Questions, Projects, and Labs Chapter Thirteen

13.1 Questions

1) What is the purpose of the UNPROTECTED section in a TRY..ENDTRY statement?

2) Once a TRY..ENDTRY statement has handled an exception, how can it tell the system to let a nesting TRY..ENDTRY statement also handle that (same) exception?

3) What is the difference between static and dynamic nesting (e.g., with respect to the TRY..ENDTRY statement)?

4) How you can you handle any exception that occurs without having to write an explicit EXCEPTION handler for each possible exception?

5) What HLA high level statement could you use immediately return from a procedure without jumping to the end of the procedure's body?

6) What is the difference between the CONTINUE and BREAK statements?

7) Explain how you could use the EXIT statement to break out of two nested loops (from inside the innermost loop). Provide an example.

8) The EXIT statement translates into a single 80x86 machine instruction. What is that instruction?

9) What is the algorithm for converting a conditional jump instruction to its opposite form?

10) Discuss how you could use the JF instruction and a label to simulate an HLA IF..ENDIF statement and a WHILE loop.

11) Which form requires the most instructions: complete boolean evaluation or short circuit evaluation?

12) Translate the following C/C++ statements into "pure" assembly language and complete boolean evaluation:

a) if((eax >= 0) && (ebx < eax) || (ebx < 0)) ebx = ebx + 2;

b) while((ebx != 0) && (*ebx != 0)) { *ebx = 'a'; ++ebx; }

c) if(al == 'c' || al == 'd' || bl == al) al = 'a';

d) if(al >= 'a' && al <= 'z') al = al & 0x5f;

13) Repeat question (12) using short circuit boolean evaluation.

14) Convert the following Pascal CASE statement to assembly language:

```
CASE I OF
    0: I := 5;
    1: J := J+1;
    2: K := I+J;
    3: K := I-J;
    Otherwise I := 0;
END;
```

15) Which implementation method for the CASE statement (jump table or IF form) produces the least amount of code (including the jump table, if used) for the following Pascal CASE statements?

a)

```
CASE I OF
    0:stmt;
    100:stmt;
    1000:stmt;
END;
```


b)

```
CASE I OF
    0:stmt;
    1:stmt;
    2:stmt;
    3:stmt;
    4:stmt;
END;
```

16) For question (15), which form produces the fastest code?

17) Implement the CASE statements in problem three using 80x86 assembly language.

18) What three components compose a loop?

19) What is the major difference between the WHILE, REPEAT..UNTIL, and FOREVER..ENDFOR loops?

20) What is a loop control variable?

21) Convert the following C/C++ WHILE loops to pure assembly language: (Note: don't optimize these loops, stick exactly to the WHILE loop format)

a)
```
I = 0;
while (I < 100)
        I = I + 1;
```

b)
```
CH = ' ';
while (CH <> '.')
{
        CH := getch();
        putch(CH);
}
```

22) Convert the following Pascal REPEAT..UNTIL loops into pure assembly language: (Stick exactly to the REPEAT..UNTIL loop format)

a)
```
I := 0;
REPEAT
        I := I + 1;
UNTIL   I >= 100;
```

b)
```
REPEAT
        CH := GETC;
        PUTC(CH);
UNTIL CH = '.';
```

23) What are the differences, if any, between the loops in problems (21) and (22)? Do they perform the same operations? Which versions are most efficient?

24) By simply adding a JMP instruction, convert the two loops in problem (21) into REPEAT..UNTIL loops.

25) By simply adding a JMP instruction, convert the two loops in problem (22) to WHILE loops.

26) Convert the following C/C++ FOR loops into pure assembly language (Note: feel free to use any of the routines provided in the HLA Standard Library package):

a) for(i = 0; i < 100; ++i) cout << "i = " << i << endl;

b)
```
for( i = 0; i < 8; ++i )
        for( j = 0; j < 8; ++j )
                k = k * (i - j);
```

c)
```
for( k= 255; k >= 16; --k )
        A [k] := A[240-k]-k;
```

27) How does moving the loop termination test to the end of the loop improve the performance of that loop?

28) What is a loop invariant computation?

29) How does executing a loop backwards improve the performance of the loop?

30) What does unraveling a loop mean?

31) How does unraveling a loop improve the loop's performance?

32) Give an example of a loop that cannot be unraveled.

33) Give an example of a loop that can be but shouldn't be unraveled.

34) What is the difference between unstructured and destructured code?

35) What is the principle difference between a state machine and a SWITCH statement?

36) What is the effect of the NODISPLAY procedure option?

37) What is the effect of the NOFRAME procedure option?

38) What is the effect of the NOSTKALIGN procedure option?

39) Why don't you normally use the RET instruction in a procedure that does not have the NOFRAME option?

40) What does the operand to the RET(n) instruction specify?

41) What is an activation record?

42) What part of the activation record does the *caller* construct?

43) What part of the activation record does the *callee* (the procedure) construct?

44) Provide a generic definition for "The Standard Entry Sequence."

45) What four instructions are typically found in an HLA Standard Entry Sequence?

46) Which instruction in the Standard Entry Sequence will HLA *not* generate if you specify the NOALIGNSTK option in the procedure?

47) Which instruction is the Standard Entry Sequence is optional if there are no automatic variables?

48) Provide a generic definition for "The Standard Exit Sequence."

49) What three instructions are typically found in an HLA Standard Exit Sequence?

50) What data in the activation record is probably being accessed by an address of the form "[ebp-16]"?

51) What data in the activation record is probably being accessed by an address of the form "[ebp+16]"?

52) What does the *_vars_* constant tell you?

53) What is the big advantage to using automatic variables in a procedure?

54) What is the difference between pass by reference parameters and pass by value parameters?

55) Name three different places *where* you can pass parameters.

56) Which parameter passing mechanism uses pointers?

57) For each of the following procedure prototypes and corresponding high level syntax procedure calls, provide an equivalent sequence of low-level assembly language statements. Assume all variables are *int32* objects unless otherwise specified. If the procedure call is illegal, simply state that fact and don't attempt to write any code for the call. Assume that you are passing all parameters on the stack.

a) procedure proc1(i:int32); forward;

a1) proc1(10);

a2) proc1(j);

a3) proc1(eax);

a4) proc1([eax]);

b) procedure proc2(var v:int32); forward;

b1) proc2(10);

b2) proc2(j);

b3) proc2(eax);

b4) proc2([eax]);

58) When passing parameters in the code stream, where do you find the pointer to the parameter data?

59) When passing parameter data immediately after a CALL instruction, how do you prevent the procedure call from attempting to execute the parameter data upon immediate return from the procedure?

60) Draw a picture of the activation record for each of the following procedure fragments. Be sure to label the size of each item in the activation record.

a)

```
procedure P1( val i:int16; var j:int16 ); nodisplay;
var
    c:char;
    k:uns32;
    w:word;
begin P1;
     .
     .
     .
end P1;
```

b)

```
procedure P2( r:real64; val b:boolean; var c:char ); nodisplay;
begin P2;
     .
     .
     .
end P2;
```

c)

```
procedure P3; nodisplay;
var
    i:uns32;
    j:char;
    k:boolean;
    w:word;
    r:real64;
```



```
begin P3;
     .
     .
     .
end P3;
```

61) Fill in the pseudo-code (in comments) for the following procedures:

a)

```
procedure P4( val v:uns32 ); nodisplay;
var
   w:dword;
begin P4;

   // w = v;
   // print w;

end P4;
```

b)

```
procedure P5( var v:uns32 ); nodisplay;
var
   w:dword;
begin P5;

   // w = v;
   // print w;

end P5;
```

62) Given the procedures defined in question (61) above, provide the low-level code for each of the following (pseudo-code) calls to P4 and P5. You may assume that you can use any registers you need for temporary calculations. You may also assume that all variables are *uns32* objects.

a) P4(i);

b) P4(10);

c) P4(eax+10);

d) P5(i);

e) P5(i[eax*4]);

f) P5([eax+ebx*4]);

63) This question also uses the procedure declarations for P4 and P5 in question (61). Write the low-level code for the statements in the P6 procedure below:

```
procedure p6( val v:uns32; var r:uns32 ); nodisplay;
begin P6;

   P4( v );
   P4( r );
   P5( v );
   P5( r );

end P6;
```

64) Describe the HLA hybrid parameter passing syntax and explain why you might want to use it over the low-level and high-level procedure call syntax provided by HLA.

65) 30)What is a procedure variable? Give some examples.

66) When you pass a procedure variable by value to a procedure, what do we typically call such a parameter?

67) How does an iterator return success? How does it return failure?

68) What does the *yield()* procedure do?

69) Why shouldn't you break out of a FOREACH..ENDFOR loop?

70) An extended precision ADD operation will set the carry flag, the overflow flag, and the sign flag properly. It does not set the zero flag properly. Explain how you can check to see if an extended precision ADD operation produces a zero result.

71) Since SUB and CMP are so closely related, why can't you use the SUB/SBB sequence to perform an extended precision compare? (hint: this has nothing to do with the fact that SUB/SBB actually compute a difference of their two operands.)

72) Provide the code to add together two 48-bit values, storing the sum in a third 48-bit variable. This should only take six instructions.

73) For 64-bit multiprecision operations, why is it more convenient to declare an uns64 variable as "uns32[2]" rather than as a *qword*?

74) The 80x86 INTMUL instruction provides an n x n bit (n=16 or 32) multiplication producing an n-bit result (ignoring any overflow). Provide a variant of the MUL64 routine in this chapter that produces a 64-bit result, ignoring any overflow (hint: mostly this involves removing instructions from the existing code).

75) When computing an extended precision NEG operation using the "subtract from zero" algorithm, does the algorithm work from the H.O. double word down to the L.O. double word, or from the L.O. double word to the H.O. double word?

76) When computing an extended precision logical operation (AND, OR, XOR, or NOT), does it matter what order you compute the result (H.O.->L.O. or L.O.->H.O.)? Explain.

77) Since the extended precision shift operations employ the rotate instructions, you cannot check the sign or zero flags after an extended precision shift (since the rotate instructions do not affect these flags). Explain how you could check the result of an extended precision shift for zero or a negative result.

78) Which of the two data operands does the SHRD and SHLD instructions leave unchanged?

79) What is the maximum number of digits a 128-bit unsigned integer will produce on output?

80) What is the purpose of the conv.getDelimiters function in the HLA Standard Library?

81) Why do the extended precision input routine always union in the EOS (#0) character into the HLA Standard Library delimiter characters when HLA, by default, already includes this character?

82) Suppose you have a 32-bit signed integer and a 32-bit unsigned integer, and both can contain an arbitrary value. Explain why an extended precision addition may be necessary to add these two values together.

83) Provide the code to add a 32-bit signed integer together with a 32-bit unsigned integer, producing a 64-bit result.

84) Why is binary representation more efficient than decimal (packed BCD) representation?

85) What is the one big advantage of decimal representation over binary representation?

86) How do you represent BCD literal constants in an HLA program?

87) What data type do you use to hold packed BCD values for use by the FPU?

88) How many significant BCD digits does the FPU support?

89) How does the FPU represent BCD values in memory? While inside the CPU/FPU?

90) Why are decimal operations so slow on the 80x86 architecture?

91) What are the repeat prefixes used for?

92) Which string prefixes are used with the following instructions?

a) MOVS b) CMPS c) STOS d) SCAS

93) Why aren't the repeat prefixes normally used with the LODS instruction?

94) What happens to the ESI, DDI, and DCX registers when the MOVSB instruction is executed (without a repeat prefix) and:

a) the direction flag is set. b) the direction flag is clear.

95) Explain how the MOVSB and MOVSW instructions work. Describe how they affect memory and registers with and without the repeat prefix. Describe what happens when the direction flag is set and clear.

96) How do you preserve the value of the direction flag across a procedure call?

97) How can you ensure that the direction flag always contains a proper value before a string instruction without saving it inside a procedure?

98) What is the difference between the "MOVSB", "MOVSW", and "MOVS oprnd1,oprnd2" instructions?

99) Consider the following Pascal array definition:

```
a:array [0..31] of record
                       a,b,c:char;
                       i,j,k:integer;
                   end;
```

Assuming A[0] has been initialized to some value, explain how you can use the MOVS instruction to initialize the remaining elements of A to the same value as A[0].

100) Give an example of a MOVS operation which requires the direction flag to be:

a) clear b) set

101) How does the CMPS instruction operate? (what does it do, how does it affect the registers and flags, etc.)

102) Which segment contains the source string? The destination string?

103) What is the SCAS instruction used for?

104) How would you quickly initialize an array to all zeros?

105) How are the LODS and STOS instructions used to build complex string operations?

106) Write a short loop which multiplies each element of a single dimensional array by 10. Use the string instructions to fetch and store each array element.

107) Explain how to perform an extended precision integer comparison using CMPS

108) Explain the difference in execution time between compile-time programs and execution-time programs.

109) What is the difference between the *stdout.put* and the #PRINT statements?

110) What is the purpose of the #ERROR statement?

111) In what declaration section do you declare compile-time constants?

112) In what declaration section do you declare run-time constants?

113) Where do you declare compile-time variables?

114) Where do you declare run-time variables?

115) Explain the difference between the following two computations (assume appropriate declarations for each symbol in these two examples:

a) ? i := j+k*m;

b)
```
mov( k, eax );
intmul( m, eax );
add( j, eax );
mov( eax, i );
```

116) What is the purpose of the compile-time conversion functions?

117) What is the difference between @sin(x) and fsin()? Where would you use @sin?

118) What is the difference between the #IF and the IF statement?

119) Explain the benefit of using conditional compilation statements in your program to control the emission of debugging code in the run-time program.

120) Describe how you can use conditional compilation to configure a program for different run-time environments.

121) What compile-time statement could you use to fill in the entries in a read-only table?

122) The HLA compile-time language does not support a #switch statement. Explain how you could achieve the same result as a #switch statement using existing compile-time statements.

123) What HLA compile-time object corresponds to a compile-time procedure declaration?

124) What HLA compile-time language facility provides a looping construct?

125) HLA TEXT constants let you perform simple textual substitution at compile time. What other HLA language facility provides textual substitution capabilities?

126) Because HLA's compile-time language provides looping capabilities, there is the possibility of creating an infinite loop in the compile-time language. Explain how the system would behave if you create a compile-time infinite loop.

127) Explain how to create an HLA macro that allows a variable number of parameters.

128) What is the difference between a macro and a (run-time) procedure? (Assume both constructs produce some result at run-time.)

129) When declaring macro that allows a variable number of parameters, HLA treats those "extra" (variable) parameters differently than it does the fixed parameters. Explain the difference between these two types of macro parameters in an HLA program.

130) How do you declare local symbols in an HLA macro?

131) What is a multipart macro? What three components appear in a multipart macro? Which part is optional? How do you invoke multipart macros?

132) Explain how you could use the #WHILE statement to unroll (or unravel) a loop.

133) The #ERROR statement allows only a single string operation. Explain (and provide an example) how you can display the values of compile-time variable and constant expressions along with text in a #ERROR statement.

134) Explain how to create a Domain Specific Embedded Language (DSEL) within HLA.

135) Explain how you could use the #WHILE statement to unroll (or unravel) a loop.

136) What is lexical analysis?

137) Explain how to use HLA compile-time functions like @OneOrMoreCSet and @OneCset to accomplish lexical analysis/scanning.

138) The #ERROR statement allows only a single string operation. Explain (and provide an example) how you can display the values of compile-time variable and constant expressions along with text in a #ERROR statement.

139) What are some differences between a RECORD declaration and a CLASS declaration?

140) What declaration section may not appear within a class definition?

141) What is the difference between a class and an object?

142) What is inheritance?

143) What is polymorphism?

144) What is the purpose of the OVERRIDE prefix on procedures, methods, and iterators?

145) What is the difference between a virtual and a static routine in a class? How do you declare virtual routines in HLA? How do you declare static routines in HLA?

146) Are class iterators virtual or static in HLA?

147) What is the purpose of the virtual method table (VMT)?

148) Why do you implement constructors in HLA using procedures rather than methods?

149) Can destructors be procedures? Can they be methods? Which is preferable?

150) What are the two common activities that every class constructor should accomplish?

151) Although HLA programs do not automatically call constructors for an object when you declare the object, there is an easy work-around you can use to automate calling constructors. Explain how this works and give an example.

152) When writing a constructor for a derived class, you will often want to call the corresponding constructor for the base class within the derived class' constructor. Describe how to do this.

153) When writing overridden methods for a derived class, once in a great while you may need to call the base class' method that you're overriding. Explain how to do this. What are some limitations to doing this (versus calling class procedures)?

154) What is an abstract method? What is the purpose of an abstract method?

155) Explain why you would need Run-Time Type Information (RTTI) in a program. Describe how to access this information in your code.

13.2 Programming Problems

> Note: unless otherwise specified, you may not use the HLA high level language statements (e.g., if..elseif..else..endif) in the following programming projects. One exception is the TRY..ENDTRY statement. If necessary, you may use TRY..ENDTRY in any of these programs.

1) Solve the following problem using only "pure" assembly language instructions (i.e., no high level statements).

Write a procedure, PrintArray(var ary:int32; NumRows:uns32; NumCols:uns32), that will print a two-dimensional array in matrix form. Note that calls to the PrintArray function will need to coerce the actual array to an int32. Assume that the array is always an array of INT32 values. Write the procedure as part of a UNIT with an appropriate header file. Also write a sample main program to test the PrintArray function. Include a makefile that will compile and run the program. Here is an example of a typical call to PrintArray:

```
static
   MyArray: int32[4, 5];
       .
       .
       .
```



```
PrintArray( (type int32 MyArray), 4, 5 );
```

2) Solve problem (2) using HLA's hybrid control structures.

3) Solve the following problem using pure assembly language instructions (i.e., no high level language statements):

Write a program that inputs a set of grades for courses a student takes in a given quarter. The program should then compute the GPA for that student for the quarter. Assume the following grade points for each of the following possible letter grades:

- A+ 4.0
- A 4.0
- A- 3.7
- B+ 3.3
- B 3.0
- B- 2.7
- C+ 2.3
- C 2.0
- C- 1.7
- D+ 1.3
- D 1.0
- D- 0.7
- F 0

4) Solve problem (3) using HLA's hybrid control structures (see the description above).

5) Write a "number guessing" program that attempts to guess the number a user has chosen. The number should be limited to the range 0..100. A well designed program should be able to guess the answer with seven or fewer guesses. Use only pure assembly language statements for this assignment.

6) Write a "calendar generation" program. The program should accept a month and a year from the user. Then it should print a calendar for the specific month. You should use the date.IsValid library routine to verify that the user's input date is valid (supply a day value of one). You can also use date.dateOfWeek(m, d, y); to determine whether a day is Monday, Tuesday, Wednesday, etc. Print the name of the month and the year above the calendar for that month. As usual, use only low-level "pure" machine instructions for this assignment.

7) A video producer needs a calculator to compute "time frame" values. Time on video tape is marked as HH:MM:SS:FF where HH is the hours (0..99), MM represents minutes (0..59), SS represents seconds (0..59), and FF represents frames (0..29). This producer needs to be able to add and subtract two time values. Write a pair of procedures that accept these four parameters and return their sum or difference in the following registers:

HH: DH

MM: DL

SS: AH

FF: AL

The main program for this project should ask the user to input two time values (a separate input for each component of these values is okay) and then ask whether the user wants to add these numbers or subtract them. After the inputs, the program should display the sum or difference, as appropriate, of these two times. Write this program using HLA's hybrid control structures.

8) Rewrite the code in problem (7) using only low-level, pure machine language instructions.

9) Write a program that reads an 80x25 block of characters from the screen (using console.getRect, see "Bonus Section: The HLA Standard Library CONSOLE Module" on page 192 for details) and then "scrolls" the characters up one line in the 80x25 array of characters you've copied the data into. Once the scrolling is complete (in the memory array), write the data back to the screen using the console.putRect routine. In your main program, write several lines of (different) text to the screen and call the scroll pro-

cedure several times in a row to test the program. As usual, use only low level machine language instructions in this assignment.

10) Write a program that reads an 80x25 block of characters from the screen (using console.getRect; see the previous problem) and horizontally "flips" the characters on the screen. That is, on each row of the screen swap the characters at positions zero and 79, positions one and 78, etc. After swapping the characters, write the buffer back to the display using the console.putRect procedure. Use only low level machine language statements for this exercise.

> Note: Your instructor may require that you use all low-level control structures (except for TRY..ENDTRY and FOREVER..ENDFOR) in the following assignments. Check with your instructor to find out if this is the case. Of course, where explicitly stated, always use low level or high level code.

11) Write a Blackjack/21 card game. You may utilize the code from the iterator laboratory exercise (see "Iterator Exercises" on page 1234). The game should play "21" with two hands: one for the house and one for the player. The play should be given an account with $5,000 U.S. at the beginning of the game. Before each hand, the player can bet any amount in multiples of $5. If the dealer wins, the players loses the bet; if the player wins, the player is credited twice the amount of the bet. The player is initially dealt two cards. Each card has the following value:

2-10: Face value of card

J, Q, K: 10

A: 1 or 11. Whichever is larger and does not cause the player's score to exceed 21.

The game should deal out the first four cards as follows:

1st: to the player.

2nd: to the dealer.

3rd: to the player

4th: to the dealer.

The game should let the player see the dealer's first card but it should not display the dealer's second card.

After dealing the cards and displaying the user's cards and the dealer's first card, the game should allow the user to request additional cards. The user can request as many additional cards as desired as long as the user's total does not exceed 21. If the player's total is exactly 21, the player automatically wins, regardless of the dealer's hand. If the player's total exceeds 21, the player automatically loses.

Once the player decides to stop accepting cards, the dealer must deal itself cards as long as the dealer's point total is less than 17. Once the dealer's total exceeds 17, the game ends. Whomever has the larger value that does not exceed 21 wins the hand. In the event of a tie, the player wins.

Do not reshuffle the deck after each hand. Place used cards in a storage array and reshuffle those once the card deck is exhausted. Complete the hand by dealing from the reshuffled deck. Once this hand is complete, reshuffle the entire deck and start over.

At the end of each hand, as the player if they want to "cash out" (quit) or continue. The game automatically ends if the player "goes broke" (that is, the cash account goes to zero). The house has an unlimited budget and never goes broke.

12) Modify program (11) to allow more than one card deck in play at a time. Let the player specify the number of card decks when the program first starts.

13) Modify program (12) to allow more than one player in the game. Let the initial user specify the number of players when the program first starts (hint: use HLA's dynamic arrays for this). Any player may "cash out" and exit the game at any time; in such a case the game continues as long as there is at least one remaining player. If a player goes broke, that particular player automatically exits the game.

14) Modify the "Outer Product" sample program (see "Outer Product Computation with Procedural Parameters" on page 848) to support division ("/"), logical AND ("&"), logical OR ("|"), logical XOR ("^"), and remainder ("%").

15) Modify project (14) to use the *uns32* values 0..7 to select the function to select the operation rather than a single character. Use a CALL-based SWITCH statement to call the actual function (see "Procedural Parameter Exercise" on page 1231 for details on a CALL-based SWITCH statement) rather than the current if..elseif implementation.

16) Generally, it is not a good idea to break out of a FOREACH..ENDFOR loop because of the extra data that the iterator pushes onto the stack (that it doesn't clean up until the iterator fails). While it is possible to pass information back to the iterator from the FOREACH loop body (remember, the loop body is essentially a procedure that the iterator calls) and you can use this return information to force the iterator to fail, this technique is cumbersome. The program would be more readable if you could simply break out of a FOREACH loop as you could any other loop. One solution to this problem is to save the stack pointer's value before executing the FOREACH statement and restoring this value to ESP immediately after the ENDFOR statement (you should keep the saved ESP value in a local, automatic, variable). Modify the Fibonacci number generation program in the Sample Programs section (see "Generating the Fibonacci Sequence Using an Iterator" on page 846) and eliminate the parameter to the iterator. Have the iterator return an infinite sequence of fibonacci numbers (that is, the iterator should never return failure unless there is an unsigned arithmetic overflow during the fibonacci computation). In the main program, prompt the user to enter the maximum value to allow in the sequence and let the FOREACH loop run until the iterator returns a value greater than the user-specified maximum value (or an unsigned overflow occurs). Be sure to clean up the stack after leaving the FOREACH loop.

17) Write a "Craps" game. Craps is played with two six-sided die[1] and the rules are the following:

    A player rolls a pair of dice and sums up the total of the two die. If the sum is 7 or 11, the player automatically wins. If the sum is 2, 3, or 12 then the player automatically loses ("craps out"). If the total is any other value, this becomes the player's "point." The player continues to throw the die until rolling either a seven or the player's point. If the player rolls a seven, the player loses. If the player rolls their point, they win. If the player rolls any other value, play continues with another roll of the die.

    Write a function "dice" that simulates a roll of one dice (hint: take a look at the HLA Standard Library *rand.range* function). Call this function twice for each roll of the two die. In your program, display the value on each dice and their sum for each roll until the game is over. To make the game slightly more interesting, pause for user input (e.g., stdin.ReadLn) after each roll.

18) Modify program (17) to allow wagering. Initialize the player's balance at $5,000 U.S. For each game, let the user choose how much they wish to wager (up to the amount in their account balance). If the player wins, increase their account by the amount of the wager. If the player loses, decrease their account by the amount of the wager. The whole game ends when the player's account drops to zero or the player chooses to "cash out" of the game.

19) Modify program (18) to allow multiple players. In a multi-player craps game only one player throws the dice. The other players take "sides" with the player or the house. Their wager is matched (and their individual account is credited accordingly) if they side with the winner. Their account is deducted by the amount of their wager if they side with the loser. When the program first begins execution, request the total number of players from the user and dynamically allocate storage for each of the players. After each game, rotate the player who "throws" the dice.

20) The "greatest common divisor" of two integer values A and B is the largest integer that evenly divides (that is, has a remainder of zero) both A and B. This function has the following recursive definition:

    If either A or B is zero, then gcd(A, B) is equal to the non-zero value (or zero if they are both zero.

    If A and B are not zero, then gcd(A, B) is equal to gcd(B, A mod B) where "mod" is the remainder of A divided by B.

    Write a program that inputs two unsigned values A and B from the user and computes the greatest com-

1. "Die" is the plural of "dice" in case you're wondering.

mon divisor of these two values.

21) Write a program that reads a string from the user and counts the number of characters belonging to a user-specified class. Use the HLA Standard Library character classification routines (chars.isAlpha, chars.isLower, chars.isAlpha, chars.isAlphaNum, chars.isDigit, chars.isXDigit, chars.isGraphic, chars.isSpace, chars.isASCII, and chars.isCtrl) to classify each character in the string. Let the user specify which character classification routine they wish to use when processing their input string. Use a single CALL instruction to call the appropriate chars.XXXX procedure (i.e., use a CALL table and a CALL-based switch statement, see "Procedural Parameter Exercise" on page 1231 for more details).

22) The HLA Standard Library arrays module ("array.hhf") includes an *array.element* iterator that returns each element from an array (in row major order) and fails after returning the last element of the array. Write a program that demonstrates the use of this iterator when processing elements of a single dimension dynamic array.

23) The HLA Standard Library *array.element* iterator (see (22) above) has one serious limitation. It only returns sequential elements from an array; it ignores the shape of the array. That is, it treats two-dimensional (and higher dimensional) matrices as though they were a single dimensional array. Write a pair of iterators specifically designed to process elements of a two-dimensional array: *rowIn* and *elementInRow.* The prototypes for these two iterators should be the following:

```
type
    matrix: dArray( uns32, 2 );

iterator rowIn( m:matrix );
iterator elementInRow( m:matrix; row:uns32 );
```

The *rowIn* iterator returns success for each row in the matrix. It also returns a row number in the EAX register (0..*n*-1 where *n* is the number of rows in the matrix). The *elementInRow* iterator returns success *m* times where *m* is the number of columns in the matrix. Note that the *uns32* value *m.dopeVector[0]* specifies the number of rows and the *uns32* value *m.dopeVector[4]* specifies the number of columns in the matrix (see the HLA Standard Library documentation for more details). The *elementInRow* iterator should return the value of each successive element in the specified row on each iteration of the corresponding FOREACH loop.

Write a program to test your iterators that reads the sizes for the dimensions from the user, dynamically allocates the storage for the matrix (using array.daAlloc), inputs the data for the matrix, and then uses the two iterators (in a pair of nested FOREACH loops) to display the data.

24) The sample program in the chapter on advanced arithmetic (BCDio, see "Sample Program" on page 903) "cheats" on decimal output by converting the output value to a signed 64-bit quantity and then calling *stdout.puti64* to do the actual output. You can use this same trick for input (i.e., call *stdin.geti64* and convert the input integer to BCD) if you check for BCD overflow (18 digits) prior to the conversion. Modify the sample program to use this technique to input BCD values. Be sure that your program properly handles 18 digits of '9' characters on input but properly reports an error if the value is 19 digits or longer.

25) Write a procedure that multiplies two signed 64-bit integer values producing a 128-bit signed integer result. The procedure's prototype should be the following:

```
type
    int64:  dword[2];
    int128: dword[4];

    procedure imul64( mcand:int64; mplier:int64; var product:int128 );
```

The procedure should compute *mcand*mplier* and leave the result in *product*. Create a UNIT that contains this library module and write a main program (in a separate source file) that calls and tests your division routine.

26) Write an extended precision unsigned division procedure that divides a 128-bit unsigned integer by a 32-bit unsigned integer divisor. (hint: use the extended precision algorithm involving the DIV instruction.) The procedure's prototype should be the following:

```
type
    uns128: dword[4];

    procedure div128
    (
              dividend:uns128;
              divisor:dword;
        var   quotient:uns128;
        var   remainder:dword
    );
```

27) Write an extended precision signed division procedure that divides one *int128* object by another *int128* object. Place this procedure in a UNIT and write a main program (in a separate module) that calls this routine in order to test it. Be sure to handle a division by zero error (raise the ex.DivideError exception if a division by zero occurs).

The procedure's prototype should be the following:

```
type
    int128: dword[4];

    procedure idiv128
    (
              dividend:int128;
              divisor:int128;
        var   quotient:int128;
        var   remainder:int128
    );
```

28) Extend the idiv128 procedure from (27) to use the fast division algorithm if the H.O. 96 bits of the divisor are all zero (the fast algorithm uses the DIV instruction). If the H.O. 96 bits are not all zero, fall back to the algorithm you wrote for problem (26).

29) Write a procedure, shr128, that shifts the bits in a 128-bit operand to the right n bit positions. Use the SHRD instruction to implement the shift. The procedure's prototype should be the following:

```
type
    uns128: dword[4];

    procedure shr128( var operand:uns128; n:uns32 );
```

The function should leave the result in the *operand* variable and the carry flag should contain the value of the last bit the procedure shifts out of the *operand* parameter.

30) Write an extended precision ROR procedure, ror128, that does a ROR operation on a 128-bit operand. The prototype for the procedure is

```
type
    uns128: dword[4];

    procedure ror128( var operand:uns128; n:uns32 );
```

The function should leave the result in the *operand* variable and the carry flag should contain the value of the last bit the procedure shifts out of bit zero of the *operand* parameter.

31) Write an extended precision ROL procedure, ror128, that does a ROR operation on a 128-bit operand. The prototype for the procedure is

```
type
    uns128: dword[4];

    procedure rol128( var operand:uns128; n:uns32 );
```

The function should leave the result in the *operand* variable and the carry flag should contain the value of the last bit the procedure shifts out of bit 63 of the *operand* parameter.

32) Write a 256-bit unsigned integer extended precision output routine (putu256). Place the procedure in a UNIT (IO256) and write a main program that calls the procedure to test it. The procedure prototype should be the following:

```
type
    uns256: dword[8];

    procedure putu256( operand:uns256 );
```

33) Extend the 256-bit output UNIT by adding a "puti256" procedure that prints 256-bit signed integer values. The procedure prototype should be the following:

```
type
    int256: dword[8];

    procedure puti256( operand:int256 );
```

34) A 256-bit integer (signed or unsigned) value requires a maximum of approximately 80 to 100 digits to represent (this value was approximated by noting that every ten binary bits is roughly equivalent to three or four decimal digits). Write a pair of routines (and add them to the IO256 UNIT) that will calculate the number of print positions for an *uns256* or *int256* object (don't forget to count the minus sign if the number is negative). These functions should return the result in the EAX register and they have the following procedure prototypes:

```
type
    uns256: dword[8];
    int256: dword[8]

    procedure isize256( operand:int256 );
    procedure usize256( operand:uns256 );
```

Probably the best way to determine how many digits the number will consume is to repeatedly divide the value by 10 (incrementing a digit counter) until the result is zero. Don't forget to negate negative numbers (*isize256*) prior to this process.

35) Extend the IO256 UNIT by writing *puti256Size* and *putu256Size*. These procedures should print the value of their parameter to the standard output device adding padding characters as appropriate for the parameter's value and the number of specified print positions. The function prototypes should be the following:

```
procedure puti256Size( number:int256; width:int32; fill:char );
procedure putu256Size( number:uns256; width:int32; fill:char );
```

See the HLA Standard Library documentation for the *stdout.puti32Size* and *stdout.putu32Size* routine for more information about the parameters.

36) Extend the IO256 UNIT by writing an *uns256* input routine, *getu256*. The procedure should use the following prototype:

```
type
    uns256: dword[8];

    procedure getu256( var operand:uns256 );
```

37) Add a *geti256* input routine to the IO256 UNIT. This procedure should read 256-bit signed integer values from the standard input device and store the two's complement representation in the variable the caller passes by reference. Don't forget to handle the leading minus sign on input. The prototype should be

```
type
    int256: dword[8];

    procedure geti256( var operand:int256 );
```

38) Write a procedure that multiplies two four-digit unpacked decimal values utilizing the MUL and AAM instructions. Note that the result may require as many as eight decimal digits.

39) Modify Program 8.8 by changing the "PUT32" macro to handle 8, 16, and 32-bit integers, unsigned integer, or hex (byte, word, and dword) variables.

40) Modify Program 8.9 by changing the *puti32* macro to handle 8, 16, and 32-bit integers, unsigned integers, and hexadecimal (byte, word, dword) values. (rename the macro to *puti* so that the name is a little more appropriate). Of course, you should still handle multiple parameters (calling *putXXXsize* if more than one parameter.

41) Write a SubStr function that extracts a substring from a zero terminated string. Pass a pointer to the string in esi, a pointer to the destination string in edi, the starting position in the string in eax, and the length of the substring in ecx. Be sure to handle degenerate conditions.

42) Write a word *iterator* to which you pass a string (by reference, on the stack). Each each iteration of the corresponding FOREACH loop should extract a word from this string, *strmalloc* sufficient storage for this string on the heap, copy that word (substring) to the malloc'd location, and return a pointer to the word. Write a main program that calls the iterator with various strings to test it.

43) Write a *strncpy* routine that behaves like str.cpy except it copies a maximum of *n* characters (including the zero terminating byte). Pass the source string's address in edi, the destination string's address in esi, and the maximum length in ecx.

44) The MOVSB instruction may not work properly if the source and destination blocks overlap (see "The MOVS Instruction" on page 938). Write a procedure *bcopy* to which you pass the address of a source block, the address of a destination block, and a length, that will properly copy the data even if the source and destination blocks overlap. Do this by checking to see if the blocks overlap and adjusting the source pointer, destination pointer, and direction flag if necessary.

45) As you will discover in the lab experiments, the MOVSD instruction can move a block of data much faster than MOVSB or MOVSW can move that same block. Unfortunately, it can only move a block that contains an even multiple of four bytes. Write a "fastcopy" routine that uses the MOVSD instruction to copy all but the last one to three bytes of a source block to the destination block and then manually copies the remaining bytes between the blocks. Write a main program with several boundary test cases to verify correct operation. Compare the performance of your fastcopy procedure against the use of the MOVSB instruction.

46) Write a macro that computes the absolute value of an integer or floating point parameter. It should generate the appropriate code based on the type of the macro parameter.

47) Write a program that implements and tests the _repeat.._until(expr) statement using HLA's multi-part macros. The macros should expand into JT or JF statements (i.e., don't simply expand these into HLA's repeat..until statement).

48) Write a program that implements and tests the _begin.._exit.._exitif.._end statements using HLA's multi-part macros. Your macros should expand into JMP, JT, and/or JF instructions; do not expand the

text to HLA's BEGIN..EXIT..EXITIF..END statement. Note: you do not have to handle procedure or program exits in this assignment.

49) The HLA #PRINT statement does not provide numeric formatting facilities. If you specify a constant integer expression as an operand, for example, it will simply print that integer using the minimum number of print positions the integer output requires. If you wish to display columns of numbers in a compile-time program, this can be a problem. Write a macro, *fmtInt(integer, size)*, that accepts an integer expression as its first operand and a field width as its second operand and returns a string with the integer representation of that constant right justified in a field whose width the second parameter specifies. Note that you can use the @string function to convert the integer to a string and then you can use the string concatenation operator and the @strset function to build the formatted string. Test your macro thoroughly; make sure it works properly with the specified size is less than the size the integer requires.

50) Write a function and a macro that compute signum (Signum(i): -1 if i < 0, 0 if i=0, +1 if i>0). Write a short main program that invokes the macro three times within the body of a FOR loop: once with the value -10, once with the value zero, and once with the value +10. Adjust the loop control variable so the program requires approximately ten seconds to execute. Next, add a second loop to the main program that executes the same number of iterations as the first loop; in that loop body, however, place a call to signum function you've written. Compare the execution times of the two loops.

51) Add a remainder (modulo) operator to Program 9.7. Use "\" as the symbol for the mod operator (for you "C" programmers out there, "%" is already in use for binary numbers in this program). The mod operator should have the same precedence and associativity as the multiplication and division operators.

52) Modify Program 9.7 (or the program in problem (51), above) to support signed integers rather than unsigned integers.

53) Modify Program 13.21 in the lab exercises to add _break and _continue statements to the _for.._endfor and _while.._endwhile loops. Of course, your new control structures must provide the same tracing facilities as Program 13.21 currently provides.

54) Modify Program 13.21 to include an _if.._elseif.._else.._endif statement with the tracing facilties.

55) Modify Program 13.21 to include _repeat.._until and _forever.._endfor loops. Be sure to provide _break and _continue macros for each of these loops. Of course, your new control structures must provide the same tracing facilities that Program 13.21 currently provides.

56) Modify Program 13.21 to include an _switch.._case.._default.._endswitch statement with tracing facilties. The trace output (if engaged) should display entry into the _switch statement, display the particular _case (or _default) the statement executes, and the display an appropriate message upon exit (i.e., executing _endswitch).

57) Add a "triangle" class to the shapes class in the Classes chapter's sample program. The object should draw images that look like the following:

The minimum width and height of a triangle object should be 2x2. The object itself will always have a width that is equal to (height - 1) * 2.

See the *diamond* class implementation to get a good idea how to draw a triangle. Don't forget to handle filled and unfilled shapes (based on the value of the *fillShape* field).

Modify the main program to draw several triangles in addition to the other shapes (demonstrate the correct operation of filled and unfilled triangles in your output).

58) Add a "parallelogram" class to the Classes chapter's sample program. The parallelogram class should draw images that look like the following:

```
    --------
   /       /
  /       /
 --------
```

The width of a parallelogram will always be equal to the width of the base plus the height minus one (e.g., in the example above, the base width is eight and the height is four, so the overall width is 8+(4-1) = 11).

59) Modify the parallelogram class from the project above to include a boolean data field *slantRight* that draws the above parallelogram if true and draws the following if false:

```
    --------
    \       \
     \       \
      --------
```

Don't forget to initialize the *slantRight* field (to true) inside the constructor.

60) Add a "Tab" class to the Class chapter's sample program. The tab class should introduce a new boolean data field, *up*, and draw the following images based on the value of the *up* field.

```
up=false:          up=true:

  ------             /\
  |    |            /  \
   \  /             |    |
    \/              |    |
                    ------
```

Note that the width must always be an even value (just like diamonds and triangles). The height should always be at least width-2. If the height is greater than this, extend the size of the tab by drawing additional vertical bar sections.

5) (Term project) Add a mouse-based user interface to the ASCIIdraw drawing program (the sample program in the Classes chapter). See the HLA console library module for information about reading the mouse position and other console related routines you can use.

13.3 Laboratory Exercises

Accompanying this text is a significant amount of software. The software can be found in the AoA_Software directory. Inside this directory is a set of directories with names like Vol3 and Vol4, each containing additional subdirectories like *Ch06* and *Ch07*, with the names obviously corresponding to chapters in this textbook. All the source code to the example programs in this chapter can be found in the Vol4\Ch08 subdirectory. Please see this directory for more details.

13.3.1 Dynamically Nested TRY..ENDTRY Statements

In this laboratory experiment you will explore the effect of dynamically nesting TRY..ENDTRY statements in an HLA program. This lab exercise uses the following program:

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

    // NestedINTO-
    //
    // This routine reads two hexadecimal values from
    // the user, adds them together (checking for
    // signed integer overflow), and then displays their
    // sum if there was no overflow.

    procedure NestedINTO;
    begin NestedINTO;

        try

            stdout.put( "Enter two hexadecimal values: " );
            stdin.get( al, ah );
            add( ah, al );
            into();
            stdout.put( "Their sum was $", al, nl );

          exception( ex.ValueOutOfRange )

            stdout.put( "The values must be in the range $0..$FF" nl );

          exception( ex.ConversionError )

            stdout.put( "There was a conversion error during input" nl );

        endtry;

    end NestedINTO;

begin DynNestTry;

    // The following code calls NestedINTO to read and add
    // two hexadecimal values.  If an overflow occurs during
    // the addition, an INTO instruction triggers the following
    // exception.

    try

        stdout.put( "Calling NestedINTO" nl );
        NestedINTO();
        stdout.put( "Returned from NestedINTO" nl );

      /*
      exception( ex.IntoInstr );
      */
      AnyException

        stdout.put( "INTO detected an integer overflow" nl );
```



```
    endtry;
    stdout.put( "After ENDTRY" nl );

end DynNestTry;
```

Program 13.1 Dynamically Nested TRY..ENDTRY Statements

Exercise A: Compile and run this program. Supply as input the two values "12" and "34". Describe the output, and why you got this particular output in your lab report.

Exercise B: Supply the two values "7F" and "10" as input to this program. Include the output from the program in your lab report and explain the results.

Exercise C: Supply the two values "AT" and "XY" as input to this program. Include the output from the program in your lab report and explain the results.

Exercise D: Supply the two values "1234" and "100" as input to this program. Include the output from the program in your lab report and explain the results.

Explain the difference between a statically nested control structure and a dynamically nested control structure in your lab report. Also explain why the TRY..ENDTRY statements in this program are dynamically nested.

13.3.2 The TRY..ENDTRY Unprotected Section

In this laboratory exercise you will explore what happens if you attempt to break out of a TRY..ENDTRY statement in an inappropriate fashion. This exercise makes use of the following program:

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

begin UnprotectedClause;

    // The following loop forces the user to enter
    // a pair of valid eight-bit hexadecimal values.
    // Note that the "unprotected" clause is commented
    // out (this is a defect in the code).  Follow the
    // directions in the lab exercise concerning this
    // statement.

    forever

        try

            stdout.put( "Enter two hexadecimal values: " );
            stdin.get( al, ah );

          //unprotected

            break;

          exception( ex.ValueOutOfRange )

            stdout.put( "The values must be in the range $0..$FF" nl );

          exception( ex.ConversionError )
```



```
            stdout.put( "There was a conversion error during input" nl );

        endtry;

    endfor;
    add( ah, al );
    stdout.put( "Their sum was $", al, nl );

    stdout.put( "Enter another hexadecimal value: " );
    stdin.get( al );
    stdout.put( "The value you entered was $", al, nl );

end UnprotectedClause;
```

Program 13.2 The TRY..ENDTRY Unprotected Section

Exercise A: Compile and run this program. When it asks for two hexadecimal values enter the values "12" and "34". When this program asks for a third hexadecimal value, enter the text "xy". Include the output of this program in your lab report and explain what happened.

Exercise B: The UNPROTECTED statement in this program is commented out. Remove the two slashes before the UNPROTECTED keyword and repeat exercise A. Explain the difference in output between the two executions of this program.

Exercise C: Put a TRY..ENDTRY block around the second stdin.get call in this program (it must handle the ex.ConversionError exception). Remove the UNPROTECTED clause (i.e., comment it out again) in the first TRY..ENDTRY block. Repeat exercise A. Include the source code and output of the program in your lab report. Explain the difference in execution between exercises A, B, and C in your lab report.

13.3.3 Performance of SWITCH Statement

In this laboratory exercise you will get an opportunity to explore the difference in performance between the jump table implementation of a SWITCH statement versus the IF..ELSEIF implementation of a switch. Note that this code relies upon the Pentium RDTSC instruction. If you do not have access to a CPU that supports this instruction you will have to skip this exercise or rewrite the code to use timing loops to approximate the running time (see "Timing Various Arithmetic Instructions" on page 712 to see how you can use loops to increase the running time to the point you can measure the difference between the two algorithms using a stopwatch).

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

static
    Cycles: qword;

    JmpTbl:dword[11] :=
        [
            &CaseDefault,   //0
            &case123,       //1
            &case123,       //2
            &case123,       //3
            &case4,         //4
            &CaseDefault,   //5
            &CaseDefault,   //6
```



```
            &case78,        //7
            &case78,        //8
            &case9,         //9
            &case10         //10
        ];



begin switchStmt;

    stdout.put( "Switch vs. IF/ELSEIF demo" nl nl );


    stdout.put
    (
        "Counting the cycles for 10 invocations of the SWITCH stmt:"
        nl
        nl
    );

    // Start timing the number of cycles required by the following
    // loop.  Note that this code requires a Pentium or compatible
    // processor that supports the RDTSC instruction.

    rdtsc();
    push( edx );
    push( eax );

    // The following loop cycles through the values 0..15
    // in order to ensure that we hit all the cases and
    // then some (not accounted for in the jump table).

    mov( 15, esi );
    xor( edi, edi );
    Rpt16TimesA:

        cmp( esi, 10 );         // Cases beyond 10 go to the default label
        ja CaseDefault;         // 'cause we only have 11 entries in the tbl.
        jmp( JmpTbl[ esi*4 ] ); // Jump to the specified case handler.


            case123:            // Handls cases 1, 2, and 3.

                jmp EndCase;

            case4:              // Handles case ESI = 4.

                jmp EndCase;



            case78:             // Handles cases ESI = 7 or 8.
                jmp EndCase;

            case9:              // Handle case ESI = 9.
                jmp EndCase;

            case10:             // Handles case ESI = 10.
                jmp EndCase;

            CaseDefault:        // Handles cases ESI = 0, 5, 6, and >=11.
```



```
    EndCase:
    dec( esi );
    jns Rpt16TimesA;


// Calculate the number of cycles required by the code above:
// Note: This requires a Pentium processor (or other CPU that
// has a RDTSC instruction).

rdtsc();
pop( ebx );
sub( ebx, eax );
mov( eax, (type dword Cycles[0] ));

pop( eax );
sub( eax, edx );
mov( edx, (type dword Cycles[4] ));
stdout.put( "Cycles for SWITCH stmt: " );
stdout.putu64( Cycles );




stdout.put
(
    nl
    nl
    "Counting the cycles for 10 invocations of the IF/ELSEIF sequence:"
    nl
    nl
);

// Begin counting the number of cycles required by the IF..ELSEIF
// implementation of the SWITCH statement.

rdtsc();
push( edx );
push( eax );

mov( 15, esi );
xor( edi, edi );
Rpt16TimesB:

    cmp( esi, 1 );
    jb  Try4;
    cmp( esi, 3 );
    ja Try4;

        jmp EndElseIf;

    Try4:
    cmp( esi, 4 );
    jne Try78;

        jmp EndElseIf;

    Try78:
    cmp( esi, 7);
    je Is78;
    cmp( esi, 8);
```



```
            jne Try9;

                Is78:
                jmp EndElseIf;

            Try9:
            cmp( esi, 9 );
            jne Try10;

                jmp EndElseIf;

            Try10:
            cmp( esi, 10 );
            jne DefaultElseIf;

                jmp EndElseIf;


            DefaultElseIf:

            EndElseIf:
            dec( esi );
            jns Rpt16TimesB;


        // Okay, compute the number of cycles required
        // by the IF..ELSEIF version of the SWITCH.
        rdtsc();
        pop( ebx );
        sub( ebx, eax );
        mov( eax, (type dword Cycles[0] ));

        pop( eax );
        sub( eax, edx );
        mov( edx, (type dword Cycles[4] ));
        stdout.put( "Cycles for IF/ELSEIF stmt: " );
        stdout.putu64( Cycles );




end switchStmt;
```

Program 13.3 Performance of SWITCH Statement.

Exercise A: Compile and run this program several times and average the cycles times for the two different implementations. Include the results in your lab report and discuss the difference in timings.

Exercise B: Remove the cases (from both implementations) one at a time until the running time is identical for both implementations. How many cases is the break-even point for using a jump table? Include the source code of the modified program in your lab report.

Exercise C: One feature of this particular program is that the loop control variable cycles through all the possible case values (and then some). Modify both loops so that they still repeat sixteen times, but they do not use the loop control variable as the value to select the particular case. Instead, fix the case so that it aways uses the value one rather than the value of the loop control variable. Rerun the experiment and describe your findings.

Exercise D: Repeat exercise C, except fix the case value at 15 rather than zero. Report your findings in your lab report.

13.3.4 Complete Versus Short Circuit Boolean Evaluation

In this laboratory exercise you will measure the execution time of two instruction sequence. One sequence computes a boolean result using complete boolean evaluation; the other uses short circuit boolean evaluation. Like the previous exercises, the code for this exercise uses the Pentium RDTSC instruction. You will need to modify this code if you intend to run it on a processor that does not support RDTSC.

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

static

    Cycles:     qword;
    theRslt:    string;

    FalseCBE:   string := "Complete Boolean Evaluation result was true";
    TrueCBE:    string := "Complete Boolean Evaluation result was false";

    FalseSC:    string := "Short Circuit Evaluation result was true";
    TrueSC:     string := "Short Circuit Evaluation result was false";

    input:      int8;

    a:          boolean;
    b:          boolean;
    c:          boolean;
    d:          boolean;


begin booleanEvaluation;

    // Get some input from the user that
    // we can use to initialize our boolean
    // varibles with:

    forever

        try

            stdout.put( "Enter an eight-bit signed integer value: " );
            stdin.get( input );
            unprotected break;

          exception( ex.ConversionError )

            stdout.put( "Input contained illegal characters" );

          exception( ex.ValueOutOfRange )

            stdout.put( "Value must be between -128 and +127" );

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



```
endfor;

// Okay, set our boolean variables to the following
// values:
//
//  a := input < 0;
//  b := input >= -10;
//  c := input <= 10;
//  d := input = 0;

cmp( input, 0 );
setl( a );
cmp( input, -10 );
setge( b );
cmp( input, 10 );
setle( c );
cmp( input, 0 );
sete( d );


// Now compute (not a) && (b || c) || d
//
//  (1) using Complete Boolean Evaluation.
//  (2) using Short Circuit Evaluation.



// Start timing the number of cycles required by the following
// code.  Note that this code requires a Pentium or compatible
// processor that supports the RDTSC instruction.

rdtsc();
push( edx );
push( eax );


mov( a,  al );
xor( 1,  al );  // not a
mov( b,  bl );
or ( c,  bl );
and( bl, al );
or ( d,  al );
jz CBEwasFalse;

    mov( FalseCBE, theRslt );
    jmp EndCBE;

CBEwasFalse:

    mov( TrueCBE, theRslt );

EndCBE:

// Calculate the number of cycles required by the code above:
// Note: This requires a Pentium processor (or other CPU that
// has a RDTSC instruction).

rdtsc();
pop( ebx );
sub( ebx, eax );
mov( eax, (type dword Cycles[0] ));
```



```
        pop( eax );
        sub( eax, edx );
        mov( edx, (type dword Cycles[4] ));
        stdout.put( "Cycles for Complete Boolean Evaluation: " );
        stdout.putu64( Cycles );

        stdout.put( nl nl, theRslt, nl );




        // Start timing the number of cycles required by short circuit evaluation.

        rdtsc();
        push( edx );
        push( eax );

        cmp( d, true );
        je SCwasTrue;
        cmp( a, true );
        je SCwasFalse;
        cmp( b, true );
        je SCwasTrue;
        cmp( c, true );
        je SCwasTrue;
        SCwasFalse:

            mov( FalseSC, theRslt );
            jmp EndSC;

        SCwasTrue:

            mov( TrueSC, theRslt );

        EndSC:

        // Calculate the number of cycles required by the code above:
        // Note: This requires a Pentium processor (or other CPU that
        // has a RDTSC instruction).

        rdtsc();
        pop( ebx );
        sub( ebx, eax );
        mov( eax, (type dword Cycles[0] ));

        pop( eax );
        sub( eax, edx );
        mov( edx, (type dword Cycles[4] ));
        stdout.put( "Cycles for Short Circuit Boolean Evaluation: " );
        stdout.putu64( Cycles );

        stdout.put( nl nl, theRslt, nl );

    end booleanEvaluation;
```

Program 13.4 Complete vs. Short Circuit Boolean Evaluation

Exercise A: Compile and run this program. Run the program several times in a row and compute the average execution time, in cycles, for each of the two methods. Be sure to specify the input value you use (use the same value for each run) in your lab report.

Exercise B: Repeat exercise A for each of the following input values: -100, -10, -5, 0, 5, 10, 100. Provide a graph of the average execution times for Complete Boolean Evaluation and Short Circuit Boolean evaluation in your laboratory report.

13.3.5 Conversion of High Level Language Statements to Pure Assembly

For this exercise you will write a short demonstration program that uses the following HLA statements: if..elseif..else..endif, switch..case..default..endswitch (from the HLA Standard Library hll.hhf module), while..endwhile, repeat..until, forever..breakif..endfor, for..endfor, and begin..exit..end (the program doesn't have to do anything particularly useful, though the bodies of these statements should not be empty).

Exercise A: Write, compile, run, and test your program. Describe what the program does in your lab report. Include a copy of the program in your lab report.

Exercise B: Convert the if..elseif..else..endif, while..endwhile, and repeat..until statements to the hybrid control statements that HLA provides (see "Hybrid Control Structures in HLA" on page 802). Rerun the program with appropriate inputs and verify that its behavior is the same as the original program. Describe the changes you've made and include the source code in your lab report.

Exercise C: Create a new version of the program you created in exercise A, this time convert the control structures to their low-level, pure assembly language form. Include the source code with your laboratory report. Comment on the readability of the three programs.

13.3.6 Activation Record Exercises

In this laboratory exercise you will construct and examine procedure activation records. This exercise involves letting HLA automatically construct the activation record for you as well as manual construction of activation records and manually accessing data in the activation record.

13.3.6.1 Automatic Activation Record Generation and Access

The following program calls a procedure and returns the values of EBP and ESP from the procedure after it has constructed the activation record. The main program then computes the size of the activation record by subtracting the difference between ESP before the call and ESP during the call.

```
// This program computes and displays the size of
// a procedure's activation record at run-time.
// This code relies on HLA's high level syntax
// and code generation to automatically construct
// the activation record.

program ActRecSize;
#include( "stdlib.hhf" )

type
    rec:record

            u:uns32;
            i:int32;
```



```
            r:real64;

        endrecord;


    // The following procedure allocates storage for the activation
    // record on the stack and then returns the pointer to the bottom
    // of the activation record (ESP) in the EAX register.  It also
    // returns the pointer to the activation record's base address (EBP)
    // in the EBX register.

    procedure RtnARptr( first:uns32; second:real64; var third:rec ); @nodisplay;
    var
        b:byte;
        c:char;
        w:word;
        d:dword;
        a:real32[4];
        r:rec[4];

    begin RtnARptr;

        mov( esp, eax );
        mov( ebp, ebx );

    end RtnARptr;

var
    PassToRtnARptr: rec;


begin ActRecSize;

    // Begin by saving the ESP and EBP register values in
    // ECX and EDX (respectively) so we can display their
    // values and compute the size of RtnARptr's activation
    // record after the call to RtnARptr.

    mov( esp, ecx );
    mov( ebp, edx );
    RtnARptr( 1, 2.0, PassToRtnARptr );

    // Display ESP/EBP value before and after the call to RtnARptr:

    mov( esp, edi );
    stdout.put( "ESP before call: $", ecx, "  ESP after call: $", edi, nl );

    mov( ebp, edi );
    stdout.put( "EBP before call: $", edx, "  EBP after call: $", edi, nl );

    // Display the activation record information:

    stdout.put( "EBP value within RtnARptr: $", ebx, nl );
    stdout.put( "ESP value within RtnARptr: $", eax, nl );
    sub( eax, ecx );
    stdout.put
    (
        "Size of RtnARptr's activation record: ",
        (type uns32 ecx),
        nl
    );
```



```
end ActRecSize;
```

Program 13.5 Computing the Size of a Procedure's Activation Record

Exercise A: Execute this program and discuss the results in your lab report. Draw a stack diagram of *RtnARptr*'s activation record that carefully shows the position of each named variable in the *RtnARptr* procedure.

Exercise B: Change the parameter *third* from pass by reference to pass by value. Recompile and rerun this program. Discuss the differences between the results from Exercise A and the results in this exercise. Provide a stack diagram that describes the activation record for this version of the program.

13.3.6.2 The _vars_ and _parms_ Constants

Whenever the HLA compiler encounters a procedure declaration, it automatically defines two local *uns32* constants, *_vars_* and *_parms_*, in the procedure. The *_vars_* constant specifies the number of bytes of local (automatic) variables the procedure declares. The *_parms_* constant specifies the number of bytes of parameters the caller passes to the procedure. The following program displays these two values for a typical procedure.

```
// This program demonstrates the use of the _vars_
// and _parms_ constants in an HLA procedure.

program VarsParmsDemo;
#include( "stdlib.hhf" )

type
    rec:record

            u:uns32;
            i:int32;
            r:real64;

        endrecord;


    // The following procedure allocates storage for the activation
    // record and then displays the values of the _vars_ and _parms_
    // constants that HLA automatically creates for the procedure.
    // This procedure also returns the ESP/EBP values in the EAX
    // and EBX registers (respectively).

    procedure VarsAndParms
    (
            first:uns32;
            second:real64;
        var third:rec
    );  @nodisplay;

    var
        b:byte;
```



```
            c:char;
            w:word;
            d:dword;
            a:real32[4];
            r:rec[4];

        begin VarsAndParms;

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

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

            mov( esp, eax );
            mov( ebp, ebx );

        end VarsAndParms;

    var
        PassToProc: rec;


    begin VarsParmsDemo;

        // Begin by saving the ESP and EBP register values in
        // ECX and EDX (respectively) so we can display their
        // values and compute the size of RtnARptr's activation
        // record after the call to RtnARptr.

        mov( esp, ecx );
        mov( ebp, edx );
        VarsAndParms( 2, 3.1, PassToProc );

        // Display ESP/EBP value before and after the call to RtnARptr:

        mov( esp, edi );
        stdout.put( "ESP before call: $", ecx, "  ESP after call: $", edi, nl );

        mov( ebp, edi );
        stdout.put( "EBP before call: $", edx, "  EBP after call: $", edi, nl );

        // Display the activation record information:

        stdout.put( "EBP value within VarsAndParms: $", ebx, nl );
        stdout.put( "ESP value within VarsAndParms: $", eax, nl );
        sub( eax, ecx );
        stdout.put
        (
            "Size of VarsAndParms's activation record: ",
            (type uns32 ecx),
            nl
        );
```



```
end VarsParmsDemo;
```

Program 13.6 Demonstration of the _vars_ and _parms_ Constants

Exercise A: Run this program and describe the output you obtain in your lab report. Explain why the sum of the two constants *_vars_* and *_parms_* does not equal the size of the activation record.

Exercise B: Comment out the "c:char;" declaration in the *VarsAndParms* procedure. Recompile and run the program. Note the output of the program. Now comment out the "d:dword;" declaration in the *VarsAndParms* procedure. In your lab report, explain why eliminating the first declaration did not produce any difference while commenting out the second declaration did (hint: see "The Standard Entry Sequence" on page 813).

13.3.6.3 Manually Constructing an Activation Record

The *_vars_* and *_parms_* constants come in real handy if you decide to construct and destroy activation records manually. The *_vars_* constant specifies how many bytes of local variables you must allocate in the standard entry sequence and the *_parms_* constant specifies how many bytes of parameters you need to remove from the stack in the standard exit sequence. The following program demonstrates the manual construction and destruction of a procedure's activation record using these constants.

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

type
    rec:record

            u:uns32;
            i:int32;
            r:real64;

        endrecord;


    // The following procedure manually allocates storage for the activation
    // record. This procedure also returns the ESP/EBP values in the EAX
    // and EBX registers (respectively).

    procedure VarsAndParms
    (
            first:uns32;
            second:real64;
        var third:rec
    );  @nodisplay; @noframe;

    var
        b:byte;
        c:char;
        w:word;
        d:dword;
        a:real32[4];
```



```
        r:rec[4];

    begin VarsAndParms;

        // The standard entry sequence.
        // Note that the stack alignment instruction is
        // commented out because we know that the stack
        // is properly dword aligned whenever the program
        // calls this procedure.

        push( ebp );                // The standard entry sequence.
        mov( esp, ebp );
        sub( _vars_, esp );         // Allocate storage for local variables.
        //and( $FFFF_FFFC, esp );   // Dword-align ESP.

        stdout.put
        (   nl
            "VarsAndParms allocates ",
            _vars_,
            " bytes of local variables and " nl
            "automatically removes ",
            _parms_,
            " parameter bytes on return." nl
            nl
        );
        mov( esp, eax );
        mov( ebp, ebx );

        // Standard exit sequence:

        mov( ebp, esp );
        pop( ebp );
        ret( _parms_ );             // Removes parameters from stack.

    end VarsAndParms;

static
    PassToProc: rec;
    FourPt2: real64 := 4.2;

begin ManActRecord;

    // Begin by saving the ESP and EBP register values in
    // ECX and EDX (respectively) so we can display their
    // values and compute the size of RtnARptr's activation
    // record after the call to RtnARptr.

    mov( esp, ecx );
    mov( ebp, edx );

    // Though not really necessary for this example, the
    // following code manually constructs the parameters
    // portion of the activation record by pushing the
    // data onto the stack.  This program could have used
    // the HLA high level syntax for the code as well.

    pushd( 3 );                     // Push value of "first" parameter.
    push((type dword FourPt2[4]));  // Push value of "second" parameter.
    push((type dword FourPt2[0]));
    pushd( &PassToProc );           // Push address of record item.
    call VarsAndParms;
```



```
    // Display ESP/EBP value before and after the call to RtnARptr:

    mov( esp, edi );
    stdout.put( "ESP before call: $", ecx, "  ESP after call: $", edi, nl );

    mov( ebp, edi );
    stdout.put( "EBP before call: $", edx, "  EBP after call: $", edi, nl );

    // Display the activation record information:

    stdout.put( "EBP value within VarsAndParms: $", ebx, nl );
    stdout.put( "ESP value within VarsAndParms: $", eax, nl );
    sub( eax, ecx );
    stdout.put
    (
        "Size of VarsAndParms's activation record: ",
        (type uns32 ecx),
        nl
    );

end ManActRecord;
```

Program 13.7 Manual Construction and Destruction of an Activation Record

Exercise A: Compile and run this program. Discuss the output in your lab report. Especially note the values of ESP and EBP before and after the call to the procedure. Does the procedure properly restore all values?

Exercise B: Change the *third* parameter from pass by reference to pass by value. You will also need to change the call to *VarsAndParms* so that you pass the record by value (the easiest way to do this is to use the HLA high level procedure call syntax and let HLA generate the code that copies the record; if you manually write this code, be sure to push 16 bytes on the stack for the *third* parameter). Recompile and run the program. Describe the results in your lab manual.

Exercise C: Add several additional local (automatic) variables to the *VarsAndParms* procedure. Recompile and run the program. Explain why using the *_vars_* and *_parms_* constants when manually constructing the activation record is far better than specifying literal constants in the "sub(xxx, esp);" and "ret(yyy);" instructions in the standard entry and exit sequences.

13.3.7 Reference Parameter Exercise

In this laboratory exercise you will explore the behavior of pass by reference versus pass by value parameters. This program passes a pair of global static variables by value and by reference to some procedures that modify their formal parameters. The program prints the result of the modifications before and after the procedure calls (and the actual modifications). This program also demonstrates passing formal value parameters by reference and passing formal reference parameters by value.

```
// This program demonstrates pass by reference
// and pass by value parameters.

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



```
static
    GlobalV: uns32;
    GlobalR: uns32;

    // ValParm-
    //
    // Demonstrates immutability of actual value parameters.

    procedure ValParm( v:uns32 );
    begin ValParm;

        stdout.newln();
        stdout.put( "ValParm, v(before) = ", v, nl );
        mov( 1, v );
        stdout.put( "ValParm, v(after) = ", v, nl );
        stdout.put( "ValParm, GlobalV  = ", GlobalV, nl );

    end ValParm;



    // RefParm-
    //
    // Demonstrates how to access the value of a pass by
    // reference parameter.  Also demonstrates the mutability
    // of an actual parameter when passed by reference.
    //
    //  Note that on all calls in this program, "r" and "GlobalR"
    //  are aliases of one another.

    procedure RefParm( var r:uns32 );
    begin RefParm;

        push( eax );
        push( ebx );

        // Display the address and value of the reference parameter
        // "r" prior to making any changes to that value.

        mov( r, ebx );          // Get address of value into EBX.
        mov( [ebx], eax );      // Fetch the value into EAX.
        stdout.newln();
        stdout.put( "RefParm, Before assignment:" nl nl );
        stdout.put( "r(address)= $", ebx, nl );
        stdout.put( "r( value )=  ", (type uns32 eax), nl );

        // Display the address and value of the GlobalR variable
        // so we can compare it against the "r" parameter.

        mov( &GlobalR, eax );
        stdout.put( "GlobalR(address) =$", ebx, nl );
        stdout.put( "GlobalR( value ) = ", GlobalR, nl );


        // Okay, change the value of "r" from its current
        // value to "1" and redisplay everything.

        stdout.newln();
        stdout.put( "RefParm, after assignment:" nl nl );
```



```
    mov( 1, (type uns32 [ebx]) );
    mov( [ebx], eax );
    stdout.put( "r(address)= $", ebx, nl );
    stdout.put( "r( value )=  ", (type uns32 eax), nl );

    mov( &GlobalR, eax );
    stdout.put( "GlobalR(address) =$", ebx, nl );
    stdout.put( "GlobalR( value ) = ", GlobalR, nl );

    pop( ebx );
    pop( eax );

end RefParm;



// ValAndRef-
//
// This procedure has a pass by reference parameter and a pass
// by value parameter. It demonstrates what happens when you
// pass formal parameters in one procedure as actual parameters
// to another procedure.

procedure ValAndRef( v:uns32; var r:uns32 );
begin ValAndRef;

    push( eax );
    push( ebx );

    // Reset the global objects to some value other
    // than one and then print the values and addresses
    // of the local and global objects.

    mov( 25, GlobalV );
    mov( 52, v );
    mov( 75, GlobalR );

    stdout.put( nl nl "ValAndRef: " nl );
    lea( eax, v );
    stdout.put( "v's address is $", eax, nl );
    lea( eax, r );
    stdout.put( "r's address is $", eax, nl );
    stdout.put( "r's value is $", (type dword r), nl );
    mov( r, ebx );
    mov( [ebx], eax );
    stdout.put( "r points at ", (type uns32 eax), nl nl );

    // Pass value by value and reference by reference to
    // the ValParm and RefParm procedures.

    ValParm( v );
    RefParm( r );

    stdout.put
    (
        nl
        "Inside ValAndRef after passing v by value, v=",
        v,
        nl
    );
```



```
        // Reset the global parameter values and then pass
        // the reference parameter by value and pass the
        // value parameter by reference.

        mov( 67, GlobalV );
        mov( 76, v );
        mov( 89, GlobalR );
        ValParm( r );
        RefParm( v );

        // Display v's value before we leave.

        stdout.put
        (
            nl
            "Inside ValAndRef after passing v by reference, v=",
            v,
            nl
        );

        pop( ebx );
        pop( eax );

    end ValAndRef;


begin PassByValRef;

    mov( 123435, GlobalV );
    mov( 67890, GlobalR );
    ValParm( GlobalV );
    RefParm( GlobalR );

    ValAndRef( GlobalV, GlobalR );


end PassByValRef;
```

Program 13.8 Parameter Passing Demonstration

Exercise A: Compile and run this program. Include the program output in your lab report. Annotate the output and explain the results it produces.

Exercise B: Modify the main program in this example to manually call ValParm, RefParm, and ValAndRef using the low-level syntax rather than the high level syntax (i.e., you must write the code to push the parameters onto the stack). Verify that you get the same results as before the modification.

13.3.8 Procedural Parameter Exercise

This exercise demonstrates an interesting feature of assembly language: the ability to create a SWITCH-like control structure that directly calls one of several functions rather than simply jumping to a statement via a jump table. This example takes advantage of the fact that the CALL instruction supports memory indirect forms, just like JMP, that allows an indexed addressing mode. The same logic you would use to simulate a SWITCH statement with an indirect JMP (see "SWITCH..CASE..DEFAULT..ENDSWITCH" on page 747) applies to the indirect CALL as well.

This program requests two inputs from the user. The first is a value in the range '1',,'4' that the program uses to select one of four different procedures to call. The second input is an arbitrary unsigned integer input that the program passes as a parameter to the procedure the user selects.

```
// This program demonstrates a CALL-based SWITCH statement.

program callSwitch;
#include( "stdlib.hhf" )

type
    tProcPtr:       procedure( i:uns32 );
    tProcPtrArray:  tProcPtr[4];


    // Here is a set of procedure that we will
    // call indirectly (One, Two, Three, and Four).

    procedure One( i:uns32 ); @nodisplay;
    begin One;

        stdout.put( "One: ", i, nl );

    end One;

    procedure Two( i:uns32 ); @nodisplay;
    begin Two;

        stdout.put( "Two: ", i, nl );

    end Two;

    procedure Three( i:uns32 ); @nodisplay;
    begin Three;

        stdout.put( "Three: ", i, nl );

    end Three;

    procedure Four( i:uns32 ); @nodisplay;
    begin Four;

        stdout.put( "Four: ", i, nl );

    end Four;


static

    UserIn:     uns32;

    // CallTbl is an array of pointers that this program uses
    // to create a "switch" statement that does a call rather
    // than a jump.

    CallTbl:    tProcPtrArray := [ &One, &Two, &Three, &Four ];
```



```
begin callSwitch;

    stdout.put( "Call-based Switch Statement Demo: " nl nl );
    repeat

        stdout.put( "Enter a value in the range 1..4: " );
        stdin.flushInput();
        stdin.getc();
        if( al not in '1'..'4' ) then

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

        endif;

    until( al in '1'..'4' );
    and( $f, al );              // Convert '1'..'4' to 1..4.
    dec( al );                  // Convert 1..4 to 0..3.
    movzx( al, eax );           // Need a 32-bit value for use as index.

    // Get a user input value for use as the parameter.

    push( eax );    // Preserve in case there's an exception.
    forever

        try

            stdout.put( "Enter a parameter value: " );
            stdin.get( UserIn );
            unprotected break;

          exception( ex.ValueOutOfRange )

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

          exception( ex.ConversionError )

            stdout.put
            (
                "Input contained illegal characters, please reenter"
                nl
            );

        endtry;

    endfor;
    pop( eax );     // Restore index into "call table".


    // Using an indirect call rather than an indirect jump,
    // SWITCH off to the appropriate subroutine based on
    // the user's input.

    CallTbl[ eax*4 ]( UserIn ); // Call the specified routine.

end callSwitch;
```


Program 13.9 A CALL-based SWITCH Statement

Exercise A: Compile and run this program four times. Select procedures One, Two, Three, and Four on each successive run of the program (you may supply any user input you desire). Include the program's output in your lab report.

Exercise B: Modify the program to print a short message immediately after the *CallTbl* procedure call. Recompile the program and verify that it really does return immediately after the "CallTbl[eax*4](UserIn);" statement.

Exercise C: Add procedures *Six*, *Eight*, *Nine*, and *DefaultProc* to this program. Modify the program so that it lets the user input an *uns32* value rather than a single character to select the procedure to call. If the user inputs a value other than 1, 2, 3, 4, 6, 8, or 9, call the *DefaultProc* procedure. Be sure to test your program with input values zero and values beyond nine (e.g., 10234). See "SWITCH..CASE..DEFAULT..ENDSWITCH" on page 747 for details if you don't remember how to properly encode a SWITCH statement. Include the source code and the output of a sample run in your lab report. Explain how you designed the CALL/SWITCH statement.

13.3.9 Iterator Exercises

The HLA Standard Library includes several iterators. One such iterator appears in the random number generators package ("rand.hhf"). The *rand.deal* iterator takes a single integer parameter. It returns success the number of times specified by this parameter and then fails (e.g., rand.deal(10) will succeed ten times and fail on the eleventh iteration). On each iteration of the corresponding FOREACH loop, the *rand.deal* iterator will return a randomized value between zero and one less than the parameter value; however, *rand.deal* will return each value only once. That is, "rand.deal(n)" will return the values in the range 0..n-1 in a random order. This iterator was given the name *deal* because it simulates dealing a set of cards from a shuffled deck. of cards (indeed, if you iterate over "rand.deal(52)" you can deal out all 52 possible card values from a standard playing card deck). The following program uses the *rand.deal* function to shuffle a deck of standard playing cards and it displays four hands of five card dealt from this shuffled deck.

```
// This program demonstrates the use of the
// rand.deal iterator.


program dealDemo;
#include( "stdlib.hhf" )

type
    card:   record

                face:   string;
                suite:  char;

            endrecord;

    deck:   card[52];



readonly
    CardValue: string[13] :=
        ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K" ];
```



```
    CardGroup: char[4] :=
        [
            #3, // Symbol for hearts
            #4, // Symbol for diamonds
            #5, // Symbol for clubs
            #6  // Symbol for spades
        ];


    procedure shuffle( var theDeck: deck ); @nodisplay;
    begin shuffle;

        push( eax );
        push( ebx );
        push( ecx );
        push( edx );

        mov( theDeck, ecx );
        foreach rand.deal( 52 ) do

            cdq();              // Zero extend EAX into EDX:EAX.
            div( 13, edx:eax ); // Divide into suites.

            // EAX contains the suite index (0..3) and EDX contains
            // the face index (0..12).
            //
            // Get the suite character and store it away into the
            // suite field of the current card.

            mov( CardGroup[ eax ], bl );
            mov( bl, (type card [ecx]).suite );

            // Get the face value and store it away into the
            // face field of the current card in the deck.

            mov( CardValue[ edx*4 ], edx );
            mov( edx, (type card [ecx]).face );

            // Move on to the next card in the deck.

            add( @size( card ), ecx );

        endfor;
        pop( edx );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end shuffle;




static
    Deck1: deck;
    Hands: card[4, 5];

begin dealDemo;

    // Call the randomize function so we don't deal out the same
```



```
// hands each time this program runs.

rand.randomize();

// Create a shuffled deck and deal out four hands of five cards each.
// Note that the outer loop selects which card a particular player
// receives while the inner loop alternates between the players.

shuffle( Deck1 );
mov( 0, edx );  // EDX selects a card from Deck1.

for( mov( 0, ebx ); ebx < 5; inc( ebx )) do

    for( mov( 0, ecx ); ecx < 4; inc( ecx )) do

        // Compute row-major order into Hands to
        // select Hands[ ecx, ebx ]

        intmul( 5, ecx, edi );
        add( ebx, edi );
        intmul( @size( card ), edi );

        // Copy the next available card in Deck1 (selected by
        // EDX) to the current player:

        mov( Deck1.face[ edx ], eax );
        mov( eax, Hands.face[ edi ] );
        mov( Deck1.suite[ edx ], al );
        mov( al, Hands.suite[ edi ] );

        // Move on to the next card in Deck1.

        add( @size( card ), edx );

    endfor;

endfor;

// Okay, display the hands dealt to each player:

stdout.newln();
for( mov( 0, ecx ); ecx < 4; inc( ecx )) do //Player loop.

    lea( eax, [ecx+1]);     // EAX = ECX+1
    stdout.put( "Player ", (type uns32 eax), ": " );
    for( mov( 0, ebx ); ebx < 5; inc( ebx )) do //Card loop.

        // Compute row-major order into Hands to
        // select Hands[ ecx, ebx ]

        intmul( 5, ecx, edi );
        add( ebx, edi );
        intmul( @size( card ), edi );

        // Display the current card for the current player.

        stdout.put( Hands.face[ edi ]:2, Hands.suite[ edi ], ' ' );

    endfor;
    stdout.newln();
```



```
    endfor;


end dealDemo;
```

Program 13.10 Card Shuffle/Deal Program that uses the rand.deal Iterator

Exercise A: Compile and run this program several times. Include the program output in your lab report. Verify that you get a different set of hands on each execution of the program.

Exercise B: Comment out the call to rand.randomize in the main program. Recompile and run the program several times. Discuss the output of this program versus the output from Exercise A.

Exercise C: Using the console.setOutputAttr function, modify this program so that it prints the player's hands with a white background. Cards from the hearts and diamonds suites should have a red foreground color and cards from the spades and clubs suites should have a black foreground color.

13.3.10 Performance of Multiprecision Multiplication and Division Operations

The extended precision multiplication and division routines appearing in the chapter on Advanced Arithmetic can be found in the files "div128.hla", "div128b.hla", and "mul64.hla" in the appropriate subdirectory. These sample programs contain a main program that provides a brief test of each of these functions. Extract the multiplication and divisions procedures (and any needed support routines) and place these procedure in a new program. In the main program, write some code that times the execution of the calls to these three procedures using the RDTSC instruction (see the laboratory exercises at the end of Chapter Six for details). If you are using a CPU that doesn't support the RDTSC instruction, then put the calls in a loop and measure their time using a stopwatch. Also include code in the main program that times the single precision 32-bit MUL, IMUL, DIV, and IDIV instructions.

Exercise A: Run the program and report the running times of the extended precision and standard operations in your lab report.

Exercise B: The multiplication and division operations take a varying amount of time depending on the values of their operands (both the extended precision procedures and the machine instructions exhibit this behavior). Modify the program to generate a pair of random operands for these operations. Repeatedly call the procedures (or execute the machine instructions) and report the average execution time for these operations. Note: be sure to check for division by zero and any other illegal operations that can occur when using random numbers. Use the random number generation facilities of the HLA Standard Library as your source of random numbers. You should measure about a 1,000 calls to the procedures (or 1,000 different executions of the machine instructions).

13.3.11 Performance of the Extended Precision NEG Operation

Write a program that uses the RDTSC instruction to time the execution of a 64-bit, a 128-bit and a 256-bit NEG operation using the two different algorithms presented in this chapter (NEG/SBB and subtract from zero).

Exercise A: Run the programs and report the timings for the two different forms in your lab report. Also point out which version is smaller in your lab report.

Exercise B: You can also perform an extended precision negation operation by applying the definition of the two's complement operation to an extended precision value (i.e., invert all the bits and add one). Add the code to implement this third form of extended precision negation to your program and report the results in your lab report. Also discuss the size of this third negation algorithm.

13.3.12 Testing the Extended Precision Input Routines

The file "uin128.hla" provides an input procedure to read 128-bit unsigned integers from the standard input device. It is not uncommon for such routines, during initial development to contain defects. Devise a set of tests to help verify that the getu128 procedure is operating properly. You tests should check the range of possible values, properly processing of delimiter characters, proper rejection of illegal characters, proper overflow handling, and so on. Describe the tests, and the programs output with your test data, in your lab report.

13.3.13 Illegal Decimal Operations

The DAA, DAS, AAA, AAS, AAM, and AAD instructions generally assume that they are adjusting for the result of some operation whose operands were legal BCD values. For example, DAA adjusts the value in the AL register after an addition assuming that the packed BCD values added together were legal BCD values. In this laboratory exercise, you will force the use of illegal BCD values just to see what happens.

Exercise A: Add together two illegal decimal values (e.g., $1F and $A2) and follow their addition with the execution of the DAA instruction. Repeat this for several pairs of illegal BCD values. Include the results in your lab report. Try to explain the results in your lab report.

Exercise B: Repeat Exercise A using the SUB and DAS instructions.

Exercise C: Repeat Exercise A using the ADD and AAA instructions.

Exercise D: Repeat Exercise B using the SUB and AAS instructions.

13.3.14 MOVS Performance Exercise #1

The movsb, movsw, and movsd instructions operate at different speeds, even when moving around the same number of bytes. In general, the movsw instruction is twice as fast as movsb when moving the same number of bytes. Likewise, movsd is about twice as fast as movsw (and about four times as fast as movsb) when moving the same number of bytes. Ex15_1.asm is a short program that demonstrates this fact. This program consists of three sections that copy 2048 bytes from one buffer to another 100,000 times. The three sections repeat this operation using the movsb, movsw, and movsd instructions. Run this program and time each phase. **For your lab report:** present the timings on your machine. Be sure to list processor type and clock frequency in your lab report. Discuss why the timings are different between the three phases of this program. Explain the difficulty with using the movsd (versus movsw or movsb) instruction in any program on an 80386 or later processor. Why is it not a general replacement for movsb, for example? How can you get around this problem?

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

static

    Buffer1: byte[2048];
    Buffer2: byte[2048];

begin MovsDemo;

    // Demo of the movsb, movsw, and movsd instructions

    stdout.put
    (
```



```
    "The following code moves a block of 2,048 bytes "
    "around 10,000,000 times."
    nl
    "The first phase does this using the movsb instruction; the second"
    nl
    "phase does this using the movsw instruction; the third phase does"
    nl
    "this using the movsd instruction."
    nl
    nl
    "Press RETURN to begin phase one:"
);
stdin.readLn();

for( mov( 10_000_000, edx ); edx > 0; dec( edx )) do

    lea( esi, Buffer1 );
    lea( edi, Buffer2 );
    cld();
    mov( 2048, ecx );
    rep.movsb();

endfor;

stdout.put
(
    "Phase one complete"
    nl nl
    "Press any key to begin phase two:"
);
stdin.readLn();


for( mov( 10_000_000, edx ); edx > 0; dec( edx )) do

    lea( esi, Buffer1 );
    lea( edi, Buffer2 );
    cld();
    mov( 1024, ecx );
    rep.movsw();

endfor;

stdout.put
(
    "Phase two complete"
    nl nl
    "Press any key to begin phase two:"
);
stdin.readLn();


for( mov( 10_000_000, edx ); edx > 0; dec( edx )) do

    lea( esi, Buffer1 );
    lea( edi, Buffer2 );
    cld();
    mov( 512, ecx );
    rep.movsd();

endfor;
```



```
    stdout.put( “Phase Three Done!” nl );

end MovsDemo;
```

Program 13.11 MOVS Demonstration Program

13.3.15 MOVS Performance Exercise #2

In this exercise you will once again time the computer moving around blocks of 2,048 bytes. Like Ex15_1.asm in the previous exercise, Ex15_2.asm contains three phases; the first phase moves data using the movsb instruction; the second phase moves the data around using the lodsb and stosb instructions; the third phase uses a loop with simple mov instructions. Run this program and time the three phases. **For your lab report:** include the timings and a description of your machine (CPU, clock speed, etc.). Discuss the timings and explain the results (consult Appendix D as necessary).

```
program MovsDemo2;
#include( “stdlib.hhf” )

static

    Buffer1: byte[2048];
    Buffer2: byte[2048];

begin MovsDemo2;

    // Demo of the movsb, movsw, and movsd instructions

    stdout.put
    (
        “The following code moves a block of 2,048 bytes “
        “around 1,000,000 times.”
        nl
        “The first phase does this using the movsb instruction; the second”
        nl
        “phase does this using the lodsb/stosb instructions; “
        “the third phase does”
        nl
        “this using the movsb instruction in a loop.”
        nl
        nl
        “Press RETURN to begin phase one:”
    );
    stdin.readLn();

    for( mov( 1_000_000, edx ); edx > 0; dec( edx )) do

        lea( esi, Buffer1 );
        lea( edi, Buffer2 );
        cld();
        mov( 2048, ecx );
        rep.movsb();
```



```
    endfor;

    stdout.put
    (
        "Phase one complete"
        nl nl
        "Press any key to begin phase two:"
    );
    stdin.readLn();


    for( mov( 1_000_000, edx ); edx > 0; dec( edx )) do

        lea( esi, Buffer1 );
        lea( edi, Buffer2 );
        cld();
        mov( 2048, ecx );
        repeat

            lodsb();
            stosb();
            dec( ecx );

        until( @z );

    endfor;

    stdout.put
    (
        "Phase two complete"
        nl nl
        "Press any key to begin phase two:"
    );
    stdin.readLn();


    for( mov( 1_000_000, edx ); edx > 0; dec( edx )) do

        lea( esi, Buffer1 );
        lea( edi, Buffer2 );
        cld();
        mov( 2048, ecx );
        repeat

            movsb();
            dec( ecx );

        until( @z );

    endfor;

    stdout.put( "Phase Three Done!" nl );


end MovsDemo2;
```

Program 13.12 MOVS Demonstration Program #2

13.3.16 Memory Performance Exercise

In the previous two exercises, the programs accessed a maximum of 4K of data. Since most modern on-chip CPU caches are at least this big, most of the activity took place directly on the CPU (which is very fast). The following exercise is a slight modification that moves the array data in such a way as to destroy cache performance. Run this program and time the results. **For your lab report:** based on what you learned about the 80x86's cache mechanism in Chapter Three, explain the performance differences.

```
program MovsDemo3;
#include( "stdlib.hhf" )

var

    Buffer1: byte[256*1024];
    Buffer2: byte[256*1024];

begin MovsDemo3;

    // Demo of the movsb, movsw, and movsd instructions

    stdout.put
    (
        "The following code moves a block of 256K bytes "
        "around 10,000 times."
        nl
        "The first phase does this using the movsb instruction; the second"
        nl
        "phase does this using the lodsb/stosb instructions; "
        "the third phase does"
        nl
        "this using the movsb instruction in a loop."
        nl
        nl
        "Press RETURN to begin phase one:"
    );
    stdin.readLn();

    for( mov( 10_000, edx ); edx > 0; dec( edx )) do

        lea( esi, Buffer1 );
        lea( edi, Buffer2 );
        cld();
        mov( 256*1024, ecx );
        rep.movsb();

    endfor;

    stdout.put
    (
        "Phase one complete"
        nl nl
        "Press any key to begin phase two:"
    );
    stdin.readLn();


    for( mov( 10_000, edx ); edx > 0; dec( edx )) do
```



```
            lea( esi, Buffer1 );
            lea( edi, Buffer2 );
            cld();
            mov( 256*1024, ecx );
            repeat

                lodsb();
                stosb();
                dec( ecx );

            until( @z );

        endfor;

        stdout.put
        (
            "Phase two complete"
            nl nl
            "Press any key to begin phase three:"
        );
        stdin.readLn();


        for( mov( 10_000, edx ); edx > 0; dec( edx )) do

            lea( esi, Buffer1 );
            lea( edi, Buffer2 );
            cld();
            mov( 256*1024, ecx );
            repeat

                movsb();
                dec( ecx );

            until( @z );

        endfor;

        stdout.put( "Phase Three Done!" nl );


    end MovsDemo3;
```

Program 13.13 MOVS Demonstration Program #3

13.3.17 The Performance of Length-Prefixed vs. Zero-Terminated Strings

The following program (Ex15_4.asm on the companion CD-ROM) executes two million string operations. During the first phase of execution, this code executes a sequence of length-prefixed string operations 1,000,000 times. During the second phase it does a comparable set of operation on zero terminated strings. Measure the execution time of each phase. **For your lab report:** report the differences in execution times and comment on the relative efficiency of length prefixed vs. zero terminated strings. Note that the relative performances of these sequences will vary depending upon the processor you use. Based on what you learned in Chapter Three and the cycle timings in Appendix D, explain some possible reasons for relative performance differences between these sequences among different processors.


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

static
    LStr1:      byte := 17;
                byte "This is a string.";

    LResult:    byte[256];

    ZStr1:      byte; @nostorage;
                byte "This is a string", 0;

    ZResult:    byte[256];

    // LStrCpy: Copies a length prefixed string pointed at by SI to
    //          the length prefixed string pointed at by DI.

    procedure LStrCpy( var src:byte; var dest:byte ); @nodisplay;
    begin LStrCpy;

        push( ecx );
        push( edi );
        push( esi );
        pushfd();
        cld();

        mov( src, esi );
        mov( dest, edi );

        movzx( (type byte [esi]), ecx );    // Get length of string in ECX.
        inc( ecx );                         // Include length byte in cnt.
        rep.movsb();                        // Copy the string.

        popfd();
        pop( esi );
        pop( edi );
        pop( ecx );

    end LStrCpy;



    // LStrCat- Concatenates the string pointed at by SI to the end
    //  of the string pointed at by DI using length
    //  prefixed strings.

    procedure LStrCat( var src:byte; var dest:byte ); @nodisplay;
    begin LStrCat;

        push( ecx );
        push( edi );
        push( esi );
        pushfd();
        cld();

        mov( src, esi );
        mov( dest, edi );

        // Compute the final length of the concatenated string
```



```
    mov( [edi], cl );       // Get dest length.
    mov( [esi], ch );       // Get source Length.
    add( ch, [edi] );       // Compute new length.

    // Move SI to the first byte beyond the end of the dest string.

    movzx( cl, ecx );       // Zero extend orig len.
    add( ecx, edi );        // Skip past str.
    inc( edi );             // Skip past length byte.

    // Concatenate the source string (ESI) to the end of
    // the Destination string (EDI)

    rep.movsb();            // Copy 2nd to end of orig.

    popfd();
    pop( esi );
    pop( edi );
    pop( ecx );

end LStrCat;


// LStrCmp- String comparison using two length prefixed strings.
//  ESI points at the first string, EDI points at the
//  string to compare it against.

procedure LStrCmp( var src:byte; var dest:byte ); @nodisplay;

// Mask to clear condition code bits in FLAGS:

const CCmask: word := !%0000_1000_1101_0101 & $FFFF;

begin LStrCmp;

    push( ecx );
    push( edi );
    push( esi );
    pushfd();
    cld();

    mov( src, esi );
    mov( dest, edi );

    // When comparing the strings, we need to compare the strings
    // up to the length of the shorter string.  The following code
    // computes the minimum length of the two strings.

    mov( [esi], cl );       //Get the minimum of the two lengths
    mov( [edi], ch );
    cmp( cl, ch );
    if( @nb ) then

        mov( ch, cl );

    endif;
    movzx( cl, ecx );

    repe.cmpsb();           //Compare the two strings.
    if( @ne ) then
```



```
            // We need to set the condition code bits in the FLAGS
            // value we've saved without affecting any of the other
            // bits, here's the code to do that:

            pop( ecx );         // Retrieve original flags.
            pushfd();           // Push condition codes.
            and( CCmask, (type word [esp] ));
            and( CCmask, cx );
            or( cx, [esp] );

            popfd();
            pop( esi );
            pop( edi );
            pop( ecx );
            exit LStrCmp;

        endif;

        // If the strings are equal through the length of the shorter string,
        // we need to compare their lengths

        mov( [esi], cl );
        cmp( cl, [edi] );

        pop( ecx );         // See comments above regarding flags
        pushfd();           // restoration.
        and( CCmask, (type word [esp] ));
        and( CCmask, cx );
        or( cx, [esp] );

        pop( esi );
        pop( edi );
        pop( ecx );

    end LStrCmp;




    // ZStrCpy- Copies the zero terminated string pointed at by SI
    //          to the zero terminated string pointed at by DI.

    procedure ZStrCpy( var src:byte; var dest:byte ); @nodisplay;
    begin ZStrCpy;

        push( ecx );
        push( edi );
        push( esi );
        push( eax );
        pushfd();
        cld();

        mov( src, esi );
        mov( dest, edi );

        repeat

            mov( [esi], al );
            mov( al, [esi] );
```



```
        inc( esi );
        inc( edi );
        cmp( al, 0 );

    until( @z );

    popfd();
    pop( eax );
    pop( esi );
    pop( edi );
    pop( ecx );

end ZStrCpy;


// ZStrCat- Concatenates the string pointed at by SI to the end
//  of the string pointed at by DI using zero terminated
//  strings.

procedure ZStrCat( var src:byte; var dest:byte ); @nodisplay;
begin ZStrCat;

    push( ecx );
    push( edi );
    push( esi );
    push( eax );
    pushfd();
    cld();

    mov( src, esi );
    mov( dest, edi );

    // Find the end of the destination string:

    mov( $FFFF_FFFF, ecx );     // Search for arbitrary length.
    mov( 0, al );               // Look for zero byte.
    repne.scasb();              // Points EDI beyond zero byte.

    // Copy the source string to the end of the destination string:

    repeat

        mov( [esi], al );
        mov( al, [edi] );
        inc( esi );
        inc( edi );
        cmp( al, 0 );

    until( @z );

    popfd();
    pop( eax );
    pop( esi );
    pop( edi );
    pop( ecx );

end ZStrCat;
```



```
        //  ZStrCmp-    Compares two zero terminated strings.
        //      This is actually easier than the length
        //      prefixed comparison.

        procedure ZStrCmp( var src:byte; var dest:byte ); @nodisplay;

        // Mask to clear condition code bits in FLAGS:

        const CCmask: word := !%0000_1000_1101_0101 & $FFFF;

        begin ZStrCmp;

            push( ecx );
            push( edi );
            push( esi );
            push( eax );
            pushfd();
            cld();

            mov( src, esi );
            mov( dest, edi );

            // Compare the two strings until they are not equal
            // or until we encounter a zero byte.  They are equal
            // if we encounter a zero byte after comparing the
            // two characters from the strings.

            repeat

                mov( [esi], al );
                inc( esi );
                cmp( al, [edi] );
                breakif( @ne );
                inc( edi );
                cmp( al, 0 );

            until( @z );

            // Restore the flags saved on the stack while keeping
            // the current condition code bits.

            pop( ecx );         // Retrieve original flags.
            pushfd();           // Push condition codes.
            and( CCmask, (type word [esp] ));
            and( CCmask, cx );
            or( cx, [esp] );

            popfd();
            pop( eax );
            pop( esi );
            pop( edi );
            pop( ecx );

        end ZStrCmp;


    begin StringComparisons;

        stdout.put
        (
            "The following code does 10,000,000 string "
```



```
            "operations using"
            nl
            "length prefixed strings.  Measure the amount "
            "of time this code"
            nl
            "takes to run."
            nl nl
            "Press ENTER to begin:"
        );
        stdin.readLn();

        for( mov( 10_000_000, edx ); EDX <> 0; dec( edx )) do

            LStrCpy( LStr1, LResult );
            LStrCat( LStr1, LResult );
            LStrCmp( LStr1, LResult );

        endfor;



        stdout.put
        (
            "The following code does 10,000,000 string "
            "operations using"
            nl
            "zero terminated strings.  Measure the amount "
            "of time this code"
            nl
            "takes to run."
            nl nl
            "Press ENTER to begin:"
        );
        stdin.readLn();

        for( mov( 10_000_000, edx ); EDX <> 0; dec( edx )) do

            ZStrCpy( LStr1, LResult );
            ZStrCat( LStr1, LResult );
            ZStrCmp( LStr1, LResult );

        endfor;

        stdout.put( nl "All Done!" nl );

    end StringComparisons;
```

Program 13.14 Performance Evaluation of String Formats

13.3.18 Introduction to Compile-Time Programs

In this laboratory exercise you will run a sample compile-time program that has no run-time component. This compile-time program demonstrates the use of the #while and #print statements as well as the declaration and use of compile-time constants and variables (VAL objects).


```
// Demonstration of a simple compile-time program:

program ctPgmDemo;
const
    MaxIterations:= 10;

val
    i:int32;

#print( “Compile-time loop:” );
#while( i < MaxIterations )

    #print( “i=”, i );
    ?i := i + 1;

#endwhile
#print( “End of compile-time program” );

begin ctPgmDemo;
end ctPgmDemo;
```

Program 13.15 A Simple Compile-Time Program

Exercise A. Compile this program. Describe its output during compilation in your lab report.

Exercise B. Although this source contains only a compile-time program, its compilation also produces a run-time program. After compiling the code, execute the corresponding EXE file this program produces. Explain the result of executing this program in your lab report.

Exercise C. What happens if you move the compile-time program from its current location (in the declaration section) to the body of the main program (i.e., between the BEGIN and END clauses)? Do this and repeat exercises A and B. Include the source code and explain the results in your lab report.

13.3.19 Conditional Compilation and Debug Code

Although the conditional compilation statements (#IF, #ELSE, etc.) are quite useful for many tasks, a principle use for these compile-time statements is to control the emission of debugging code. By using a single constant (e.g., "debug") declared as *true* or *false* at the beginning of your program, you can easily control the compilation of statements that should only appear in the run-time code during debugging runs. The following program has a subtle bug (you will get the opportunity to discover this problem for yourself). The use of debugging statements make locating this problem much easier. You can easily enable or disable the debugging statements by changing the value of the *debug* variable (between *true* and *false*).

```
// Demonstration of conditional compilation:

program condCompileDemo;
#include( “stdlib.hhf” );

// The following constant declaration controls the automatic
// compilation of debug code in the program.  Set “debug” to
// true in order to enable the compilation of the debugging
// code;  set this constant to false to disable the debug code.
```



```
const debug := false;

// Normal variables and constants this program uses:

const
    MaxElements := 10;

static
    uArray:uns32[ MaxElements ];
    LoopControlVariable:uns32;

begin condCompileDemo;

    try

        for
        (
            mov( MaxElements-1, LoopControlVariable );
            LoopControlVariable >= 0;
            dec( LoopControlVariable )
        ) do

            #if( debug )

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

            #endif
            mov( LoopControlVariable, ebx );
            mov( 0, uArray[ ebx*4 ] );

        endfor;

      anyexception

        stdout.put( "Exception $", eax, " raised in loop", nl );
    endtry;

end condCompileDemo;
```

Program 13.16 Using Conditional Compilation to Control Debugging Code

Exercise A. Set the *debug* constant to *false*. Run this program and explain the results in your lab report. Do not try to correct the defect yet (even if the defect is obvious to you).

Exercise B. Set the *debug* constant to *true*. Rerun the program and explain the results in your lab report (if you cannot figure out what the problem is, ask your instructor for help).

Exercise C. Correct the defect and rerun the program. The value of the *debug* constant should still be *true*. Include the program's output in your laboratory report.

Exercise D. Set the value of the *debug* constant to *false* and recompile and run your program. Include the program's output in your lab report. In your lab report, explain how you could use conditional compilation and this *debug* variable to help track down problems in your own programs. In particular, explain why conditional compilation is helpful here (i.e., why not simply insert and remove the debugging code without using conditional compilation?).

13.3.20 The Assert Macro

The HLA Standard Library "excepts.hhf"[2] header file contains a macro, *assert*, that is quite useful for debugging purposes. This macro is defined as follows:

```
macro assert( expr ):
    skipAssertion,
    msg;

    #if( !ex.NDEBUG )

        readonly

            msg:string := @string:expr;

        endreadonly;
        jt( expr ) skipAssertion;

            mov( msg, ex.AssertionStr );
            raise( ex.AssertionFailed );

        skipAssertion:

    #endif

endmacro;
```

The purpose of the *assert* macro is to test a boolean expression. If the boolean expression evaluates *true* then the assert macro does nothing; however, if the boolean expression evaluates *false*, then assert raises an exception (*ex.AssertionFailed*). This macro is quite handy for checking the validity of certain expressions while your program is running (e.g., checking to see if an array index is within the appropriate bounds).

If you take a close look at the *assert* macro definition, you'll discover that an #IF statement surrounds the body of the macro. If the symbol *ex.NDEBUG* (No DEBUG) is true, the *assert* macro does not generate any code; conversely, *assert* will generate the code to test the boolean expression if *ex.NDEBUG* is false. The reason for the #IF statement is to allow you to insert debugging assertions throughout your code and easily disable all of them with a single statement at the beginning of your program. By default, assertions are active and will generate code (i.e., the VAL object *ex.NDEBUG* initially contains *false*). You may disable code generation for assertions by including the following statement at the beginning of your program (after the #Include("stdlib.hhf") or #Include("excepts.hhf") directive which defines the *ex.NDEBUG* VAL object):

```
?ex.NDEBUG := true;
```

You can even sprinkle statements throughout your program to selective turn code emission for the *assert* macro on and off by setting *ex.NDEBUG* to false and true (respectively). However, turning on all asserts or turning off all asserts at the beginning of your program should prove sufficient.

During testing, you should leave all assertions active so the program can help you locate defects (by raising an exception if an assertion fails). Later, when you've debugged your code and are confident that it behaves correctly, you can eliminate the overhead of the *assert* macros by setting the *ex.NDEBUG* object to true.

The following sample program demostrates how to detect a common error (array bounds violation) useing an *assert* macro. This is the program from the previous exercise (with the same problem) adapted to use the assert macro rather than explicit debugging code.

2. The "stdlib.hhf" header file automatically includes the "excepts.hhf" header file.


```
// Demonstration of the Assert macro:

program assertDemo;
#include( "stdlib.hhf" );

// Set the following variable to true to tell HLA
// not to generate code for ASSERT debug macros
// (i.e., if NDEBUG [no debug] is true, turn off
// the debugging code).
//
//  Conversely, set this to false to activate the
// debugging code associated with the assert macros.

val ex.NDEBUG := false;

// Normal variables and constants this program uses:

const
    MaxElements := 10;

static
    uArray:uns32[ MaxElements ];
    LoopControlVariable:uns32;

begin assertDemo;

    for
    (
        mov( MaxElements-1, LoopControlVariable );
        LoopControlVariable >= 0;
        dec( LoopControlVariable )
    ) do

        mov( LoopControlVariable, ebx );

        // The following assert verifies that
        // EBX is the range legal range for
        // elements of the uArray object:

        assert( ebx in 0..@elements(uArray)-1 );
        mov( 0, uArray[ ebx*4 ] );

    endfor;

end assertDemo;
```

Program 13.17 Demonstration of the Assert Macro

Exercise A. Compile and run this program. Describe the results in your laboratory report. (Do not correct the defect in the program.)

Exercise B. Change the statement "?ex.NDEBUG := false;" so that the *ex.NDEBUG* VAL object is set to true. Compile and run the program (with the defect still present). Describe the results in your laboratory report. If you were debugging this code and didn't know the cause of the error, which exception message do you think would help you locate the defect faster? Why? Explain this in your lab report.

Exercise C. Correct the defect (ask your instructor if you don't see the problem) and rerun the program with *ex.NDEBUG* set to *true* and *false* (on separate runs). How does this affect the execution of your (correct) program?

13.3.21 Demonstration of Compile-Time Loops (#while)

In this laboratory exercise you will use the #WHILE compile-time statement for two purposes. First, this program uses #WHILE to generate data for a table during the compilation of the program. Second, this program uses #WHILE in order to unroll a loop (see "Unraveling Loops" on page 800 for more details on unrolling loops). This sample program also uses the #IF compile-time statement, along with the *UnrollLoop* constant, in order to control whether this program generates a FOR loop to manipulate an array or unroll the FOR loop using the #WHILE statement. Here's the source code for this exercise:

```
// Demonstration of the #while statement:

program whileDemo;
#include( "stdlib.hhf" );

// Set the following constant to true in order to use loop unrolling
// when initializing the array at run-time.  Set this constant to
// false to use a run-time FOR loop to initialize the data:

const UnrollLoop := true;

// Normal variables and constants this program uses:

const
    MaxElements := 16;

static
    index:uns32;
    iArray:int32[ MaxElements ] :=
        [

            // The following #while loop initializes
            // each element of the array (except the
            // last element) with the negative version
            // of the index into the array.

            ?i := 0;
            #while( i < MaxElements-1 )

                -i,
                ?i := i + 1;

            #endwhile

            // Initialize the last element (special case here
            // because we can't have a comma at the end of the
            // list).

            -i
        ];

begin whileDemo;
```



```
    // Display the current elements in the array:

    stdout.put( "Initialized elements of iArray:", nl, nl );
    for( mov( 0, ebx ); ebx < MaxElements; inc( ebx )) do

        assert( ebx in 0..MaxElements-1 );
        stdout.put( "iArray[", (type uns32 ebx), "] = ", iArray[ ebx*4], nl );

    endfor;

    #if( UnrollLoop )

        // Reset the array elements using an unrolled loop.

        ?i := 0;
        #while( i < MaxElements )

            mov( MaxElements-i, iArray[ i*4 ] );
            ?i := i + 1;

        #endwhile

    #else // Reset the array using a run-time FOR loop.

        for( mov( 0, ebx ); ebx < MaxElements; inc( ebx )) do

            mov( MaxElements, eax );
            sub( ebx, eax );
            assert( ebx < MaxElements );
            mov( eax, iArray[ ebx*4 ] );

        endfor;

    #endif

    // Display the new array elements (should be MaxElements downto 1):

    stdout.put( nl, "Reinitialized elements of iArray:", nl, nl );
    for( mov( 0, ebx ); ebx < MaxElements; inc( ebx )) do

        assert( ebx in 0..MaxElements-1 );
        stdout.put( "iArray[", (type uns32 ebx), "] = ", iArray[ ebx*4], nl );

    endfor;
    stdout.put( nl, "All done!", nl, nl );




end whileDemo;
```

Program 13.18 #While Loop Demonstration

Exercise A. Compile and execute this program. Include the output in your laboratory report.

Exercise B. Change the *UnrollLoop* constant from true to false. Recompile and run the program. Include the output in your laboratory report. Describe the differences between the two programs (in particular, you should take care to describe how these two runs produce their output).

13.3.22 Writing a Trace Macro

When debugging HLA programs, especially in the absence of a good debugging tool, a common need is to print a brief message that effectively says "here I am" while the program is running. The execution of such a statement lets you know that the program has reached a certain point in the source code during execution.

If you only need a single such statement, probably the easiest way to achieve this is to use the *stdout.put* statement as follows:

```
stdout.put( "Here I am" nl );
```

Of course, if you have more than one such statement in your program you will need to modify the string your print so that each stdout.put statement prints a different message (so you can easily identify which statements execute in the program). A typical solution is to print a unique number with each string, e.g.,

```
stdout.put( "Here I am at point 1" nl );
    .
    .
    .
stdout.put( "Here I am at point 2" nl );
    .
    .
    .
stdout.put( "Here I am at point 3" nl );
    .
    .
    .
```

There is a big problem with this approach: it's very easy to become confused and repeat the same number twice. This, of course, does you no good when the program prints that particular value. One way to handle this is to print the line number of the *stdout.put* statement. Unless you put two such statements on the same line (which would be very unusual), each call to *stdout.put* would produce a unique output value. You can easily display the line number of the statement using the HLA @LineNumber compile-time function:

```
stdout.put( "Here I am at line ", @LineNumber, nl );
```

There is a problem with inserting code like this into your program: you might forget to remove it later. As noted in section 7.7, you can use the conditional compilation directives to let you easily turn debugging code on and off in your program. E.g., you could replace the statement above by

```
#if( debug )

    stdout.put( "Here I am at line ", @LineNumber, nl );

#endif
```

Now, by setting the constant *debug* to *true* or *false* at the beginning of your program, you can easily turn the code generation of these "trace" statements on and off.

While this is very close to what we need, there is still a problem with this approach: it's a lot of code. That makes it difficult to write and the amount of incidental code in your program obscures the statements that do actual work, making your program harder to read and understand. What would really be nice is a single, short, statement that automatically generates code like the above if *debug* is *true* (and doesn't generate

anything if *debug* is *false*). I.e., what would would really like is to be able to write something like the following:

```
trace;
```

Presumably, *trace* expands into code similar to the above.

In this laboratory exercise you will get to use just such a macro. The actual definition of *trace* is somewhat complicated by the behavior of eager vs. deferred macro parameter expansion (See “Eager vs. Deferred Macro Parameter Evaluation” on page 977.). Also, *trace* is actually a TEXT object rather than a macro so that *trace* can automatically expand to a macro invocation and pass in a line number parameter (this saves having to type something like "trace(@LineNumber)" manually). Here is a program that defines and uses *trace* as described above:

```
// Demonstration of macros and the development of a debugging tool:

program macroDemo;
#include( “stdlib.hhf” );


// The TRACE “macro”.
//
//  Putting “trace;” at a point in an executable section of
// your program tells HLA to print the line number of that
// statement when it encounters the statement during program
// execution.  Setting the “traceOn” variable to true or false
// turns on and off the display of the trace line numbers
// during execution.
//
// Note that the “trace” object is actually a text constant
// that expands to the “tracestmt” macro invocation.  This
// saves you from having to type “@linenumber” as a parameter
// to every tracestmt invocation (see the text to learn why
// you can’t simply bury “@linenumber” within the tracestmt
// macro body).

const traceOn := true;

const trace:text := “tracestmt( @eval( @LineNumber ))”;
macro tracestmt( LineNumber );

    #if( traceOn )

        stdout.put( “line: “, LineNumber, nl );

    #endif

endmacro;

// Normal variables and constants this program uses:

const
    MaxElements := 16;

static
    index:uns32;
    iArray:int32[ MaxElements ];

begin macroDemo;
```



```
    trace;
    mov( 0, ebx );
    trace;
    while(  ebx < MaxElements ) do

        mov( MaxElements, eax );
        sub( ebx, eax );
        assert( ebx < MaxElements );
        mov( eax, iArray[ ebx*4 ] );
        inc( ebx );

    endwhile;
    trace;

    // Display the new array elements (should be MaxElements downto 1):

    stdout.put( nl, "Elements of iArray:", nl, nl );
    trace;
    for( mov( 0, ebx ); ebx < MaxElements; inc( ebx )) do

        assert( ebx in 0..MaxElements-1 );
        stdout.put( "iArray[", (type uns32 ebx), "] = ", iArray[ ebx*4], nl );

    endfor;
    trace;
    stdout.put( nl, "All done!", nl, nl );

end macroDemo;
```

Program 13.19 Trace Macro

Exercise A. Compile and run this program. Include the output in your laboratory report.

Exercise B. Add additional "trace;" statements to the program (e.g., stick them inside the WHILE and FOR loops). Recompile and run the program. Include the output in your lab report. Comment on the usefulness of the *trace* macro in your lab report.

Exercise C. Change the *traceOn* constant to *false*. Recompile and run your program. Explain the output in your lab report.

13.3.23 Overloading

A nifty feature in the C++ language is *function overloading*. Function overloading lets you use the same function (procedure) name for different functions leaving the compiler to differentiate the functions by the number and types of the function's parameters. Although HLA does not directly support procedure overloading, it is very easy to simulate this using HLA's compile-time language. The following sample program demonstrates how to write a *Max* "function" that computes the maximum of two values whose types can be *uns32*, *int32*, or *real32*.

```
// Demonstration of using macros to implement function overloading:

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



```
procedure i32Max( val1:int32; val2:int32; var dest:int32 );
begin i32Max;

    push( eax );
    push( ebx );

    mov( val1, eax );
    mov( dest, ebx );
    if( eax < val2 ) then

        mov( val2, eax );

    endif;
    mov( eax, [ebx] );
    pop( ebx );
    pop( eax );

end i32Max;


procedure u32Max( val1:uns32; val2:uns32; var dest:uns32 );
begin u32Max;

    push( eax );
    push( ebx );

    mov( val1, eax );
    mov( dest, ebx );
    if( eax < val2 ) then

        mov( val2, eax );

    endif;
    mov( eax, [ebx] );
    pop( ebx );
    pop( eax );

end u32Max;


procedure r32Max( val1:real32; val2:real32; var dest:real32 );
begin r32Max;

    push( eax );
    push( ebx );

    mov( dest, ebx );
    fld( val1 );
    fld( val2 );
    fcompp();
    fstsw( ax );
    sahf();
    if( @b ) then

        mov( val1, eax );   // Since real32 fit in EAX, just use EAX

    else

        mov( val2, eax );
```



```
    endif;
    mov( eax, [ebx] );
    pop( ebx );
    pop( eax );

end r32Max;



macro Max( val1, val2, dest );

    #if( @typeName( val1 ) = "uns32" )

        u32Max( val1, val2, dest );

    #elseif( @typeName( val1 ) = "int32" )

        i32Max( val1, val2, dest );

    #elseif( @typeName( val1 ) = "real32" )

        r32Max( val1, val2, dest );

    #else

        #error
        (
            "Unsupported type in 'Max' function: '" +
            @typeName( val1 ) +
            "'"
        )

    #endif

endmacro;


static
    u1:uns32 := 5;
    u2:uns32 := 6;
    ud:uns32;

    i1:int32 := -5;
    i2:int32 := -6;
    id:int32;

    r1:real32 := 5.0;
    r2:real32 := -6.0;
    rd:real32;


begin overloadDemo;

    Max( u1, u2, ud );
    Max( i1, i2, id );
    Max( r1, r2, rd );

    stdout.put
    (
        "Max( ", u1, ", ", u2, " ) = ", ud, nl,
        "Max( ", i1, ", ", i2, " ) = ", id, nl,
```



```
        "Max( ", r1, ", ", r2, " ) = ", rd, nl
    );

end overloadDemo;
```

Program 13.20 Procedure Overloading Demonstration

Exercise A. Compile and run this program. Include the output in your laboratory report.

Exercise B. Add an additional statement to the main program that attempts to compute the maximum of an *int32* and *real32* object (storing the result in an *uns32* object). Compile the program and explain the results in your lab report.

Exercise C. Extend the *Max* macro so that it can handle *real64* objects in addition to *uns32*, *int32*, and *real32* objects. Note that you will have to write an *r64Max* function as well as modify the *Max* macro (hint: you will not be able to simply clone the *r32Max* procedure to achieve this).

13.3.24 Multi-part Macros and RatASM (Rational Assembly)

While the *trace* macro of the previous section is very nice and quite useful for debugging purpose, it does have one major drawback, you have to explicitly insert *trace* macro invocations into your source code in order to take advantage of this debugging facility. If you have an existing program, into which you have not inserted any *trace* invocations, it might be a bit of effort to instrument your program by inserting dozens or hundreds of *trace* invocations into the program. Wouldn't it be nice if HLA code do this for you automatically?

Unfortunately, HLA cannot automatically insert *trace* invocations into your program for you. However, with a little preparation, you can almost achieve this goal. To do this, we'll define a special Domain Specific Embedded Language (See "Domain Specific Embedded Languages" on page 1003.) that defines some of the high level language statements found in HLA (similar to what appears in "Implementing the Standard HLA Control Structures" on page 1003). However, rather than simply mimic the existing control structures, our new control structures will also automatically inject some output statements into the code if the *traceOn* constant is *true*. We'll call this DSEL *RatASM* (after *RatC*, which adds the same type of tracing features to the C/C++ language). *RatASM* is short for Rational Assembly (note that the names *RatASM* and *RatC* are derivations of RATFOR, Kernighan and Plauger's RATional FORtran preprocessor).

RatASM works as follows: rather than using statements like WHILE..DO..ENDWHILE or FOR..DO..ENDFOR, you use the *_while.._do.._endwhile* and *_for.._do.._endfor* macros to do the same thing. These macros essentially expand into the equivalent HLA high level language statements. If the *traceOn* constant is *true*, then these macros also emit some additional code to display the name of the control structure and the corresponding line number. The following sample program provides multi-part macros for the *_for* and *_while* statements that support this tracing feature:

```
// Demonstration of multi-part macros and the development
// of yet another debugging tool:

program RatASMdemo;
#include( "stdlib.hhf" );


// The TRACE "macro".
```



```
//
//  Putting "trace;" at a point in an executable section of
// your program tells HLA to print the line number of that
// statement when it encounters the statement during program
// execution.  Setting the "traceOn" variable to true or false
// turns on and off the display of the trace line numbers
// during execution.
//
// Note that the "trace" object is actually a text constant
// that expands to the "tracestmt" macro invocation.  This
// saves you from having to type "@linenumber" as a parameter
// to every tracestmt invocation (see the text to learn why
// you can't simply bury "@linenumber" within the tracestmt
// macro body).

const traceOn := true;

const trace:text := "tracestmt( @eval( @LineNumber ))";
macro tracestmt( LineNumber );

    #if( traceOn )

        stdout.put( "trace(", LineNumber, ")", nl );

    #endif

endmacro;


macro traceRatASM( LineNumber, msg );

    #if( traceOn )

        stdout.put( msg, ": ", LineNumber, nl );

    #endif

endmacro;


// The "RatASM" (rational assembly) "FOR" statement.
//
//  Behaves just like the standard HLA FOR statement
// except it provides the ability to transparently
// trace the execution of FOR statements in a program.

const _for:text := "?_CurStmtLineNumber := @LineNumber; raFor";
macro raFor( init, expr, increment ):ForLineNumber;
    ?ForLineNumber := _CurStmtLineNumber;
    traceRatASM( _CurStmtLineNumber, "FOR" );
    for( init; expr; increment ) do

        traceRatASM( _CurStmtLineNumber, "for" );

  keyword _do;
  terminator _endfor;

    endfor;
    traceRatASM( _CurStmtLineNumber, "endfor" );

endmacro;
```



```
// The “RatASM” (rational assembly) “WHILE” statement.
//
//  Behaves just like the standard HLA WHILE statement
// except it provides the ability to transparently
// trace the execution of WHILE statements in a program.

const _while:text := “?_CurStmtLineNumber := @LineNumber; raWhile”;
macro raWhile( expr ):WhileLineNumber;
    ?WhileLineNumber := _CurStmtLineNumber;
    traceRatASM( _CurStmtLineNumber, “WHILE” );
    while( expr ) do

        traceRatASM( _CurStmtLineNumber, “while” );

  keyword _do;
  terminator _endwhile;


    endwhile;
    traceRatASM( _CurStmtLineNumber, “endwhile” );

endmacro;



// Normal variables and constants this program uses:

const
    MaxElements := 16;

static
    index:uns32;
    iArray:int32[ MaxElements ];

begin RatASMdemo;

    trace;
    mov( 0, ebx );
    _while(  ebx < MaxElements ) _do

        mov( MaxElements, eax );
        sub( ebx, eax );
        assert( ebx < MaxElements );
        mov( eax, iArray[ ebx*4 ] );
        inc( ebx );

    _endwhile;

    // Display the new array elements (should be MaxElements downto 1):

    stdout.put( nl, “Elements of iArray:”, nl, nl );
    _for( mov( 0, ebx ), ebx < MaxElements, inc( ebx )) _do

        assert( ebx in 0..MaxElements-1 );
        stdout.put( “iArray[“, (type uns32 ebx), “] = “, iArray[ ebx*4], nl );

    _endfor;
    stdout.put( nl, “All done!”, nl, nl );
```



```
end RatASMdemo;
```

Program 13.21 Demonstration of RatASM _WHILE and _FOR Loops

Exercise A. Compile and run this program. Include the output in your lab report. Note that some trace messages display their text in uppercase while others display their text in lowercase. Figure out the difference between these two messages and describe the difference in your lab report.

Exercise B. Set the *traceOn* constant to *false*. Recompile and run the program. Describe the output in your lab report.

Exercise C. Using the *_while* and *_for* definitions as a template, create a *_repeat.._until* RatASM statement and modify the main program by adding this statement (your choice what the loop will actually do). Include the source code in your lab report. Compile and run the program with the *traceOn* constant set to *true* and then set to *false*. Include the output of the program in your lab report.

13.3.25 Virtual Methods vs. Static Procedures in a Class

Class methods and procedures are generally interchangeable in a program. One exception occurs with polymorphism. If you have a pointer to an object rather than a standard object variable, the semantics of method versus procedure calls are different. In this exercise you will explore those differences.

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

type
    baseClass: class

        procedure Create; returns( "esi" );
        procedure aProc;
        method aMethod;

    endclass;

    derivedClass: class inherits( baseClass )

        override procedure Create;
        override procedure aProc;
        override method aMethod;

    endclass;


// Methods for the baseClass class type:

procedure baseClass.Create; @nodisplay; @noframe;
begin Create;

    stdout.put( "called baseClass.Create", nl );

    push( eax );
    if( esi = 0 ) then

        mov( malloc( @size( baseClass ) ), esi );
```



```
    endif;
    mov( &baseClass._VMT_, this._pVMT_ );
    pop( eax );
    ret();

end Create;


procedure baseClass.aProc; @nodisplay; @noframe;
begin aProc;

    stdout.put( "Called baseClass.aProc" nl );
    ret();

end aProc;

method baseClass.aMethod; @nodisplay; @noframe;
begin aMethod;

    stdout.put( "Called baseClass.aMethod" nl );
    ret();

end aMethod;



// Methods for the derivedClass class type:

procedure derivedClass.Create; @nodisplay; @noframe;
begin Create;

    stdout.put( "called derivedClass.Create", nl );

    push( eax );
    if( esi = 0 ) then

        mov( malloc( @size( derivedClass ) ), esi );

    endif;
    mov( &derivedClass._VMT_, this._pVMT_ );
    pop( eax );
    ret();

end Create;


procedure derivedClass.aProc; @nodisplay; @noframe;
begin aProc;

    stdout.put( "Called derivedClass.aProc" nl );
    ret();

end aProc;

method derivedClass.aMethod; @nodisplay; @noframe;
begin aMethod;

    stdout.put( "Called derivedClass.aMethod" nl );
    ret();
```



```
end aMethod;



static
    vmt( baseClass );
    vmt( derivedClass );
var
    b: baseClass;
    d: derivedClass;

    bPtr: pointer to baseClass;
    dPtr: pointer to derivedClass;

    generic: pointer to baseClass;

begin PolyMorphDemo;

    // Deal with the b and d objects:

    stdout.put( "Manipulating 'b' object:" nl );

    b.Create();
    b.aProc();
    b.aMethod();

    stdout.put( nl "Manipulating 'd' object:" nl );

    d.Create();
    d.aProc();
    d.aMethod();

    // Now work with pointers to the objects:

    stdout.put( nl "Manipulating 'bPtr' object:" nl );

    mov( baseClass.Create(), bPtr );
    bPtr.aProc();
    bPtr.aMethod();

    stdout.put( nl "Manipulating 'dPtr' object:" nl );

    mov( derivedClass.Create(), dPtr );
    dPtr.aProc();
    dPtr.aMethod();

    // Demonstrate polymorphism using the 'generic' pointer.

    stdout.put( nl "Manipulating 'generic' object:" nl );

    mov( bPtr, generic );
    generic.aProc();
    generic.aMethod();

    mov( dPtr, generic );
    generic.aProc();
    generic.aMethod();
```



```
end PolyMorphDemo;
```

Program 13.22 Polymorphism Demonstration

Exercise A: Compile and run the program. Include the output in your lab report. Describe the output and explain why you got the output you did. Especially explain the output of the *generic* procedure and method invocations.

Exercise B: Add a second pointer variable, *generic2*, whose type is a pointer to *derivedClass*. Initialize this pointer with the value from the *dPtr* variable and invoke the *dPtr.aProc* procedure and *dPtr.aMethod* method. Run the program and include the output in your lab report. Explain why these procedure/method invocations produce different output than the output from the *generic* procedure and method invocations.

13.3.26 Using the _initialize_ and _finalize_ Strings in a Program

Although HLA does not provide a mechanism that automatically invokes an object's constructors when the procedure, iterator, method, or program that contains the object's declaration begins execution, you can simulate this by using HLA's *_initialize_* string. Likewise, HLA does not automatically call a class destructor associated with an object when that object goes out of scope; still, you can simulate this by using the *_finalize_* string. In this laboratory exercise you will experiment with these two string values.

Note: the following program demonstrates the use of a class definition macro that manipulates the *_initialize_* string in order to automatically invoke a class' constructor. Adding a class destructor and modifying the value of the *_finalize_* string is one of the activities you will do in this laboratory exercise.

```
// Using _initialize_ and _finalize_ in a program.

program InitFinalDemo;
#include( "stdlib.hhf" );

// Define a simple class with a constructor and
// a method to demonstrate the use of the _initialize
// string.
type
    _myClass: class

        var
            s:string;

        procedure Create; returns( "esi" );
        method put;
        method assign( theValue:string );

    endclass;

// Define a "pseudo-type".  That is a macro that we use
// in place of the "_myClass" name.  In addition to actually
// defining the macro name, this macro will modify the
// _initialize_ string so that the procedure/program/whatever
// containing the object declaration does an automatic call to
// the constructor.

macro myClass:theID;
```



```
    forward( theID );
    ?_initialize_ := _initialize_ + @string:theID + ".Create();";
    theID:_myClass

endmacro;



// Methods for the baseClass class type:

procedure _myClass.Create; @nodisplay; @noframe;
begin Create;

    push( eax );
    if( esi = 0 ) then

        mov( malloc( @size( _myClass ) ), esi );

    endif;
    mov( &_myClass._VMT_, this._pVMT_ );

    // Within the constructor, initialize the string
    // to a reasonable value:

    mov( str.a_cpy( "" ), this.s );

    pop( eax );
    ret();

end Create;


method _myClass.assign( theValue:string ); @nodisplay;
begin assign;

    push( eax );

    // First, free the current value of the s field.

    strfree( this.s );

    // Now make a copy of the parameter value and point
    // the s field at this copy:

    mov( str.a_cpy( theValue ), this.s );

    pop( eax );

end assign;


method _myClass.put; @nodisplay; @noframe;
begin put;

    stdout.put( this.s );
    ret();

end put;
```



```
static
    vmt( _myClass );

var
    mc:myClass;

begin InitFinalDemo;

    stdout.put( "Initial value of mc.s = """ );
    mc.put();
    stdout.put( """" nl );

    mc.assign( "Hello World" );
    stdout.put( "After mc.assign( ""Hello World"" ), mc.s=""" );
    mc.put();
    stdout.put( """" nl );

end InitFinalDemo;
```

Program 13.23 Code for _initialize_/_finialize_ Laboratory Exercise

Exercise A: compile and run this program. Include the output of this program in your laboratory report. Explain the output you obtain (and, in particular, explain why you get the output that you do). Specifically, describe how this code automatically calls the constructor for the *mc* object.

Exercise B: the *_myClass* class does not have a destructor. Write a destructor for this class. Note that class objects have a string variable that is allocated on the heap. Therefore, one of the tasks for this destructor is to free the storage associated with that string. You should also print a short message in the destructor to let you know that it has been called. Modify the main program to call *mc.Destroy* (*Destroy* being the conventional name for a class destructor) at the very end of the program.

For extra credit: this program assumes that the *s* field always points at an object that has been allocated on the heap. In the HLA Standard Library memory allocation module there is a function that will tell you whether a pointer is pointing into the heap. Use this function to verify that *s* contains a valid pointer you can free before freeing the storage associated with this string point. To test this, try initializing the s field with an address that is not on the heap and then call the destructor to see how it responds.

Exercise C: modify the *myClass* macro so that the program automatically calls the destructor when the object loses scope. To do this, you will need to modify the value of the *_finalize_* string. Use the existing modification of the *_initialize_* string as a template (and, of course, look up how to do this earlier in this chapter). Include a sample run in your lab report and explain what is happening and how the destructor is activating.

Exercise D: using a macro to simulate automatic constructor and destructor calls is not a panacea. Try declaring the following and compiling your program and explain what happens during the compilation:

- Declare a "pointer to myClass" variable.
- Declare a new type name which simply renames the "myClass" type.

Bonus exercise: declare an array of *myClass* objects. Syntactically the compiler should accept this. Explain the problem with this declaration considering the purpose of this exercise. Include a sample program with your lab report that demonstrates the problem when declaring arrays of *myClass* objects.

13.3.27 Using RTTI in a Program

Run-time type information (RTTI) lets you determine the specific type of a generic object. This is quite useful when you might need to do some special processing for certain objects if they have a given type. In this exercise you will see a simple demonstration of RTTI in a generic program that demonstrates how you can use RTTI to achieve this goal.

```
// Using RTTI in a program.

program RTTIdemo;
#include( "stdlib.hhf" );

// Define some classes so we can demonstrate RTTI:

type
    baseClass: class

        procedure Create; returns( "esi" );
        procedure aProc;
        method aMethod;

    endclass;

    derivedClass: class inherits( baseClass )

        override procedure Create;
        override procedure aProc;
        override method aMethod;
        method dcNewMethod;

    endclass;

    anotherDerivedClass: class inherits( baseClass )

        override procedure Create;
        override procedure aProc;
        override method aMethod;
        method adcNewMethod;

    endclass;


/*************************************************/

// Methods for the baseClass class type:

procedure baseClass.Create; @nodisplay; @noframe;
begin Create;

    stdout.put( "called baseClass.Create", nl );

    push( eax );
    if( esi = 0 ) then

        mov( malloc( @size( baseClass ) ), esi );

    endif;
    mov( &baseClass._VMT_, this._pVMT_ );
```



```
    pop( eax );
    ret();

end Create;


procedure baseClass.aProc; @nodisplay; @noframe;
begin aProc;

    stdout.put( "Called baseClass.aProc" nl );
    ret();

end aProc;

method baseClass.aMethod; @nodisplay; @noframe;
begin aMethod;

    stdout.put( "Called baseClass.aMethod" nl );
    ret();

end aMethod;



/*************************************************/

// Methods for the derivedClass class type:

procedure derivedClass.Create; @nodisplay; @noframe;
begin Create;

    stdout.put( "called derivedClass.Create", nl );

    push( eax );
    if( esi = 0 ) then

        mov( malloc( @size( derivedClass ) ), esi );

    endif;
    mov( &derivedClass._VMT_, this._pVMT_ );
    pop( eax );
    ret();

end Create;


procedure derivedClass.aProc; @nodisplay; @noframe;
begin aProc;

    stdout.put( "Called derivedClass.aProc" nl );
    ret();

end aProc;

method derivedClass.aMethod; @nodisplay; @noframe;
begin aMethod;

    stdout.put( "Called derivedClass.aMethod" nl );
    ret();

end aMethod;
```



```
method derivedClass.dcNewMethod; @nodisplay; @noframe;
begin dcNewMethod;

    stdout.put( "Called derivedClass.dcNewMethod" nl );
    ret();

end dcNewMethod;



/************************************************/

// Methods for the anotherDerivedClass class type:

procedure anotherDerivedClass.Create; @nodisplay; @noframe;
begin Create;

    stdout.put( "called anotherDerivedClass.Create", nl );

    push( eax );
    if( esi = 0 ) then

        mov( malloc( @size( anotherDerivedClass ) ), esi );

    endif;
    mov( &anotherDerivedClass._VMT_, this._pVMT_ );
    pop( eax );
    ret();

end Create;


procedure anotherDerivedClass.aProc; @nodisplay; @noframe;
begin aProc;

    stdout.put( "Called anotherDerivedClass.aProc" nl );
    ret();

end aProc;

method anotherDerivedClass.aMethod; @nodisplay; @noframe;
begin aMethod;

    stdout.put( "Called anotherDerivedClass.aMethod" nl );
    ret();

end aMethod;



method anotherDerivedClass.adcNewMethod; @nodisplay; @noframe;
begin adcNewMethod;

    stdout.put( "Called anotherDerivedClass.adcNewMethod" nl );
    ret();

end adcNewMethod;
```



```
/************************************************/

static
    vmt( baseClass );
    vmt( derivedClass );
    vmt( anotherDerivedClass );

    bc: baseClass;
    dc: derivedClass;
    dc2: anotherDerivedClass;

    pbc: pointer to baseClass;


    // Randomly select one of the above class variables and
    // return its address in EAX:

    procedure randomSelect
    (
        var b:baseClass;
        var d:derivedClass;
        var a:anotherDerivedClass
    );
        @nodisplay; returns( "eax" );

    begin randomSelect;

        // Get a pseudo-random number between zero and two and
        // use this value to select one of the addresses passed
        // in as a parameter:

        rand.urange( 0, 2 );
        if( al == 0 ) then

            mov( b, eax );

        elseif( al = 1 ) then

            mov( d, eax );

        else

            mov( a, eax );

        endif;

    end randomSelect;



begin RTTIdemo;


    // Warning: This code only works on Pentium and
    // compatible chips that have an enabled RDTSC
    // instruction.  If your CPU doesn't support this,
    // you will have to replace this code with something
    // else. (Suggestion: read a value from the user
    // and call rand.uniform the specified number of times.)
```



```
rand.randomize();


// Okay, initialize our class objects:

bc.Create();
dc.Create();
dc2.Create();

// Demonstrate calling a common method for each of these
// objects:

bc.aMethod();
dc.aMethod();
dc2.aMethod();

// Randomly select one of the objects and use RTTI to
// exactly determine which one we want:

for( mov( 0, ecx ); ecx < 10; inc( ecx )) do

    // Do the random selection:

    mov( randomSelect( bc, dc, dc2), pbc );

    // Print a separator:

    stdout.put
    (
        nl
        "---------------------------------------"
        nl
    );

    // Call aProc just to verify that this is a baseClass variable:

    pbc.aProc();

    // Call aMethod to display the actual type:

    pbc.aMethod();

    // Okay, use RTTI to determine the actual type of this object
    // and call a method specific to that type (if appropriate)
    // to demonstrate RTTI.

    mov( pbc, eax );

    // If the object's VMT pointer field contains the address of
    // derivedClass' VMT, then this must be a derivedClass object.

    if( (type baseClass [eax])._pVMT_ = &derivedClass._VMT_ ) then

        (type derivedClass [eax]).dcNewMethod();

    // If the object's VMT pointer field contains the address of
    // anotherDerivedClass' VMT, then this must be an
    // anotherDerivedClass object.

    elseif((type baseClass [eax])._pVMT_ = &anotherDerivedClass._VMT_ ) then
```



```
            (type derivedClass [eax]).dcNewMethod();

        // If the object's VMT pointer field contains the address of
        // baseClass' VMT, then this must be a baseClass object.

        elseif( (type baseClass [eax])._pVMT_ = &baseClass._VMT_ ) then

            stdout.put
            (
                "This is the base class, there are no special methods"
                nl
            );

        // This case should never happen...

        else

            stdout.put( "Whoa! Something weird is going on..." nl );

        endif;

    endfor;

end RTTIdemo;
```

Program 13.24 Code for RTTI Laboratory Exercise

Exercise A: Run this program and include the output in your laboratory report. Explain the results that you're getting.

Exercise B: Run the program a second time. Is the output the same as the first run? If not, explain this in your laboratory report.

Optional: If your CPU does not support the RDTSC instruction, modify the *randomSelect* function to read a "random" value from the user in order to make the selection.

