14 HLA Language Reference and User Manual

14.1 High Level Language Statements

HLA provides several control structures that provide a high level language flavor to assembly language programming. The statements HLA provides are

```
try..unprotect..exception..anyexception..endtry
try..always..endtry
raise
if..then..elseif..else..endif
switch..case..default..endswitch
while..endwhile
repeat..until
for..endfor
foreach..endfor
forever..endfor
break, breakif
continue, continueif
begin..end, exit, exitif

JT
JF
```

These HLL statements provide two basic improvements to assembly language programs: (1) they make many algorithms much easier to read; (2) they eliminate the need to create tons of labels in a program (which also helps make the program easier to read).

Generally, these instructions are "macros" that emit one or two machine instructions. Therefore, these instructions are not always as flexible as their HLL counterparts. Nevertheless, they are suitable for about 85% of the uses people typically have for these instructions.

Do keep in mind, that even though these statements compile to efficient machine code, writing assembly language using a HLL mindset produces intrinsically inefficient programs. If speed or size is your number one priority in a program, you should be sure you understand exactly which instructions each of these statements emits before using them in your code.

The JT and JF statements are actually "medium level language" statements. They are intended for use in macros when constructing other HLL control statements; they are not intended for use as standard statements in your program (not that they don't work, they're just not true HLL statements).

Note: The FOREACH..ENDFOR loop is mentioned above only for completeness. The full discussion of the FOREACH..ENDFOR statement appears a little later in the section on iterators.

14.2 Exception Handling in HLA:try..exception..endtry

HLA uses the TRY..EXCEPTION..ENDTRY and RAISE statements to implement exception handling. The syntax for these statements is as follows:

```
try
   << HLA Statements to execute >>

<<  unprotected // Optional unprotected section.
   << HLA Statements to execute >>
>>
```

```
exception( const1 )

   << Statements to execute if exception const1 is raised >>

<< optional exception statements for other exceptions >>

<<  anyexception //Optional anyexception section.
   << HLA Statements to execute >>
>>

endtry;


raise( const2 );
```

Const1 and *const2* must be unsigned integer constants. Usually, these are values defined in the excepts.hhf header file. Some examples of predefined values include the following:

```
ex.StringOverflow
ex.StringIndexError

ex.ValueOutOfRange
ex.IllegalChar
ex.ConversionError

ex.BadFileHandle
ex.FileOpenFailure
ex.FileCloseError
ex.FileWriteError
ex.FileReadError
ex.DiskFullError
ex.EndOfFile

ex.MemoryAllocationFailure

ex.AttemptToDerefNULL

ex.WidthTooBig
ex.TooManyCmdLnParms

ex.ArrayShapeViolation
ex.ArrayBounds

ex.InvalidDate
ex.InvalidDateFormat
ex.TimeOverflow
ex.AssertionFailed
ex.ExecutedAbstract
```

Hardware related exception values:

```
ex.AccessViolation
ex.Breakpoint
ex.SingleStep

ex.PrivInstr
```

```
ex.IllegalInstr

ex.BoundInstr
ex.IntoInstr

ex.DivideError

ex.fDenormal
ex.fDivByZero
ex.fInexactResult
ex.fInvalidOperation
ex.fOverflow
ex.fStackCheck
ex.fUnderflow

ex.InvalidHandle
ex.StackOverflow

ex.ControlC
```

This list is constantly changing as the HLA Standard Library grows, so it is impossible to provide a compete list of standard exceptions at this time. Please see the excepts.hhf header file for a complete list of standard exceptions. As this was being written, the *NIX-specific exceptions (signals) had not been added to the list. See the excepts.hhf file on your *NIX system to see if these have been added. Note that not all OSes support every hardware-related exception value.

The HLA Standard Library currently reserves exception numbers zero through 1023 for its own internal use. User-defined exceptions should use an integer value greater than or equal to 1024 and less than or equal to 65535 ($FFFF). Exception value $10000 and above are reserved for use by Windows Structured Exception Handler and *NIX signals.

The TRY..ENDTRY statement contains two or more blocks of statements. The statements to *protect* immediately follow the TRY reserved word. During the execution of the protected statements, if the program encounters the first exception block, control immediately transfers to the first statement following the endtry reserved word. The program will skip all the statements in the exception blocks.

If an exception occurs during the execution of the protected block, control is immediate transferred to an exception handling block that begins with the exception reserved word and the constant that specifies the type of exception.

Example:

```
repeat

    mov( false, GoodInput );
    try
        stdout.put( "Enter an integer value:" );
        stdin.get( i );
        mov( true, GoodInput );

    exception( ex.ValueOutOfRange )

        stdout.put( "Numeric overflow, please reenter ", nl );

    exception( ex.ConversionError )

        stdout.put( "Conversion error, please reenter", nl );
```

```
    endtry;

until( GoodInput = true );
```

In this code, the program will repeatedly request the input of an integer value as long as the user enters a value that is out of range (+/- 2 billion) or as long as the user enters a value containing illegal characters.

TRY..ENDTRY statements can be *nested*. If an exception occurs within a nested TRY protected block, the EXCEPTION blocks in the innermost try block containing the offending statement get first shot at the exceptions. If none of the EXCEPTION blocks in the enclosing TRY..ENDTRY statement handle the specified exception, then the next innermost TRY..ENDTRY block gets a crack at the exception. This process continues until some exception block handles the exception or there are no more TRY..ENDTRY statements.

If an exception goes unhandled, the HLA run-time system will handle it by printing an appropriate error message and aborting the program. Generally, this consists of printing "Unhandled Exception" (or a similar message) and stopping the program. If you include the excepts.hhf header file in your main program, then HLA will automatically link in a somewhat better default exception handler that will print the number (and name, if known) of the exception before stopping the program.

Note that TRY..ENDTRY blocks are dynamically nested, not statically nested. That is, a program must actually execute the TRY in order to activate the exception handler. You should never jump into the middle of a protected block, skipping over the TRY. Doing so may produce unpredictable results.

You should not use the TRY..ENDTRY statement as a general control structure. For example, it will probably occur to someone that one could easily create a switch/case selection statement using TRY..ENDTRY as follows:

```
try
    raise( SomeValue );

    exception( case1_const)
        <code for case 1>

    exception( case2_const)
        <code for case 2>

    etc.
endtry
```

While this might work in some situations, there are two problems with this code.

First, if an exception occurs while using the TRY..ENDTRY statement as a switch statement, the results may be unpredictable. Second, HLA's run-time system assumes that exceptions are rare events. Therefore, the code generated for the exception handlers doesn't have to be efficient. You will get much better results implementing a switch/case statement using a table lookup and indirect jump (see the Art of Assembly) rather than a TRY..ENDTRY block.

Warning: The TRY statement pushes data onto the stack upon initial entry and pops data off the stack upon leaving the TRY..ENDTRY block. Therefore, jumping into or out of a TRY..ENDTRY block is an absolute no-no. As explained so far, then, there are only two reasonable ways to exit a TRY statement, by falling off the end of the protected block or by an exception (handled by the TRY statement or a surrounding TRY statement).

The UNPROTECTED clause in the TRY..ENDTRY statement provides a safe way to exit a TRY..ENDTRY block without raising an exception or executing all the statements in the protected portion of the TRY..ENDTRY statement. An unprotected section is a sequence of statements, between the protected block and the first exception handler, that begins with the keyword UNPROTECTED. E.g.,

```
try
```

```
    << Protected HLA Statements >>

  unprotected

    << Unprotected HLA Statements >>

  exception( SomeExceptionID )

    << etc. >>

endtry;
```

Control flows from the protected block directly into the unprotected block as though the UNPROTECTED keyword were not present. However, between the two blocks HLA compiler-generated code removes the data pushed on the stack. Therefore, it is safe to transfer control to some spot outside the TRY..ENDTRY statement from within the unprotected section.

If an exception occurs in an unprotected section, the TRY..ENDTRY statement containing that section does not handle the exception. Instead, control transfers to the (dynamically) nesting TRY..ENDTRY statement (or to the HLA run-time system if there is no enclosing TRY..ENDTRY).

If you're wondering why the UNPROTECTED section is necessary (after all, why not simply put the statements in the UNPROTECTED section after the ENDTRY?), just keep in mind that both the protected sequence and the handled exceptions continue execution after the ENDTRY. There may be some operations you want to perform after exceptions are released, but only if the protected block finished successfully. The UNPROTECTED section provides this capability. Perhaps the most common use of the UNPROTECTED section is to break out of a loop that repeats a TRY..ENDTRY block until it executes without an exception occuring. The following code demonstrates this use:

```
forever

   try

          stdout.put( "Enter an integer: " );
          stdin.geti8();   // May raise an exception.

      unprotected

          break;

      exception( ex.ValueOutOfRange )

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

      exception( ex.ConversionError )

          stdout.put( "Value contained illegal chars" nl );

   endtry;

endfor;
```

This simple example repeatedly asks the user to input an `int8` integer until the value is legal and within the range of valid integers.

Another clause in the TRY..EXCEPT statement is the *ANYEXCEPTION* clause. If this clause is present, it must be the last clause in the *TRY..EXCEPT* statement, e.g.,

```
try
   << protected statements >>

<<
   unprotected

      Optional unprotected statements
>>

<< exception( constant ) // Note: may be zero or more of
                            of these.

      Optional exception handler statements
>>

   anyexception
      << Exception handler if none of the others execute >>

endtry;
```

Without the ANYEXCEPTION clause present, if the program raises an exception that is not specifically handled by one of the exception clauses, control transfers to the enclosing TRY..ENDTRY statement. The ANYEXCEPTION clause gives a TRY..ENDTRY statement the opportunity to handle any exception, even those that are not explicitly listed. Upon entry into the ANYEXCEPTION block, the EAX register contains the actual exception number.

14.3 Exception Handling in HLA:try..always..endtry

The HLA TRY..ALWAYS..ENDTRY statement is a variant of the try..endtry statement that has a single ALWAYS block (no EXCEPTION or ANYEXCEPTION clauses). This statement takes the following form:

```
try
   << protected statements >>

   always

      Statements that always execute

endtry;
```

The ALWAYS block in this statement always executes, whether an exception occurs or no exception occurs. The ALWAYS block is useful for executing code that must happen regardless of the successful execution of the protected statements. Examples including closing files that were opened prior to the TRY statement, freeing memory allocated on the heap, leaving critical sections, and so on.

If the ALWAYS block executes because an exception occurred, then the code will re-raise the exception immediately after the AWAYS block finishes execution. An outer TRY..ENDTRY statement can handle the exception at that point.

If no exeception occurs, then the ALWAYS block executes immediately after the last protected statement and once the ALWAYS block finishes, control resumes with the first statement after the ENDTRY.

Note that there is no way inside the ALWAYS block to determine if execution occurs because of an exception or because the protected statements completed execution without raising an exception. If you absolutely, positively, need to do something special if an exception occurs, then

insert a TRY..ANYEXCEPTION..ENDTRY statement around the protected statements or enclose the TRY..ALWAYS..ENDTRY statement inside a TRY .. EXCEPTION .. ANYEXCEPTION .. ENDTRY statement:

The following code executes the ANYEXCEPTION block prior to executing the code in the ALWAYS section:

```
try
    try
        << protected statements >>

        anyexception

            // Handle the statement before executing the ALWAYS clause
            raise( eax );

    endtry;


    always

        // Statements that always execute

endtry;
```

The following version excutes the ALWAYS block first and then an ANYEXCEPTION block if there was an exception

```
try
    try
        << protected statements >>

        always

            // Statements that always execute

    endtry;

  anyexception

        // Statements that execute after ALWAYS bloc
        // if there was an exception

endtry;
```

14.4 Exception Handling in HLA:raise

The HLA RAISE statement generates an exception. The single parameter is an 8, 16, or 32-bit ordinal constant. Control is (ultimately) transferred to the first (most deeply nested) TRY..ENDTRY statement that has a corresponding exception handler (including ANYEXCEPTION).

If the program executes the RAISE statement within the protected block of a TRY..ENDTRY statement, then the enclosing TRY..ENDTRY gets first shot at handling the exception. If the RAISE statement occurs in an UNPROTECTED block, or in an exception handler (including ANYEXCEPTION), then the next higher level (nesting) TRY..ENDTRY statement will handle the exception. This allows *cascading* exceptions; that is, exceptions that the system handles in two or more exception handlers. Consider the following example:

```
try
   << Protected statements >>

  exception( someException )
   << Code to process this exception >>

   // The following re-raises this exception, allowing
   // an enclosing try..endtry statement to handle
   // this exception as well as this handler.

   raise( someException );

  << Additional, optional, exception handlers >>

endtry;
```

14.5 IF..THEN..ELSEIF..ELSE..ENDIF Statement in HLA

HLA provides a limited IF..THEN.ELSEIF..ELSE..ENDIF statement that can help make your programs easier to read. For the most part, HLA's if statement provides a convenient substitute for a CMP and a conditional branch instruction pair (or chain of such instructions when employing ELSEIF's).

The generic syntax for the HLA if statement is the following:

```
if( conditional_expression ) then

   << Statements to execute if expression is true >>

endif;


if( conditional_expression ) then

   << Statements to execute if expression is true >>

else

   << Statements to execute if expression is false >>

endif;

if( expr1 ) then

   << Statements to execute if expr1 is true >>

elseif( expr2 ) then

   << Statements to execute if expr1 is false
        and expr2 is true >>

endif;


if( expr1 ) then
```

```
    << Statements to execute if expr1 is true >>

elseif( expr2 ) then

    << Statements to execute if expr1 is false
        and expr2 is true >>

else

    << Statements to execute if both expr1 and
        expr2 are false >>

endif;
```

Note: HLA's if statement allows multiple ELSEIF clauses. All ELSEIF clauses must appear between IF clause and the ELSE clause (if present) or the ENDIF (if an ELSE clause is not present).

See the next section for a discussion of valid boolean expressions within the IF statement (this section appears first because the section on boolean expressions uses IF statements in its examples).

14.6 Boolean Expressions for High-Level Language Statements

The primary limitation of HLA's IF and other HLL statements has to do with the conditional expressions allowed in these statements. These expressions must take one of the following forms:

```
operand1 relop operand2

register in constant .. constant
register not in constant .. constant

memory in constant .. constant
memory not in constant .. constant

reg8 in CSet_Constant
reg8 in CSet_Variable

reg8 not in CSet_Constant
reg8 not in CSet_Variable

register
!register

memory
!memory

Flag

( boolean_expression )
!( boolean_expression )

boolean_expression && boolean_expression

boolean_expression || boolean_expression
```

For the first form, "operand1 *relop* operand2", *relop* is one of:

```
=  or ==          (either one, both are equivalent)
<> or !=          (either one)
<
<=
>
>=
```

`Operand1` and `operand2` must be operands that would be legal for a "`cmp(operand1, operand2);`" instruction.

For the IF statement, HLA emits a CMP instruction with the two operands specified and an appropriate conditional jump instruction that skips over the statements following the "THEN" reserved word if the condition is false. For example, consider the following code:

```
if( al = 'a' ) then

    stdout.put( "Option 'a' was selected", nl );

endif;
```

Like the CMP instruction, the two operands cannot both be memory operands.

Unlike the conditional branch instructions, the six relational operators cannot differentiate between signed and unsigned comparisons (for example, HLA uses "<" for both signed and unsigned less than comparisons). Since HLA must emit different instructions for signed and unsigned comparisons, and the relational operators do not differentiate between the two, HLA must rely upon the types of the operands to determine which conditional jump instruction to emit.

By default, HLA emits unsigned conditional jump instructions (i.e., JA, JAE, JB, JBE, etc.). If either (or both) operands are signed values, HLA will emit signed conditional jump instructions (i.e., JG, JGE, JL, JLE, etc.) instead.

HLA considers the 80x86 registers to be *unsigned*. This can create some problems when using the HLA if statement. Consider the following code:

```
if( eax < 0 ) then

    << do something if eax is negative >>

endif;
```

Since neither operand is a signed value, HLA will emit the following code:

```
        cmp( eax, 0 );
        jnb SkipThenPart;
        << do something if eax is negative >>
SkipThenPart:
```

Unfortunately, it is never the case that the value in EAX is below zero (since zero is the minimum unsigned value), so the body of this if statement never executes. Clearly, the programmer intended to use a signed comparison here. The solution is to ensure that at least one operand is signed. However, as this example demonstrates, what happens when both operands are intrinsically unsigned?

The solution is to use coercion to tell HLA that one of the operands is a signed value. In general, it is always possible to coerce a register so that HLA treats it as a signed, rather than unsigned, value. The IF statement above could be rewritten (correctly) as

```
if( (type int32 eax) < 0 ) then

    << do something if eax is negative >>

endif;
```

HLA will emit the JNL instruction (rather than JNB) in this example. Note that if either operand is signed, HLA will emit a signed condition jump instruction. Therefore, it is not necessary to coerce both unsigned operands in this example.

The second form of a conditional expression that the IF statement accepts is a register or memory operand followed by "in" and then two constants separated by the ".." operator, e.g.,

```
if( al in 0..10 ) then ...
```

This code checks to see if the first operand is in the range specified by the two constants. The constant value to the left of the ".." must be less than the constant to the right for this expression to make any sense. The result is true if the operand is within the specified range. For this instruction, HLA emits a pair of compare and conditional jump instructions to test the operand to see if it is in the specified range.

HLA also allows a exclusive range test specified by an expression of the form:

```
if( al not in 0..10 ) then ...
```

In this case, the expression is true if the value in AL is outside the range 0..10.

In addition to integer ranges, HLA also lets you use the IN operator with CSET constants and variables. The generic form is one of the following:

reg$_8$ in *CSetConst*
reg$_8$ not in *CSetConst*
reg$_8$ in *CSetVariable*
reg$_8$ not in *CSetVariable*

For example, a statement of the form "if(al in {'a'..'z'}) then ..." checks to see if the character in the AL register is a lower case alphabetic character. Similarly,

```
if( al not in {'a'..'z', 'A'..'Z'}) then...
```

checks to see if AL is not an alphabetic character.

The fifth form of a conditional expression that the IF statement accepts is a single register name (eight, sixteen, or thiry-two bits). The IF statement will test the specified register to see if it is zero (false) or non-zero (true) and branches accordingly. If you specify the not operator ("!") before the register, HLA reverses the sense of this test.

The sixth form of a conditional expression that the IF staement accepts is a single memory location. The type of the memory location must be boolean, byte, word, or dword. HLA will emit code that compares the specified memory location against zero (false) and generate an appropriate branch depending upon the value in the memory location. If you put the not operator ("!") before the variable, HLA reverses the sense of the test.

The seventh form of a conditional expression that the IF statement accepts is a Flags register bit or other condition code combination handled by the 80x86 conditional jump instructions. The following reserved words are acceptable as IF statement expressions:

```
@c, @nc, @o, @no, @z, @nz, @s, @ns, @a, @na, @ae, @nae, @b, @nb, @be,
@nbe, @l, @nl, @g, @ne, @le, @nle, @ge, @nge, @e, @ne
```

These items emit an appropriate jump (of the opposite sense) around the THEN portion of the IF statement if the condition is false.

If you supply any legal boolean expression in parenthesis, HLA simply uses the value of the internal expression for the value of the whole expression. This allows you to override default precedence for the AND, OR, and ! operators.

The !(boolean_expression) evaluates the expression and does just the opposite. That is, if the interior expression is false, then !(boolean_expression) is true and vice versa. This is mainly useful with conjunction and disjunction since all of the other interesting terms already allow the not operator in front of them. Note that in general, the "!" operator must precede some parentheses. You cannot say "! AX < BX", for example.

Originally, HLA did not include support for the conjunction (&&) and disjunction (||) operators. This was explicitly left out of the design so that beginning students would be forced to rethink their logical operations in assembly language. Unfortunately, it was so inconvenient not to have these operators that they were eventually added. So a compromise was made: these operators were added to HLA but "The Art of Assembly Language Programming/Win32 Edition" doesn't bother to mention them until an advanced chapter on control structures.

The conjunction and disjunction operators are the operators && and ||. They expect two valid HLA boolean expressions around the operator, e.g.,

```
eax < 5 && ebx <> ecx
```

Since the above forms a valid boolean expression, it, too, may appear on either side of the && or | operator, e.g.,

```
eax < 5 && ebx <> ecx || !dl
```

HLA gives && higher precedence than ||. Both operators are left-associative so if multiple operators appear within the same expression, they are evaluated from left to right if the operators have the same precedence. Note that you can use parentheses to override HLA's default precedence.

One wrinkle with the addition of && and || is that you need to be careful when using the flags in a boolean expression. For example, "eax < ecx && @nz" hides the fact that HLA emits a compare instruction that affects the Z flag. Hence, the "@nz" adds nothing to this expression since EAX must not equal ECX if eax<ecx. So take care when using && and ||.

HLA uses short-circuit evaluation when evaluating expressions containing the conjunction and disjunction operators. For the && operator, this means that the resulting code will not compute the right-hand expression if the left-hand expression evaluates false. Similarly, the code will not compute the right-hand expression of the || operator if the left-hand expression evaluates true.

Note that the evaluation of complex boolean expressions involving the !(---), &&, and || operators does not change any register or memory values. HLA strictly uses flow control to implement these operations.

Note that the "&" and "|" operators are for compile-time only expression while the "&&" and "||" operators are for run-time boolean expressions. These two groups of operators are not synonyms and you cannot use them interchangably.

If you would prefer to use a less abstract scheme to evaluate boolean expressions, one that lets you see the low-level machine instructions, HLA provides a solution that allows you to write code to evaluate complex boolean expressions within the HLL statements using low-level instructions. Consider the following syntax:

```
if
(#{
   <<arbitrary HLA statements >>
}#) then

   << "True" section >>

else //or elseif...

   << "False" section >>
```

```
endif;
```

The "#{" and "}#" brackets tell HLA that an arbitrary set of HLA statements will appear between the braces. HLA will *not* emit any code for the IF expression. Instead, it is the programmer's responsibility to provide the appropriate test code within the "#{---}#" section. Within the sequence, HLA allows the use of the boolean constants "`true`" and "`false`" as targets of conditional jump instructions. Jumping to the "`true`" label transfers control to the true section (i.e., the code after the "THEN" reserved word). Jumping to the "`false`" label transfers control to the false section. Consider the following code that checks to see if the character in AL is in the range "a".."z":

```
if
(#{
   cmp( al, 'a' );
   jb false;
   cmp( al, 'z' );
   ja false;
}#) then

   << code to execute if AL in {'a'..'z'} goes here >>

endif;
```

With the inclusion of the #{---}# operand, the IF statement becomes much more powerful, allowing you to test any condition possible in assembly language. Of course, the #{---}# expression is legal in the ELSEIF expression as well as the IF expression.

It would be a good idea for you to write some code using the HLA if statement and study the MASM code produced by HLA for these IF statements. By becoming familiar with the code that HLA generates for the IF statement, you will have a better idea about when it is appropriate to use the if statement versus standard assembly language statements.

14.7 WHILE..WELSE..ENDWHILE Statement in HLA

The while..endwhile statement allows the following syntax:

```
while( boolean_expression ) do

   << while loop body>>

endwhile;


while( boolean_expression ) do

   << while loop body>>
else

   << Code to execute when expression is false >>

endwhile;


while(#{ HLA_statements }#) do

   << while loop body>>
```

```
endwhile;

while(#{ HLA_statements }#) do

    << while loop body>>

welse

    << Code to execute when expression is false >>

endwhile;
```

The WHILE statement allows the same boolean expressions as the HLA IF statement. Like the HLA IF statement, HLA allows you to use the boolean constants "`true`" and "`false`" as labels in the #{...}# form of the WHILE statement above. Jumping to the `true` label executes the body of the while loop, jumping to the `false` label exits the while loop.

For the "while(expr) do" forms, HLA moves the test for loop termination to the bottom of the loop and emits a jump at the top of the loop to transfer control to the termination test. For the "while(#{stmts}#)" form, HLA compiles the termination test at the top of the emitted code for the loop. Therefore, the standard WHILE loop may be slightly more efficient (in the typical case) than the hybrid form.

The HLA while loop supports an optional "welse" (while-else) section. The while loop will execute the code in this section only when then the expression evaluates false. Note that if you exit the loop vra a "break" or "breakif" statement the welse section does not execute. This provides logic that is sometimes useful when you want to do something different depending upon whether you exit the loop via the expression going false or by a break statement.

14.8 REPEAT..UNTIL Statement in HLA

HLA's REPEAT..UNTIL statement uses the following syntax:

```
repeat

    << statements to execute repeatedly >>

until( boolean_expression );


repeat

    << statements to execute repeatedly >>

until(#{ HLA_statements }#);
```

For those unfamiliar with REPEAT..UNTIL, the body of the loop always executes at least once with the test for loop termination ocurring at the bottom of the loop. The REPEAT..UNTIL loop (unlike C/C++'s do..while statement) terminates loop execution when the expression is true (that is, REPEAT..UNTIL repeats while the expression is false).

As you can see, the syntax for this is very similar to the WHILE loop. About the only major difference is the fact that jump to the "`true`" label in the #{---}# sequence exits the loop while jumping to the "`false`" label in the #{---}# sequence transfers control back to the top of the loop.

14.9 The FOR..ENDFOR Statement in HLA

The HLA for..endfor statement is very similar to the C/C++ for loop. The FOR clause consists of three components:

```
for( initialize_stmt; if_boolean_expression; increment_statement ) do
```

The *initialize_statement* component is a single machine instruction. This instruction typically initializes a loop control variable. HLA emits this statement before the loop body so that it executes only once, before the test for loop termination.

The *if_boolean_expression* component is a simple boolean expression (same syntax as for the IF statement). This expression determines whether the loop body executes. Note that the FOR statement tests for loop termination before executing the body of the loop.

The *increment_statement* component is a single machine instruction that HLA emits at the bottom of the loop, just before jumping back to the top of the loop. This instruction is typically used to modify the loop control variable.

The syntax for the HLA for statement is the following:

```
for( initStmt; BoolExpr; incStmt ) do

   << loop body >>

endfor;

-or-

for( initStmt; BoolExpr; incStmt ) do

   << loop body >>

felse

   << statements to execute when BoolExpr evaluates false >>

endfor;
```

Semantically, this statement is identical to the following while loop:

```
initStmt;
while( BoolExpr ) do
   << loop body >>
   incStmt;
endwhile;

-or-

initStmt;
while( BoolExpr ) do
   << loop body >>
   incStmt;

welse

   << statements to execute when BoolExpr evaluates false >>
endwhile;
```

Note that HLA does not include a form of the FOR loop that lets you bury a sequence of statements inside the boolean expression. Use the WHILE loop if you want to do that. If this is inconvenient, you can always create your own version of the FOR loop using HLA's macro facilities.

The FELSE section in the FOR..FELSE..ENDFOR loop executes when the boolean expression evaluates false. Note that the FELSE section does not execute if you break out of the FOR loop with a BREAK or BREAKIF statement. You can use this fact to do different logic depending on whether the code exits the loop via the boolean expression going false or via some sort of BREAK.

14.10 The FOREVER..ENDFOR Statement in HLA

The forever statement creates an infinite loop. Its syntax is

```
forever

   << Statements to execute repeatedly >>

endfor
```

This HLA statement simply emits a single JMP instruction that unconditionally transfers control from the ENDFOR clause back up to the beginning of the loop.

In addition to creating infinite loops, the FOREVER..ENDFOR loop is very useful for creating loops that test for loop termination somewhere in the middle of the loop's body. For more details, see the BREAK and BREAKIF statements, next.

14.11 The BREAK and BREAKIF Statements in HLA

The BREAK and BREAKIF statements allow you to exit a loop at some point other than the normal test for loop termination. These two statements allow the following syntax:

```
break;
breakif( boolean_expression );
breakif(#{ stmts }#);
```

There are two very important things to note about these statements. First, unlike many HLA machine instructions, you do not follow the BREAK statement with a pair of empty parentheses. The 80x86 machine instructions behave like compile-time functions, so it made sense to require empty parentheses after those instructions. The HLA HLL statements do not behave like compile-time functions; the lack of parentheses after BREAK (and other HLL statements, e.g., ELSE) makes sense here if you think about it for a moment.

The second thing to note is that the BREAK and BREAKIF statements are legal only inside WHILE, FOREACH, FOREVER, and REPEAT loops. HLA does not recognize loops you've coded yourself using discrete assembly language instructions (of course, you can probably write a macro to provide a BREAK function for your own loops). Note that the FOREACH loop pushes data on the stack that the BREAK statement is unaware of. Therefore, if you break out of a FOREACH loop, garbage will be left on the stack. The HLA BREAK statement will issue a warning if this occurs. It is your responsibility to clean up the stack upon exiting a FOREACH loop if you break out of it.

14.12 The CONTINUE and CONTINUEIF Statements in HLA

The continue and continueif statements allow you to restart a loop. These two statements allow the following syntax:

```
continue;
continueif( boolean_expression );
continueif(#{ stmts }#);
```

There are two very important things to note about these statements. First, unlike many HLA machine instructions, you do not follow the CONTINUE statement with a pair of empty

parentheses. The 80x86 machine instructions behave like compile-time functions, so it made sense to require empty parentheses after those instructions. The HLA HLL statements do not behave like compile-time functions; the lack of parentheses after continue (and other HLL statements, e.g., else) makes sense here if you think about it for a moment.

The CONTINUE and CONTINUEIF statements are legal only inside WHILE, FOREACH, FOREVER, and REPEAT loops. HLA does not recognize loops you've coded yourself using discrete assembly language instructions (of course, you can probably write a macro to provide a CONTINUE function for your own loops).

For the WHILE and REPEAT statements, the CONTINUE and CONTINUEIF statements transfer control to the test for loop termination. For the FOREVER loop, the CONTINUE and CONTINUEIF statements transfer control the the first statement in the loop. For the FOREACH loop, CONTINUE and CONTINUEIF transfer control to the bottom of the loop (i.e., forces a return from the `yield`() call).

14.13 The BEGIN..END, EXIT, and EXITIF Statements in HLA

The BEGIN..END statement block provides a structured goto statement for HLA. The BEGIN and END clauses surround a group of statements; the EXIT and EXITIF statements allow you to exit such a block of statements in much the same way that the BREAK and BREAKIF statements allow you to exit a loop. Unlike BREAK and BREAKIF, which can only exit the loop that immediately contains the BREAK or BREAKIF, the exit statements allow you to specify a BEGIN label so you can exit several nested contexts at once. The syntax for the BEGIN..END, EXIT, and EXITIF statements is as follows:

```
begin contextLabel ;

    << statements within the specified context >>

end contextLabel;

exit contextLabel;
exitif( boolean_expression ) contextLabel;
exitif(#{ stmts }#) contextLabel;
```

The BEGIN..END clauses do not generate any machine code (although END does emit a label to the assembly output file). The EXIT statement simply emits a JMP to the first instruction following the END clause. The EXITIF statement emits a compare and a conditional jump to the statement following the specified end.

If you break out of a FOREACH loop using the EXIT or EXITIF statements, there will be garbage left on the stack. It is your responsibility to be aware of this situation (i.e., HLA doesn't warn you about it) and clean up the stack, if necessary.

You can nest BEGIN..END blocks and EXIT out of any enclosing BEGIN..END block at any time. The BEGIN label provides this capability. Consider the following example:

```
program ContextDemo;

#include( "stdio.hhf" );

static
    i:int32;

begin ContextDemo;

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

    begin c1;
```

```
        begin c2;

            stdout.put( "Inside c2" nl );
            exitif( i < 0 ) c1;

        end c2;
        stdout.put( "Inside c1" nl );
        exitif( i = 0 ) c1;
        stdout.put( "Still inside c1" nl );

    end c1;
    stdout.put( "Outside of c1" nl );

end ContextDemo;
```

The EXIT and EXITIF statements let you exit any BEGIN..END block; including those associated with a program unit such as a procedure, iterator, method, or even the main program. Consider the following (unusable) program:

```
program mainPgm;

    procedure LexLevel1;

        procedure LexLevel2;
        begin LexLevel2;

            exit LexLevel2;     // Returns from this procedure.
            exit LexLevel1;     // Returns from this procedure and
                                //  and the LexLevel1 procedure
                                //  (including cleaning up the stack).
            exit mainPgm;       // Terminates the main program.

        end LexLevel2;

    begin LexLevel1;
        .
        .
        .
    end LexLevel1;

begin mainPgm;
    .
    .
    .
end mainPgm;
```

Note: You may only exit from procedures that have a display and all nested procedures from the procedure you wish to exit from through to the EXIT statement itself must have displays. In the example above, both `LexLevel1` and `LexLevel2` must have displays if you wish to exit from the `LexLevel1` procedure from inside `LexLevel2`. By default, HLA emits code to build the display unless you use the "`@nodisplay`" procedure option.

Note that to exit from the current procedure, you must not have specified the "`@noframe`" procedure option. This applies only to the current procedure. You may exit from nesting (lower lex level) procedures as long as the display has been built.

14.14 The SWITCH/CASE/DEFAULT/ENDSWITCH Statement in HLA

As of HLA v1.102, a multi-way switch statement is available in the HLA language (prior to HLA v1.102, the switch statement was handled by a macro provided in the HLA Standard Library). This statement uses syntax similar to the following:

```
switch( reg32 )

    case( constant_list )

        <statements>


    << any number of additional case clauses >>

    default// This is optional

        <statements>

endswitch;
```

The case clause argument list is either a single ordinal constant, or a list of ordinal constants separated by commas. The following is an example of a legal switch statement with multiple case clauses:

```
switch( eax )

    case( 0 )

        mov( 1, eax );

    case( 1, 2 )

        mov( 2, eax );

    case( 5 )

        add( 4, eax );

endswitch;
```

The *switch* statement, like it's HLL counterpart, transfers control to the statements following the *case* clause containing the value held in the 32-bit register passed into the *switch* statement.

The *case* constant values in a single *case* statement must all be unique. HLA will report an error if two *cases* contain the same constant value.

During the execution of the *switch* statement, if the value in the 32-bit register passed as an argument to the *switch* statement is not present any any of the *case* clauses, then control transfers to the statements associated with the *default* clause (if one is present) or to the first statement following the *endswitch* class if there is no *default* section present.

In general, HLA compiles the *switch* statement into a jump table and an indirect jmp instruction that transfers control to the code associated with the specified case. However, in a couple of special cases HLA will not compile a switch into an indirect jump instruction. To understand when this occurs, there are a couple of terms you'll need to understand.

Jump tables created for *switch* statements will have one entry for every ordinal value between the smallest *case* value and the largest *case* value in the table. The difference between the largest

and smallest case values (plus one) is called the *spread*. This means that a jump table's size in bytes will be four times the spread. Note that the spread value is independent of the number of cases. Consider the following *switch* statement fragnents:

```
switch( eax )

    case( 1 )

        << code to execute if EAX = 1 >>

    case( 10 )

        << code to execute if EAX = 10 >>

endswitch;
```

The jump table associated with this switch entry will have ten entries, not two. This is because the spread is 10 for this switch statement. Consider the following example:

```
switch( eax )

    case( 1 )

        << code to execute if EAX = 1 >>

    case( 3 )

        << code to execute if EAX = 3 >>

    case( 6 )

        << code to execute if EAX = 6 >>

    case( 10 )

        << code to execute if EAX = 10 >>

endswitch;
```

In this examples the spread is still 10 and the jump table will have the same number of entries (10) as the previous example. This is true even though this latter example has twice as many cases as the earlier example.

The case clause lets you specify multiple values in a comma-separated list. Consider the following example:

```
switch( eax )

    case( 1 )

        << code to execute if EAX = 1 >>

    case( 3, 6, 12 )

        << code to execute if EAX = 3, 6, or 12 >>

    case( 10 )
```

```
        << code to execute if EAX = 10 >>

endswitch;
```

It is important to realize that this *switch* statement has five cases, not three. It just happens that three of the cases (3, 6, and 12) share the same set of instructions to execute. Also note that the spread is 12 in this example as the minimum *case* value is 1 and the largest is 12. Note that the default *case* does not count as a *case* for the purposes of counting the number of *case* values. The default case simply provides a sequence of instructions to execute for all the "holes" in the spread of case values (as well as all values below and greater than the minimum and maximum case values).

Because the jump table will have one entry for each integer value between the smallest and largest *case* values, you can easily generate a huge table with a very simple *switch* statement. Consider the following example:

```
switch( eax )

   case( 1 )

      << code to execute if EAX = 1 >>

   case( 1000 )

      << code to execute if EAX = 1000 >>

endswitch;
```

Even though this example has only two cases, the jump table will contain 1,000 entries (and be 4,000 bytes long). A set of widely spaced *case* values produces a sparse jump table (that is, only a few of the entries in the jump table contain pointers to sections of code associated with the cases, most entries contain a pointer to the *default* case (or the address of the first statement following the *endswitch* if there isn't a default section).

To improve efficiency and reduce the space consumed by large, sparse, jump tables, HLA specially handles a couple of situations. First of all, if the number of cases is three or less, HLA will not emit a jump table. Instead, it will emit a sequence of CMP and JNE instructions to test the three or fewer case values. Second, if the spread is 256 or greater but there are 32 or fewer cases, then HLA will emit a sequence of CMP and JNE instructions to implement the *switch* statement. In all other situations, HLA will emit a jump table implementation of the *switch*.

If the spread is 16384 or greater (this is an implementation-dependent constant an may change in the future), HLA will generate an error and refuse to compile the *switch* statement. If you really want to generate a *switch* statement whose jump table consumes 64K (or more) of data, you will have to implement the statement manually (or modify the switch macro in the "switch.hhf" header file).

If the spread is 4096 or greater but less than 16384, HLA will generate the code but issue a warning telling you that the jump table is going to be very large. If the spread is 16 times (or more) the number of cases, HLA will emit a warning telling you that the jump table is going to be very sparse.

All the case values in a particular *switch* statement must be unique. If there are any duplicate case values in a particular *switch* statement HLA will issue an error message.

14.15 The JT and JF Medium Level Instructions in HLA

The JT (jump if true) and JF (jump if false) instructions are a cross between the 80x86 conditional jump instruction and the HLA IF statement. These two instructions use the following syntax:

```
JT ( booleanExpression ) targetLabel;
JF ( booleanExpression ) targetLabel;
```

The *booleanExpression* component can be any legal HLA boolean expression that you'd use in an IF, WHILE, REPEAT..UNTIL, or other HLA HLL statement. The HLA compiler emits code that will transfer control to the specified target label in your program if the condition is true.

These instructions are primarily intended for use in macros when creating your own HLL control statements. For a discussion of macros and creating your own control structures, see the HLA documentation on the compile-time language.

14.16 Iterators and the HLA Foreach Loop

HLA provides a very powerful user-defined looping control structure, the FOREACH..ENDFOR loop. The FOREACH loop uses the following syntax:

```
foreach iteratorProc( parameters ) do
   << foreach loop body >>
endfor;
```

The *iteratorProc(parameters)* component is a call to a special kind of procedure known as an iterator[1]. Iterators have the special property that they return one of two states, success or failure. If an iterator returns success, it generally also returns a function result. If an iterator returns success, the foreach loop will execute the loop body and reenter the iterator (more on that later) at the top of the loop. If an iterator returns failure, then the loop terminates.

If you've never used true iterators before, you may be thinking "big deal, an iterator is simply a function that returns a boolean value." This, however, isn't entirely true. An iterator behaves like a value returning function when it succeeds, it behaves like a procedure when it fails. The success or failure state of the iterator call is *not* the return value. To understand the difference, consider the syntax for an iterator:

```
iterator iteratorName <<( optional_parameters )>>;
<< procedure options >>
<< local declarations >>
begin iteratorName;

   << iterator statements >>

end iteratorName;
```

Other than the use of the "ITERATOR" keyword rather than "PROCEDURE," this declaration looks just like a procedure or method declaration. However, there are some crucial differences. First of all, HLA emits different code for building iterator activation records than it does for procedures and methods. Furthermore, whenever you declare an iterator, HLA automatically creates a special thunk variable named "`yield`". Also, HLA will not let you call an iterator directly by specifying the iterator's name as an HLA statement (although you can still use the CALL instruction to call an iterator procedure, though you'd better have set the stack up properly before doing so).

If an iterator returns via a EXIT(*iteratorname*) or RET() statement, or returns by "falling off the end of the function" (i.e., executing the "end" clause), then the iterator returns failure to the calling FOREACH loop (hence, the loop will terminate). To return success, and return a value to the body of the FOREACH loop, you must invoke the "`yield`" thunk. `Yield` doesn't actually return to the FOREACH loop, instead, it calls the body of the FOREACH loop and at the bottom of

1. HLA's iterators are based on the similar control structure from the CLU language. CLU's iterators are considerably more powerful than the misnamed "iterators" found in the C/C++ language/library (which, technically, should be called "cursors" not iterators).

the FOREACH loop HLA emits a return instruction that transfers control back into the iterator (to the first statement following the `yield`). This may seem counter-intuitive, but it has some important ramifications. First of call, an iterator maintains its context until it fails. This means that local variables maintain their values across the `yield` calls. Likewise, when a FOREACH loop reenters an iterator, it picks up immediately after the `yield`, it does not pass new parameters and begin execution at the top of the iterator code.

Consider the following typical iterator code:

```
program iteratorDemo;

#include( "stdio.hhf" );

iterator range( start:int32; stop:int32 ); @nodisplay;
begin range;

    forever

        mov( start, eax );
        breakif( eax > stop );
        yield();
        inc( start );

    endfor;

end range;


static
    i:int32;

begin iteratorDemo;

    foreach range( 1, 10 ) do

        stdout.put( "eax = ", eax, nl );

    endfor;

end iteratorDemo;
```

This example demonstrates how to create a standard "for" loop like those found in Pascal or C++[2]. The `range` iterator is passed two parameters, a starting value and an ending value. It returns a sequence of values between the starting and ending values (respectively) and fails once the return value would exceed the ending value. The FOREACH loop in this example prints the values one through ten to the display.

Warning: because the iterator's activation is left on the stack while executing a FOREACH loop, you should take care when breaking out of a FOREACH loop using BREAK, BREAKIF, EXIT, EXITIF, or some sort of jump. Cavalierly jumping out of a loop in this fashion leaves the iterator's activation record on the stack. You will need to clean this up manually if you exit an iterator in this fashion. Since HLA cannot determine the myriad of ways one could jump out of a FOREACH loop body, it is up to you to make sure you don't do this (or that you handle the garbage on the stack in an appropriate way).

Keep in mind that the body of a FOREACH loop is actually a procedure your program calls when it encounters the `yield` statement[3]. Therefore, any registers whose values you change will be changed when control returns to the code following the `yield`. If you need to preserve any

2. Mind you, this is not a very efficient implementation of a standard for loop.

registers across a `yield`, either push and pop them at the beginning of the FOREACH loop body or place the PUSH and POP instructions around the `yield`.

3. Technically, yield is a variable of type thunk, not a statement. However, this discussion is somewhat clearly if we think of yield as a statement rather than a variable.