY..ENDTRY Statements

If the program scans through all the exception clauses in a TRY..ENDTRY statement and does match the current exception value, then the program searches through the EXCEPTION clauses of a *dynamically nested* TRY..ENDTRY block in an attempt to find an appropriate exception handler. For example, consider the following code:

```
program testBadInput2;
#include( "stdlib.hhf" );

static
    u:      uns16;

begin testBadInput2;

    try

        try

            stdout.put( "Enter an unsigned integer:" );
            stdin.get( u );
            stdout.put( "You entered: ", u, nl );

          exception( ex.ConversionError )

            stdout.put( "Your input contained illegal characters" nl );

        endtry;

        stdout.put( "Input did not fail due to a value out of range" nl );


      exception( ex.ValueOutOfRange )

        stdout.put( "The value was too large" nl );

    endtry;

end testBadInput2;
```

Program 1.2 Nested TRY..ENDTRY Statements

In this example one TRY statement is nested inside another. During the execution of the *stdin.get* statement, if the user enters a value greater than four billion and some change, then *stdin.get* will raise the *ex.ValueOutOfRange* exception. When the HLA run-time system receives this exception, it first searches through all the EXCEPTION clauses in the TRY..ENDTRY statement immediately surrounding the statement that raised the exception (this would be the nested TRY..ENDTRY in the example above). If the HLA run-time system fails to locate an exception handler for *ex.ValueOutOfRange* then it checks to see if the current TRY..ENDTRY is nested inside another TRY..ENDTRY (as is the case in Program 1.2). If so, the HLA

run-time system searches for the appropriate EXCEPTION clause in that TRY..ENDTRY statement. Within this TRY..ENDTRY block the program finds an appropriate exception handler, so control transfers to the statements after the "exception(ex.ValueOutOfRange)" clause.

After leaving a TRY..ENDTRY block, the HLA run-time system no longer considers that block active and will not search through its list of exceptions when the program raises an exception[2]. This allows you to handle the same exception differently in other parts of the program.

If two nested TRY..ENDTRY statements handle the same exception, and the program raises an exception while executing in the innermost TRY..ENDTRY seqeuence, then HLA transfers control directly to the exception handler provided by that TRY..ENDTRY block. HLA does not automatically transfer control to the exception handler provided by the outer TRY..ENDTRY sequence.

If the program raises an exception for which there is no appropriate EXCEPTION clause active, control transfers to the HLA run-time system. It will stop the program and print an appropriate error message.

In the previous example (Program 1.2) the second TRY..ENDTRY statement was statically nested inside the enclosing TRY..ENDTRY statement[3]. As mentioned without comment earlier, if the most recently activated TRY..ENDTRY statement does not handle a specific exception, the program will search through the EXCEPTION clauses of any dynamically nesting TRY..ENDTRY blocks. Dynamic nesting does not require the nested TRY..ENDTRY block to physically appear within the enclosing TRY..ENDTRY statement. Instead, control could transfer from inside the enclosing TRY..ENDTRY protected block to some other point in the program. Execution of a TRY..ENDTRY statement at that other point dynamically nests the two TRY statements. Although you will see lots of ways to dynamically nest code a little later in this chapter, there is one method you are familiar with that will let you dynamically nest these statements: the procedure call. The following program provides yet another example of nested TRY..ENDTRY statements, this example demonstrates dynamic nesting via a procedure call:

```
program testBadInput3;
#include( "stdlib.hhf" );


    procedure getUns;
    static
        u:      uns16;

    begin getUns;

        try

            stdout.put( "Enter an unsigned integer:" );
            stdin.get( u );
            stdout.put( "You entered: ", u, nl );

          exception( ex.ConversionError )

            stdout.put( "Your input contained illegal characters" nl );

        endtry;

    end getUns;


begin testBadInput3;
```

2. Unless, of course, the program re-enters the TRY..ENDTRY block via a loop or other control structure.

3. Statically nested means that one statement is physically nested within another in the source code. When we say one statement is nested within another, this typically means that the statement is statically nested within the other statement.


```
    try

        getUns();
        stdout.put( "Input did not fail due to a value out of range" nl );

      exception( ex.ValueOutOfRange )

        stdout.put( "The value was too large" nl );

    endtry;

end testBadInput3;
```

Program 1.3 Dynamic Nesting of TRY..ENDTRY Statements

In Program 1.3 the main program executes the TRY statement that activates a value out of range exception handler, then it calls the *getUns* procedure. Inside the *getUns* procedure, the program executes a second TRY statement. This dynamically nests this TRY..ENDTRY block inside the TRY of the main program. Because the main program has not yet encountered its ENDTRY, the TRY..ENDTRY block in the main program is still active. However, upon execution of the TRY statement in *getUns*, the nested TRY..ENDTRY block takes precedence. If an exception occurs inside the *stdin.get* procedure, control transfers to the most recently activated TRY..ENDTRY block of statements and the program scans through the EXCEPTION clauses looking for a match to the current exception value. In the program above, if the exception is a conversion error exception, then the exception handler inside *getUns* will handle the error and print an appropriate message. After the execution of the exception handler, the program falls through to the bottom of *getUns* and it returns to the main program and prints the message "Input did not fail due to a value out of range". Note that if a nested exception handler processes an exception, the program does not automatically reraise this exception in other active TRY..ENDTRY blocks, even if they handle that same exception (*ex.ConversionError*, in this case).

Suppose, however, that *stdin.get* raises the *ex.ValueOutOfRange* exception rather than the *ex.ConversionError* exception. Since the TRY..ENDTRY statement inside *getUns* does not handle this exception, the program will search through the exception list of the enclosing TRY..ENDTRY statement. Since this statement is in the main program, the exception will cause the program to automatically return from the *getUns* procedure. Since the program will find the value out of range exception handler in the main program, it transfers control directly to the exception handler. Note that the program will not print the string "Input did not fail due to a value out of range" since control transfers directly from *stdin.get* to the exception handler.

1.3.2 The UNPROTECTED Clause in a TRY..ENDTRY Statement

Whenever a program executes the TRY clause, it preserves the current exception environment and sets up the system to transfer control to the EXCEPTION clauses within that TRY..ENDTRY statement should an exception occur. If the program successfully completes the execution of a TRY..ENDTRY protected block, the program restores the original exception environment and control transfers to the first statement beyond the ENDTRY clause. This last step, restoring the execution environment, is very important. If the program skips this step, any future exceptions will transfer control to this TRY..ENDTRY statement even though the program has already left the TRY..ENDTRY block. The following program demonstrates this problem:

```
program testBadInput4;
#include( "stdlib.hhf" );
```



```
static
    input:  uns32;

begin testBadInput4;

    // This forever loop repeats until the user enters
    // a good integer and the BREAK statement below
    // exits the loop.

    forever

        try

            stdout.put( "Enter an integer value: " );
            stdin.get( input );
            stdout.put( "The first input value was: ", input, nl );
            break;

          exception( ex.ValueOutOfRange )

            stdout.put( "The value was too large, reenter." nl );

          exception( ex.ConversionError )

            stdout.put( "The input contained illegal characters, reenter." nl );

        endtry;

    endfor;

    // Note that the following code is outside the loop and there
    // is no TRY..ENDTRY statement protecting this code.

    stdout.put( "Enter another number: " );
    stdin.get( input );
    stdout.put( "The new number is: ", input, nl );

end testBadInput4;
```

Program 1.4 Improperly Exiting a TRY..ENDTRY Statement

This example attempts to create a robust input system by putting a loop around the TRY..ENDTRY statement and forcing the user to reenter the data if the *stdin.get* routine raises an exception (because of bad input data). While this is a good idea, there is a big problem with this implementation: the BREAK statement immediately exits the FOREVER..ENDFOR loop without first restoring the exception environment. Therefore, when the program executes the second *stdin.get* statement, at the bottom of the program, the HLA exception handling code still thinks that it's inside the TRY..ENDTRY block. If an exception occurs, HLA transfers control back into the TRY..ENDTRY statement looking for an appropriate exception handler. Assuming the exception was *ex.ValueOutOfRange* or *ex.ConversionError*, Program 1.4 will print an appropriate error message *and then force the user to reenter the first value*. This isn't desirable.

Transferring control to the wrong TRY..ENDTRY exception handlers is only part of the problem. Another big problem with the code in Program 1.4 has to do with the way HLA preserves and restores the exception environment: specifically, HLA saves the old execution environment information on the stack. If you exit a TRY..ENDTRY without restoring the exception environment, this leaves garbage on the stack (the old execution environment information) and this extra data on the stack could cause your program to malfunction.

Although it is quite clear that a program should not exit from a TRY..ENDTRY statement in the manner that Program 1.4 uses, it would be nice if you could use a loop around a TRY..ENDTRY block to force the re-entry of bad data as this program attempts to do. To allow for this, HLA's TRY..ENDTRY provides an UNPROTECTED section. Consider the following program code:

```
program testBadInput5;
#include( "stdlib.hhf" );

static
    input:  uns32;

begin testBadInput5;

    // This forever loop repeats until the user enters
    // a good integer and the BREAK statement below
    // exits the loop.  Note that the BREAK statement
    // appears in an UNPROTECTED section of the TRY..ENDTRY
    // statement.

    forever

        try

            stdout.put( "Enter an integer value: " );
            stdin.get( input );
            stdout.put( "The first input value was: ", input, nl );

          unprotected

            break;

          exception( ex.ValueOutOfRange )

            stdout.put( "The value was too large, reenter." nl );

          exception( ex.ConversionError )

            stdout.put( "The input contained illegal characters, reenter." nl );

        endtry;

    endfor;

    // Note that the following code is outside the loop and there
    // is no TRY..ENDTRY statement protecting this code.

    stdout.put( "Enter another number: " );
    stdin.get( input );
    stdout.put( "The new number is: ", input, nl );

end testBadInput5;
```

Program 1.5 The TRY..ENDTRY UNPROTECTED Section

Whenever the TRY..ENDTRY statement hits the UNPROTECTED clause, it immediate restores the exception environment from the stack. As the phrase suggests, the execution of statements in the UNPRO-

TECTED section is no longer protected by the enclosing TRY..ENDTRY block (note, however, that any dynamically nesting TRY..ENDTRY statements will still be active, UNPROTECTED only turns off the exception handling of the TRY..ENDTRY statement that immediately contains the UNPROTECTED clause). Since the BREAK statement in Program 1.5 appears inside the UNPROTECTED section, it can safely transfer control out of the TRY..ENDTRY block without "executing" the ENDTRY since the program has already restored the former exception environment.

Note that the UNPROTECTED keyword must appear in the TRY..ENDTRY statement immediately after the protected block. I.e., it must precede all EXCEPTION keywords.

If an exception occurs during the execution of a TRY..ENDTRY sequence, HLA automatically restores the execution environment. Therefore, you may execute a BREAK statement (or any other instruction that transfers control out of the TRY..ENDTRY block) within an EXCEPTION clause without having to do anything special.

Since the program restores the exception environment upon encountering an UNPROTECTED block or an EXCEPTION block, an exception that occurs within one of these areas immediately transfers control to the previous (dynamically nesting) active TRY..ENDTRY sequence. If there is no nesting TRY..ENDTRY sequence, the program aborts with an appropriate error message.

1.3.3 The ANYEXCEPTION Clause in a TRY..ENDTRY Statement

In a typical situation, you will use a TRY..ENDTRY statement with a set of EXCEPTION clauses that will handle all possible exceptions that can occur in the protected section of the TRY..ENDTRY sequence. Often, it is important to ensure that a TRY..ENDTRY statement handles all possible exceptions to prevent the program from prematurely aborting due to an unhandled exception. If you have written all the code in the protected section, you will know the exceptions it can raise so you can handle all possible exceptions. However, if you are calling a library routine (especially a third-party library routine), making a OS API call, or otherwise executing code that you have no control over, it may not be possible for you to anticipate all possible exceptions this code could raise (especially when considering past, present, and future versions of the code). If that code raises an exception for which you do not have an EXCEPTION clause, this could cause your program to fail. Fortunately, HLA's TRY..ENDTRY statement provides the ANYEXCEPTION clause that will automatically trap any exception the existing EXCEPTION clauses do not handle.

The ANYEXCEPTION clause is similar to the EXCEPTION clause except it does not require an exception number parameter (since it handles any exception). If the ANYEXCEPTION clause appears in a TRY..ENDTRY statement with other EXCEPTION sections, the ANYEXCEPTION section must be the last exception handler in the TRY..ENDTRY statement. An ANYEXCEPTION section may be the only exception handler in a TRY..ENDTRY statement.

If an otherwise unhandled exception transfers control to an ANYEXCEPTION section, the EAX register will contain the exception number. Your code in the ANYEXCEPTION block can test this value to determine the cause of the exception.

1.3.4 Raising User-Defined Exceptions

Although you typically use the TRY..ENDTRY statement to catch exceptions that the hardware, the OS or the HLA Standard Library raises, it is also possible to create your own exceptions and process them via the TRY..ENDTRY statement. You accomplish this by assigning an unused exception number to your exception and raising this exception with the HLA RAISE statement.

The parameter you supply to the EXCEPTION statement is really nothing more than an unsigned integer value. The HLA "excepts.hhf" header file provides definitions for the standard HLA Standard Library and hardware/OS exception types. These names are nothing more than dword constants that have been given a descriptive name. HLA reserves the values zero through 1023 for HLA and HLA Standard Library exceptions; it also reserves all exception values greater than $FFFF (65,535) for use by the operating system. The values in the range 1024 through 65,535 are available for user-defined exceptions.

To create a user-defined exception, you would generally begin by defining a descriptive symbolic name for the exception[4]. Then within your code you can use the RAISE statement to raise that exception. The following program provides a short example of some code that uses a user-defined exception to trap empty strings:

```
program userDefinedExceptions;
#include( "stdlib.hhf" );

    // Provide a descriptive name for the
    // user-defined exception.

const   EmptyString:dword := 1024;

    // readAString-
    //
    //  This procedure reads a string from the user
    // and returns a pointer to that string in the
    // EAX register.  It raises the "EmptyString"
    // exception if the string is empty.

    procedure readAString;
    begin readAString;

        stdin.a_gets();
        if( (type str.strRec [eax]).length == 0 ) then

            strfree( eax );
            raise( EmptyString );

        endif;

    end readAString;

begin userDefinedExceptions;

    try

        stdout.put( "Enter a non-empty string: " );
        readAString();
        stdout.put
        (
            "You entered the string '",
            (type string eax),
            "'"
            nl
        );
        strfree( eax );

      exception( EmptyString )

        stdout.put( "You entered an empty string", nl );

    endtry;

end userDefinedExceptions;
```

4. Technically, you could use a literal numeric constant, e.g., EXCEPTION(1024), but this is extremely poor programming style.

Program 1.6 User-Defined Exceptions and the RAISE Statement

One important thing to notice in this example: the *readAString* procedure frees the string storage before raising the exception. It has to do this because the RAISE statement loads the EAX register with the exception number (1024 in this case), effectively obliterating the pointer to the string. Therefore, this code frees the storage before the exception and assumes that EAX does not contain a valid string pointer if an exception occurs.

In addition to raising exceptions you've defined, you can also use the RAISE statement to raise any exception. Therefore, if your code encounters an error converting some data, you could raise the *ex.ConversionError* exception to denote this condition. There is nothing sacred about the predefined exception values. Feel free to use their values as exceptions if they are descriptive of the error you need to handle.

1.3.5 Reraising Exceptions in a TRY..ENDTRY Statement

Once a program transfers control to an exception handling section, the exception is effectively dead. That is, after executing the associated EXCEPTION block, control transfers to the first statement after the ENDTRY and program execution continues as though an exception had not occurred. HLA assumes that the exception handler has taken care of the problem and it is okay to continue program execution after the ENDTRY statement. In some instances, this isn't an appropriate response.

Although falling through and executing the statements after the ENDTRY when an exception handler finishes is probably the most common response, another possibility is to reraise the exception at the end of the EXCEPTION sequence. This lets the current TRY..ENDTRY block accommodate the exception as best it can and then pass control to an enclosing TRY..ENDTRY statement to complete the exception handling process. To reraise an exception all you need do is execute a RAISE statement at the end of the exception handler. Although you would typically reraise the same exception, there is nothing preventing you from raising a different exception at the end of your exception handler. For example, after handling a user-defined exception you've defined, you might want to raise a different exception (e.g., *ex.MemoryAllocationFailure*) and let an enclosing TRY..ENDTRY statement finish handling your exception.

1.3.6 A List of the Predefined HLA Exceptions

Appendix G in this text provides a listing of the HLA exceptions and the situations in which the hardware, the operating system, or the HLA Standard Library raises these exceptions. The HLA Standard Library reference in Appendix F also lists the exceptions that each HLA Standard Library routine raises. You should skim over these appendices to familiarize yourself with the types of exceptions HLA raises and refer to these sections when calling Standard Library routines to ensure that you handle all the possible exceptions.

1.3.7 How to Handle Exceptions in Your Programs

When an exception occurs in a program there are four general ways to handle the exception: (1) correct the problem in the exception handler and restart the offending instruction (if this is possible), (2) report an error message and loop back to the offending code and re-execute the entire sequence, preferably with better input data that won't cause an exception, (3) report an error and reraise the exception (or raise a different exception) and leave it up to a dynamically nesting exception handler to deal with the problem, or (4) clean up the program's data as much as possible and abort program execution. The HLA run-time system only supports the last three options (i.e., it does not allow you to restart the offending instruction after some sort of correction), so we will ignore the first option in this text.

Reporting an error and looping back to repeat the offending code is an extremely common solution when then program raises an exception because of bad user input. The following program provides a typical example of this solution that forces a user to enter a valid unsigned integer value:

```
program repeatingBadCode;
#include( "stdlib.hhf" );

static
    u:      uns16;

begin repeatingBadCode;

    forever

        try
            // Protected block.  Read an unsigned integer
            // from the user and display that value if
            // there wasn't an error.

            stdout.put( "Enter an unsigned integer:" );
            stdin.get( u );


          // Clean up the exception and break out of
          // the forever loop if all went well.

          unprotected

            break;


          // If the user entered an illegal character,
          // print an appropriate error message.

          exception( ex.ConversionError )

            stdout.put( "Your input contained illegal characters" nl );

          // If the user entered a value outside the range
          // 0..65535 then print an error message.

          exception( ex.ValueOutOfRange )

            stdout.put( "The value was too large" nl );

        endtry;

        // If we get down here, it's because there was an exception.
        // Loop back and make the user reenter the value.

    endfor;

    // Only by executed the BREAK statement do we wind up down here.
    // That occurs if the user entered a value unsigned integer value.

    stdout.put( "You entered: ", u, nl );

end repeatingBadCode;
```


Program 1.7 Repeating Code via a Loop to Handle an Exception

Another way to handle an exception is to print an appropriate message (or take other corrective action) and then re-raise the exception or raise a different exception. This allows an enclosing exception handler to handle the exception. The big advantage to this scheme is that it minimizes the code you have to write to handle a given exception throughout your code (i.e., passing an exception on to a different handler that contains some complex code is much easier than replicating that complex code everywhere the exception can occur). However, this approach has its own problems. Primary among the problems is ensuring that there is some enclosing TRY..ENDTRY statement that will handle the exception for you. Of course, HLA automatically encloses your entire program in one big TRY..ENDTRY statement, but the default handler simply prints a short message and then stops your program. This is unacceptable behavior in a robust program. At the very least, you should supply your own exception handler that surrounds the code in your main program that attempts to clean up the system before shutting it down if an otherwise unhandled exception comes along. Generally, however, you would like to handle the exception without shutting down the program. Ensuring that this always occurs if you reraise an exception can be difficult.

The last alternative, and certainly the least desirable of the four, is to clean up the system as much as possible and terminate program execution. Cleaning up the system includes writing transient data in memory to files, closing the files, releasing system resources (i.e., peripheral devices and memory), and, in general, preserving as much of the user's work as possible before quitting the program. Although you would like to continue program execution whenever an exception occurs, sometimes it is impossible to recover from an invalid operation (either on the part of the user or because of an error in your program) and continue execution. In such a situation you want to shut the program down as gracefully as possible so the user can restart the program and continue where they left off.

Of course, the absolute worst thing you can do is allow the program to terminate without attempting to save user data or release system resources. The user of your application will not have kind things to say about your program if they use it for three or four hours and the program aborts and loses all the data they've entered (requiring them to spend another three or four hours entering that data). Telling the user to "save your data often" is not a good substitute for automatically saving their data when an exception occurs.

The easiest way to handle an arbitrary (and unexpected) exception is to place a TRY..ANYEXCEPTION..ENDTRY statement around your main program. If the program raises an exception, you should save the value in EAX upon entry into the ANYEXCEPTION section (this contains the exception number) and then save any important data and release any resources your program is using. After this, you can re-raise the exception and let the default handler print the error message and terminate your program.

1.3.8 Registers and the TRY..ENDTRY Statement

The TRY..ENDTRY statement preserves about 16 bytes of data on the stack whenever you enter a TRY..ENDTRY statement. Upon leaving the TRY..ENDTRY block (or hitting the UNPROTECTED clause), the program restores the exception environment by popping this data off the stack. As long as no exception occurs, the TRY..ENDTRY statement does not affect the values of any registers upon entry to or upon exit from the TRY..ENDTRY statement. However, this claim is not true if an exception occurs during the execution of the protected statements.

Upon entry into an EXCEPTION clause the EAX register contains the exception number and the state of all other general purpose registers is undefined. Since the operating system may have raised the exception in response to a hardware error (and, therefore, has played around with the registers), you can't even assume that the general purpose registers contain whatever values they happened to contain at the point of the exception. The underlying code that HLA generates for exceptions is subject to change in different versions of the compiler, and certainly it changes across operating systems, so it is never a good idea to experimentally determine what values registers contain in an exception handler and depend upon those values in your code.

Since entry into an exception handler can scramble all the register values, you must ensure that you reload important registers if the code following your ENDTRY clause assumes that the registers contain valid values (i.e., values set in the protected section or values set prior to executing the TRY..ENDTRY statement). Failure to do so will introduce some nasty defects into your program (and these defects may be very intermittent and difficult to detect since exceptions rarely occur and may not always destroy the value in a particular register). The following code fragment provides a typical example of this problem and its solution:

```
static
    array: uns32[8];
        .
        .
        .
    for( mov( 0, ebx ); ebx < 8; inc( ebx )) do

        push( ebx );  // Must preserve EBX in case there is an exception.
        forever
            try

                stdin.geti32();
                unprotected break;

              exception( ex.ConversionError )

                stdout.put( "Illegal input, please reenter value: " );

            endtry;
        endfor;
        pop( ebx );  // Restore EBX's value.
        mov( eax, array[ ebx*4 ] );

    endfor;
```

Because the HLA exception handling mechanism messes with the registers, and because exception handling is a relatively inefficient process, you should never use the TRY..ENDTRY statement as a generic control structure (e.g., using it to simulate a SWITCH/CASE statement by raising an integer exception value and using the EXCEPTION clauses as the cases to process). Doing so will have a very negative impact on the performance of your program and may introduce subtle defects because exceptions scramble the registers.

For proper operation, the TRY..ENDTRY statement assumes that you only use the EBP register to point at *activation records* (the chapter on intermediate procedures discusses activation records). By default, HLA programs automatically use EBP for this purpose; as long as you do not modify the value in EBP, your programs will automatically use EBP to maintain a pointer to the current activation record. If you attempt to use the EBP register as a general purpose register to hold values and compute arithmetic results, HLA's exception handling capabilities will no longer function properly (not to mention you will lose access to procedure parameters and variables in the VAR section). Therefore, you should never use the EBP register as a general purpose register. Of course, this same discussion applies to the ESP register.

1.4 BEGIN..EXIT..EXITIF..END

HLA provides a structured GOTO via the EXIT and EXITIF statements. The EXIT and EXITIF statements let you exit a block of statements surrounded by a BEGIN..END pair. These statements behave much like the BREAK and BREAKIF statements (that let you exit from an enclosing loop) except, of course, they jump out of a BEGIN..END block rather than a loop. The EXIT and EXITIF statements are *structured gotos*

because they do not let you jump to an arbitrary point in the code, they only let you exit from a block delimited by the BEGIN..END pair.

The EXIT and EXITIF statements take the following forms:

```
exit identifier;
exitif( boolean_expression) identifier;
```

The *identifier* component at the end of these two statements must match the identifier following the BEGIN and END keywords (e.g., a procedure or program name). The EXIT statement immediately transfers control to the "end identifier;" clause. The EXITIF statement evaluates the boolean expression immediately following the EXITIF reserved word and transfers control to the specified END clause only if the expression evaluates true. If the boolean expression evaluates false, the control transfers to the first statement following the EXITIF statement.

If you specify the name of a procedure as the identifier for an EXIT statement, the program will return from the procedure upon encountering the EXIT statement[5]. *Note that the EXIT statement does not automatically restore any registers you pushed on the stack upon entry into the procedure.* If you need to pop data off the stack, you must do this before executing the EXIT statement.

If you specify the name of your main program as the identifier following the EXIT (or EXITIF) statement, your program will terminate upon encountering the EXIT statement. With EXITIF, your program will only terminate if the boolean expression evaluates true. Note that your program will still terminate even if you execute the "exit MainPgmName;" statement within a procedure nested inside your main program. You do not have to execute the EXIT statement in the main program to terminate the main program.

HLA lets you place arbitrary BEGIN..END blocks within your program, they are not limited to surrounding your procedures or main program. The syntax for an arbitrary BEGIN..END block is

```
begin identifier;

    <statements>

end identifier;
```

The identifier following the END clause must match the identifier following the corresponding BEGIN statement. Naturally, you can nest BEGIN..END blocks, but the identifier following an END clause must match the identifier following the previous unmatched BEGIN clause.

One interesting use of the BEGIN..END block is that it lets you easily escape a deeply nested control structure without having to completely restructure the program. Typically, you would use this technique to exit a block of code on some special condition and the TRY..ENDTRY statement would be inappropriate (e.g., you might need to pass values in registers to the outside code, an EXCEPTION clause can't guarantee register status). The following program demonstrates the use of the BEGIN..EXIT..END sequence to bail out of some deeply nested code.

```
program beginEndDemo;
#include( "stdlib.hhf" );

static
    m:uns8;
    d:uns8;
    y:uns16;

readonly
    DaysInMonth: uns8[13] :=
        [
```

5. This is true for the EXITIF statement as well, though, of course, the program will only exit the procedure if the boolean expression in the EXITIF statement evaluates true.


```
                0,  // No month zero.
                31, // Jan
                28, // Feb is a special case, see the code.
                31, // Mar
                30, // Apr
                31, // May
                30, // Jun
                31, // Jul
                31, // Aug
                30, // Sep
                31, // Oct
                30, // Nov
                31  // Dec
            ];

    begin beginEndDemo;

        forever

            try

                stdout.put( "Enter month, day, year: " );
                stdin.get( m, d, y );

              unprotected

                break;

              exception( ex.ValueOutOfRange )

                stdout.put( "Value out of range, please reenter", nl );

              exception( ex.ConversionError )

                stdout.put( "Illegal character in value, please reenter", nl );

            endtry;

        endfor;
        begin goodDate;

            mov( y, bx );
            movzx( m, eax );
            mov( d, dl );

            // Verify that the year is legal
            // (this program allows years 2000..2099.)

            if( bx in 2000..2099 ) then

                // Verify that the month is legal

                if( al in 1..12 ) then

                    // Quick check to make sure the
                    // day is half-way reasonable.

                    if( dl <> 0 ) then

                        // To verify that the day is legal,
                        // we have to handle Feb specially
```



```
                    // since this could be a leap year.

                    if( al = 2 ) then

                        // If this is a leap year, subtract
                        // one from the day value (to convert
                        // Feb 29 to Feb 28) so that the
                        // last day of Feb in a leap year
                        // will pass muster.  (This could
                        // set dl to zero if the date is
                        // Feb 1 in a leap year, but we've
                        // already handled dl=0 above, so
                        // we don't have to worry about this
                        // anymore.)

                        date.isLeapYear( bx );
                        sub( al, dl );

                    endif;

                    // Verify that the number of days in the month
                    // is valid.

                    exitif( dl <= DaysInMonth[ eax ] ) goodDate;

                endif;

            endif;

        endif;
        stdout.put( "You did not enter a valid date!", nl );

    end goodDate;

end beginEndDemo;
```

Program 1.8 Demonstration of BEGIN..EXIT..END Sequence

In this program, the "begin goodDate;" statement surrounds a section of code that checks to see if the date entered by a user is a valid date in the 100 years from 2000..2099. If the user enters an invalid date, it prints an appropriate error message, otherwise the program quits without further user interaction. While you could restructure this code to avoid the use of the EXITIF statement, the resulting code would probably be more difficult to understand. The nice thing about the design of the code is that it uses refinement to test for a legal date. That is, it tests to see if one component is legal, then tests to see if the next component of the date is legal, and works downward in this fashion. In the middle of the tests, this code determines that the date is legal. To restructure this code to work without the EXITIF (or other GOTO type instruction) would require using negative logic at each step (asking is this component *not* a legal date). That logic would be quite a bit more complex and much more difficult to read, understand, and verify. Hence, this example is preferable even if it contains a structured form of the GOTO statement.

Because the BEGIN..END statement uses a label, that the EXIT and EXITIF statements specify, you can nest BEGIN..END blocks and break out of several nested blocks with a single EXIT/EXITIF statement. Figure 1.1 provides a schematic of this capability.

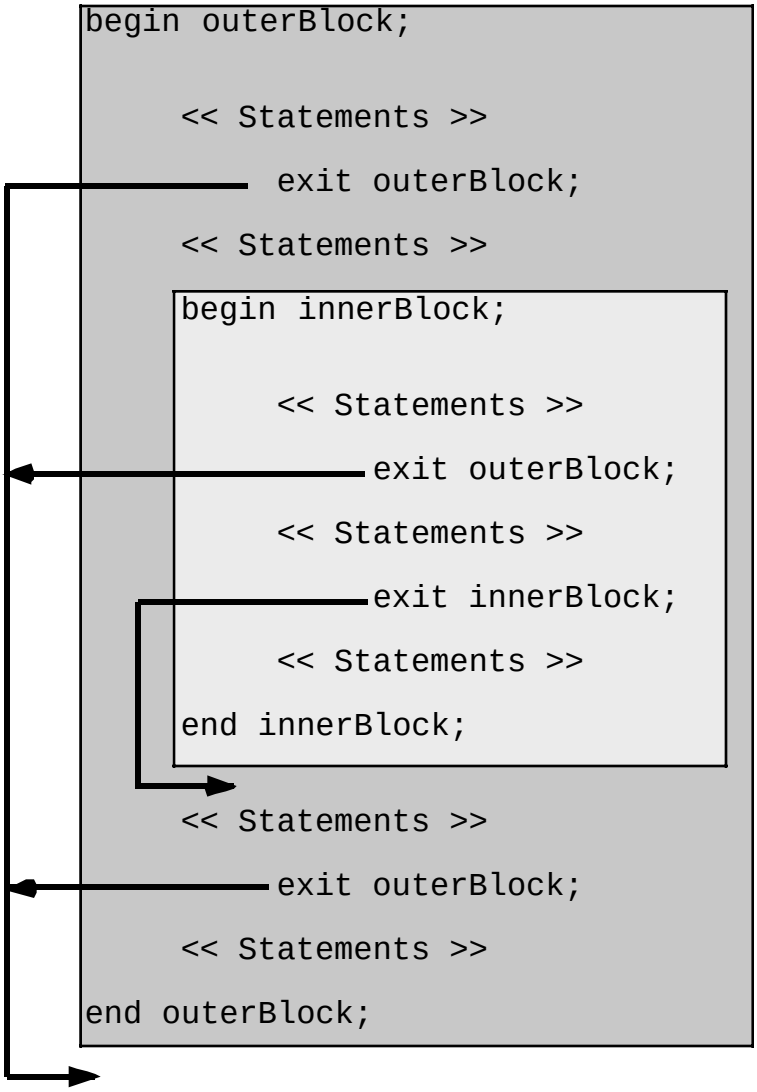


Figure 1.1 Nesting BEGIN..END Blocks

This ability to break out of nested BEGIN..END blocks is very powerful. Contrast this with the BREAK and BREAKIF statements that only let you exit the loop that immediately contains the BREAK or BREAKIF. Of course, if you need to exit out of multiple nested loops you won't be able to use the BREAK/BREAKIF statement to achieve this, but you can surround your loops with a BEGIN..END sequence and use the EXIT/EXITIF statement to leave those loops. The following program demonstrates how this could work, using the EXITIF statement to break out of two nested loops.

```
program brkNestedLoops;
#include( "stdlib.hhf" )

static
    i:int32;

begin brkNestedLoops;
```



```
    // DL contains the last value to print on each line.

    for( mov(0, dl ); dl <= 7; inc( dl )) do

        begin middleLoop;

            // DH ranges over the values to print.

            for( mov( 0, dh ); dh <= 7; inc( dh )) do

                // "i" specifies the field width
                // when printing DH, it also specifies
                // the maximum number of times to print DH.

                for( mov( 2, i ); i <= 4; inc( i )) do

                    // Break out of both inner loops
                    // when DH becomes equal to DL.

                    exitif( dh >= dl ) middleLoop;

                    // The following statement prints
                    // a triangular shaped object composed
                    // of the values that DH goes through.

                    stdout.puti8Size( dh, i, '.' );

                endfor;

            endfor;

        end middleLoop;
        stdout.newln();

    endfor;

end brkNestedLoops;
```

Program 1.9 Breaking Out of Nested Loops Using EXIT/EXITIF

1.5 CONTINUE..CONTINUEIF

The CONTINUE and CONTINUEIF statements are very similar to the BREAK and BREAKIF statements insofar as they affect control flow within a loop. The CONTINUE statement immediately transfers control to the point in the loop where the current iteration completes and the next iteration begins. The CONTINUEIF statement first checks a boolean expression and transfers control if the expression evaluates false.

The phrase "where the current iteration completes and the next iteration begins" has a different meaning for nearly every loop in HLA. For the WHILE..ENDWHILE loop, control transfers to the top of the loop at the start of the test for loop termination. For the FOREVER..ENDFOR loop, control transfers to the top of the loop (no test for loop termination). For the FOR..ENDFOR loop, control transfers to the bottom of the loop where the increment operation occurs (i.e., to execute the third component of the FOR loop). For the REPEAT..UNTIL loop, control transfers to the bottom of the loop, just before the test for loop termination. The following diagrams show how the CONTINUE statement behaves.

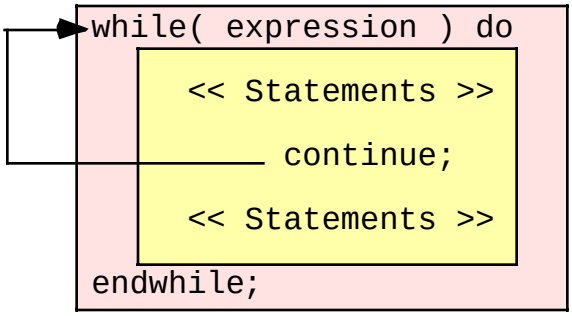


Figure 1.2 Behavior of CONTINUE in a WHILE Loop

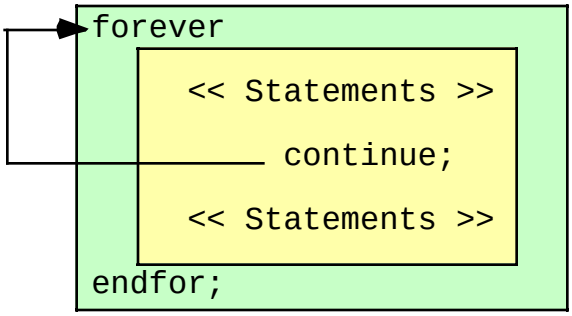
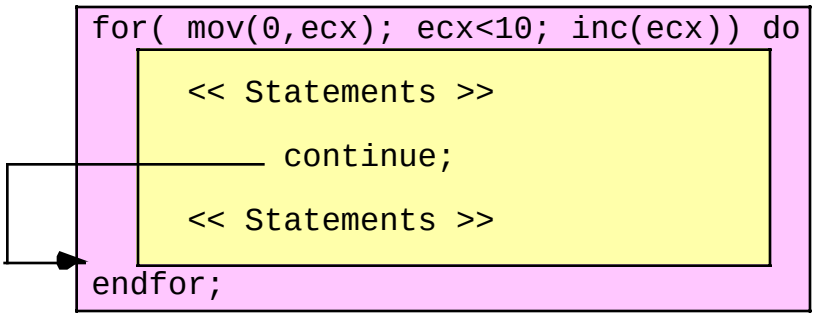
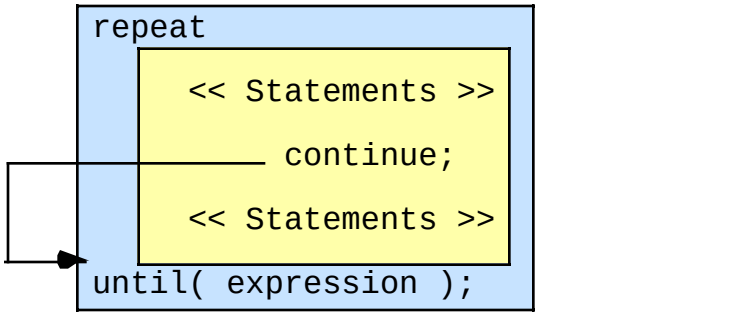


Figure 1.3 Behavior of CONTINUE in a FOREVER Loop




```
// Note that CONTINUE will cause
// the execution of the inc(ecx)
// instruction.
```

Figure 1.4 Behavior of CONTINUE in a FOR Loop




```
// Note that CONTINUE will transfer
// control to the test for loop
// termination.
```

Figure 1.5 Behavior of CONTINUE in a REPEAT..UNTIL Loop

It turns out that CONTINUE is rarely needed in common programs. Most of the time an IF..ENDIF statement provides the same functionality as CONTINUE (or CONTINUEIF) while being much more readable. Nevertheless, there are a few instances you will encounter where the CONTINUE or CONTINUEIF statements provide exactly what you need. However, if you find yourself using the CONTINUE or CONTINUEIF statements on a frequent basis, you should probably reconsider the logic in your programs.

1.6 SWITCH..CASE..DEFAULT..ENDSWITCH

The HLA language does not provide a selection statement similar to SWITCH in C/C++ or CASE in Pascal/Delphi. This omission was intentional; by leaving the SWITCH statement out of the language it is possible to demonstrate how to extend the HLA language by adding new control structures. In the chapters on Macros and the HLA Compile-time Language, this text will demonstrate how you can add your own statements, like SWITCH, to the HLA language. In the meantime, although HLA does not provide a SWITCH statement, the HLA Standard Library provides a macro that provides this capability for you. If you include the “hll.hhf” header file (which “stdlib.hhf” automatically includes for you), then you can use the SWITCH statement exactly as though it were a part of the HLA language.

The HLA Standard Library SWITCH statement has the following syntax:

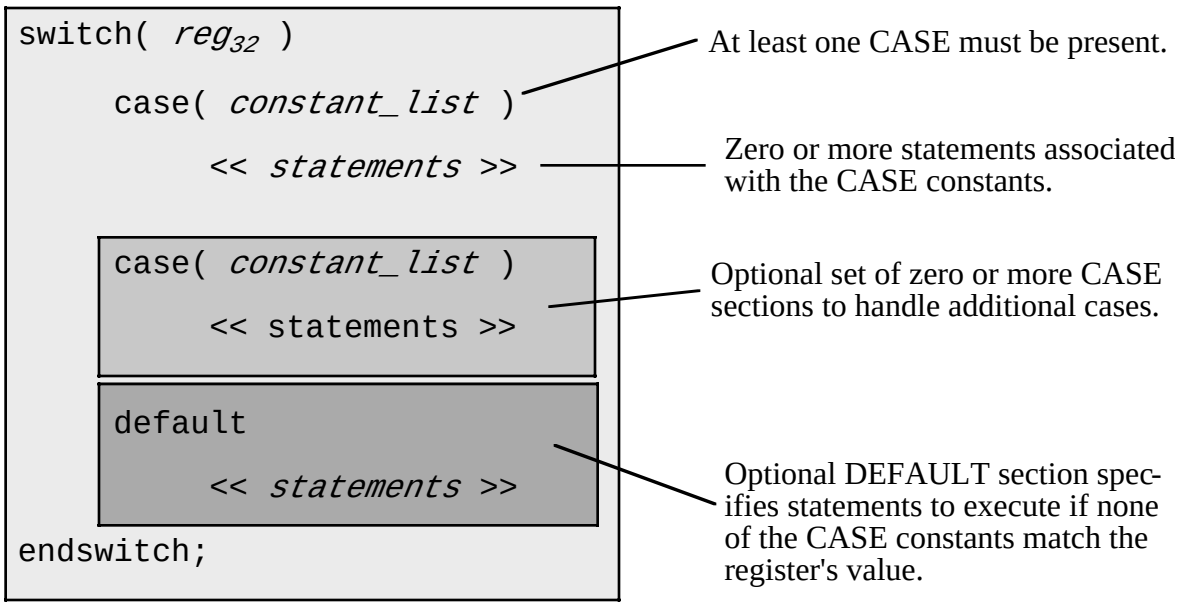


Figure 1.6 Syntax for the SWITCH..CASE..DEFAULT..ENDSWITCH Statement

Like most HLA high level language statements, there are several restrictions on the SWITCH statement. First of all, the SWITCH clause does not allow a general expression as the selection value. The SWITCH clause will only allow a value in a 32-bit general purpose register. In general you should only use EAX, EBX, ECX, EDX, ESI, and EDI since EBP and ESP are reserved for special purposes.

The second restriction is that the HLA SWITCH statement supports a maximum of 256 different case values. Few SWITCH statements use anywhere near this number, so this shouldn't prove to be a problem. Note that each CASE in Figure 1.6 allows a constant list. This could be a single unsigned integer value or a comma separated list of values, e.g.,

```
        case( 10 )
```

-or-

```
        case( 5, 6, 8 )
```

Each value in the list of constants counts as one case constant towards the maximum of 256 possible constants. So the second CASE clause above contributes three constants towards the total maximum of 256 constants.

Another restriction on the HLA SWITCH statement is that the difference between the largest and smallest values in the case list must be 1,024. Therefore, you cannot have CASEs (in the same SWITCH statement) with values like 1, 10, 100, 1,000, and 10,000 since the difference between the smallest and largest values, 9999, exceeds 1,024.

The DEFAULT section, if it appears in a SWITCH statement, must be the last section in the SWITCH statement. If no DEFAULT section is present and the value in the 32-bit register does not match one of the CASE constants, then control transfers to the first statement following the ENDSWITCH clause.

Here is a typical example of a SWITCH..ENDSWITCH statement:

```
switch( eax )

    case( 1 )

        stdout.put( "Selection #1:" nl );
        << Code for case #1 >>
```



```
        case( 2, 3 )

            stdout.put( "Selections (2) and (3):" nl );
            << code for cases 2 & 3 >>

        case( 5,6,7,8,9 )

            stdout.put( "Selections (5)..(9)" nl );
            << code for cases 5..9 >

        default

            stdout.put( "Selection outside range 1..9" nl );
            << default case code >>

    endswitch;
```

The SWITCH statement in a program lets your code choose one of several different code paths depending upon the value of the case selection variable. Among other things, the SWITCH statement is ideal for processing user input that selects a menu item and executes different code depending on the user's selection.

Later in this volume you will see how to implement a SWITCH statement using low-level machine instructions. Once you see the implementation you will understand the reasons behind these limitations in the SWITCH statement. You will also see why the CASE constants must be constants and not variables or registers.

1.7 Putting It All Together

This chapter completes the discussion of the high level control structures built into the HLA language or provided by the HLA Standard Library (i.e., SWITCH). First, this chapter gave a complete discussion of the TRY..ENDTRY and RAISE statements. Although Volume One provided a brief discussion of exception handling and the TRY..ENDTRY statement, this particular statement is too complex to fully describe earlier in this text. This chapter completes the discussions of this important statement and suggests ways to use it in your programs that will help make them more robust.

After discussing TRY..ENDTRY and RAISE, this chapter discusses the EXIT and EXITIF statements and describes how to use them to prematurely exit a procedure or the program. This chapter also discusses the BEGIN..END block and describes how to use the EXIT and EXITIF statements to exit such a block. These statements provide a structured GOTO (JMP) in the HLA language.

Although you will not use them as frequently as the BREAK and BREAKIF statements, the CONTINUE and CONTINUEIF statements are helpful once in a while for jumping over the remainder of a loop body and starting the next loop iteration. This chapter discusses the syntax of these statements and warns against overusing them.

This chapter concludes with a discussion of the SWITCH/CASE/DEFAULT/ENDCASE statement. This statement isn't actually a part of the HLA language - instead it is provided by the HLA Standard Library as an example of how you can extend the language. If you would like details on extending the HLA language yourself, see the chapter on "Domain Specific Languages."

