

Volume Four: Intermediate Assembly Language

This volume completes the material traditionally taught in a 10 or 15 week course on assembly language programming (the difference between such courses is how many chapters they've skipped up to this point). This volume also completes this text's discussion of the essential material you need to know to start using assembly language effectively. Although there is still much for you to learn, after you complete this chapter any further study of assembly language tends to be more specialized. In any case, mastery of the material up to the end of this volume is an important milestone. Once you absorb and are able to apply this material, you can start calling yourself an "Assembly Language Programmer."

Chapter One: Advanced High Level Control Structures I

This chapter completes the discussion of HLA's high level control structures. It completely discusses TRY..ENDTRY and introduces several new high level control structures.

Chapter Two: Low Level Control Structures

This chapter discusses the "real" way to do control structures, using "pure" assembly language. This is a very important chapter; you cannot call yourself an assembly language programmer if you haven't mastered the low-level control structures.

Chapter Three: Intermediate Procedures

This chapter extends the information on procedures found in the previous volume. This chapter discusses some of the low-level implementation details of procedures and describes how to call procedures and pass parameters using "pure" assembly language.

Chapter Four: Advanced Arithmetic

This chapter discusses multiprecision and binary coded decimal arithmetic. It also describes how to input and output very large values (code included!).

Chapter Five: Bit Manipulation

This chapter discusses bit operations in assembly language. You'll learn how to deal with packed data, insert and extract bit strings, count bits in an operand, and do all other sorts of bit-related stuff.

Chapter Six: The String Instructions

This chapter discusses the 80x86 string instructions which are convenient for manipulating large blocks of memory.

Volume Four:

Intermediate Assembly Language

Chapter Seven: The HLA Compile-Time Language

This chapter begins the discussion of one of HLA's most powerful features - the HLA compile time language. In this chapter you'll learn about conditional compilation, compile-time loops, compile-time functions, generating tables, and lots of other features that make assembly language programming easier.

Chapter Eight: Macros

This chapter continues the discussion of the HLA compile-time language with a discussion of one of HLA's most powerful features – the HLA macro processor. In this chapter you'll learn how to extend the HLA language and do all those neat things that the HLA Standard Library provides.

Chapter Nine: Domain Specific Languages

This chapter describes how to design and implement your own programming language inside HLA.

Chapter Ten: Classes

This chapter describes classes and object-oriented programming in HLA.

Chapter Eleven: The MMX Instruction Set

This chapter describes the special MMX multimedia extensions on the Pentium and later chips.

Chapter Twelve: Mixed Language Programming

This chapter describes how to call HLA procedures and access HLA data from other languages.

Chapter Thirteen: Questions, Projects, and Laboratory Exercises

Test your knowledge and see how well you've learned the material in this chapter!

Advanced High Level Control Structures Chapter One

1.1 Chapter Overview

Volume One introduced some basic HLA control structures like the IF and WHILE statements (see “Some Basic HLA Control Structures” on page 21.). This section elaborates on some of those control structures (discussing features that were a little too advanced to present in Volume One) and it introduces the remaining high level language control structures that HLA provides.

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

1.2 TRY..ENDTRY

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

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

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

static
    u: uns16;

begin testBadInput;

    try

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

        exception( ex.ConversionError )

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

        exception( ex.ValueOutOfRange )

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

    endtry;

end testBadInput;
```

Program 1.1 TRY..ENDTRY Example

HLA calls the statements between the TRY clause and the first EXCEPTION clause the *protected* statements. If an exception occurs within the protected statements, then the program will scan through each of the exceptions and compare the value of the current exception against the value in the parentheses after each of the EXCEPTION clauses¹. This exception value is simply an *uns32* value. The value in the parentheses after each EXCEPTION clause, therefore, must be an unsigned 32-bit value. The HLA “excepts.hhf” header file predefines several exception constants; for example, *ex.ConversionError* is equal to five and *ex.ValueOutOfRange* is equal to three. Other than it would be an incredibly bad style violation, you could substitute “exception(5)” and “exception(3)” for the two EXCEPTION clauses above.

1.2.1 Nesting TRY..ENDTRY Statements

If the program scans through all the exception clauses in a TRY..ENDTRY statement and does not match the current exception value, then the program searches through the EXCEPTION clauses of a *dynamically nested* TRY..ENDTRY block in an attempt to find an appropriate exception handler. For example, consider the following code:

```

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

static
    u:      uns16;

begin testBadInput2;

    try

        try

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

            exception( ex.ConversionError )

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

        endtry;

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

        exception( ex.ValueOutOfRange )

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

        endtry;

    end testBadInput2;

```

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

Program 1.2 Nested TRY..ENDTRY Statements

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

After leaving a TRY..ENDTRY block, the HLA run-time system no longer considers that block active and will not search through its list of exceptions when the program raises an exception². This allows you to handle the same exception differently in different parts of the program.

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

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

In the previous example (Program 1.2) the second TRY..ENDTRY statement was statically nested inside the enclosing TRY..ENDTRY statement³. As mentioned without comment earlier, if the most recently activated TRY..ENDTRY statement does not handle a specific exception, the program will search through the EXCEPTION clauses of any dynamically nesting TRY..ENDTRY blocks. Dynamic nesting does not require the nested TRY..ENDTRY block to physically appear within the enclosing TRY..ENDTRY statement. Instead, control could transfer from inside the enclosing TRY..ENDTRY protected block to some other point in the program. Execution of a TRY..ENDTRY statement at that other point dynamically nests the two TRY statements. Although you will see lots of ways to dynamically nest code a little later in this chapter, there is one method you are familiar with that will let you dynamically nest these statements: the procedure call. The following program provides yet another example of the sample programs in this section, this example demonstrates dynamic nesting via a procedure call:

```

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

    procedure getUns;
    static
        u:          uns16;

    begin getUns;

        try

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

```

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

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

```

        exception( ex.ConversionError )

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

    endtry;

end getUns;

begin testBadInput3;

    try

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

        exception( ex.ValueOutOfRange )

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

    endtry;

end testBadInput3;

```

Program 1.3 Dynamic Nesting of TRY..ENDTRY Statements

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

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

1.2.2 The UNPROTECTED Clause in a TRY..ENDTRY Statement

Whenever a program executes the TRY clause, it preserves the current exception environment and sets up the system to transfer control to the EXCEPTION clauses within that TRY..ENDTRY statement should an exception occur. If the program successfully completes the execution of a TRY..ENDTRY protected block, the program restores the original exception environment and control transfers to the first statement

beyond the ENDTRY clause. This last step, restoring the execution environment, is very important. If the program skips this step, any future exceptions will transfer control to this TRY..ENDTRY statement even though the program has already left the TRY..ENDTRY block. The following program demonstrates this problem:

```

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

static
    input:  uns32;

begin testBadInput4;

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

    forever

        try

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

        exception( ex.ValueOutOfRange )

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

        exception( ex.ConversionError )

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

        endtry;

    endfor;

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

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

end testBadInput4;

```

Program 1.4 Improperly Exiting a TRY..ENDTRY Statement

This example attempts to create a robust input system by putting a loop around the TRY..ENDTRY statement and forcing the user to reenter the data if the *stdin.get* routine raises an exception (because of bad input data). While this is a good idea, there is a big problem with this implementation: the BREAK statement immediately exits the FOREVER..ENDFOR loop without first restoring the exception environment. Therefore, when the program executes the second *stdin.get* statement, at the bottom of the program, the HLA exception handling code still thinks that it's inside the TRY..ENDTRY block. If an exception occurs, HLA transfers control back into the TRY..ENDTRY statement looking for an appropriate exception handler.

Assuming the exception was *ex.ValueOutOfRangeException* or *ex.ConversionError*, Program 1.4 will print an appropriate error message *and then force the user to reenter the first value*. This isn't desirable.

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

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

```

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

static
    input: uns32;

begin testBadInput5;

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

    forever

        try

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

        unprotected

            break;

        exception( ex.ValueOutOfRangeException )

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

        exception( ex.ConversionError )

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

    endtry;

endfor;

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

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

```

```
end testBadInput5;
```

Program 1.5 The TRY..ENDTRY UNPROTECTED Section

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

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

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

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

1.2.3 The ANYEXCEPTION Clause in a TRY..ENDTRY Statement

In a typical situation, you will use a TRY..ENDTRY statement with a set of EXCEPTION clauses that will handle all possible exceptions that can occur in the protected section of the TRY..ENDTRY sequence. Often, it is important to ensure that a TRY..ENDTRY statement handles all possible exceptions to prevent the program from prematurely aborting due to an unhandled exception. If you have written all the code in the protected section, you will know the exceptions it can raise so you can handle all possible exceptions. However, if you are calling a library routine (especially a third-party library routine), making a Win32 API call, or otherwise executing code that you have no control over, it may not be possible for you to anticipate all possible exceptions this code could raise (especially when considering past, present, and future versions of this code). If that code raises an exception for which you do not have an EXCEPTION clause, this could cause your program to fail. Fortunately, HLA’s TRY..ENDTRY statement provides the ANYEXCEPTION clause that will automatically trap any exception the existing EXCEPTION clauses do not handle.

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

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

1.2.4 Raising User-Defined Exceptions

Although you typically use the TRY..ENDTRY statement to catch exceptions that Windows or the HLA Standard Library raises, it is also possible to create your own exceptions and process them via the

TRY..ENDTRY statement. You accomplish this by assigning an unused exception number to your exception and raising this exception with the HLA RAISE statement.

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

To create a user-defined exception, you would generally begin by defining a descriptive symbolic name for the exception⁴. Then within your code you can use the RAISE statement to raise that exception. The following program provides a short example of some code that uses a user-defined exception to trap empty strings:

```

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

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

const    EmptyString:dword := 1024;

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

procedure readAString;
begin readAString;

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

        strfree( eax );
        raise( EmptyString );

    endif;

end readAString;

begin userDefinedExceptions;

    try

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

```

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

```

exception( EmptyString )

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

endtry;

end userDefinedExceptions;

```

Program 1.6 User-Defined Exceptions and the RAISE Statement

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

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

1.2.5 Reraising Exceptions in a TRY..ENDTRY Statement

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

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

1.2.6 A List of the Predefined HLA Exceptions

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

1.2.7 How to Handle Exceptions in Your Programs

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

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

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

static
    u:      uns16;

begin repeatingBadCode;

    forever

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

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

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

            unprotected

                break;

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

            exception( ex.ConversionError )

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

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

            exception( ex.ValueOutOfRange )

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

        endtry;
```

```

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

    endfor;

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

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

end repeatingBadCode;

```

Program 1.7 Repeating Code via a Loop to Handle an Exception

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

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

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

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

1.2.8 Registers and the TRY..ENDTRY Statement

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

from the TRY..ENDTRY statement. However, this claim is not true if an exception occurs during the execution of the protected statements.

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

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

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

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

        stdin.geti32();
        unprotected break;

      exception( ex.ConversionError )

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

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

  endfor;
```

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

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

1.3 BEGIN..EXIT..EXITIF..END

HLA provides a structured GOTO via the EXIT and EXITIF statements. The EXIT and EXITIF statements let you exit a block of statements surrounded by a BEGIN..END pair. These statements behave much like the BREAK and BREAKIF statements (that let you exit from an enclosing loop) except, of course, they jump out of a BEGIN..END block rather than a loop. The EXIT and EXITIF statements are *structured gotos* because they do not let you jump to an arbitrary point in the code, they only let you exit from a block delimited by the BEGIN..END pair.

The EXIT and EXITIF statements take the following forms:

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

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

If you specify the name of a procedure as the identifier for an EXIT statement, the program will return from the procedure upon encountering the EXIT statement⁵. *Note that the EXIT statement does not automatically restore any registers you pushed on the stack upon entry into the procedure.* If you need to pop data off the stack, you must do this before executing the EXIT statement.

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

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

```
begin identifier;

    <statements>

end identifier;
```

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

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

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

static
    m:uns8;
```

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

```

d:uns8;
y:uns16;

readonly
DaysInMonth: uns8[13] :=
[
    0, // No month zero.
    31, // Jan
    28, // Feb is a special case, see the code.
    31, // Mar
    30, // Apr
    31, // May
    30, // Jun
    31, // Jul
    31, // Aug
    30, // Sep
    31, // Oct
    30, // Nov
    31 // Dec
];

begin beginEndDemo;

    forever

        try

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

            unprotected

                break;

            exception( ex.ValueOutOfRange )

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

            exception( ex.ConversionError )

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

        endtry;

    endfor;
    begin goodDate;

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

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

        if( bx in 2000..2099 ) then

            // Verify that the month is legal

            if( al in 1..12 ) then

                // Quick check to make sure the

```

```

// day is half-way reasonable.

if( dl <> 0 ) then

    // To verify that the day is legal,
    // we have to handle Feb specially
    // since this could be a leap year.

    if( al = 2 ) then

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

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

    endif;

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

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

endif;

endif;

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

end goodDate;

end beginEndDemo;

```

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

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

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

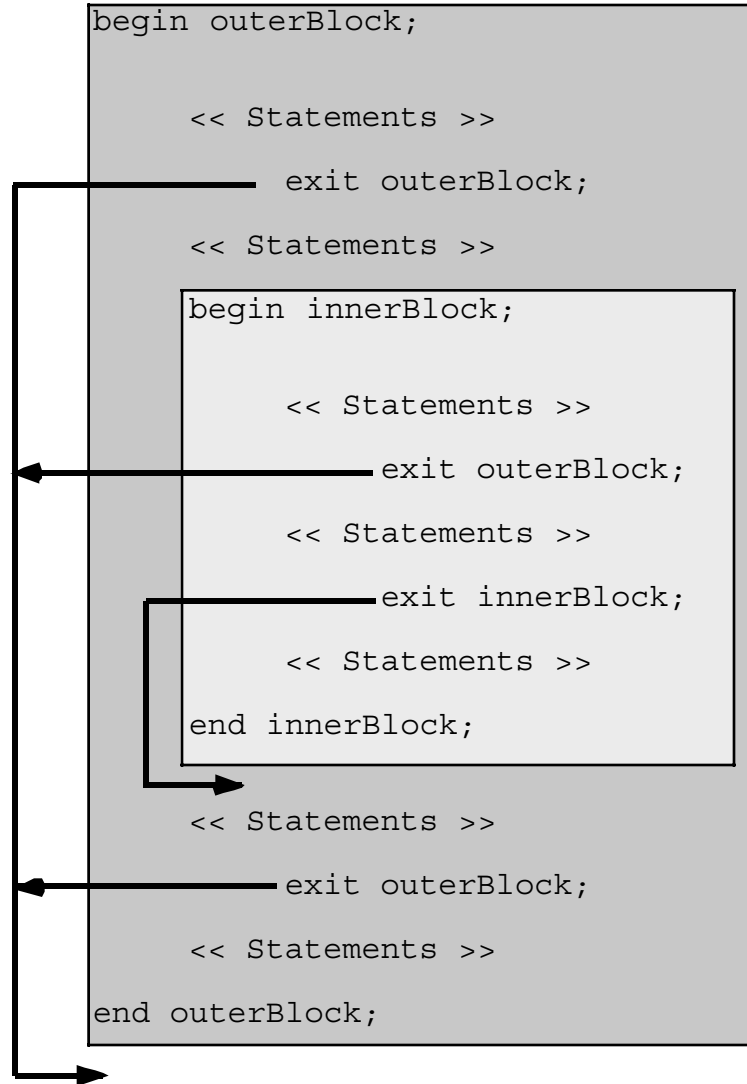


Figure 1.1 **Nesting BEGIN..END Blocks**

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

```

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

static
    i:int32;

```

```

begin brkNestedLoops;

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

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

        begin middleLoop;

            // DH ranges over the values to print.

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

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

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

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

                    exitif( dh >= dl ) middleLoop;

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

                    stdout.puti8size( dh, i, '.' );

                endfor;

            endfor;

        end middleLoop;
        stdout.newln();

    endfor;

end brkNestedLoops;

```

Program 1.9 Breaking Out of Nested Loops Using EXIT/EXITIF

1.4 CONTINUE..CONTINUEIF

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

The phrase “where the current iteration completes and the next iteration begins” has a different meaning for nearly every loop in HLA. For the WHILE..ENDWHILE loop, control transfers to the top of the loop at the start of the test for loop termination. For the FOREVER..ENDFOR loop, control transfers to the top of the loop (no test for loop termination). For the FOR..ENDFOR loop, control transfers to the bottom of the loop where the increment operation occurs (i.e., to execute the third component of the FOR loop). For the

REPEAT..UNTIL loop, control transfers to the bottom of the loop, just before the test for loop termination. The following diagrams show how the CONTINUE statement behaves.

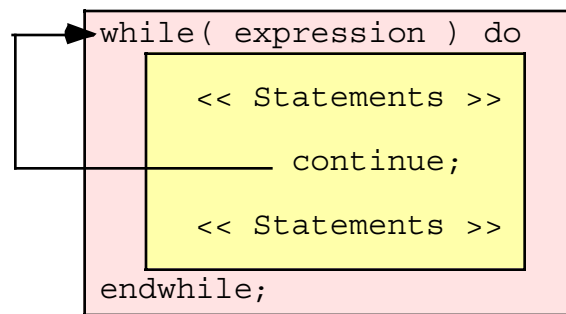


Figure 1.2 Behavior of CONTINUE in a WHILE Loop

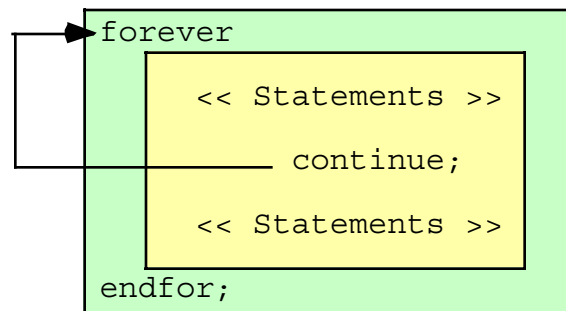


Figure 1.3 Behavior of CONTINUE in a FOREVER Loop

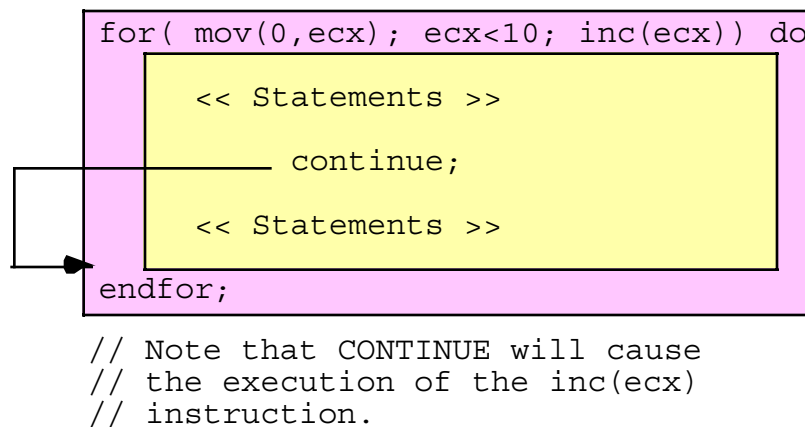


Figure 1.4 Behavior of CONTINUE in a FOR Loop

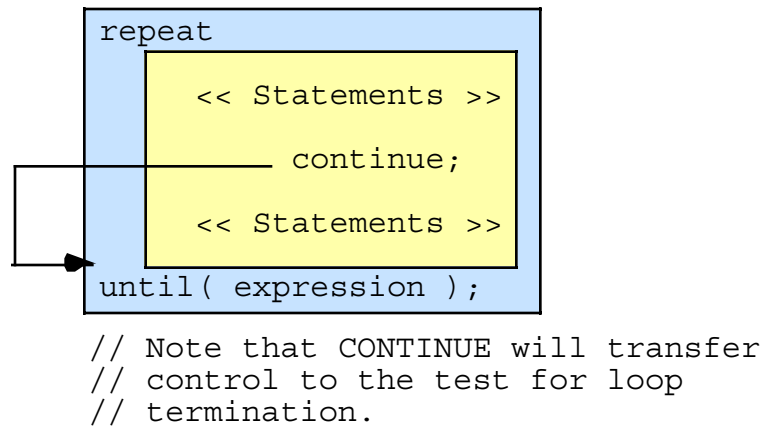


Figure 1.5 Behavior of CONTINUE in a REPEAT..UNTIL Loop

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

1.5 SWITCH..CASE..DEFAULT..ENDSWITCH

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

The HLA Standard Library SWITCH statement has the following syntax:

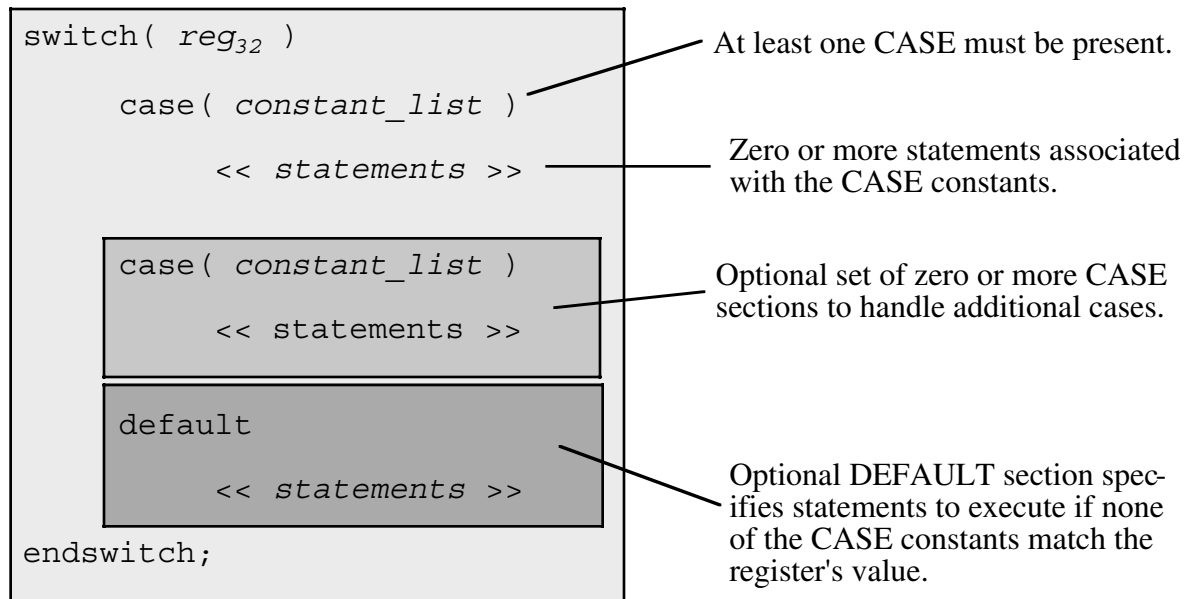


Figure 1.6 Syntax for the SWITCH..CASE..DEFAULT..ENDSWITCH Statement

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

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

```

case( 10 )
-or-
case( 5, 6, 8 )

```

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

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

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

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

```

switch( eax )

case( 1 )

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

```

```

case( 2, 3 )

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

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

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

default

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

endswitch;

```

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

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

1.6 Putting It All Together

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

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

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

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

Low-Level Control Structures

Chapter Two

2.1 Chapter Overview

This chapter discusses “pure” assembly language control statements. The last section of this chapter discusses *hybrid* control structures that combine the features of HLA’s high level control statements with the 80x86 control instructions.

2.2 Low Level Control Structures

Until now, most of the control structures you’ve seen and have used in your programs have been very similar to the control structures found in high level languages like Pascal, C++, and Ada. While these control structures make learning assembly language easy they are not true assembly language statements. Instead, the HLA compiler translates these control structures into a sequence of “pure” machine instructions that achieve the same result as the high level control structures. This text uses the high level control structures to avoid your having to learn too much all at once. Now, however, it’s time to put aside these high level language control structures and learn how to write your programs in *real* assembly language, using low-level control structures.

Since this text is attempting to teach you assembly language programming, many of the examples that appear in this text from this point forward will use these low-level control structures rather than the HLA high level control structures. This will better expose you to the use of these statements by removing the “crutch” you’ve been (unknowingly) using up to this point. One exception to this rule will be the TRY..ENDTRY statement. The TRY..ENDTRY statement is complex and difficult to implement in “pure” assembly language, so this text will continue to use the TRY..ENDTRY statement to guard against exceptions.

2.3 Statement Labels

HLA low level control structures make extensive use of *labels* within your code. A low level control structure usually transfers control from one point in your program to another point in your program. You typically specify the destination of such a transfer using a statement label. A statement label consists of a valid (unique) HLA identifier and a colon, e.g.,

```
label:
```

Of course, like procedure, variable, and constant identifiers, you should attempt to choose descriptive and meaningful names for your labels. The identifier “label” is hardly descriptive or meaningful.

Statement labels have one important attribute that differentiates them from most other identifiers in HLA: you don’t have to declare a label before you use it. This is important, because low-level control structures must often transfer control to a label at some point later in the code, therefore the label may not be defined at the point you reference it.

Labels are the target of jump (goto) instructions and you can take the address of a label. There is very little else you can directly do with a label (of course, there is very little else you would want to do with a label, so this is hardly a restriction. The following program demonstrates two ways to take the address of a label in your program and print out the address (using the LEA instruction and using the “&” address-of operator):

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

```

begin labelDemo;

    lbl1:

        lea( ebx, lbl1 );
        mov( &lbl2, eax );
        stdout.put( "&lbl1=$", ebx, " &lbl2=", eax, nl )

    lbl2:

end labelDemo;

```

Program 2.1 Displaying the Address of Statement Labels in a Program

HLA also allows you to initialize *dword* variables with the addresses of statement labels. However, there are some restrictions on labels that appear in the initialization portions of variable declarations. The most important restriction is that you must define the statement label at the same lex level as the variable declaration. That is, if you reference a statement label in the initialization section of a variable declaration appearing in the main program, the statement label must also be in the main program. Conversely, if you take the address of a statement label in a local variable declaration, that symbol must appear in the same procedure as the local variable. The following program demonstrates the use of statement labels in variable initialization:

```

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

static
    labels:dword[2] := [ &lbl1, &lbl2 ];

procedure hasLabels;
static
    stmtLbls: dword[2] := [ &label1, &label2 ];

begin hasLabels;

    label1:

        stdout.put
        (
            "stmtLbls[0]= $", stmtLbls[0], nl,
            "stmtLbls[1]= $", stmtLbls[4], nl
        );

    label2:

end hasLabels;

begin labelArrays;

    hasLabels();
    lbl1:

        stdout.put( "labels[0]= $", labels[0], " labels[1]=", labels[4], nl )

    lbl2:

```

```
end labelArrays;
```

Program 2.2 Initializing DWORD Variables with the Address of Statement Labels

2.4 Unconditional Transfer of Control (JMP)

The JMP (jump) instruction unconditionally transfers control to another point in the program. There are three forms of this instruction: a direct jump, and two indirect jumps. These instructions take one of the following three forms:

```
jmp label;
jmp( reg32 );
jmp( mem32 );
```

For the first (direct) jump above, you normally specify the target address using a statement label (see the previous section for a discussion of statement labels). The statement label is usually on the same line as an executable machine instruction or appears by itself on a line preceding an executable machine instruction. The direct jump instruction is the most commonly used of these three forms. It is completely equivalent to a GOTO statement in a high level language¹. Example:

```
<< statements >>
jmp laterInPgm;
.
.
.
laterInPgm:
<< statements >>
```

The second form of the JMP instruction above, “jmp(reg₃₂);”, is a *register indirect* jump instruction. This instruction transfers control to the instruction whose address appears in the specified 32-bit general purpose register. To use this form of the JMP instruction you must load the specified register with the address of some machine instruction prior to the execution of the JMP. You could use this instruction to implement a state machine (see “State Machines and Indirect Jumps” on page 758) by loading a register with the address of some label at various points throughout your program; then, arriving along different paths, a point in the program can determine what path it arrived upon by executing the indirect jump. The following short sample program demonstrates how you could use the JMP in this manner:

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

static
    i:int32;

begin regIndJmp;

    // Read an integer from the user and set EBX to
    // denote the success or failure of the input.

    try

        stdout.put( "Enter an integer value between 1 and 10: " );
        stdin.get( i );
```

1. Unlike high level languages, where your instructors usually forbid you to use GOTO statements, you will find that the use of the JMP instruction in assembly language is absolutely essential.

```

    mov( i, eax );
    if( eax in 1..10 ) then

        mov( &GoodInput, ebx );

    else

        mov( &valRange, ebx );

    endif;

exception( ex.ConversionError )

    mov( &convError, ebx );

exception( ex.ValueOutOfRange )

    mov( &valRange, ebx );

endtry;

// Okay, transfer control to the appropriate
// section of the program that deals with
// the input.

jmp( ebx );

valRange:
    stdout.put( "You entered a value outside the range 1..10" nl );
    jmp Done;

convError:
    stdout.put( "Your input contained illegal characters" nl );
    jmp Done;

GoodInput:
    stdout.put( "You entered the value ", i, nl );

Done:

end regIndJmp;

```

Program 2.3 Using Register Indirect JMP Instructions

The third form of the JMP instruction is a memory indirect JMP. This form of the JMP instruction fetches a *dword* value from the specified memory location and transfers control to the instruction at the address specified by the contents of the memory location. This is similar to the register indirect JMP except the address appears in a memory location rather than in a register. The following program demonstrates a rather trivial use of this form of the JMP instruction:

```

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

static
    LabelPtr:dword := &stmtLabel;

```

```

begin memIndJump;

    stdout.put( "Before the JMP instruction" nl );
    jmp( LabelPtr );

    stdout.put( "This should not execute" nl );

stmtLabel:

    stdout.put( "After the LabelPtr label in the program" nl );

end memIndJump;

```

Program 2.4 Using Memory Indirect JMP Instructions

Warning: unlike the HLA high level control structures, the low-level JMP instructions can get you into a lot of trouble. In particular, if you do not initialize a register with the address of a valid instruction and you jump indirect through that register, the results are undefined (though this will usually cause a Windows general protection fault). Similarly, if you do not initialize a *dword* variable with the address of a legal instruction, jumping indirect through that memory location will probably crash your program.

2.5 The Conditional Jump Instructions

Although the JMP instruction provides transfer of control, it does not allow you to make any serious decisions. The 80x86's conditional jump instructions handle this task. The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the IF..ENDIF statement.

The conditional jumps test one or more flags in the flags register to see if they match some particular pattern (just like the SET_{cc} instructions). If the flag settings match a particular pattern control transfers to the target location. If the match fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some conditional jump instructions simply test the setting of the sign, carry, overflow, and zero flags. For example, after the execution of a SHL instruction, you could test the carry flag to determine if the SHL shifted a one out of the H.O. bit of its operand. Likewise, you could test the zero flag after a TEST instruction to see if any specified bits were one. Most of the time, however, you will probably execute a conditional jump after a CMP instruction. The CMP instruction sets the flags so that you can test for less than, greater than, equality, etc.

The conditional JMP instructions take the following form:

```
Jcc label;
```

The “cc” in Jcc indicates that you must substitute some character sequence that specifies the type of condition to test. These are the same characters the SET_{cc} instruction uses. For example, “JS” stands for jump if the sign flag is set.” A typical JS instruction looks like this

```
js ValueIsNegative;
```

In this example, the JS instruction transfers control to the *ValueIsNegative* statement label if the sign flag is currently set; control falls through to the next instruction following the JS instruction if the sign flag is clear.

Unlike the unconditional JMP instruction, the conditional jump instructions do not provide an indirect form. The only form they allow is a branch to a statement label in your program. Conditional jump instructions have a restriction that the target label must be within 32,768 bytes of the jump instruction. However, since this generally corresponds to somewhere between 8,000 and 32,000 machine instructions, it is unlikely you will ever encounter this restriction.

Note: Intel's documentation defines various synonyms or instruction aliases for many conditional jump instructions. The following tables list all the aliases for a particular instruction. These tables also list out the opposite branches. You'll soon see the purpose of the opposite branches.

Table 1: Jcc Instructions That Test Flags

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1		JNS
JNS	Jump if no sign	Sign = 0		JS
JO	Jump if overflow	Ovrflw=1		JNO
JNO	Jump if no Ovrflw	Ovrflw=0		JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

Table 2: Jcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Aliases	Opposites
JA	Jump if above (>)	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=)	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (>=)	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not <)	Carry = 0	JNC, JAE	JB
JB	Jump if below (<)	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=)	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (<=)	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not >)	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

Table 3: Jcc Instructions for Signed Comparisons

Instruction	Description	Condition	Aliases	Opposite
JG	Jump if greater (>)	Sign = Ovrflw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign = Ovrflw	JNL	JGE
JNL	Jump if not less than (not <)	Sign = Ovrflw	JGE	JL
JL	Jump if less than (<)	Sign ≠ Ovrflw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign ≠ Ovrflw	JL	JGE
JLE	Jump if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	JLE	JG
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

One brief comment about the “opposites” column is in order. In many instances you will need to be able to generate the opposite of a specific branch instructions (lots of examples of this appear throughout the remainder of this chapter). With only two exceptions, a very simple rule completely describes how to generate an opposite branch:

- If the second letter of the *Jcc* instruction is *not* an “n”, insert an “n” after the “j”. E.g., JE becomes JNE and JL becomes JNL.
- If the second letter of the *Jcc* instruction *is* an “n”, then remove that “n” from the instruction. E.g., JNG becomes JG and JNE becomes JE.

The two exceptions to this rule are JPE (jump if parity is even) and JPO (jump if parity is odd). These exceptions cause few problems because (a) you’ll hardly ever need to test the parity flag, and (b) you can use the aliases JP and JNP synonyms for JPE and JPO. The “N/No N” rule applies to JP and JNP.

Though you *know* that JGE is the opposite of JL, get in the habit of using JNL rather than JGE as the opposite jump instruction for JL. It’s too easy in an important situation to start thinking “greater is the opposite of less” and substitute JG instead. You can avoid this confusion by always using the “N/No N” rule.

The 80x86 conditional jump instructions give you the ability to split program flow into one of two paths depending upon some logical condition. Suppose you want to increment the AX register if BX is equal to CX. You can accomplish this with the following code:

```

cmp( bx, cx );
jne SkipStmts;
inc( ax );
SkipStmts:

```

The trick is to use the *opposite* branch to skip over the instructions you want to execute if the condition is true. Always use the “opposite branch (N/no N)” rule given earlier to select the opposite branch.

You can also use the conditional jump instructions to synthesize loops. For example, the following code sequence reads a sequence of characters from the user and stores each character in successive elements of an array until the user presses the Enter key (carriage return):

```

        mov( 0, edi );
RdLnLoop:
        stdin.getc();           // Read a character into the AL register.
        mov( al, Input[ edi ] ); // Store away the character
        inc( edi );             // Move on to the next character
        cmp( al, stdio.cr );     // See if the user pressed Enter
        jne RdLnLoop;

```

For more information concerning the use of the conditional jumps to synthesize IF statements, loops, and other control structures, see “Implementing Common Control Structures in Assembly Language” on page 734.

Like the SETcc instructions, the conditional jump instructions come in two basic categories – those that test specific processor flags (e.g., JZ, JC, JNO) and those that test some condition (less than, greater than, etc.). When testing a condition, the conditional jump instructions almost always follow a CMP instruction. The CMP instruction sets the flags so you can use a JA, JAE, JB, JBE, JE, or JNE instruction to test for unsigned less than, less than or equal, equality, inequality, greater than, or greater than or equal. Simultaneously, the CMP instruction sets the flags so you can also do a signed comparison using the JL, JLE, JE, JNE, JG, and JGE instructions.

The conditional jump instructions only test flags, they do not affect any of the 80x86 flags.

2.6 “Medium-Level” Control Structures: JT and JF

HLA provides two special conditional jump instructions: JT (jump if true) and JF (jump if false). These instructions take the following syntax:

```

        jt( boolean_expression ) target_label;
        jf( boolean_expression ) target_label;

```

The *boolean_expression* is the standard HLA boolean expression allowed by IF..ENDIF and other HLA high level language statements. These instructions evaluate the boolean expression and jump to the specified label if the expression evaluates true (JT) or false (JF).

These are not real 80x86 instructions. HLA compiles them into a sequence of one or more 80x86 machine instructions that achieve the same result. In general, you should not use these two instructions in your main code; they offer few benefits over using an IF..ENDIF statement and they are no more readable than the pure assembly language sequences they compile into. HLA provides these “medium-level” instructions so that you may create your own high level control structures using macros (see the chapters on Macros, the HLA Run-Time Language, and Domain Specific Languages for more details).

2.7 Implementing Common Control Structures in Assembly Language

Since a primary goal of this chapter is to teach you how to use the low-level machine instructions to implement decisions, loops, and other control constructs, it would be wise to show you how to simulate these high level statements using “pure” assembly language. The following sections provide this information.

2.8 Introduction to Decisions

In its most basic form, a decision is some sort of branch within the code that switches between two possible execution paths based on some condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the IF..THEN..ENDIF statement in HLA:

```

        if( expression ) then
            << statements >>

```

```
endif;
```

Assembly language, as usual, offers much more flexibility when dealing with conditional statements. Consider the following C/C++ statement:

```
if( ( ( x < y ) && ( z > t ) ) || ( a != b ) )
    stmt1;
```

A “brute force” approach to converting this statement into assembly language might produce:

```
mov( x, eax );
cmp( eax, y );
setl( bl );          // Store X<Y in bl.
mov( z, eax );
cmp( eax, t );
setg( bh );          // Store Z > T in bh.
and( bh, bl );       // Put (X<Y) && (Z>T) into bl.
mov( a, eax );
cmp( eax, b );
setne( bh );         // Store A != B into bh.
or( bh, bl );        // Put (X<Y) && (Z>T) || (A!=B) into bl
je SkipStmt1;        // Branch if result is false (OR sets Z-Flag if false).
```

```
<Code for stmt1 goes here>
```

```
SkipStmt1:
```

As you can see, it takes a considerable number of conditional statements just to process the expression in the example above. This roughly corresponds to the (equivalent) C/C++ statements:

```
bl = x < y;
bh = z > t;
bl = bl && bh;
bh = a != b;
bl = bl || bh;
if( bl )
    stmt1;
```

Now compare this with the following “improved” code:

```
mov( a, eax );
cmp( eax, b );
jne DoStmt;
mov( x, eax );
cmp( eax, y );
jnl SkipStmt;
mov( z, eax );
cmp( eax, t );
jng SkipStmt;
DoStmt:
    << Place code for Stmt1 here >>
SkipStmt:
```

Two things should be apparent from the code sequences above: first, a single conditional statement in C/C++ (or some other HLL) may require several conditional jumps in assembly language; second, organization of complex expressions in a conditional sequence can affect the efficiency of the code. Therefore, care should be exercised when dealing with conditional sequences in assembly language.

Conditional statements may be broken down into three basic categories: IF statements, SWITCH/CASE statements, and indirect jumps. The following sections will describe these program structures, how to use them, and how to write them in assembly language.

2.8.1 IF..THEN..ELSE Sequences

The most commonly used conditional statement is the `if..then` or `IF..THEN..ELSE` statement. These two statements take the following form shown in Figure 2.1:

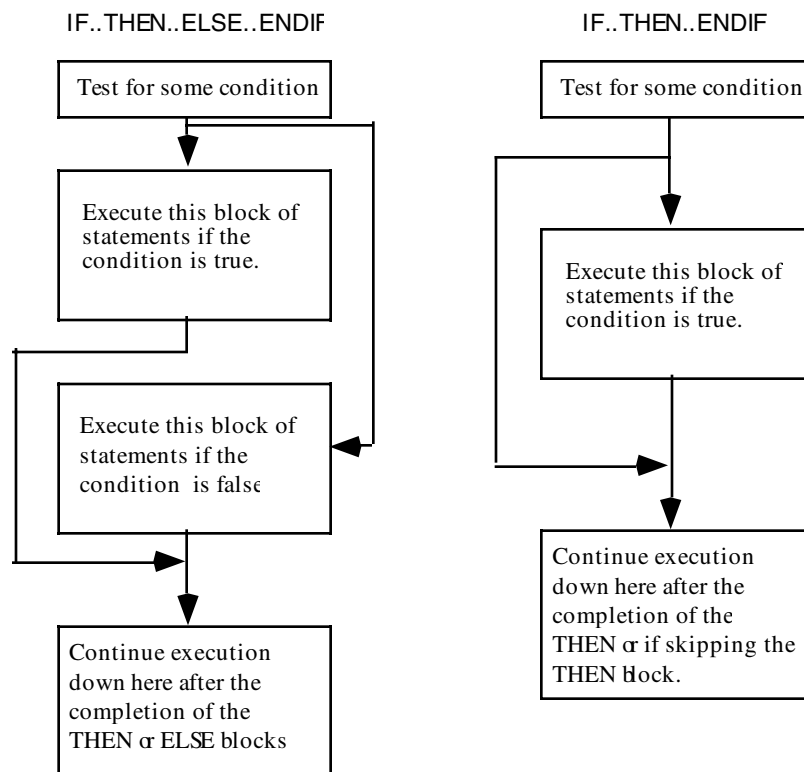


Figure 2.1 IF..THEN..ELSE..ENDIF and IF..ENDIF Statement Flow

The `IF..ENDIF` statement is just a special case of the `IF..ELSE..ENDIF` statement (with an empty `ELSE` block). Therefore, we'll only consider the more general `IF..ELSE..ENDIF` form. The basic implementation of an `IF..THEN..ELSE` statement in 80x86 assembly language looks something like this:

```

{Sequence of statements to test some condition}
    Jcc ElseCode
{Sequence of statements corresponding to the THEN block}

    jmp EndOfIF

ElseCode:
    {Sequence of statements corresponding to the ELSE block}

EndOfIF:
  
```

Note: `Jcc` represents some conditional jump instruction.

For example, to convert the C/C++ statement:

```

if( a == b )
    c = d;
else
    b = b + 1;
  
```

to assembly language, you could use the following 80x86 code:

```

        mov( a, eax );
        cmp( eax, b );
        jne ElsePart;
        mov( d, c );
        jmp EndOfIf;

ElseBlk:
        inc( b );

EndOfIf:

```

For simple expressions like “(a == b)” generating the proper code for an IF..ELSE..ENDIF statement is almost trivial. Should the expression become more complex, the associated assembly language code complexity increases as well. Consider the following C/C++ IF statement presented earlier:

```

if( ( ( x > y ) && ( z < t ) ) || ( a != b ) )
    c = d;

```

When processing complex IF statements such as this one, you’ll find the conversion task easier if you break this IF statement into a sequence of three different IF statements as follows:

```

if( a != b ) C = D;
else if( x > y )
    if( z < t )
        C = D;

```

This conversion comes from the following C/C++ equivalences:

```

if( expr1 && expr2 ) stmt;

```

is equivalent to

```

if( expr1 ) if( expr2 ) stmt;

```

and

```

if( expr1 || expr2 ) stmt;

```

is equivalent to

```

if( expr1 ) stmt;
else if( expr2 ) stmt;

```

In assembly language, the former IF statement becomes:

```

// if( ( ( x > y ) && ( z < t ) ) || ( a != b ) )
//     c = d;

        mov( a, eax );
        cmp( eax, b );
        jne DoIf;
        mov( x, eax );
        cmp( eax, y );
        jng EndOfIf;
        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;

DoIf:
        mov( d, c );

EndOfIf:

```

As you can probably tell, the code necessary to test a condition can easily become more complex than the statements appearing in the ELSE and THEN blocks. Although it seems somewhat paradoxical that it may take more effort to test a condition than to act upon the results of that condition, it happens all the time. Therefore, you should be prepared for this situation.

Probably the biggest problem with the implementation of complex conditional statements in assembly language is trying to figure out what you've done after you've written the code. Probably the biggest advantage high level languages offer over assembly language is that expressions are much easier to read and comprehend in a high level language. This is one of the primary reasons HLA supports high level language control structures. The high level language version is self-documenting whereas assembly language tends to hide the true nature of the code. Therefore, well-written comments are an essential ingredient to assembly language implementations of `if..then..else` statements. An elegant implementation of the example above is:

```
// IF ((X > Y) && (Z < T)) OR (A != B) C = D;
// Implemented as:
// IF (A != B) THEN GOTO DoIF;

    mov( a, eax );
    cmp( eax, b );
    jne DoIF;

// if NOT (X > Y) THEN GOTO EndOfIF;

    mov( x, eax );
    cmp( eax, y );
    jng EndOfIF;

// IF NOT (Z < T) THEN GOTO EndOfIF ;

    mov( z, eax );
    cmp( eax, t );
    jnl EndOfIf;

// THEN Block:

DoIf:
    mov( d, c );

// End of IF statement

EndOfIF:
```

Admittedly, this appears to be going overboard for such a simple example. The following would probably suffice:

```
// if( ( ( x > y ) && ( z < t ) ) || ( a != b ) ) c = d;
// Test the boolean expression:

    mov( a, eax );
    cmp( eax, b );
    jne DoIF;
    mov( x, eax );
    cmp( eax, y );
    jng EndOfIF;
    mov( z, eax );
    cmp( eax, t );
    jnl EndOfIf;

; THEN Block:

DoIf:
    mov( d, c );

; End of IF statement

EndOfIF:
```

However, as your IF statements become complex, the density (and quality) of your comments become more and more important.

2.8.2 Translating HLA IF Statements into Pure Assembly Language

Translating HLA IF statements into pure assembly language is very easy. The boolean expressions that the HLA IF supports were specifically chosen to expand into a few simple machine instructions. The following paragraphs discuss the conversion of each supported boolean expression into pure machine code.

if(*flag_specification*) then <<stmts>> endif;

This form is, perhaps, the easiest HLA IF statement to convert. To execute the code immediately following the THEN keyword if a particular flag is set (or clear), all you need do is skip over the code if the flag is clear (set). This requires only a single conditional jump instruction for implementation as the following examples demonstrate:

```
// if( @c ) then inc( eax ); endif;

    jnc SkipTheInc;

    inc( eax );

SkipTheInc:

// if( @ns ) then neg( eax ); endif;

    js SkipTheNeg;

    neg( eax );

SkipTheNeg:
```

if(*register*) then <<stmts>> endif;

This form of the IF statement uses the TEST instruction to check the specified register for zero. If the register contains zero (false), then the program jumps around the statements after the THEN clause with a JZ instruction. Converting this statement to assembly language requires a TEST instruction and a JZ instruction as the following examples demonstrate:

```
// if( eax ) then mov( false, eax ); endif;

    test( eax, eax );
    jz DontSetFalse;

    mov( false, eax );

DontSetFalse:

// if( al ) then mov( bl, cl ); endif;

    test( al, al );
    jz noMove;

    mov( bl, cl );

noMove:
```

if(!*register*) then <<stmts>> endif;

This form of the IF statement uses the TEST instruction to check the specified register to see if it is zero. If the register is not zero (true), then the program jumps around the statements after the THEN clause with a

JNZ instruction. Converting this statement to assembly language requires a TEST instruction and a JNZ instruction in a manner identical to the previous examples.

if(*boolean_variable*) then <<stmts>> endif;

This form of the IF statement compares the boolean variable against zero (false) and branches around the statements if the variable does contain false. HLA implements this statement by using the CMP instruction to compare the boolean variable to zero and then it uses a JZ (JE) instruction to jump around the statements if the variable is false. The following example demonstrates the conversion:

```
// if( bool ) then mov( 0, al ); endif;
```

```
    cmp( bool, false );
    je SkipZeroAL;
```

```
    mov( 0, al );
```

```
SkipZeroAL:
```

if(!*boolean_variable*) then <<stmts>> endif;

This form of the IF statement compares the boolean variable against zero (false) and branches around the statements if the variable contains true (i.e., the opposite condition of the previous example). HLA implements this statement by using the CMP instruction to compare the boolean variable to zero and then it uses a JNZ (JNE) instruction to jump around the statements if the variable contains true. The following example demonstrates the conversion:

```
// if( !bool ) then mov( 0, al ); endif;
```

```
    cmp( bool, false );
    jne SkipZeroAL;
```

```
    mov( 0, al );
```

```
SkipZeroAL:
```

if(*mem_reg relop mem_reg_const*) then <<stmts>> endif;

HLA translates this form of the IF statement into a CMP instruction and a conditional jump that skips over the statements on the opposite condition specified by the *relop* operator. The following table lists the correspondence between operators and condition jump instructions:

Table 4: IF Statement Conditional Jump Instructions

Relop	Conditional jump instruction if both operands are unsigned	Conditional jump instruction if either operand is signed
= or ==	JNE	JNE
<> or !=	JE	JE
<	JNB	JNL
<=	JNBE	JNLE
>	JNA	JNG
>=	JNAE	JNGE

Here are a few examples of IF statements translated into pure assembly language that use expressions involving relational operators:

```
// if( al == ch ) then inc( cl ); endif;

    cmp( al, ch );
    jne SkipIncCL;

    inc( cl );

SkipIncCL:

// if( ch >= 'a' ) then and( $5f, ch ); endif;

    cmp( ch, 'a' );
    jnae NotLowerCase

    and( $5f, ch );

NotLowerCase:

// if( (type int32 eax) < -5 ) then mov( -5, eax ); endif;

    cmp( eax, -5 );
    jnl DontClipEAX;

    mov( -5, eax );

DontClipEAX:

// if( si <> di ) then inc( si ); endif;

    cmp( si, di );
    je DontIncSI;

    inc( si );

DontIncSI:
```

if(*reg/mem* in *LowConst..HiConst*) then <<stmts>> endif;

HLA translates this IF statement into a pair of CMP instructions and a pair of conditional jump instructions. It compares the register or memory location against the lower valued constant and jumps if less than (below) past the statements after the THEN clause. If the register or memory location's value is greater than or equal to *LowConst*, the code falls through to the second CMP/conditional jump pair that compares the register or memory location against the higher constant. If the value is greater than (above) this constant, a conditional jump instruction skips the statements in the THEN clause. Example:

```
// if( eax in 1000..125_000 ) then sub( 1000, eax ); endif;

    cmp( eax, 1000 );
    jb DontSub1000;
    cmp( eax, 125_000 );
    ja DontSub1000;

    sub( 1000, eax );

DontSub1000:

// if( i32 in -5..5 ) then add( 5, i32 ); endif;
```

```

cmp( i32, -5 );
jl NoAdd5;
cmp( i32, 5 );
jg NoAdd5;

    add(5, i32 );

NoAdd5:

```

if(*reg/mem* not in *LowConst..HiConst*) then <<stmts>> endif;

This form of the HLA IF statement tests a register or memory location to see if its value is outside a specified range. The implementation is very similar to the code above except you branch to the THEN clause if the value is less than the *LowConst* value or greater than the *HiConst* value and you branch over the code in the THEN clause if the value is within the range specified by the two constants. The following examples demonstrate how to do this conversion:

```

// if( eax not in 1000..125_000 ) then add( 1000, eax ); endif;

    cmp( eax, 1000 );
    jb Add1000;
    cmp( eax, 125_000 );
    jbe SkipAdd1000;

    Add1000:
        add( 1000, eax );

    SkipAdd1000:

// if( i32 not in -5..5 ) then mov( 0, i32 ); endif;

    cmp( i32, -5 );
    jl Zeroi32;
    cmp( i32, 5 );
    jle SkipZero;

    Zeroi32:
        mov( 0, i32 );

    SkipZero:

```

if(*reg₈* in *CSetVar/CSetConst*) then <<stmts>> endif;

This statement checks to see if the character in the specified eight-bit register is a member of the specified character set. HLA emits code that is similar to the following for instructions of this form:

```

movzx( reg8, eax );
bt( eax, CsetVar/CsetConst );
jnc SkipPastStmts;

    << stmts >>

    SkipPastStmts:

```

This example modifies the EAX register (the code HLA generates does not, because it pushes and pops the register it uses). You can easily swap another register for EAX if you've got a value in EAX you need to preserve. In the worst case, if no registers are available, you can push EAX, execute the MOVZX and BT instructions, and then pop EAX's value from the stack. The following are some actual examples:

```

// if( al in { 'a'..'z' } ) then or( $20, al ); endif;

movzx( al, eax );
bt( eax, { 'a'..'z' } ); // See if we've got a lower case char.
jnc DontConvertCase;

or( $20, al ); // Convert to uppercase.

DontConvertCase:

// if( ch in { '0'..'9' } ) then and( $f, ch ); endif;

push( eax );
movzx( ch, eax );
bt( eax, { 'a'..'z' } ); // See if we've got a lower case char.
pop( eax );
jnc DontConvertNum;

and( $f, ch ); // Convert to binary form.

DontConvertNum:

```

2.8.3 Implementing Complex IF Statements Using Complete Boolean Evaluation

Although the HLA IF statements are very convenient, their inability to handle complex boolean expressions (particularly those involving conjunction [and] and disjunction [or]) creates many problems. Although HLA high level language IF statement does not support conjunction and disjunction, it is very easy to solve this problem using the HLA low level control statements.

Using complete boolean evaluation to evaluate a boolean expression for an IF statement is almost identical to converting arithmetic expressions into assembly language. Indeed, the previous volume covers this conversion process (see “Logical (Boolean) Expressions” on page 584). About the only thing worth noting about that process is that you do not need to store the ultimate boolean result in some variable; once the evaluation of the expression is complete you check to see if you have a false (zero) or true (one, or non-zero) result to determine whether to branch around the THEN portion of the IF statement. As you can see in the examples in the preceding sections, you can often use the fact that the last boolean instruction (AND/OR) sets the zero flag if the result is false and clears the zero flag if the result is true. This lets you avoid explicitly testing the result. Consider the following IF statement and its conversion to assembly language using complete boolean evaluation:

```

if( ( ( x < y ) && ( z > t ) ) || ( a != b ) )
    Stmt1;

mov( x, eax );
cmp( eax, y );
setl( bl ); // Store x<y in bl.
mov( z, eax );
cmp( eax, t );
setg( bh ); // Store z > t in bh.
and( bh, bl ); // Put (x<y) && (z>t) into bl.
mov( a, eax );
cmp( eax, b );
setne( bh ); // Store a != b into bh.
or( bh, bl ); // Put (x<y) && (z>t) || (a != b) into bl
je SkipStmt1; // Branch if result is false (OR sets Z-Flag if false).

<< Code for Stmt1 goes here >>

```

SkipStmt1:

This code computes a boolean value in the BL register and then, at the end of the computation, tests this resulting value to see if it contains true or false. If the result is false, this sequence skips over the code associated with *Stmt1*. The important thing to note in this example is that the program will execute each and every instruction that computes this boolean result (up to the JE instruction).

For more details on complete boolean evaluation, see “Logical (Boolean) Expressions” on page 584.

2.8.4 Short Circuit Boolean Evaluation

If you are willing to spend a little more effort studying a complex boolean expression, you can usually convert it to a much shorter and faster sequence of assembly language instructions using *short-circuit boolean evaluation*. Short-circuit boolean evaluation attempts to determine whether an expression is true or false by executing only a portion of the instructions that compute the complete expression. By executing only a portion of the instructions, the evaluation is often much faster.

To understand how short-circuit boolean evaluation works, consider the C/C++ expression “A && B”. Once we determine that A is false, there is no need to evaluate B since there is no way the expression can be true. If A and B represent sub-expressions rather than simple variables, you can begin to see the savings that are possible with short-circuit boolean evaluation. As a concrete example, consider the sub-expression “((x<y) && (z>t))” from the previous section. Once you determine that x is not less than y, there is no need to check to see if z is greater than t since the expression will be false regardless of z and t’s values. The following code fragment shows how you can implement short-circuit boolean evaluation for this expression:

```
// if( (x<y) && (z>t) ) then ...

    mov( x, eax );
    cmp( eax, y );
    jnl TestFails;
    mov( z, eax );
    cmp( eax, t );
    jng TestFails;

    << Code for THEN clause of IF statement >>

TestFails:
```

Notice how the code skips any further testing once it determines that x is not less than y. Of course, if x is less than y, then the program has to test z to see if it is greater than t; if not, the program skips over the THEN clause. Only if the program satisfies both conditions does the code fall through to the THEN clause.

For the logical OR operation the technique is similar. If the first sub-expression evaluates to true, then there is no need to test the second operand. Whatever the second operand’s value is at that point, the full expression still evaluates to true. The following example demonstrates the use of short-circuit evaluation with disjunction (OR):

```
// if( ch < 'A' || ch > 'Z' ) then stdout.put( "Not an upper case char" ); endif;

    cmp( ch, 'A' );
    jnb ItsNotUC
    cmp( ch, 'Z' );
    jna ItWasUC;

    ItsNotUC:
        stdout.put( "Not an upper case char" );

    ItWasUC:
```

Since the conjunction and disjunction operators are commutative, you can evaluate the left or right operand first if it is more convenient to do so. As one last example in this section, consider the full boolean expression from the previous section:

```
// if( ( ( x < y ) && ( z > t ) ) || ( a != b ) )    Stmt1;

    mov( a, eax );
    cmp( eax, b );
    jne DoStmt1;
    mov( x, eax );
    cmp( eax, y );
    jnl SkipStmt1;
    mov( z, eax );
    cmp( eax, t );
    jng SkipStmt1;

    DoStmt1:
    << Code for Stmt1 goes here >>

    SkipStmt1:
```

Notice how the code in this example chose to evaluate “a != b” first and the remaining sub-expression last. This is a common technique assembly language programmers use to write better code.

2.8.5 Short Circuit vs. Complete Boolean Evaluation

One fact about complete boolean evaluation is that every statement in the sequence will execute when evaluating the expression. Short-circuit boolean evaluation may not require the execution of every statement associated with the boolean expression. As you’ve seen in the previous two sections above, code based on short-circuit evaluation is probably faster². So it would seem that short-circuit evaluation is the technique of choice when converting complex boolean expressions to assembly language.

Sometimes, unfortunately, short-circuit boolean evaluation may not produce the correct result. In the presence of *side-effects* in an expression, short-circuit boolean evaluation will produce a different result than complete boolean evaluation. Consider the following C/C++ example:

```
if( ( x == y ) && ( ++z != 0 ) ) stmt;
```

Using complete boolean evaluation, you might generate the following code:

```
    mov( x, eax );          // See if x == y
    cmp( eax, y );
    sete( bl );
    inc( z );                // ++z
    cmp( z, 0 );             // See if incremented z is zero.
    setne( bh );
    and( bh, bl );           // Test x == y && ++z != 0
    jz SkipStmt;

    << code for stmt goes here >>

SkipStmt:
```

Using short-circuit boolean evaluation, you might generate the following code:

```
    mov( x, eax );          // See if x == y
    cmp( eax, y );
    jne SkipStmt;
```

2. Note that this does not always mean that the program will run faster. Jumps (conditional or otherwise) are often very slow executing instructions. Sometimes it’s faster to execute several instructions in a row rather than execute a few instructions that include a conditional jump.

```

inc( z );           // ++z
cmp( z, 0 );        // See if incremented z is zero.
je SkipStmt;

<< code for stmt goes here >>

```

SkipStmt:

Notice a very subtle, but important difference between these two conversions: if it turns out that x is equal to y , then the first version above *still increments* z and compares it to zero before it executes the code associated with *stmt*; the short-circuit version, on the other hand skips over the code that increments z if it turns out that x is equal to y . Therefore, the behavior of these two code fragments is different with respect to what happens to z if x is equal to y . Neither implementation is particularly wrong, depending on the circumstances you may or may not want the code to increment z if x is equal to y . However, it is important that you realize that these two conversion methods produce different results so you can choose an appropriate implementation if the effect of this code on z matters to your program.

Many programs take advantage of short-circuit boolean evaluation and rely upon the fact that the program may not evaluate certain components of the expression. The following C/C++ code fragment demonstrates what is probably the most common example that requires short-circuit boolean evaluation:

```

if( Ptr != NULL && *Ptr == 'a' ) stmt;

```

If it turns out that *Ptr* is NULL in this IF statement, then the expression is false and there is no need to evaluate the remainder of the expression (and, therefore, code that uses short-circuit boolean evaluation will not evaluate the remainder of this expression). This statement relies upon the semantics of short-circuit boolean evaluation for correct operation. Were C/C++ to use complete boolean evaluation, and the variable *Ptr* contained NULL, then the second half of the expression would attempt to dereference a NULL pointer (which tends to crash most programs). Consider the translation of this statement using complete and short-circuit boolean evaluation:

// Complete boolean evaluation:

```

mov( Ptr, eax );
test( eax, eax );    // Check to see if EAX is zero (NULL is zero).
setne( bl );
mov( [eax], al );    // Get *Ptr into AL.
cmp( al, 'a' );
sete( bh );
and( bh, bl );
jz SkipStmt;

<< code for stmt goes here >>

```

SkipStmt:

Notice in this example that if *Ptr* contains NULL (zero), then this program will attempt to access the data at location zero in memory via the “mov([eax], al);” instruction. Under Win32 this will cause a memory access fault (general protection fault). Now consider the short-circuit boolean conversion:

// Short-circuit boolean evaluation

```

mov( Ptr, eax );    // See if Ptr contains NULL (zero) and
test( eax, eax );    // immediately skip past Stmt if this
jz SkipStmt;        // is the case

mov( [eax], al );    // If we get to this point, Ptr contains
cmp( al, 'a' );      // a non-NULL value, so see if it points
jne SkipStmt;        // at the character 'a'.

<< code for stmt goes here >>

```

SkipStmt:

As you can see in this example, the problem with dereferencing the NULL pointer doesn't exist. If *Ptr* contains NULL, this code skips over the statements that attempt to access the memory address *Ptr* contains.

2.8.6 Efficient Implementation of IF Statements in Assembly Language

Encoding IF statements efficiently in assembly language takes a bit more thought than simply choosing short-circuit evaluation over complete boolean evaluation. To write code that executes as quickly as possible in assembly language you must carefully analyze the situation and generate the code appropriately. The following paragraphs provide some suggestions you can apply to your programs to improve their performance.

Know your data!

A mistake programmers often make is the assumption that data is random. In reality, data is rarely random and if you know the types of values that your program commonly uses, you can use this knowledge to write better code. To see how, consider the following C/C++ statement:

```
if(( a == b ) && ( c < d )) ++i;
```

Since C/C++ uses short-circuit evaluation, this code will test to see if *a* is equal to *b*. If so, then it will test to see if *c* is less than *d*. If you expect *a* to be equal to *b* most of the time but don't expect *c* to be less than *d* most of the time, this statement will execute slower than it should. Consider the following HLA implementation of this code:

```
mov( a, eax );
cmp( eax, b );
jne DontIncI;

mov( c, eax );
cmp( eax, d );
jnl DontIncI;

inc( i );

DontIncI:
```

As you can see in this code, if *a* is equal to *b* most of the time and *c* is not less than *d* most of the time, you will have to execute the first three instructions nearly every time in order to determine that the expression is false. Now consider the following implementation of the above C/C++ statement that takes advantage of this knowledge and the fact that the “&&” operator is commutative:

```
mov( c, eax );
cmp( eax, d );
jnl DontIncI;

mov( a, eax );
cmp( eax, b );
jne DontIncI;

inc( i );

DontIncI:
```

In this example the code first checks to see if *c* is less than *d*. If most of the time *c* is less than *d*, then this code determines that it has to skip to the label *DontIncI* after executing only three instructions in the typical case (compared with six instructions in the previous example). This fact is much more obvious in assembly language than in a high level language; this is one of the main reasons that assembly programs are often faster than their high level language counterparts: optimizations are more obvious in assembly lan-

guage than in a high level language. Of course, the key here is to understand the behavior of your data so you can make intelligent decisions such as the one above.

Rearranging Expressions

Even if your data is random (or you can't determine how the input values will affect your decisions), there may still be some benefit to rearranging the terms in your expressions. Some calculations take far longer to compute than others. For example, the DIV instruction is much slower than a simple CMP instruction. Therefore, if you have a statement like the following you may want to rearrange the expression so that the CMP comes first:

```
if( (x % 10 == 0) && (x != y) ) ++x;
```

Converted to assembly code, this IF statement becomes:

```
mov( x, eax );           // Compute X % 10
cdq();                   // Must sign extend EAX -> EDX:EAX
imod( 10, edx:eax );     // Remember, remainder goes into EDX
test( edx, edx );        // See if EDX is zero.
jnz SkipIF

mov( x, eax );
cmp( eax, y );
je SkipIF

inc( x );

SkipIF:
```

The IMOD instruction is very expensive (often 50-100 times slower than most of the other instructions in this example). Unless it is 50-100 times more likely than the remainder is zero rather than x is equal to y, it would be better to do the comparison first and the remainder calculation afterwards:

```
mov( x, eax );
cmp( eax, y );
je SkipIF

mov( x, eax );           // Compute X % 10
cdq();                   // Must sign extend EAX -> EDX:EAX
imod( 10, edx:eax );     // Remember, remainder goes into EDX
test( edx, edx );        // See if EDX is zero.
jnz SkipIF

inc( x );

SkipIF:
```

Destructuring Your Code

Although there is a lot of good things to be said about structured programming techniques, there are some drawbacks to writing structured code. Specifically, structured code is sometimes less efficient than unstructured code. Most of the time this is tolerable because unstructured code is difficult to read and maintain; it is often acceptable to sacrifice some performance in exchange for maintainable code. In certain instances, however, you may need all the performance you can get. In those rare instances you might choose to compromise the readability of your code in order to gain some additional performance.

One classic way to do this is to use code movement to move code your program rarely uses out of the way of code that executes most of the time. For example, consider the following pseudo C/C++ statement:

```
if( See_If_an_Error_Has_Occurred )
{
```

```

    << Statements to execute if no error >>
}
else
{
    << Error handling statements >>
}

```

In normal code, one does not expect errors to be frequent. Therefore, you would normally expect the THEN section of the above IF to execute far more often than the ELSE clause. The code above could translate into the following assembly code:

```

cmp( See_If_an_Error_Has_Occurred, true );
je HandleTheError

    << Statements to execute if no error >>
    jmp EndOfIf;

HandleTheError:
    << Error handling statements >>

EndOfIf:

```

Notice that if the expression is false this code falls through to the normal statements and then jumps over the error handling statements. Instructions that transfer control from one point in your program to another (e.g., JMP instructions) tend to be slow. It is much faster to execute a sequential set of instructions rather than jump all over the place in your program. Unfortunately, the code above doesn't allow this. One way to rectify this problem is to move the ELSE clause of the code somewhere else in your program. That is, you could rewrite the code as follows:

```

cmp( See_If_an_Error_Has_Occurred, true );
je HandleTheError

    << Statements to execute if no error >>

EndOfIf:

```

At some other point in your program (typically after a JMP instruction) you would insert the following code:

```

HandleTheError:
    << Error handling statements >>
    jmp EndOfIf;

```

Note that the program isn't any shorter. The JMP you removed from the original sequence winds up at the end of the ELSE clause. However, since the ELSE clause rarely executes, moving the JMP instruction from the THEN clause (which executes frequently) to the ELSE clause is a big performance win because the THEN clause executes using only straight-line code. This technique is surprisingly effective in many time-critical code segments.

There is a difference between writing *destructured* code and writing *unstructured* code. Unstructured code is written in an unstructured way to begin with. It is generally hard to read, difficult to maintain, and it often contains defects. Destructured code, on the other hand, starts out as structured code and you make a conscious decision to eliminate the structure in order to gain a small performance boost. Generally, you've already tested the code in its structured form before destructuring it. Therefore, destructured code is often easier to work with than unstructured code.

Calculation Rather than Branching

On many processors in the 80x86 family, branches are very expensive compared to many other instructions (perhaps not as bad as IMOD or IDIV, but typically an order of magnitude worse than instructions like ADD and SUB). For this reason it is sometimes better to execute more instructions in a sequence rather than

fewer instructions that involve branching. For example, consider the simple assignment “EAX = abs(EAX);” Unfortunately, there is no 80x86 instruction that computes the absolute value of an integer value. The obvious way to handle this is with an instruction sequence like the following:

```
test( eax, eax );
jns ItsPositive;

neg( eax );

ItsPositive:
```

However, as you can plainly see in this example, it uses a conditional jump to skip over the NEG instruction (that creates a positive value in EAX if EAX was negative). Now consider the following sequence that will also do the job:

```
// Set EDX to $FFFF_FFFF if EAX is negative, $0000_0000 if EAX is
// zero or positive:

cdq();

// If EAX was negative, the following code inverts all the bits in EAX,
// otherwise it has no effect on EAX.

xor( edx, edx);

// If EAX was negative, the following code adds one to EAX, otherwise
// it doesn't modify EAX's value.

and( 1, edx );      // EDX = 0 or 1 (1 if EAX was negative).
add( edx, eax );
```

This code will invert all the bits in EAX and then add one to EAX if EAX was negative prior to the sequence (i.e., it takes the two's complement [negates] the value in EAX). If EAX was zero or positive, then this code does not change the value in EAX.

Note that this sequence takes four instructions rather than the three the previous example requires. However, since there are no transfer of control instructions in this sequence, it may execute faster on many CPUs in the 80x86 family.

2.8.7 SWITCH/CASE Statements

The HLA SWITCH statement takes the following form :

```
switch( reg32 )
  case( const1 )
    <<stmts1>>

  case( const2 )
    <<stmts2>>
  .
  .
  .
  case( constn )
    <<stmtsn>>

  default      // Note that the default section is optional
    <<stmtsdefault>>

endswitch;
```

When this statement executes, it checks the value of register against the constants $\text{const}_1 \dots \text{const}_n$. If a match is found then the corresponding statements execute. HLA places a few restrictions on the SWITCH statement. First, the HLA SWITCH statement only allows a 32-bit register as the SWITCH expression. Second, all the constants appearing as CASE clauses must be unique. The reason for these restrictions will become clear in a moment.

Most introductory programming texts introduce the SWITCH/CASE statement by explaining it as a sequence of IF..THEN..ELSEIF statements. They might claim that the following two pieces of HLA code are equivalent:

```
switch( eax )
    case(0) stdout.put("I=0");
    case(1) stdout.put("I=1");
    case(2) stdout.put("I=2");
endswitch;

if( eax = 0 ) then stdout.put("I=0")
elseif( eax = 1 ) then stdout.put("I=1")
elseif( eax = 2 ) then stdout.put("I=2");
```

While semantically these two code segments may be the same, their implementation is usually different. Whereas the IF..THEN..ELSEIF chain does a comparison for each conditional statement in the sequence, the SWITCH statement normally uses an indirect jump to transfer control to any one of several statements with a single computation. Consider the two examples presented above, they could be written in assembly language with the following code:

```
// IF..THEN..ELSE form:

    mov( i, eax );
    test( eax, eax );    // Check for zero.
    jnz Not0;
        stdout.put( "I=0" );
        jmp EndCase;

Not0:
    cmp( eax, 1 );
    jne Not1;
        stdout.put( "I=1" );
        jmp EndCase;

Not1:
    cmp( eax, 2 );
    jne EndCase;
        stdout.put( "I=2" );
EndCase:

// Indirect Jump Version

readonly
    JumpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];
    .
    .
    .
    mov( i, eax );
    jmp( JumpTbl[ eax*4 ] );

    Stmt0:
        stdout.put( "I=0" );
        jmp EndCase;

    Stmt1:
        stdout.put( "I=1" );
```

```

        jmp EndCase;

Stmt2:
    stdout.put ( "I=2" );

EndCase:

```

The implementation of the IF.THEN.ELSEIF version is fairly obvious and doesn't need much in the way of explanation. The indirect jump version, however, is probably quite mysterious to you; so let's consider how this particular implementation of the SWITCH statement works.

Remember that there are three common forms of the JMP instruction (see "Unconditional Transfer of Control (JMP)" on page 729). The standard unconditional JMP instruction, like the "jmp EndCase;" instructions in the previous examples, transfer control directly to the statement label specified as the JMP operand. The second form of the JMP instruction (i.e., "jmp Reg32;") transfers control to the memory location specified by the address found in a 32-bit register. The third form of the JMP instruction, the one the example above uses, transfers control to the instruction specified by the contents of *dword* memory location. As this example clearly illustrates, that memory location can use any addressing mode. You are not limited to the displacement-only addressing mode. Now let's consider exactly how this second implementation of the SWITCH statement works.

To begin with, a SWITCH statement requires that you create an array of pointers with each element containing the address of a statement label in your code (those labels must be attached to the sequence of instructions to execute for each case in the SWITCH statement). In the example above, the *JmpTbl* array serves this purpose. Note that this code initializes *JmpTbl* with the address of the statement labels *Stmt0*, *Stmt1*, and *Stmt2*. The program places this array in the READONLY section because the program should never change these values during execution.

Warning: whenever you initialize an array with a set of address of statement labels as in this example, the declaration section in which you declare the array (e.g., READONLY in this case) must be in the same procedure that contains the statement labels³.

During the execution of this code sequence, the program loads the EAX register with *I*'s value. Then the program uses this value as an index into the *JmpTbl* array and transfers control to the four-byte address found at the specified location. For example, if EAX contains zero, the "jmp(JmpTbl[eax*4]);" instruction will fetch the *dword* at address *JmpTbl+0* (*eax*4=0*). Since the first *dword* in the table contains the address of *Stmt0*, the JMP instruction will transfer control to the first instruction following the *Stmt0* label. Likewise, if *I* (and therefore, EAX) contains one, then the indirect JMP instruction fetches the *dword* at offset four from the table and transfers control to the first instruction following the *Stmt1* label (since the address of *Stmt1* appears at offset four in the table). Finally, if *I*/EAX contains two, then this code fragment transfers control to the statements following the *Stmt2* label since it appears at offset eight in the *JmpTbl* table.

Two things should become readily apparent: the more (consecutive) cases you have, the more efficient the jump table implementation becomes (both in terms of space and speed). Except for trivial cases, the SWITCH statement is almost always faster and usually by a large margin. As long as the CASE values are consecutive, the SWITCH statement version is usually smaller as well.

What happens if you need to include non-consecutive CASE labels or you cannot be sure that the SWITCH value doesn't go out of range? In HLA, such an occurrence will transfer control to the first statement after the ENDSWITCH clause. However, this doesn't happen in the example above. If variable *I* does not contain zero, one, or two, the result of executing the code above is undefined. For example, if *I* contains five when you execute the code in the previous example, the indirect JMP instruction will fetch the *dword* at offset 20 (5*4) in *JmpTbl* and transfer control to that address. Unfortunately, *JmpTbl* doesn't have six entries; so the program will wind up fetching the value of the third double word following *JmpTbl* and using that as the target address. This will often crash your program or transfer control to an unexpected location. Clearly this code does not behave like the HLA SWITCH statement, nor does it have desirable behavior.

3. If the SWITCH statement appears in your main program, you must declare the array in the declaration section of your main program.

The solution is to place a few instructions before the indirect JMP to verify that the SWITCH selection value is within some reasonable range. In the previous example, we'd probably want to verify that *I*'s value is in the range 0..2 before executing the JMP instruction. If *I*'s value is outside this range, the program should simply jump to the *EndCase* label (this corresponds to dropping down to the first statement after the ENDSWITCH clause). The following code provides this modification to the previous example:

```
readonly
    JumpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];
    .
    .
    .
    mov( i, eax );
    cmp( eax, 2 );           // Verify that I is in the range
    ja EndCase;             // 0..2 before the indirect JMP.
    jmp( JumpTbl[ eax*4 ] );

    Stmt0:
        stdout.put( "I=0" );
        jmp EndCase;

    Stmt1:
        stdout.put( "I=1" );
        jmp EndCase;

    Stmt2:
        stdout.put( "I=2" );

    EndCase:
```

Although the example above handles the problem of selection values being outside the range zero through two, it still suffers from a couple of severe restrictions:

- The cases must start with the value zero. That is, the minimum CASE constant has to be zero in this example.
- The case values must be contiguous; there cannot be any gaps between any two case values.

Solving the first problem is easy and you deal with it in two steps. First, you must compare the case selection value against a lower and upper bounds before determining if the case value is legal, e.g.,

```
// SWITCH statement specifying cases 5, 6, and 7:
// WARNING: this code does *NOT* work. Keep reading to find out why.
```

```
mov( i, eax );
cmp( eax, 5 );
jb EndCase
cmp( eax, 7 );           // Verify that I is in the range
ja EndCase;             // 5..7 before the indirect JMP.
jmp( JumpTbl[ eax*4 ] );

    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

    Stmt7:
        stdout.put( "I=7" );

    EndCase:
```

As you can see, this code adds a pair of extra instructions, `CMP` and `JB`, to test the selection value to ensure it is in the range five through seven. If not, control drops down to the *EndCase* label, otherwise control transfers via the indirect `JMP` instruction. Unfortunately, as the comments point out, this code is broken. Consider what happens if variable *i* contains the value five: The code will verify that five is in the range five through seven and then it will fetch the dword at offset 20 (`5*@size(dword)`) and jump to that address. As before, however, this loads four bytes outside the bounds of the table and does not transfer control to a defined location. One solution is to subtract the smallest case selection value from `EAX` before executing the `JMP` instruction. E.g.,

```
// SWITCH statement specifying cases 5, 6, and 7:
// WARNING: there is a better way to do this. Keep reading.

readonly
    JumpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
    .
    .
    .
    mov( i, eax );
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );           // Verify that I is in the range
    ja EndCase;             // 5..7 before the indirect JMP.
    sub( 5, eax );           // 5->0, 6->1, 7->2.
    jmp( JumpTbl[ eax*4 ] );

    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

    Stmt7:
        stdout.put( "I=7" );

    EndCase:
```

By subtracting five from the value in `EAX` this code forces `EAX` to take on the values zero, one, or two prior to the `JMP` instruction. Therefore, case selection value five jumps to *Stmt5*, case selection value six transfers control to *Stmt6*, and case selection value seven jumps to *Stmt7*.

There is a sneaky way to slightly improve the code above. You can eliminate the `SUB` instruction by merging this subtraction into the `JMP` instruction's address expression. Consider the following code that does this:

```
// SWITCH statement specifying cases 5, 6, and 7:

readonly
    JumpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
    .
    .
    .
    mov( i, eax );
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );           // Verify that I is in the range
    ja EndCase;             // 5..7 before the indirect JMP.
    jmp( JumpTbl[ eax*4 - 5*@size(dword)] );
```

```

Stmt5:
    stdout.put( "I=5" );
    jmp EndCase;

Stmt6:
    stdout.put( "I=6" );
    jmp EndCase;

Stmt7:
    stdout.put( "I=7" );

EndCase:

```

The HLA SWITCH statement provides a DEFAULT clause that executes if the case selection value doesn't match any of the case values. E.g.,

```

switch( ebx )

    case( 5 )  stdout.put( "ebx=5" );
    case( 6 )  stdout.put( "ebx=6" );
    case( 7 )  stdout.put( "ebx=7" );
    default
        stdout.put( "ebx does not equal 5, 6, or 7" );

endswitch;

```

Implementing the equivalent of the DEFAULT clause in pure assembly language is very easy. Just use a different target label in the JB and JA instructions at the beginning of the code. The following example implements an HLA SWITCH statement similar to the one immediately above:

// SWITCH statement specifying cases 5, 6, and 7 with a DEFAULT clause:

```

readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
    .
    .
    .
mov( i, eax );
cmp( eax, 5 );
jb DefaultCase;
cmp( eax, 7 );           // Verify that I is in the range
ja DefaultCase;          // 5..7 before the indirect JMP.
jmp( JmpTbl[ eax*4 - 5*@size(dword)] );

Stmt5:
    stdout.put( "I=5" );
    jmp EndCase;

Stmt6:
    stdout.put( "I=6" );
    jmp EndCase;

Stmt7:
    stdout.put( "I=7" );

DefaultCase:
    stdout.put( "I does not equal 5, 6, or 7" );
EndCase:

```

The second restriction noted earlier, that the case values need to be contiguous, is easy to handle by inserting extra entries into the jump table. Consider the following HLA SWITCH statement:

```
switch( ebx )

    case( 1 ) stdout.put( "ebx = 1" );
    case( 2 ) stdout.put( "ebx = 2" );
    case( 4 ) stdout.put( "ebx = 4" );
    case( 8 ) stdout.put( "ebx = 8" );
    default
        stdout.put( "ebx is not 1, 2, 4, or 8" );

endswitch;
```

The minimum switch value is one and the maximum value is eight. Therefore, the code before the indirect JMP instruction needs to compare the value in EBX against one and eight. If the value is between one and eight, it's still possible that EBX might not contain a legal case selection value. However, since the JMP instruction indexes into a table of dwords using the case selection table, the table must have eight dword entries. To handle the values between one and eight that are not case selection values, simply put the statement label of the default clause (or the label specifying the first instruction after the ENDSWITCH if there is no DEFAULT clause) in each of the jump table entries that don't have a corresponding CASE clause. The following code demonstrates this technique for the above HLA SWITCH statement:

```
readonly
    JumpTbl2: dword :=
        [
            &Case1, &Case2, &dfltCase, &Case4,
            &dfltCase, &dfltCase, &dfltCase, &Case8
        ];
    .
    .
    .
    cmp( ebx, 1 );
    jb dfltCase;
    cmp( ebx, 8 );
    ja dfltCase;
    jmp( JumpTbl2[ ebx*4 - 1*@size(dword)] );

    Case1:
        stdout.put( "ebx = 1" );
        jmp EndOfSwitch;

    Case2:
        stdout.put( "ebx = 2" );
        jmp EndOfSwitch;

    Case4:
        stdout.put( "ebx = 4" );
        jmp EndOfSwitch;

    Case8:
        stdout.put( "ebx = 8" );
        jmp EndOfSwitch;

    dfltCase:
        stdout.put( "ebx is not 1, 2, 4, or 8" );

    EndOfSwitch:
```

There is a problem with this implementation of the SWITCH statement. If the CASE values contain non-consecutive entries that are widely spaced the jump table could become exceedingly large. The following SWITCH statement would generate an extremely large code file:

```
switch( ebx )

    case( 1      ) stmt1;
    case( 100    ) stmt2;
    case( 1_000  ) stmt3;
    case( 10_000 ) stmt4;
    default stmt5;

endswitch;
```

In this situation, your program will be much smaller if you implement the SWITCH statement with a sequence of IF statements rather than using an indirect jump statement. However, keep one thing in mind—the size of the jump table does not normally affect the execution speed of the program. If the jump table contains two entries or two thousand, the SWITCH statement will execute the multi-way branch in a constant amount of time. The IF statement implementation requires a linearly increasing amount of time for each case label appearing in the case statement.

Probably the biggest advantage to using assembly language over a HLL like Pascal or C/C++ is that you get to choose the actual implementation. In some instances you can implement a SWITCH statement as a sequence of IF..THEN..ELSEIF statements, or you can implement it as a jump table, or you can use a hybrid of the two:

```
switch( eax )

    case( 0 ) stmt0;
    case( 1 ) stmt1;
    case( 2 ) stmt2;
    case( 100 ) stmt3;
    default stmt4;

endswitch;
```

could become:

```
cmp( eax, 100 );
je DoStmt3;
cmp( eax, 2 );
ja TheDefaultCase;
jmp( JmpTbl[ eax*4 ] );
etc.
```

Of course, you could do this in HLA using the following code high-level control structures:

```
if( ebx = 100 ) then stmt3
else
    switch( eax )
        case(0) stmt0;
        case(1) stmt1;
        case(2) stmt2;
        Otherwise stmt4
    endswitch;
endif;
```

But this tends to destroy the readability of the program. On the other hand, the extra code to test for 100 in the assembly language code doesn't adversely affect the readability of the program (perhaps because it's so hard to read already). Therefore, most people will add the extra code to make their program more efficient.

The C/C++ SWITCH statement is very similar to the HLA SWITCH statement. There is only one major semantic difference: the programmer must explicitly place a BREAK statement in each CASE clause to

transfer control to the first statement beyond the SWITCH. This BREAK corresponds to the JMP instruction at the end of each CASE sequence in the assembly code above. If the corresponding BREAK is not present, C/C++ transfers control into the code of the following CASE. This is equivalent to leaving off the JMP at the end of the CASE'S sequence:

```
switch (i)
{
    case 0: stmt1;
    case 1: stmt2;
    case 2: stmt3;
        break;
    case 3: stmt4;
        break;
    default: stmt5;
}
```

This translates into the following 80x86 code:

```
readonly
JumpTbl: dword[4] := [ &case0, &case1, &case2, &case3 ];
.
.
.
mov( i, ebx );
cmp( ebx, 3 );
ja DefaultCase;
jmp( JumpTbl[ ebx*4 ] );

case0:
    stmt1;

case1:
    stmt2;

case2:
    stmt3;
    jmp EndCase;    // Emitted for the break stmt.

case3:
    stmt4;
    jmp EndCase;    // Emitted for the break stmt.

DefaultCase:
    stmt5;

EndCase:
```

2.9 State Machines and Indirect Jumps

Another control structure commonly found in assembly language programs is the *state machine*. A state machine uses a *state variable* to control program flow. The FORTRAN programming language provides this capability with the assigned GOTO statement. Certain variants of C (e.g., GNU's GCC from the Free Software Foundation) provide similar features. In assembly language, the indirect jump provides a mechanism to easily implement state machines.

So what is a state machine? In very basic terms, it is a piece of code⁴ which keeps track of its execution history by entering and leaving certain "states". For the purposes of this chapter, we'll not use a very formal

4. Note that state machines need not be software based. Many state machines' implementation are hardware based.

definition of a state machine. We'll just assume that a state machine is a piece of code which (somehow) remembers the history of its execution (its *state*) and executes sections of code based upon that history.

In a very real sense, all programs are state machines. The CPU registers and values in memory constitute the “state” of that machine. However, we'll use a much more constrained view. Indeed, for most purposes only a single variable (or the value in the EIP register) will denote the current state.

Now let's consider a concrete example. Suppose you have a procedure which you want to perform one operation the first time you call it, a different operation the second time you call it, yet something else the third time you call it, and then something new again on the fourth call. After the fourth call it repeats these four different operations in order. For example, suppose you want the procedure to ADD EAX and EBX the first time, subtract them on the second call, multiply them on the third, and divide them on the fourth. You could implement this procedure as follows:

```
procedure StateMachine;
static
    State:byte := 0;
begin StateMachine;

    cmp( State, 0 );
    jne TryState1;

    // State 0: Add EBX to EAX and switch to state 1:

    add( ebx, eax );
    inc( State );
    exit StateMachine;

TryState1:
    cmp( State, 1 );
    jne TryState2;

    // State 1: subtract ebx from EAX and switch to state 2:

    sub( ebx, eax );
    inc( State );      // State 1 becomes State 2.
    exit StateMachine;

TryState2:
    cmp( State, 2 );
    jne MustBeState3;

    // If this is state 2, multiply EAX by EAX and switch to state 3:

    intmul( ebx, eax );
    inc( State );      // State 2 becomes State 3.
    exit StateMachine;

    // If it isn't one of the above states, we must be in state 3
    // So divide eax by ebx and switch back to state zero.

MustBeState3:
    push( edx );      // Preserve this 'cause it gets whacked by DIV.
    xor( edx, edx );   // Zero extend EAX into EDX.
    div( ebx, edx:eax );
    pop( edx );        // Restore EDX's value preserved above.
    mov( 0, State );   // Reset the state back to zero.

end StateMachine;
```

Technically, this procedure is not the state machine. Instead, it is the variable *State* and the CMP/JNE instructions which constitute the state machine.

There is nothing particularly special about this code. It's little more than a SWITCH statement implemented via the IF..THEN..ELSEIF construct. The only thing special about this procedure is that it remembers how many times it has been called⁵ and behaves differently depending upon the number of calls. While this is a *correct* implementation of the desired state machine, it is not particularly efficient. The astute reader, of course, would recognize that this code could be made a little faster using an actual SWITCH statement rather than the IF..THEN..ELSEIF implementation. However, there is a better way...

The more common implementation of a state machine in assembly language is to use an *indirect jump*. Rather than having a state variable which contains a value like zero, one, two, or three, we could load the state variable with the *address* of the code to execute upon entry into the procedure. By simply jumping to that address, the state machine could save the tests above needed to execute the proper code fragment. Consider the following implementation using the indirect jump:

```
procedure StateMachine;
static
    State:dword := &State0;
begin StateMachine;

    jmp( State );

    // State 0: Add EBX to EAX and switch to state 1:

State0:
    add( ebx, eax );
    mov( &State1, State );
    exit StateMachine;

State1:

    // State 1: subtract ebx from EAX and switch to state 2:

    sub( ebx, eax );
    mov( &State2, State );    // State 1 becomes State 2.
    exit StateMachine;

State2:

    // If this is state 2, multiply EAX by EAX and switch to state 3:

    intmul( ebx, eax );
    mov( &State3, State );    // State 2 becomes State 3.
    exit StateMachine;

    // State 3: divide eax by ebx and switch back to state zero.

State3:
    push( edx );              // Preserve this 'cause it gets whacked by DIV.
    xor( edx, edx );          // Zero extend EAX into EDX.
    div( ebx, edx:eax );
    pop( edx );               // Restore EDX's value preserved above.
    mov( &State0, State );    // Reset the state back to zero.

end StateMachine;
```

The JMP instruction at the beginning of the *StateMachine* procedure transfers control to the location pointed at by the *State* variable. The first time you call *StateMachine* it points at the *State0* label. Thereafter, each subsection of code sets the *State* variable to point at the appropriate successor code.

5. Actually, it remembers how many times, *MOD 4*, that it has been called.

2.10 Spaghetti Code

One major problem with assembly language is that it takes several statements to realize a simple idea encapsulated by a single HLL statement. All too often an assembly language programmer will notice that s/he can save a few bytes or cycles by jumping into the middle of some programming structure. After a few such observations (and corresponding modifications) the code contains a whole sequence of jumps in and out of portions of the code. If you were to draw a line from each jump to its destination, the resulting listing would end up looking like someone dumped a bowl of spaghetti on your code, hence the term “spaghetti code”.

Spaghetti code suffers from one major drawback- it’s difficult (at best) to read such a program and figure out what it does. Most programs start out in a “structured” form only to become spaghetti code at the altar of efficiency. Alas, spaghetti code is rarely efficient. Since it’s difficult to figure out exactly what’s going on, it’s very difficult to determine if you can use a better algorithm to improve the system. Hence, spaghetti code may wind up less efficient than structured code.

While it’s true that producing some spaghetti code in your programs may improve its efficiency (e.g., destructuring your code, see “Efficient Implementation of IF Statements in Assembly Language” on page 747), doing so should always be a last resort after you’ve tried everything else and you still haven’t achieved what you need. Always start out writing your programs with straight-forward IFs and SWITCH statements. Start combining sections of code (via JMP instructions) once everything is working and well understood. Of course, you should never obliterate the structure of your code unless the gains are worth it.

A famous saying in structured programming circles is “After GOTOs, pointers are the next most dangerous element in a programming language.” A similar saying is “Pointers are to data structures what GOTOs are to control structures.” In other words, avoid excessive use of pointers. If pointers and gotos are bad, then the indirect jump must be the worst construct of all since it involves both GOTOs and pointers! Seriously though, the indirect jump instructions should be avoided for casual use. They tend to make a program harder to read. After all, an indirect jump can (theoretically) transfer control to any label within a program. Imagine how hard it would be to follow the flow through a program if you have no idea what a pointer contains and you come across an indirect jump using that pointer. Therefore, you should always exercise care when using jump indirect instructions.

2.11 Loops

Loops represent the final basic control structure (sequences, decisions, and loops) that make up a typical program. Like so many other structures in assembly language, you’ll find yourself using loops in places you’ve never dreamed of using loops. Most HLLs have implied loop structures hidden away. For example, consider the BASIC statement “IF A\$ = B\$ THEN 100”. This IF statement compares two strings and jumps to statement 100 if they are equal. In assembly language, you would need to write a loop to compare each character in A\$ to the corresponding character in B\$ and then jump to statement 100 if and only if all the characters matched. In BASIC, there is no loop to be seen in the program. In assembly language, this very simple IF statement requires a loop to compare the individual characters in the string⁶. This is but a small example which shows how loops seem to pop up everywhere.

Program loops consist of three components: an optional initialization component, a loop termination test, and the body of the loop. The order with which these components are assembled can dramatically change the way the loop operates. Three permutations of these components appear over and over again. Because of their frequency, these loop structures are given special names in high-level languages: WHILE loops, REPEAT..UNTIL loops (do..while in C/C++), and infinite loops (e.g., FOREVER..ENDFOR in HLA).

6. Of course, HLA provides the *str:eq* routine that compares the strings for you, effectively hiding the loop even in an assembly language program.

2.11.1 While Loops

The most general loop is the WHILE loop. In HLA it takes the following form:

```
while( expression ) do <<statements>> endwhile;
```

There are two important points to note about the WHILE loop. First, the test for termination appears at the beginning of the loop. Second as a direct consequence of the position of the termination test, the body of the loop may never execute. If the termination condition always exists, the loop body will always be skipped over.

Consider the following HLA WHILE loop:

```
mov( 0, I );
while( I < 100 ) do

    inc( I );

endwhile;
```

The “mov(0, I);” instruction is the initialization code for this loop. *I* is a loop control variable, because it controls the execution of the body of the loop. “I<100” is the loop termination condition. That is, the loop will not terminate as long as *I* is less than 100. The single instruction “inc(I);” is the loop body. This is the code that executes on each pass of the loop.

Note that an HLA WHILE loop can be easily synthesized using IF and JMP statements. For example, the HLA WHILE loop presented above can be replaced by:

```
mov( 0, I );
WhileLp:
if( I < 100 ) then

    inc( i );
    jmp WhileLp;

endif;
```

More generally, any WHILE loop can be built up from the following:

```
<< optional initialization code>>

UniqueLabel:
if( not_termination_condition ) then

    <<loop body>>
    jmp UniqueLabel;

endif;
```

Therefore, you can use the techniques from earlier in this chapter to convert IF statements to assembly language along with a single JMP instruction to produce a WHILE loop. The example we’ve been looking at in this section translates to the following “pure” 80x86 assembly code:

```
mov( 0, i );
WhileLp:
    cmp( i, 100 );
    jnl WhileDone;
    inc( i );
    jmp WhileLp;

WhileDone:
```

2.11.2 Repeat..Until Loops

The REPEAT..UNTIL (do..while) loop tests for the termination condition at the end of the loop rather than at the beginning. In HLA, the REPEAT..UNTIL loop takes the following form:

```
<< optional initialization code >>
repeat

    <<loop body>>

until( termination_condition );
```

This sequence executes the initialization code, the loop body, then tests some condition to see if the loop should repeat. If the boolean expression evaluates to false, the loop repeats; otherwise the loop terminates. The two things to note about the REPEAT..UNTIL loop are that the termination test appears at the end of the loop and, as a direct consequence of this, the loop body always executes at least once.

Like the WHILE loop, the repeat..until loop can be synthesized with an IF statement and a JMP . You could use the following:

```
<< initialization code >>
SomeUniqueLabel:

    << loop body >>

if( not_the_termination_condition ) then jmp SomeUniqueLabel; endif;
```

Based on the material presented in the previous sections, you can easily synthesize REPEAT..UNTIL loops in assembly language. The following is a simple example:

```
repeat

    stdout.put( "Enter a number greater than 100: " );
    stdin.get( i );

until( i > 100 );

// This translates to the following IF/JMP code:

RepeatLbl:

    stdout.put( "Enter a number greater than 100: " );
    stdin.get( i );

    if( i <= 100 ) then jmp RepeatLbl; endif;

// It also translates into the following "pure" assembly code:

RepeatLabel:

    stdout.put( "Enter a number greater than 100: " );
    stdin.get( i );

    cmp( i, 100 );
    jng RepeatLbl;
```

2.11.3 FOREVER..ENDFOR Loops

If WHILE loops test for termination at the beginning of the loop and REPEAT..UNTIL loops check for termination at the end of the loop, the only place left to test for termination is in the middle of the loop. The HLA FOREVER..ENDFOR loop, combined with the BREAK and BREAKIF statements, provide this capability. The FOREVER..ENDFOR loop takes the following form:

```
forever

    << loop body >>

endfor;
```

Note that there is no explicit termination condition. Unless otherwise provided for, the FOREVER..ENDFOR construct simply forms an infinite loop. Loop termination is typically handled by a BREAKIF statement. Consider the following HLA code that employs a FOREVER..ENDFOR construct:

```
forever

    stdin.get( character );
    breakif( character = '.' );
    stdout.put( character );

endfor;
```

Converting a FOREVER loop to pure assembly language is trivial. All you need is a label and a JMP instruction. The BREAKIF statement in this example is really nothing more than an IF and a JMP instruction. The “pure” assembly language version of the code above looks something like the following:

```
foreverLabel:

    stdin.get( character );
    cmp( character, '.' );
    je ForIsDone;
    stdout.put( character );
    jmp foreverLabel;

ForIsDone:
```

2.11.4 FOR Loops

The FOR loop is a special form of the WHILE loop that repeats the loop body a specific number of times. In HLA, the FOR loop takes the following form:

```
for( <<Initialization Stmt>>; <<Termination Expression>>; <<inc_Smt>> ) do

    << statements >>

endfor;
```

This is completely equivalent to the following:

```
<< Initialization Stmt>>;
while( <<Termination Expression>> ) do

    << statements >>

    <<inc_Smt>>

endwhile;
```

Traditionally, the FOR loop has been used to process arrays and other objects accessed in sequential numeric order. One normally initializes a loop control variable with the initialization statement and then uses the loop control variable as an index into the array (or other data type), e.g.,

```
for( mov(0, esi); esi < 7; inc( esi ) ) do

    stdout.put( "Array Element = ", SomeArray[ esi*4], nl );

endfor;
```

To convert this to “pure” assembly language, begin by translating the FOR loop into an equivalent WHILE loop:

```
mov( 0, esi );
while( esi < 7 ) do

    stdout.put( "Array Element = ", SomeArray[ esi*4], nl );

    inc( esi );
endwhile;
```

Now, using the techniques from the section on WHILE loops (see “While Loops” on page 762), translate the code into pure assembly language:

```
mov( 0, esi );
WhileLp:
cmp( esi, 7 );
jnl EndWhileLp;

    stdout.put( "Array Element = ", SomeArray[ esi*4], nl );

    inc( esi );
    jmp WhileLp;

EndWhileLp:
```

2.11.5 The BREAK and CONTINUE Statements

The HLA BREAK and CONTINUE statements both translate into a single JMP instruction. The BREAK instruction exits the loop that immediately contains the BREAK statement; the CONTINUE statement restarts the loop that immediately contains the CONTINUE statement.

Converting a BREAK statement to “pure” assembly language is very easy. Just emit a JMP instruction that transfers control to the first statement following the ENDxxxx clause of the loop to exit. This is easily accomplished by placing a label after the associated END clause and jumping to that label. The following code fragments demonstrate this technique for the various loops:

// Breaking out of a forever loop:

```
forever
    <<stmts>>
        //break;
        jmp BreakFromForever;
    <<stmts>>
endfor;
BreakFromForever:
```

// Breaking out of a FOR loop;

```

for( <<initStmt>>; <<expr>>; <<incStmt>> ) do
    <<stmts>>
    //break;
    jmp BrkFromFor;
    <<stmts>>
endfor;
BrkFromFor:

// Breaking out of a WHILE loop:

while( <<expr>> ) do
    <<stmts>>
    //break;
    jmp BrkFromWhile;
    <<stmts>>
endwhile;
BrkFromWhile:

// Breaking out of a REPEAT..UNTIL loop:

repeat
    <<stmts>>
    //break;
    jmp BrkFromRpt;
    <<stmts>>
until( <<expr>> );
BrkFromRpt:

```

The CONTINUE statement is slightly more difficult to implement than the BREAK statement. The implementation still consists of a single JMP instruction, however the target label doesn't wind up going in the same spot for each of the different loops. The following figures show where the CONTINUE statement transfers control for each of the HLA loops:

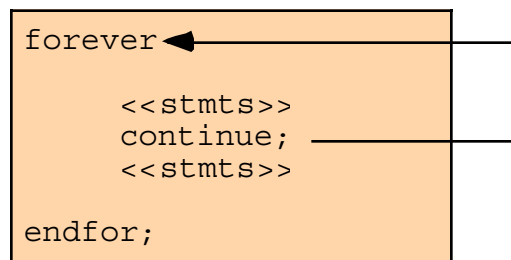


Figure 2.2 CONTINUE Destination for the FOREVER Loop

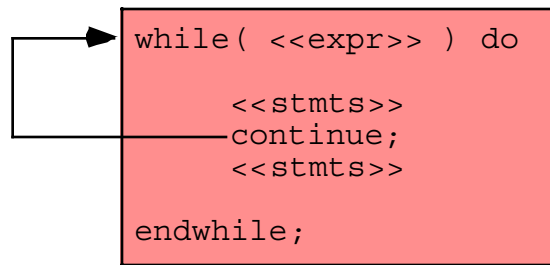
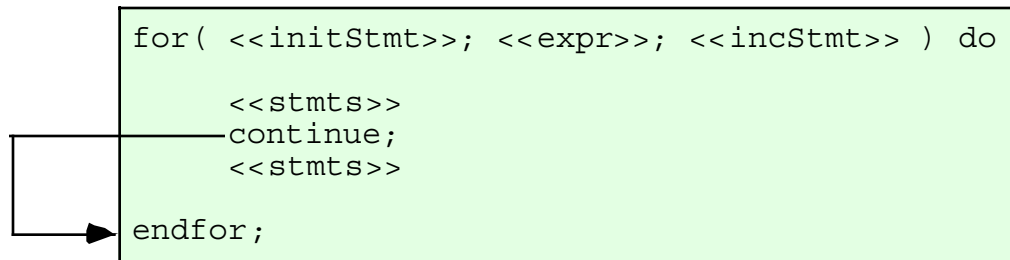


Figure 2.3 CONTINUE Destination and the WHILE Loop



Note: CONTINUE forces the execution of the `<<incStmt>>` clause and then transfers control to the test for loop termination.

Figure 2.4 CONTINUE Destination and the FOR Loop

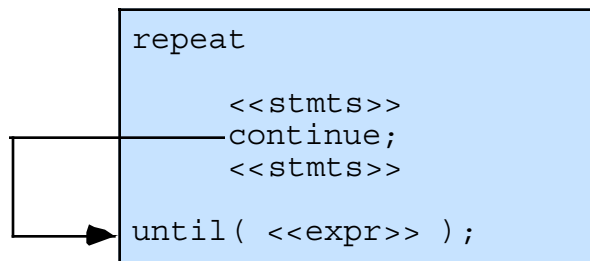


Figure 2.5 CONTINUE Destination and the REPEAT..UNTIL Loop

The following code fragments demonstrate how to convert the CONTINUE statement into an appropriate JMP instruction for each of these loop types.

forever..continue..endfor

```

// Conversion of forever loop w/continue
// to pure assembly:
forever
    <<stmts>>
    continue;
    <<stmts>>

```

```
endfor;
```

```
// Converted code:
```

```
foreverLbl:
    <<stmts>>
    jmp foreverLbl;
    <<stmts>>
```

while..continue..endwhile

```
// Conversion of while loop w/continue
// into pure assembly:
```

```
while( <<expr>> ) do
    <<stmts>>
    continue;
    <<stmts>>
endwhile;
```

```
// Converted code:
```

```
whlLabel:
<<Code to evaluate expr>>
Jcc EndOfWhile; // Skip loop on expr failure.
    <<stmts>>
    jmp whlLabel; // Jump to start of loop on continue.
    <<stmts>>
    jmp whlLabel; // Repeat the code.
EndOfwhile:
```

for..continue..endfor

```
// Conversion for a for loop w/continue
// into pure assembly:
```

```
for( <<initStmt>>; <<expr>>; <<incStmt>> ) do
    <<stmts>>
    continue;
    <<stmts>>
endfor;
```

```
// Converted code
```

```
<<initStmt>>
ForLpLbl:
<<Code to evaluate expr>>
Jcc EndOfFor; // Branch if expression fails.
    <<stmts>>
    jmp ContFor; // Branch to <<incStmt>> on continue.
    <<stmts>>

    ContFor:
        <<incStmt>>
        jmp ForLpLbl;
EndOfFor:
```

repeat..continue..until

```
repeat
    <<stmts>>
```

```

        continue;
        <<stmts>>
until( <<expr>> );

// Converted Code:

RptLpLbl:
    <<stmts>>
    jmp ContRpt; // Continue branches to loop termination test.
    <<stmts>>
ContRpt:
    <<code to test expr>>
    Jcc RptLpLbl; // Jumps if expression evaluates false.

```

2.11.6 Register Usage and Loops

Given that the 80x86 accesses registers much faster than memory locations, registers are the ideal spot to place loop control variables (especially for small loops). However, there are some problems associated with using registers within a loop. The primary problem with using registers as loop control variables is that registers are a limited resource. Therefore, the following will not work properly:

```

        mov( 8, cx );
loop1:
        mov( 4, cx );
loop2:
        <<stmts>>
        dec( cx );
        jnz loop2;

        dec( cx );
        jnz loop1;

```

The intent here, of course, was to create a set of nested loops, that is, one loop inside another. The inner loop (*Loop2*) should repeat four times for each of the eight executions of the outer loop (*Loop1*). Unfortunately, both loops use the same register as a loop control variable.. Therefore, this will form an infinite loop since CX will be set to zero at the end of the first loop. Since CX is always zero upon encountering the second DEC instruction, control will always transfer to the LOOP1 label (since decrementing zero produces a non-zero result). The solution here is to save and restore the CX register or to use a different register in place of CX for the outer loop:

```

        mov( 8, cx );
loop1:
        push( cx );
        mov( 4, cx );
loop2:
        <<stmts>>
        dec( cx );
        jnz loop2;

        pop( cx );
        dec( cx );
        jnz loop1;

```

or:

```

        mov( 8, dx );
loop1:
        mov( 4, cx );
loop2:
        <<stmts>>
        dec( cx );

```

```

        jnz loop2;

        dec( dx );
        jnz loop1;

```

Register corruption is one of the primary sources of bugs in loops in assembly language programs, always keep an eye out for this problem.

2.12 Performance Improvements

The 80x86 microprocessors execute sequences of instructions at blinding speeds. Therefore, you'll rarely encounter a program that is slow which doesn't contain any loops. Since loops are the primary source of performance problems within a program, they are the place to look when attempting to speed up your software. While a treatise on how to write efficient programs is beyond the scope of this chapter, there are some things you should be aware of when designing loops in your programs. They're all aimed at removing unnecessary instructions from your loops in order to reduce the time it takes to execute one iteration of the loop.

2.12.1 Moving the Termination Condition to the End of a Loop

Consider the following flow graphs for the three types of loops presented earlier:

REPEAT..UNTIL loop:

```

Initialization code
  Loop body
  Test for termination
  Code following the loop

```

WHILE loop:

```

Initialization code
  Loop termination test
  Loop body
  Jump back to test
  Code following the loop

```

FOREVER..ENDFOR loop:

```

Initialization code
  Loop body, part one
  Loop termination test
  Loop body, part two
  Jump back to loop body part 1
  Code following the loop

```

As you can see, the REPEAT..UNTIL loop is the simplest of the bunch. This is reflected in the assembly language code required to implement these loops. Consider the following REPEAT..UNTIL and WHILE loops that are identical:

// Example involving a WHILE loop:

```

mov( edi, esi );
sub( 20, esi );
while( esi <= edi ) do

    <<stmts>>
    inc( esi );

endwhile;

```

```
// Conversion of the code above into pure assembly language:
```

```
mov( edi, esi );
sub( 20, esi );
whlLbl:
cmp( esi, edi );
jnl EndOfWhile;
```

```
<<stmts>>
inc( esi );
<<stmts>>
jmp whlLbl;
```

```
EndOfWhile:
```

```
//Example involving a REPEAT..UNTIL loop:
```

```
mov( edi, esi );
sub( 20, esi );
repeat
```

```
<<stmts>>
inc( esi );
```

```
until( esi > edi );
```

```
// Conversion of the REPEAT..UNTIL loop into pure assembly:
```

```
rptLabel:
<<stmts>>
inc( esi );
cmp( esi, edi );
jng rptLabel;
```

As you can see by carefully studying the conversion to pure assembly language, testing for the termination condition at the end of the loop allowed us to remove a JMP instruction from the loop. This can be significant if this loop is nested inside other loops. In the preceding example there wasn't a problem with executing the body at least once. Given the definition of the loop, you can easily see that the loop will be executed exactly 20 times. This suggests that the conversion to a REPEAT..UNTIL loop is trivial and always possible. Unfortunately, it's not always quite this easy. Consider the following HLA code:

```
while( esi <= edi ) do
<<stmts>>
inc( esi );
endwhile;
```

In this particular example, we haven't the slightest idea what ESI contains upon entry into the loop. Therefore, we cannot assume that the loop body will execute at least once. So we must test for loop termination before executing the body of the loop. The test can be placed at the end of the loop with the inclusion of a single JMP instruction:

```
jmp WhlTest;
TopOfLoop:
<<stmts>>
inc( esi );
WhlTest:
cmp( esi, edi );
jle TopOfLoop;
```

Although the code is as long as the original WHILE loop, the JMP instruction executes only once rather than on each repetition of the loop. Note that this slight gain in efficiency is obtained via a slight loss in readability. The second code sequence above is closer to spaghetti code than the original implementation. Such is often the price of a small performance gain. Therefore, you should carefully analyze your code to ensure that the performance boost is worth the loss of clarity. More often than not, assembly language programmers sacrifice clarity for dubious gains in performance, producing impossible to understand programs.

Note, by the way, that HLA translates its WHILE statement into a sequence of instructions that test the loop termination condition at the bottom of the loop using exactly the technique this section describes. Therefore, you do not have to worry about the HLA WHILE statement introducing slower code into your programs.

2.12.2 Executing the Loop Backwards

Because of the nature of the flags on the 80x86, loops which range from some number down to (or up to) zero are more efficient than any other. Compare the following HLA FOR loop and the code it generates:

```
for( mov( 1, J ); J <= 8; inc(J) ) do
    <<stmts>>
endfor;

// Conversion to pure assembly (as well as using a repeat..until form):

mov( 1, J );
ForLp:
    <<stmts>>
    inc( J );
    cmp( J, 8 );
    jnge ForLp;
```

Now consider another loop that also has eight iterations, but runs its loop control variable from eight down to one rather than one up to eight:

```
mov( 8, J );
LoopLb1:
    <<stmts>>
    dec( J );
    jnz LoopLb1;
```

Note that by running the loop from eight down to one we saved a comparison on each repetition of the loop.

Unfortunately, you cannot force all loops to run backwards. However, with a little effort and some coercion you should be able to work most FOR loops so they operate backwards. By saving the execution time of the CMP instruction on each iteration of the loop the code may run faster.

The example above worked out well because the loop ran from eight down to one. The loop terminated when the loop control variable became zero. What happens if you need to execute the loop when the loop control variable goes to zero? For example, suppose that the loop above needed to range from seven down to zero. As long as the upper bound is positive, you can substitute the JNS instruction in place of the JNZ instruction above to repeat the loop some specific number of times:

```
mov( 7, J );
LoopLb1:
    <<stmts>>
    dec( J );
    jns LoopLb1;
```

This loop will repeat eight times with *J* taking on the values seven down to zero on each execution of the loop. When it decrements zero to minus one, it sets the sign flag and the loop terminates.

Keep in mind that some values may look positive but they are negative. If the loop control variable is a byte, then values in the range 128..255 are negative. Likewise, 16-bit values in the range 32768..65535 are negative. Therefore, initializing the loop control variable with any value in the range 129..255 or 32769..65535 (or, of course, zero) will cause the loop to terminate after a single execution. This can get you into a lot of trouble if you're not careful.

2.12.3 Loop Invariant Computations

A loop invariant computation is some calculation that appears within a loop that always yields the same result. You needn't do such computations inside the loop. You can compute them outside the loop and reference the value of the computation inside the loop. The following HLA code demonstrates a loop which contains an invariant computation:

```
for( mov( 0, eax ); eax < n; inc( eax ) ) do

    mov( eax, edx );
    add( j, edx );
    sub( 2, edx );
    add( edx, k );

endfor;
```

Since *j* never changes throughout the execution of this loop, the sub-expression “*j*-2” can be computed outside the loop and its value used in the expression inside the loop:

```
mov( j, ecx );
sub( 2, ecx );
for( mov( 0, eax ); eax < n; inc( eax ) ) do

    mov( eax, edx );
    add( ecx, edx );
    add( edx, k );

endfor;
```

Still, the value in ECX never changes inside this loop, so although we've eliminated a single instruction by computing the subexpression “*j*-2” outside the loop, there is still an invariant component to this calculation. Since we note that this invariant component executes *n* times in the loop, we can translate the code above to the following:

```
mov( j, ecx );
sub( 2, ecx );
intmul( n, ecx );    // Compute n*(j-2) and add this into k outside
add( ecx, k );       // the loop.
for( mov( 0, eax ); eax < n; inc( eax ) ) do

    add( eax, k );

endfor;
```

As you can see, we've shrunk the loop body from four instructions down to one. Of course, if you're really interested in improving the efficiency of this particular loop, you'd be much better off (most of the time) computing *k* using the formula:

$$k = k + ((n + 1) \times temp) + \frac{(n + 2) \times (n + 2)}{2}$$

This computation for k is based on the formula:

$$\sum_{i=0}^n i = \frac{(n+1) \times (n)}{2}$$

However, simple computations such as this one aren't always possible. Still, this demonstrates that a better algorithm is almost always better than the trickiest code you can come up with.

Removing invariant computations and unnecessary memory accesses from a loop (particularly an inner loop in a set of nested loops) can produce dramatic performance improvements in a program.

2.12.4 Unraveling Loops

For small loops, that is, those whose body is only a few statements, the overhead required to process a loop may constitute a significant percentage of the total processing time. For example, look at the following Pascal code and its associated 80x86 assembly language code:

```
FOR I := 3 DOWNT0 0 DO A [I] := 0;

mov( 3, I );
LoopLbl:
    mov( I, ebx );
    mov( 0, A[ebx*4] );
    dec( I );
    jns LoopLbl;
```

Each execution of the loop requires four instructions. Only one instruction is performing the desired operation (moving a zero into an element of A). The remaining three instructions control the repetition of the loop. Therefore, it takes 16 instructions to do the operation logically required by four.

While there are many improvements we could make to this loop based on the information presented thus far, consider carefully exactly what it is that this loop is doing-- it's simply storing four zeros into $A[0]$ through $A[3]$. A more efficient approach is to use four MOV instructions to accomplish the same task. For example, if A is an array of dwords, then the following code initializes A much faster than the code above:

```
mov( 0, A[0] );
mov( 0, A[4] );
mov( 0, A[8] );
mov( 0, A[12] );
```

Although this is a trivial example, it shows the benefit of loop unraveling. If this simple loop appeared buried inside a set of nested loops, the 4:1 instruction reduction could possibly double the performance of that section of your program.

Of course, you cannot unravel all loops. Loops that execute a variable number of times cannot be unraveled because there is rarely a way to determine (at assembly time) the number of times the loop will execute. Therefore, unraveling a loop is a process best applied to loops that execute a known number of times (and the number of times is known at assembly time).

Even if you repeat a loop some fixed number of iterations, it may not be a good candidate for loop unraveling. Loop unraveling produces impressive performance improvements when the number of instructions required to control the loop (and handle other overhead operations) represent a significant percentage of the total number of instructions in the loop. Had the loop above contained 36 instructions in the body of the loop (exclusive of the four overhead instructions), then the performance improvement would be, at best, only 10% (compared with the 300-400% it now enjoys). Therefore, the costs of unraveling a loop, i.e., all the extra code which must be inserted into your program, quickly reaches a point of diminishing returns as the body of the loop grows larger or as the number of iterations increases. Furthermore, entering that code into your program can become quite a chore. Therefore, loop unraveling is a technique best applied to small loops.

Note that the superscalar x86 chips (Pentium and later) have *branch prediction hardware* and use other techniques to improve performance. Loop unrolling on such systems many actually *slow down* the code since these processors are optimized to execute short loops.

2.12.5 Induction Variables

Consider the following modification of the loop presented in the previous section:

```
FOR I := 0 TO 255 DO csetVar[I] := {};
```

Here the program is initializing each element of an array of character sets to the empty set. The straight-forward code to achieve this is the following:

```
mov( 0, i );
FLp:

    // Compute the index into the array (note that each element
    // of a CSET array contains 16 bytes).

    mov( i, ebx );
    shl( 4, ebx );

    // Set this element to the empty set (all zero bits).

    mov( 0, csetVar[ ebx ] );
    mov( 0, csetVar[ ebx+4 ] );
    mov( 0, csetVar[ ebx+8 ] );
    mov( 0, csetVar[ ebx+12 ] );

    inc( i );
    cmp( i, 256 );
    jb FLp;
```

Although unraveling this code will still produce a performance improvement, it will take 1024 instructions to accomplish this task, too many for all but the most time-critical applications. However, you can reduce the execution time of the body of the loop using *induction variables*. An induction variable is one whose value depends entirely on the value of some other variable. In the example above, the index into the array *csetVar* tracks the loop control variable (it's always equal to the value of the loop control variable times 16). Since *i* doesn't appear anywhere else in the loop, there is no sense in performing all the computations on *i*. Why not operate directly on the array index value? The following code demonstrates this technique:

```
mov( 0, ebx );
FLp:
    mov( 0, csetVar[ ebx ] );
    mov( 0, csetVar[ ebx+4 ] );
    mov( 0, csetVar[ ebx+8 ] );
    mov( 0, csetVar[ ebx+12 ] );

    add( 16, ebx );
    cmp( ebx, 256*16 );
    jb FLp;
```

The induction that takes place in this example occurs when the code increments the loop control variable (moved into EBX for efficiency reasons) by 16 on each iteration of the loop rather than by one. Multiplying the loop control variable by 16 allows the code to eliminate multiplying the loop control variable by 16 on each iteration of the loop (i.e., this allows us to remove the SHL instruction from the previous code). Further, since this code no longer refers to the original loop control variable (*i*), the code can maintain the loop control variable strictly in the EBX register.

2.13 Hybrid Control Structures in HLA

The HLA high level language control structures have three principle drawbacks: (1) they're not true assembly language instructions, (2) they do not support complex boolean expressions, and (3) they often introduce inefficient coding practices into a language that most people only use when they need to write high-performance code. On the other hand, while the 80x86 low level control structures let you write efficient code and handle complex boolean expressions, the resulting code is very difficult to read and maintain. HLA provides a set of hybrid control structures that allow you to use pure assembly language statements to evaluate boolean expressions while using the high level control structures to delineate the statements controlled by the boolean expressions. The result is code that is much more readable than pure assembly language without being a whole lot less efficient.

HLA provides hybrid forms of the IF..ELSEIF..ELSE..ENDIF, WHILE..ENDWHILE, REPEAT..UNTIL, BREAKIF, EXITIF, and CONTINUEIF statements (i.e., those that involve a boolean expression). For example, a hybrid IF statement takes the following form:

```
if{ <<statements>> } <<statements>> endif;
```

Note the use of curly braces (rather than parentheses) to surround the statements corresponding to the boolean expression. Also note that there is no THEN keyword to this statement. This is what differentiates the hybrid control structures from the standard high level language control structures. The remaining hybrid control structures take the following forms:

```
while{ <<statements>> } <<statements>> endwhile;
repeat <<statements>> until{ <<statements>> };
breakif{ <<statements>> };
exitif{ <<statements>> };
continueif{ <<statements>> };
```

The statements within the curly braces replace the normal boolean expression in an HLA high level control structure. These particular statements are special insofar as HLA defines two labels, *true* and *false*, within their context. HLA associates the label *true* with the code that would normally execute if a boolean expression were present and that expression's result was true. Similarly, HLA associates the label *false* with the code that would execute if a boolean expression in one of these statements evaluated false. As a simple example, consider the following two (equivalent) IF statements:

```
if( eax < ebx ) then inc( eax ); endif;
```

```
if
{
    cmp( eax, ebx );
    jnl false;
}
    inc( eax );

endif;
```

The JNL that transfers control to the *false* label in this latter example will skip over the INC instruction if EAX is not less than EBX. Note that if EAX is less than EBX then control falls through to the INC instruction. This is roughly equivalent to the following pure assembly code:

```
cmp( eax, ebx );
jnl falseLabel;
    inc( eax );
falseLabel:
```

As a slightly more complex example, consider the (illegal, though desirable) statement

```
if( eax >= J && eax <= K ) then sub( J, eax ); endif;
```

The following hybrid IF statement accomplishes the above:

```
if
{
    cmp( eax, J );
    jnge false;
    cmp( eax, K );
    jnle false;
}
    sub( J, eax );

endif;
```

As one final example of the hybrid IF statement, consider the following:

```
// if( ((eax > ebx) && (eax < ecx)) || (eax = edx)) then mov( ebx, eax ); endif;

if
{
    cmp( eax, edx );
    je true;
    cmp( eax, ebx );
    jng false;
    cmp( eax, ecx );
    jnl false;
}
    mov( ebx, eax );

endif;
```

Since these examples are rather trivial, they don't really demonstrate how much more readable the code can be when using hybrid statements rather than pure assembly code. However, one thing you notice is that the use of hybrid statements eliminate the need to insert labels throughout your code. This is what makes your programs easier to read and understand.

For the IF statement, the *true* label corresponds to the THEN clause of the statement; the *false* label corresponds to the ELSEIF, ELSE, or ENDIF clause (whichever follows the THEN clause). For the WHILE loop, the *true* label corresponds to the body of the loop while the *false* label is attached to the first statement following the corresponding ENDWHILE. For the REPEAT..UNTIL statement, the *true* label is attached to the code following the UNTIL clause while the *false* label is attached to the first statement of the body of the loop. The BREAKIF, EXITIF, and CONTINUEIF statements associate the *false* label with the statement immediately following one of these statements, they associate the *true* label with the code normally associated with a BREAK, EXIT, or CONTINUE statement.

2.14 Putting It All Together

In this chapter we've taken a look at the low-level, or "pure" assembly language, implementation of several common control structures. Although HLA's high level control structures are easy to use and quite a bit more readable than their low-level equivalents, sometimes efficiency demands a low-level implementation. This chapter presents the blueprints for such transformations.

While this chapter covers the principle high level control structures and their translation to assembly language, there are some additional control structures that this chapter does not consider. Iterators and the TRY..ENDTRY statement are two good examples. Fear not, however, the volume on Advanced Assembly Language Programming will tidy up those loose ends.

Intermediate Procedures

Chapter Three

3.1 Chapter Overview

This chapter picks up where the chapter “Introduction to Procedures” in Volume Three leaves off. That chapter presented a high level view of procedures, parameters, and local variables; this chapter takes a look at some of the low-level implementation details. This chapter begins by discussing the CALL instruction and how it affects the stack. Then it discusses activation records and how a program passes parameters to a procedure and how that procedure maintains local (automatic) variables. Next, this chapter presents an in-depth discussion of pass by value and pass by reference parameters. This chapter concludes by discussing procedure variables, procedural parameters, iterators, and the FOREACH..ENDFOR loop.

3.2 Procedures and the CALL Instruction

Most procedural programming languages implement procedures using the call/return mechanism. That is, some code calls a procedure, the procedure does its thing, and then the procedure returns to the caller. The call and return instructions provide the 80x86’s *procedure invocation mechanism*. The calling code calls a procedure with the CALL instruction, the procedure returns to the caller with the RET instruction. For example, the following 80x86 instruction calls the HLA Standard Library *stdout.newln* routine:

```
call stdout.newln;
```

stdout.newln prints a carriage return/line feed sequence to the video display and returns control to the instruction immediately following the “call stdout.newln;” instruction.

The HLA language lets you call procedures using a high level language syntax. Specifically, you may call a procedure by simply specifying the procedure’s name and (in the case of *stdout.newln*) an empty parameter list. That is, the following is completely equivalent to “call stdout.newln;”:

```
stdout.newln();
```

The 80x86 CALL instruction does two things. First, it pushes the address of the instruction immediately following the CALL onto the stack; then it transfers control to the address of the specified procedure. The value that CALL pushes onto the stack is known as the *return address*. When the procedure wants to return to the caller and continue execution with the first statement following the CALL instruction, the procedure simply pops the return address off the stack and jumps (indirectly) to that address. Most procedures return to their caller by executing a RET (return) instruction. The RET instruction pops a return address off the stack and transfers control indirectly to the address it pops off the stack.

By default, the HLA compiler automatically places a RET instruction (along with a few other instructions) at the end of each HLA procedure you write. This is why you haven’t had to explicitly use the RET instruction up to this point. To disable the default code generation in an HLA procedure, specify the following options when declaring your procedures:

```
procedure ProcName; noframe; nodisplay;
begin ProcName;
    .
    .
    .
end ProcName;
```

The NOFRAME and NODISPLAY clauses are examples of procedure *options*. HLA procedures support four such options, specifically RETURNS (See “The HLA RETURNS Option in Procedures” on page 540.), the NOFRAME, NODISPLAY, and NOALIGNSTK. You’ll see the purpose of NOALIGNSTK a little later in this chapter. These procedure options may appear in any order following the procedure name (and parameters, if any). Note that NOFRAME and NODISPLAY (as well as NOALIGNSTK) may only

appear in an actual procedure declaration. You cannot specify these options in an external procedure prototype.

The `NOFRAME` option tells HLA that you don't want the compiler to automatically generate entry and exit code for the procedure. This tells HLA not to automatically generate the `RET` instruction (along with several other instructions).

The `NODISPLAY` option tells HLA that it should not allocate storage in procedure's local variable area for a *display*. The display is a mechanism you use to access non-local VAR objects in a procedure. Therefore, a display is only necessary if you nest procedures in your programs. This chapter will not consider the display or nested procedures; for more details on the display and nested procedures see the chapter on Advanced Procedures in Volume Five. Until then, you can safely specify the `NODISPLAY` option on all your procedures. Note that you may specify the `NODISPLAY` option independently of the `NOFRAME` option. Indeed, for all of the procedures appearing in this text up to this point specifying the `NODISPLAY` option makes a lot of sense because none of those procedures have actually used the display. Procedures that have the `NODISPLAY` option are a tiny bit faster and a tiny bit shorter than those procedures that do not specify this option.

The following is an example of the minimal procedure:

```
procedure minimal; nodisplay; noframe; noalignstk;
begin minimal;

    ret();

end minimal;
```

If you call this procedure with the `CALL` instruction, `minimal` will simply pop the return address off the stack and return back to the caller. You should note that a `RET` instruction is absolutely necessary when you specify the `NOFRAME` procedure option¹. If you fail to put the `RET` instruction in the procedure, the program will not return to the caller upon encountering the "end minimal;" statement. Instead, the program will fall through to whatever code happens to follow the procedure in memory. The following example program demonstrates this problem:

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

// This first procedure has the NOFRAME
// option but does not have a RET instruction.

procedure firstProc; noframe; nodisplay;
begin firstProc;

    stdout.put( "Inside firstProc" nl );

end firstProc;

// Because the procedure above does not have a
// RET instruction, it will "fall through" to
// the following instruction. Note that there
// is no call to this procedure anywhere in
// this program.

procedure secondProc; noframe; nodisplay;
begin secondProc;
```

1. Strictly speaking, this isn't true. But some mechanism that pops the return address off the stack and jumps to the return address is necessary in the procedure's body.

```

        stdout.put( "Inside secondProc" nl );
        ret();

    end secondProc;

begin missingRET;

    // Call the procedure that doesn't have
    // a RET instruction.

    call firstProc;

end missingRET;

```

Program 3.1 Effect of Missing RET Instruction in a Procedure

Although this behavior might be desirable in certain rare circumstances, it usually represents a defect in most programs. Therefore, if you specify the NOFRAME option, always remember to explicitly return from the procedure using the RET instruction.

3.3 Procedures and the Stack

Since procedures use the stack to hold the return address, you must exercise caution when pushing and popping data within a procedure. Consider the following simple (and defective) procedure:

```

procedure MessedUp; noframe; nodisplay;
begin MessedUp;

    push( eax );
    ret();

end MessedUp;

```

At the point the program encounters the RET instruction, the 80x86 stack takes the form shown in Figure 3.1:

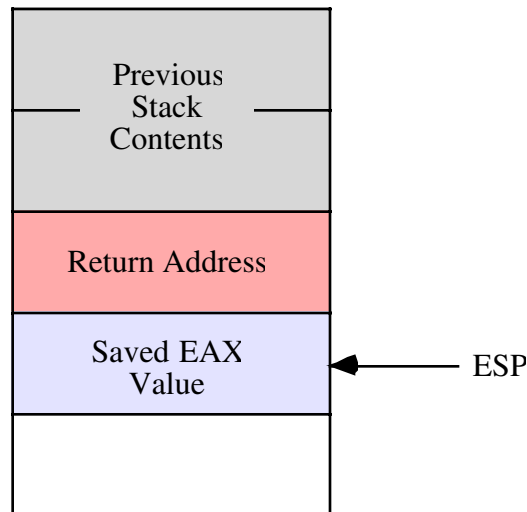


Figure 3.1 **Stack Contents Before RET in “MessedUp” Procedure**

The RET instruction isn’t aware that the value on the top of stack is not a valid address. It simply pops whatever value is on the top of the stack and jumps to that location. In this example, the top of stack contains the saved EAX value. Since it is very unlikely that EAX contains the proper return address (indeed, there is about a one in four billion chance it is correct), this program will probably crash or exhibit some other undefined behavior. Therefore, you must take care when pushing data onto the stack within a procedure that you properly pop that data prior to returning from the procedure.

Note: if you do not specify the NOFRAME option when writing a procedure, HLA automatically generates code at the beginning of the procedure that pushes some data onto the stack. Therefore, unless you understand exactly what is going on and you’ve taken care of this data HLA pushes on the stack, you should never execute the bare RET instruction inside a procedure that does not have the NOFRAME option. Doing so will attempt to return to the location specified by this data (which is not a return address) rather than properly returning to the caller. In procedures that do not have the NOFRAME option, use the EXIT or EXITIF statements to return from the procedure (See “BEGIN..EXIT..EXITIF..END” on page 717.).

Popping extra data off the stack prior to executing the RET statement can also create havoc in your programs. Consider the following defective procedure:

```
procedure MessedUpToo; noframe; nodisplay;
begin MessedUpToo;

    pop( eax );
    ret();

end MessedUpToo;
```

Upon reaching the RET instruction in this procedure, the 80x86 stack looks something like that shown in Figure 3.2:

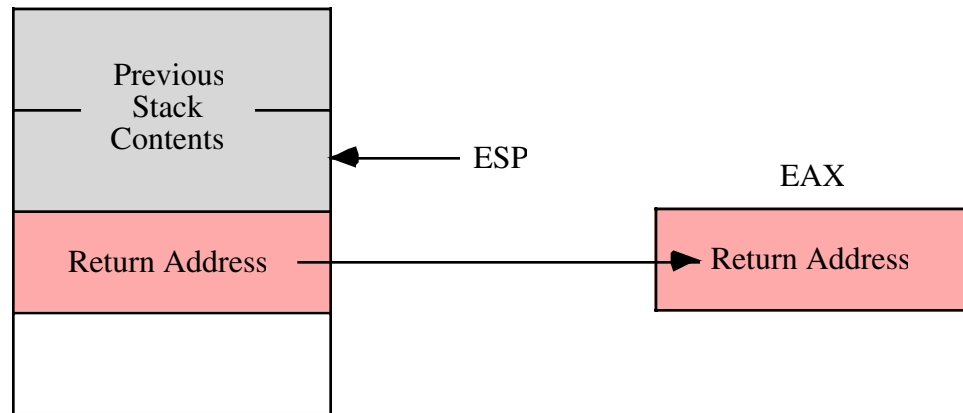


Figure 3.2 **Stack Contents Before RET in MessedUpToo**

Once again, the RET instruction blindly pops whatever data happens to be on the top of the stack and attempts to return to that address. Unlike the previous example, where it was very unlikely that the top of stack contained a valid return address (since it contained the value in EAX), there is a small possibility that the top of stack in this example *does* contain a return address. However, this will not be the proper return address for the *MessedUpToo* procedure; instead, it will be the return address for the procedure that called *MessedUpToo*. To understand the effect of this code, consider the following program:

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

// Note that the following procedure pops
// excess data off the stack (in this case,
// it pops messedUpToo's return address).

procedure messedUpToo; noframe; nodisplay;
begin messedUpToo;

    stdout.put( "Entered messedUpToo" nl );
    pop( eax );
    ret();

end messedUpToo;

procedure callsMU2; noframe; nodisplay;
begin callsMU2;

    stdout.put( "calling messedUpToo" nl );
    messedUpToo();

    // Because messedUpToo pops extra data
    // off the stack, the following code
    // never executes (since the data popped
    // off the stack is the return address that
    // points at the following code.

    stdout.put( "Returned from messedUpToo" nl );
```

```

        ret ();

    end callsMU2;

begin extraPop;

    stdout.put ( "Calling callsMU2" nl );
    callsMU2 ();
    stdout.put ( "Returned from callsMU2" nl );

end extraPop;

```

Program 3.2 Effect of Popping Too Much Data Off the Stack

Since a valid return address is sitting on the top of the stack, you might think that this program will actually work (properly). However, note that when returning from the *MessedUpToo* procedure, this code returns directly to the main program rather than to the proper return address in the *EndSkipped* procedure. Therefore, all code in the *callsMU2* procedure that follows the call to *MessedUpToo* does not execute. When reading the source code, it may be very difficult to figure out why those statements are not executing since they immediately follow the call to the *MessedUpToo* procedure. It isn't clear, unless you look very closely, that the program is popping an extra return address off the stack and, therefore, doesn't return back to *callsMU2* but, rather, returns directly to whomever calls *callsMU2*. Of course, in this example it's fairly easy to see what is going on (because this example is a demonstration of this problem). In real programs, however, determining that a procedure has accidentally popped too much data off the stack can be much more difficult. Therefore, you should always be careful about pushing and popping data in a procedure. You should always verify that there is a one-to-one relationship between the pushes in your procedures and the corresponding pops.

3.4 Activation Records

Whenever you call a procedure there is certain information the program associates with that procedure call. The return address is a good example of some information the program maintains for a specific procedure call. Parameters and automatic local variables (i.e., those you declare in the VAR section) are additional examples of information the program maintains for each procedure call. *Activation record* is the term we'll use to describe the information the program associates with a specific call to a procedure².

Activation record is an appropriate name for this data structure. The program creates an activation record when calling (activating) a procedure and the data in the structure is organized in a manner identical to records (see "Records" on page 465). Perhaps the only thing unusual about an activation record (when comparing it to a standard record) is that the base address of the record is in the middle of the data structure, so you must access fields of the record at positive and negative offsets.

Construction of an activation record begins in the code that calls a procedure. The caller pushes the parameter data (if any) onto the stack. Then the execution of the CALL instruction pushes the return address onto the stack. At this point, construction of the activation record continues in the procedure itself. The procedure pushes registers and other important state information and then makes room in the activation record for local variables. The procedure must also update the EBP register so that it points at the base address of the activation record.

To see what a typical activation record looks like, consider the following HLA procedure declaration:

```

procedure ARDemo( i:uint32; j:int32; k:dword ); nodisplay;
var
    a:int32;

```

2. Stack frame is another term many people use to describe the activation record.

```

r:real32;
c:char;
b:boolean;
w:word;
begin ARDemo;
.
.
.
end ARDemo;

```

Whenever an HLA program calls this *ARDemo* procedure, it begins by pushing the data for the parameters onto the stack. The calling code will push the parameters onto the stack in the order they appear in the parameter list, from left to right. Therefore, the calling code first pushes the value for the *i* parameter, then it pushes the value for the *j* parameter, and it finally pushes the data for the *k* parameter. After pushing the parameters, the program calls the *ARDemo* procedure. Immediately upon entry into the *ARDemo* procedure, the stack contains these four items arranged as shown in Figure 3.3

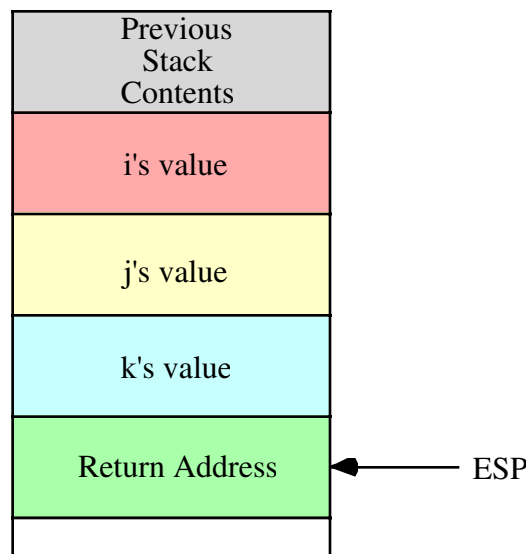


Figure 3.3 **Stack Organization Immediately Upon Entry into ARDemo**

The first few instructions in *ARDemo* (note that it does not have the *NOFRAME* option) will push the current value of *EBP* onto the stack and then copy the value of *ESP* into *EBP*. Next, the code drops the stack pointer down in memory to make room for the local variables. This produces the stack organization shown in Figure 3.4

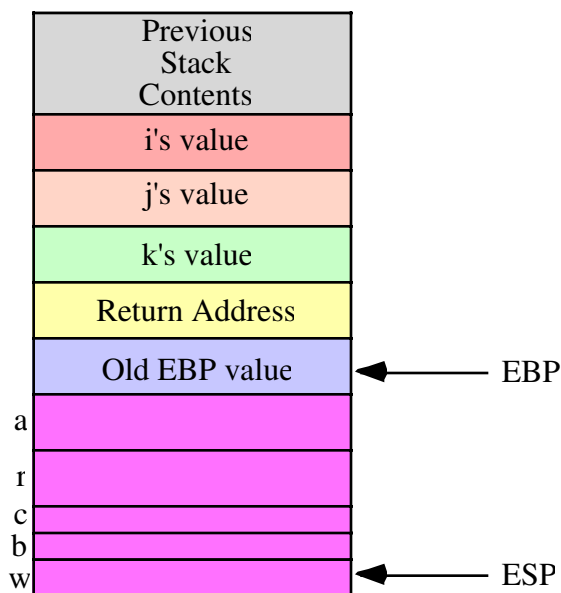


Figure 3.4 **Activation Record for ARDemo**

To access objects in the activation record you must use offsets from the EBP register to the desired object. The two items of immediate interest to you are the parameters and the local variables. You can access the parameters at positive offsets from the EBP register, you can access the local variables at negative offsets from the EBP register as Figure 3.5 shows:

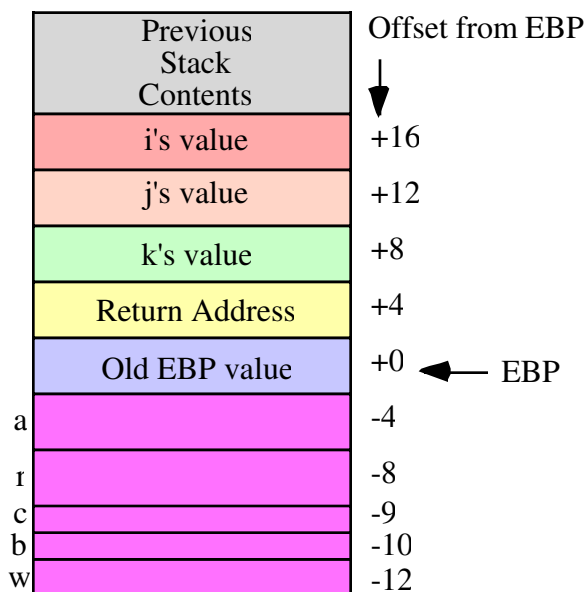


Figure 3.5 **Offsets of Objects in the ARDemo Activation Record**

Intel specifically reserves the EBP (extended base pointer) for use as a pointer to the base of the activation record. This is why you should never use the EBP register for general calculations. If you arbitrarily

change the value in the EBP register you will lose access to the current procedure's parameters and local variables.

3.5 The Standard Entry Sequence

The caller of a procedure is responsible for pushing the parameters onto the stack. Of course, the CALL instruction pushes the return address onto the stack. It is the procedure's responsibility to construct the rest of the activation record. This is typically accomplished by the following "standard entry sequence" code:

```
push( ebp );      // Save copy of old EBP value
mov( esp, ebp );  // Get ptr to base of activation record into EBP
sub( NumVars, esp ); // Allocate storage for local variables.
```

If the procedure doesn't have any local variables, the third instruction above, "sub(NumVars, esp);" isn't needed. *NumVars* represents the number of bytes of local variables needed by the procedure. This is a constant that should be an even multiple of four (so the ESP register remains aligned on a double word boundary). If the number of bytes of local variables in the procedure is not an even multiple of four, you should round the value up to the next higher multiple of four before subtracting this constant from ESP. Doing so will slightly increase the amount of storage the procedure uses for local variables but will not otherwise affect the operation of the procedure.

Warning: if the *NumVars* constant is not an even multiple of four, subtracting this value from ESP (which, presumably, contains a dword-aligned pointer) will virtually guarantee that all future stack accesses are misaligned since the program almost always pushes and pops dword values. This will have a very negative performance impact on the program. Worse still, many Windows API calls will fail if the stack is not dword-aligned upon entry into the Windows subsystem. Therefore, you must always ensure that your local variable allocation value is an even multiple of four.

Because of the problems with a misaligned stack, by default HLA will also emit a fourth instruction as part of the standard entry sequence. The HLA compiler actually emits the following standard entry sequence for the ARDemo procedure defined earlier:

```
push( ebp );
mov( esp, ebp );
sub( 12, esp );      // Make room for ARDemo's local variables.
and( $FFFF_FFC, esp ); // Force dword stack alignment.
```

The AND instruction at the end of this sequence forces the stack to be aligned on a four-byte boundary (it reduces the value in the stack pointer by one, two, or three if the value in ESP is not an even multiple of four). Although the *ARDemo* entry code correctly subtracts 12 from ESP for the local variables (12 is both an even multiple of four and the number of bytes of local variables), this only leaves ESP double word aligned if it was double word aligned immediately upon entry into the procedure. Had the caller messed with the stack and left ESP containing a value that was not an even multiple of four, subtracting 12 from ESP would leave ESP containing an unaligned value. The AND instruction in the sequence above, however, guarantees that ESP is dword aligned regardless of ESP's value upon entry into the procedure. The few bytes and CPU cycles needed to execute this instruction pay off handsomely if ESP is not double word aligned.

Although it is always safe to execute the AND instruction in the standard entry sequence, it might not be necessary. If you always ensure that ESP contains a double word aligned value, the AND instruction in the standard entry sequence above is unnecessary. Therefore, if you've specified the NOFRAME procedure option, you don't have to include that instruction as part of the entry sequence.

If you haven't specified the NOFRAME option (i.e., you're letting HLA emit the instructions to construct the standard entry sequence for you), you can still tell HLA not to emit the extra AND instruction if you're sure the stack will be dword aligned whenever someone calls the procedure. To do this, use the NOALIGNSTK procedure option, e.g.,

```
procedure NASDemo( i:uns32; j:int32; k:dword ); noalignstk;
```

```

var
    LocalVar:int32;
begin NASDemo;
    .
    .
    .
end NASDemo;

```

HLA emits the following entry sequence for the procedure above:

```

push( ebp );
mov( esp, ebp );
sub( 4, esp );

```

3.6 The Standard Exit Sequence

Before a procedure returns to its caller, it needs to clean up the activation record. Although it is possible to share the clean-up duties between the procedure and the procedure's caller, Intel has included some features in the instruction set that allows the procedure to efficiently handle all the clean up chores itself. Standard HLA procedures and procedure calls, therefore, assume that it is the procedure's responsibility to clean up the activation record (including the parameters) when the procedure returns to its caller.

If a procedure does not have any parameters, the calling sequence is very simple. It requires only three instructions:

```

mov( ebp, esp );    // Deallocate locals and clean up stack.
pop( ebp );         // Restore pointer to caller's activation record.
ret();              // Return to the caller.

```

If the procedure has some parameters, then a slight modification to the standard exit sequence is necessary in order to remove the parameter data from the stack. Procedures with parameters use the following standard exit sequence:

```

mov( ebp, esp );    // Deallocate locals and clean up stack.
pop( ebp );         // Restore pointer to caller's activation record.
ret( ParmBytes );   // Return to the caller and pop the parameters.

```

The *ParmBytes* operand of the RET instruction is a constant that specifies the number of *bytes* of parameters to remove from the stack after the return instruction pops the return address. For example, the ARDemo example code in the previous sections has three double word parameters. Therefore, the standard exit sequence would take the following form:

```

mov( ebp, esp );
pop( ebp );
ret( 12 );

```

If you've declared your parameters using HLA syntax (i.e., a parameter list follows the procedure declaration), then HLA automatically creates a local constant in the procedure, *_parms_*, that is equal to the number of bytes of parameters in that procedure. Therefore, rather than worrying about having to count the number of parameter bytes yourself, you can use the following standard exit sequence for any procedure that has parameters:

```

mov( ebp, esp );
pop( ebp );
ret( _parms_ );

```

Note that if you do not specify a byte constant operand to the RET instruction, the 80x86 will not pop the parameters off the stack upon return. Those parameters will still be sitting on the stack when you execute the first instruction following the CALL to the procedure. Similarly, if you specify a value that is too

small, some of the parameters will be left on the stack upon return from the procedure. If the RET operand you specify is too large, the RET instruction will actually pop some of the caller's data off the stack, usually with disastrous consequences.

Note that if you wish to return early from a procedure that doesn't have the NOFRAME option, and you don't particularly want to use the EXIT or EXITIF statement, you must execute the standard exit sequence to return to the caller. A simple RET instruction is insufficient since local variables and the old EBP value are probably sitting on the top of the stack.

3.7 HLA Local Variables

Your program accesses local variables in a procedure by using negative offsets from the activation record base address (EBP). For example, consider the following HLA procedure (which admittedly, doesn't do much other than demonstrate the use of local variables):

```
procedure LocalVars; nodisplay;
var
    a:int32;
    b:int32;
begin LocalVars;

    mov( 0, a );
    mov( a, eax );
    mov( eax, b );

end LocalVars;
```

The activation record for LocalVars looks like

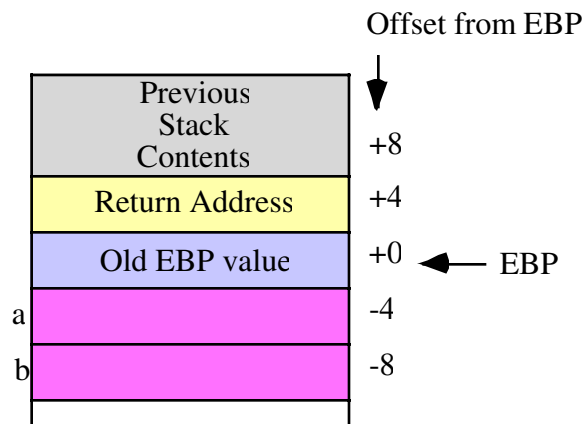


Figure 3.6 Activation Record for LocalVars Procedure

The HLA compiler emits code that is roughly equivalent to the following for the body of this procedure³:

```
mov( 0, (type dword [ebp-4]));
mov( [ebp-4], eax );
mov( eax, [ebp-8] );
```

3. Ignoring the code associated with the standard entry and exit sequences.

You could actually type these statements into the procedure yourself and they would work. Of course, using memory references like “[ebp-4]” and “[ebp-8]” rather than *a* or *b* makes your programs very difficult to read and understand. Therefore, you should always declare and use HLA symbol names rather than offsets from EBP.

The standard entry sequence for this *LocalVars* procedure will be⁴

```
push( ebp );
mov( esp, ebp );
sub( 8, esp );
```

This code subtracts eight from the stack pointer because there are eight bytes of local variables (two *dword* objects) in this procedure. Unfortunately, as the number of local variables increases, especially if those variables have different types, computing the number of bytes of local variables becomes rather tedious. Fortunately, for those who wish to write the standard entry sequence themselves, HLA automatically computes this value for you and creates a constant, `_vars_`, that specifies the number of bytes of local variables for you⁵. Therefore, if you intend to write the standard entry sequence yourself, you should use the `_vars_` constant in the SUB instruction when allocating storage for the local variables:

```
push( ebp );
mov( esp, ebp );
sub( _vars_, esp );
```

Now that you’ve seen how assembly language (and, indeed, most languages) allocate and deallocate storage for local variables, it’s easy to understand why automatic (local VAR) variables do not maintain their values between two calls to the same procedure. Since the memory associated with these automatic variables is on the stack, when a procedure returns to its caller the caller can push other data onto the stack obliterating the values of the local variable values previously held on the stack. Furthermore, intervening calls to other procedures (with their own local variables) may wipe out the values on the stack. Also, upon reentry into a procedure, the procedure’s local variables may correspond to different physical memory locations, hence the values of the local variables would not be in their proper locations.

One big advantage to automatic storage is that it efficiently shares a fixed pool of memory among several procedures. For example, if you call three procedures in a row,

```
ProcA();
ProcB();
ProcC();
```

The first procedure (*ProcA* in the code above) allocates its local variables on the stack. Upon return, *ProcA* deallocates that stack storage. Upon entry into *ProcB*, the program allocates storage for *ProcB*’s local variables *using the same memory locations just freed by ProcA*. Likewise, when *ProcB* returns and the program calls *ProcC*, *ProcC* uses the same stack space for its local variables that *ProcB* recently freed up. This memory reuse makes efficient use of the system resources and is probably the greatest advantage to using automatic (VAR) variables.

3.8 Parameters

Although there is a large class of procedures that are totally self-contained, most procedures require some input data and return some data to the caller. Parameters are values that you pass to and from a procedure. There are many facets to parameters. Questions concerning parameters include:

- *where* is the data coming from?
- *what* mechanism do you use to pass and return data?
- *how* much data are you passing?

4. This code assumes that ESP is *dword* aligned upon entry so the “AND(\$FFFF_FFFC, ESP);” instruction is unnecessary.

5. HLA even rounds this constant up to the next even multiple of four so you don’t have to worry about stack alignment.

In this chapter we will look at the two most common parameter passing mechanisms: pass by value and pass by reference. We will also discuss three popular places to pass parameters: in the registers, on the stack, and in the code stream. The amount of parameter data has a direct bearing on where and how to pass it. The following sections take up these issues.

3.8.1 Pass by Value

A parameter passed by value is just that – the caller passes a value to the procedure. Pass by value parameters are input only parameters. That is, you can pass them to a procedure but the procedure cannot return values through them. In high level languages the idea of a pass by value parameter being an input only parameter makes a lot of sense. Given the Pascal or C/C++ procedure call:

```
CallProc(I);
```

If you pass *I* by value, *CallProc* does not change the value of *I*, regardless of what happens to the parameter inside *CallProc*.

Since you must pass a copy of the data to the procedure, you should only use this method for passing small objects like bytes, words, and double words. Passing arrays and strings by value is very inefficient (since you must create and pass a copy of the structure to the procedure).

3.8.2 Pass by Reference

To pass a parameter by reference you must pass the address of a variable rather than its value. In other words, you must pass a pointer to the data. The procedure must dereference this pointer to access the data. Passing parameters by reference is useful when you must modify the actual parameter or when you pass large data structures between procedures.

Passing parameters by reference can produce some peculiar results. The following Pascal procedure provides an example of one problem you might encounter:

```
program main(input,output);
var m:integer;

(*
** Note: this procedure passes i and j by reference.
*)

procedure bletch(var i,j:integer);
begin
    i := i+2;
    j := j-i;
    writeln(i,' ',j);
end;

.
.
.

begin {main}
    m := 5;
    bletch(m,m);
end.
```

This particular code sequence will print “00” regardless of *m*’s value. This is because the parameters *i* and *j* are pointers to the actual data and they both point at the same object (that is, they are aliases). Therefore, the statement “j:=j-i;” always produces zero since *i* and *j* refer to the same variable.

Pass by reference is usually less efficient than pass by value. You must dereference all pass by reference parameters on each access; this is slower than simply using a value. However, when passing a large data structure, pass by reference is faster because you do not have to copy a large data structure before calling the procedure.

3.8.3 Passing Parameters in Registers

Having touched on how to pass parameters to a procedure, the next thing to discuss is where to pass parameters. Where you pass parameters depends on the size and number of those parameters. If you are passing a small number of bytes to a procedure, then the registers are an excellent place to pass parameters to a procedure. If you are passing a single parameter to a procedure you should use the following registers for the accompanying data types:

Data Size	Pass in this Register
Byte:	al
Word:	ax
Double Word:	eax
Quad Word:	edx:eax

This is not a hard and fast rule. If you find it more convenient to pass 16 bit values in the SI or BX register, do so. However, most programmers use the registers above to pass parameters.

If you are passing several parameters to a procedure in the 80x86's registers, you should probably use up the registers in the following order:

First	Last
eax, edx, esi, edi, ebx, ecx	

In general, you should avoid using EBP register. If you need more than six double words, perhaps you should pass your values elsewhere.

As an example, consider the following “strfill(str,c);” that copies the character *c* (passed by value in AL) to each character position in *s* (passed by reference in EDI) up to a zero terminating byte:

```
// strfill- Overwrites the data in a string with a character.
//
// EDI- pointer to zero terminated string (e.g., an HLA string)
// AL- character to store into the string.

procedure strfill; nodisplay;
begin strfill;

    push( edi ); // Preserve this because it will be modified.
    while( (type char [edi] <> #0 ) do

        mov( al, [edi] );
        inc( edi );

    endwhile;
    pop( edi );

end strfill;
```

To call the *strfill* procedure you would load the address of the string data into EDI and the character value into AL prior to the call. The following code fragment demonstrates a typical call to *strfill*:

```
mov( s, edi ); // Get ptr to string data into edi (assumes s:string).
mov( ' ', al );
strfill();
```

Don't forget that HLA string variables are pointers. This example assumes that *s* is a HLA string variable and, therefore, contains a pointer to a zero-terminated string. Therefore, the “mov(*s*, edi);” instruction loads the address of the zero terminated string into the EDI register (hence this code passes the address of the string data to *strfill*, that is, it passes the string by reference).

One way to pass parameters in the registers is to simply load the registers with the appropriate values prior to a call and then reference the values in those registers within the procedure. This is the traditional mechanism for passing parameters in registers in an assembly language program. HLA, being somewhat more high level than traditional assembly language, provides a formal parameter declaration syntax that lets you tell HLA you're passing certain parameters in the general purpose registers. This declaration syntax is the following:

```
parmName: parmType in reg
```

Where *parmName* is the parameter's name, *parmType* is the type of the object, and *reg* is one of the 80x86's general purpose eight, sixteen, or thirty-two bit registers. The size of the parameter's type must be equal to the size of the register or HLA will generate an error. Here is a concrete example:

```
procedure HasRegParms( count: uns32 in ecx; charVal:char in al );
```

One nice feature to this syntax is that you can call a procedure that has register parameters exactly like any other procedure in HLA using the high level syntax, e.g.,

```
HasRegParms( ecx, bl );
```

If you specify the same register as an actual parameter that you've declared for the formal parameter, HLA does not emit any extra code; it assumes that the parameter is already in the appropriate register. For example, in the call above the first actual parameter is the value in ECX; since the procedure's declaration specifies that that first parameter is in ECX HLA will not emit any code. On the other hand, the second actual parameter is in BL while the procedure will expect this parameter value in AL. Therefore, HLA will emit a “mov(bl, al);” instruction prior to calling the procedure so that the value is in the proper register upon entry to the procedure.

You can also pass parameters by reference in a register. Consider the following declaration:

```
procedure HasRefRegParm( var myPtr:uns32 in edi );
```

A call to this procedure always requires some memory operand as the actual parameter. HLA will emit the code to load the address of that memory object into the parameter's register (EDI in this case). Note that when passing reference parameters, the register must be a 32-bit general purpose register since addresses are 32-bits long. Here's an example of a call to *HasRefRegParm*:

```
HasRefRegParm( x );
```

HLA will emit either a “mov(&x, edi);” or “lea(edi, x);” instruction to load the address of *x* into the EDI registers prior to the CALL instruction⁶.

If you pass an anonymous memory object (e.g., “[edi]” or “[ecx]”) as a parameter to *HasRefRegParm*, HLA will not emit any code if the memory reference uses the same register that you declare for the parameter (i.e., “[edi]”). It will use a simple MOV instruction to copy the actual address into EDI if you specify an indirect addressing mode using a register other than EDI (e.g., “[ecx]”). It will use an LEA instruction to compute the effective address of the anonymous memory operand if you use a more complex addressing mode like “[edi+ecx*4+2]”.

Within the procedure's code, HLA creates text equates for these register parameters that map their names to the appropriate register. In the *HasRegParms* example, any time you reference the *count* parameter, HLA substitutes “ecx” for *count*. Likewise, HLA substitutes “al” for *charVal* throughout the procedure's body. Since these names are aliases for the registers, you should take care to always remember that you cannot use ECX and AL independently of these parameters. It would be a good idea to place a comment next to

6. The choice of instructions is dictated by whether *x* is a static variable (MOV for static objects, LEA for other objects).

each use of these parameters to remind the reader that *count* is equivalent to ECX and *charVal* is equivalent to AL.

3.8.4 Passing Parameters in the Code Stream

Another place where you can pass parameters is in the code stream immediately after the CALL instruction. Consider the following *print* routine that prints a literal string constant to the standard output device:

```
call print;
byte "This parameter is in the code stream.",0
```

Normally, a subroutine returns control to the first instruction immediately following the CALL instruction. Were that to happen here, the 80x86 would attempt to interpret the ASCII code for “This...” as an instruction. This would produce undesirable results. Fortunately, you can skip over this string when returning from the subroutine.

So how do you gain access to these parameters? Easy. The return address on the stack points at them. Consider the following implementation of *print*:

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

// print-
//
// This procedure writes the literal string
// immediately following the call to the
// standard output device. The literal string
// must be a sequence of characters ending with
// a zero byte (i.e., a C string, not an HLA
// string).

procedure print; noframe; nodisplay;
const

    // RtnAdrs is the offset of this procedure's
    // return address in the activation record.

    RtnAdrs:text := "(type dword [ebp+4])";

begin print;

    // Build the activation record (note the
    // "noframe" option above).

    push( ebp );
    mov( esp, ebp );

    // Preserve the registers this function uses.

    push( eax );
    push( ebx );

    // Copy the return address into the EBX
    // register. Since the return address points
    // at the start of the string to print, this
    // instruction loads EBX with the address of
    // the string to print.

    mov( RtnAdrs, ebx );
```

```

// Until we encounter a zero byte, print the
// characters in the string.

forever

    mov( [ebx], al ); // Get the next character.
    breakif( !al ); // Quit if it's zero.
    stdout.putc( al ); // Print it.
    inc( ebx ); // Move on to the next char.

endfor;

// Skip past the zero byte and store the resulting
// address over the top of the return address so
// we'll return to the location that is one byte
// beyond the zero terminating byte of the string.

inc( ebx );
mov( ebx, RtnAdrs );

// Restore EAX and EBX.

pop( ebx );
pop( eax );

// Clean up the activation record and return.

pop( ebp );
ret();

end print;

begin printDemo;

// Simple test of the print procedure.

call print;
byte "Hello World!", 13, 10, 0 ;

end printDemo;

```

Program 3.3 Print Procedure Implementation (Using Code Stream Parameters)

Besides showing how to pass parameters in the code stream, the *print* routine also exhibits another concept: *variable length parameters*. The string following the CALL can be any practical length. The zero terminating byte marks the end of the parameter list. There are two easy ways to handle variable length parameters. Either use some special terminating value (like zero) or you can pass a special length value that tells the subroutine how many parameters you are passing. Both methods have their advantages and disadvantages. Using a special value to terminate a parameter list requires that you choose a value that never appears in the list. For example, *print* uses zero as the terminating value, so it cannot print the NUL character (whose ASCII code is zero). Sometimes this isn't a limitation. Specifying a special length parameter is another mechanism you can use to pass a variable length parameter list. While this doesn't require any special codes or limit the range of possible values that can be passed to a subroutine, setting up the length parameter and maintaining the resulting code can be a real nightmare⁷.

Despite the convenience afforded by passing parameters in the code stream, there are some disadvantages to passing parameters there. First, if you fail to provide the exact number of parameters the procedure requires, the subroutine will get very confused. Consider the *print* example. It prints a string of characters up to a zero terminating byte and then returns control to the first instruction following the zero terminating byte. If you leave off the zero terminating byte, the *print* routine happily prints the following opcode bytes as ASCII characters until it finds a zero byte. Since zero bytes often appear in the middle of an instruction, the *print* routine might return control into the middle of some other instruction. This will probably crash the machine. Inserting an extra zero, which occurs more often than you might think, is another problem programmers have with the *print* routine. In such a case, the *print* routine would return upon encountering the first zero byte and attempt to execute the following ASCII characters as machine code. Once again, this usually crashes the machine. These are the some of the reasons why the HLA *stdout.put* code does *not* pass its parameters in the code stream. Problems notwithstanding, however, the code stream is an efficient place to pass parameters whose values do not change.

3.8.5 Passing Parameters on the Stack

Most high level languages use the stack to pass parameters because this method is fairly efficient. By default, HLA also passes parameters on the stack. Although passing parameters on the stack is slightly less efficient than passing those parameters in registers, the register set is very limited and you can only pass a few value or reference parameters through registers. The stack, on the other hand, allows you to pass a large amount of parameter data without any difficulty. This is the principal reason that most programs pass their parameters on the stack.

HLA passes parameters you specify in a high-level language form on the stack. For example, suppose you define *strfill* from the previous section as follows:

```
procedure strfill( s:string; chr:char );
```

Calls of the form “*strfill*(*s*, ‘ ’);” will pass the value of *s* (which is an address) and a space character on the 80x86 stack. When you specify a call to *strfill* in this manner, HLA automatically pushes the parameters for you, so you don’t have to push them onto the stack yourself. Of course, if you choose to do so, HLA will let you manually push the parameters onto the stack prior to the call.

To manually pass parameters on the stack, push them immediately before calling the subroutine. The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following HLA procedure call:

```
CallProc( i, j, k );
```

HLA pushes parameters onto the stack in the order that they appear in the parameter list. Therefore, the 80x86 code HLA emits for this subroutine call (assuming you’re passing the parameters by value) is

```
push( i );
push( j );
push( k );
call CallProc;
```

Upon entry into *CallProc*, the 80x86’s stack looks like that shown in Figure 3.7:

7. Especially if the parameter list changes frequently.

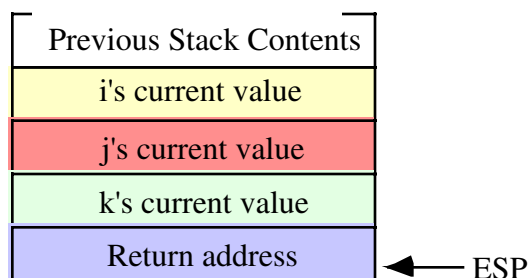


Figure 3.7 Stack Layout Upon Entry into CallProc

You could gain access to the parameters passed on the stack by removing the data from the stack as the following code fragment demonstrates:

```
// Note: to extract parameters off the stack by popping it is very important
// to specify both the nodisplay and noframe procedure options.

static
    RtnAdrs: dword;
    p1Parm: dword;
    p2Parm: dword;
    p3Parm: dword;

procedure CallProc( p1:dword; p2:dword; p3:dword ); nodisplay; noframe;
begin CallProc;

    pop( RtnAdrs );
    pop( p3Parm );
    pop( p2Parm );
    pop( p1Parm );
    push( RtnAdrs );
    .
    .
    .
    ret();

end CallProc;
```

As you can see from this code, it first pops the return address off the stack and into the *RtnAdrs* variable; then it pops (in reverse order) the values of the *p1*, *p2*, and *p3* parameters; finally, it pushes the return address back onto the stack (so the RET instruction will operate properly). Within the *CallProc* procedure, you may access the *p1Parm*, *p2Parm*, and *p3Parm* variables to use the *p1*, *p2*, and *p3* parameter values.

There is, however, a better way to access procedure parameters. If your procedure includes the standard entry and exit sequences (see “The Standard Entry Sequence” on page 787 and “The Standard Exit Sequence” on page 788), then you may directly access the parameter values in the activation record by indexing off the EBP register. Consider the layout of the activation record for *CallProc* that uses the following declaration:

```
procedure CallProc( p1:dword; p2:dword; p3:dword ); nodisplay; noframe;
begin CallProc;

    push( ebp );      // This is the standard entry sequence.
    mov( esp, ebp ); // Get base address of A.R. into EBP.
```

.

.

.

Take a look at the stack immediately after the execution of “mov(esp, ebp);” in *CallProc*. Assuming you’ve pushed three double word parameters onto the stack, it should look something like shown in Figure 3.8:

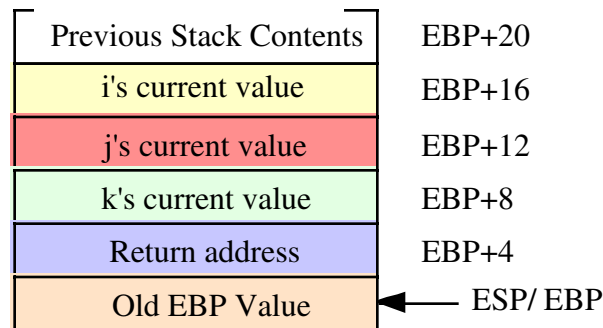


Figure 3.8 **Activation Record for CallProc After Standard Entry Sequence Execution**

.Now you can access the parameters by indexing off the EBP register:

```
mov( [ebp+16], eax );    // Accesses the first parameter.
mov( [ebp+12], ebx );    // Accesses the second parameter.
mov( [ebp+8], ecx );     // Accesses the third parameter.
```

Of course, like local variables, you’d never really access the parameter in this way. You can use the formal parameter names (*p1*, *p2*, and *p3*) and HLA will substitute a suitable “[ebp+displacement]” memory address. Even though you shouldn’t actually access parameters using address expressions like “[ebp+12]” it’s important to understand their relationship to the parameters in your procedures.

Other items that often appear in the activation record are register values your procedure preserves. The most rational place to preserve registers in a procedure is in the code immediately following the standard entry sequence. In a standard HLA procedure (one where you do not specify the NOFRAME option), this simply means that the code that preserves the registers should appear first in the procedure’s body. Likewise, the code to restore those register values should appear immediately before the END clause for the procedure⁸.

3.8.5.1 Accessing Value Parameters on the Stack

Accessing parameters passed by value is no different than accessing a local VAR object. As long as you’ve declared the parameter in a formal parameter list and the procedure executes the standard entry sequence upon entry into the program, all you need do is specify the parameter’s name to reference the value of that parameter. The following is an example program whose procedure accesses a parameter the main program passes to it by value:

```
program AccessingValueParameters;
```

8. Note that if you use the EXIT statement to exit a procedure, you must duplicate the code to pop the register values and place this code immediately before the EXIT clause. This is a good example of a maintenance nightmare and is also a good reason why you should only have one exit point in your program.

```

#include( "stdlib.hhf" )

procedure ValueParm( theParameter: uns32 ); nodisplay;
begin ValueParm;

    mov( theParameter, eax );
    add( 2, eax );
    stdout.put
    (
        "theParameter + 2 = ",
        (type uns32 eax),
        nl
    );

end ValueParm;

begin AccessingValueParameters;

    ValueParm( 10 );
    ValueParm( 135 );

end AccessingValueParameters;

```

Program 3.4 Demonstration of Value Parameters

Although you may access the value of *theParameter* using the anonymous address “[EBP+8]” within your code, there is absolutely no good reason for doing so. If you declare the parameter list using the HLA high level language syntax, you can access the value parameter by specifying its name within the procedure.

3.8.5.2 Passing Value Parameters on the Stack

As Program 3.4 demonstrates, passing a value parameter to a procedure is very easy. Just specify the value in the actual parameter list as you would for a high level language call. Actually, the situation is a little more complicated than this. Passing value parameters is easy if you’re passing constant, register, or variable values. It gets a little more complex if you need to pass the result of some expression. This section deals with the different ways you can pass a parameter by value to a procedure.

Of course, you do not have to use the HLA high level language syntax to pass value parameters to a procedure. You can push these values on the stack yourself. Since there are many times it is more convenient or more efficient to manually pass the parameters, describing how to do this is a good place to start.

As noted earlier in this chapter, when passing parameters on the stack you push the objects in the order they appear in the formal parameter list (from left to right). When passing parameters by value, you should push the values of the actual parameters onto the stack. The following program demonstrates how to do this:

```

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

procedure ThreeValueParms( p1:uns32; p2:uns32; p3:uns32 ); nodisplay;
begin ThreeValueParms;

    mov( p1, eax );
    add( p2, eax );
    add( p3, eax );
    stdout.put

```

```

    (
        "p1 + p2 + p3 = ",
        (type uns32 eax),
        nl
    );

    end ThreeValueParms;

static
    SecondParmValue:uns32 := 25;

begin ManuallyPassingValueParameters;

    pushd( 10 );           // Value associated with p1.
    pushd( SecondParmValue); // Value associated with p2.
    pushd( 15 );           // Value associated with p3.
    call ThreeValueParms;

end ManuallyPassingValueParameters;

```

Program 3.5 Manually Passing Parameters on the Stack

Note that if you manually push the parameters onto the stack as this example does, you must use the CALL instruction to call the procedure. If you attempt to use a procedure invocation of the form “ThreeValueParms();” then HLA will complain about a mismatched parameter list. HLA won’t realize that you’ve manually pushed the parameters (as far as HLA is concerned, those pushes appear to preserve some other data).

Generally, there is little reason to manually push a parameter onto the stack if the actual parameter is a constant, a register value, or a variable. HLA’s high level syntax handles most such parameters for you. There are several instances, however, where HLA’s high level syntax won’t work. The first such example is passing the result of an arithmetic expression as a value parameter. Since arithmetic expressions don’t exist in HLA, you will have to manually compute the result of the expression and pass that value yourself. There are two possible ways to do this: calculate the result of the expression and manually push that result onto the stack, or compute the result of the expression into a register and pass the register as a parameter to the procedure. Program 3.6 demonstrates these two mechanisms.

```

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

    procedure ExprParm( exprValue:uns32 ); nodisplay;
    begin ExprParm;

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

    end ExprParm;

static
    Operand1: uns32 := 5;
    Operand2: uns32 := 20;

begin PassingExpressions;

    // ExprParm( Operand1 + Operand2 );
    //

```

```

// Method one: Compute the sum and manually
// push the sum onto the stack. Then use the
// CALL instruction to call the procedure.

mov( Operand1, eax );
add( Operand2, eax );
push( eax );
call ExprParm;

// Method two: Compute the sum in a register and
// pass the register using the HLA high level
// language syntax.

mov( Operand1, eax );
add( Operand2, eax );
ExprParm( eax );

end PassingExpressions;

```

Program 3.6 Passing the Result of Some Arithmetic Expression as a Parameter

The examples up to this point in this section have made an important assumption: that the parameter you are passing is a double word value. The calling sequence changes somewhat if you're passing parameters that are not four-byte objects. Because HLA can generate relatively inefficient code when passing objects that are not four-bytes long, manually passing such objects is a good idea if you want to have the fastest possible code.

HLA requires that all value parameters be an even multiple of four bytes long⁹. If you pass an object that is less than four bytes long, HLA requires that you *pad* the parameter data with extra bytes so that you always pass an object that is at least four bytes in length. For parameters that are larger than four bytes, you must ensure that you pass an even multiple of four bytes as the parameter value, adding extra bytes at the high-order end of the object to pad it, as necessary.

Consider the following procedure prototype:

```
procedure OneByteParm( b:byte );
```

The activation record for this procedure looks like the following:

9. This only applies if you use the HLA high level language syntax to declare and access parameters in your procedures. Of course, if you manually push the parameters yourself and you access the parameters inside the procedure using an addressing mode like “[ebp+8]” then you can pass any sized object you choose.

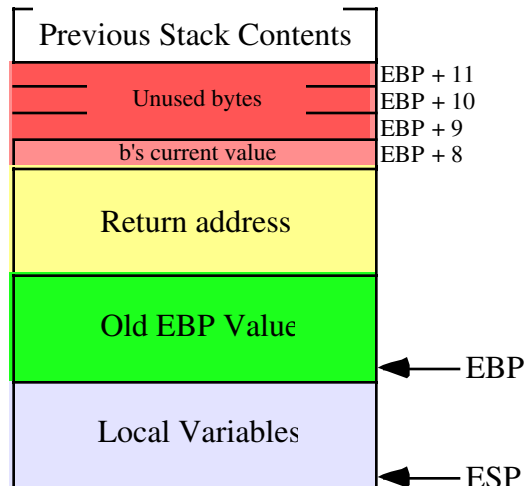


Figure 3.9 OneByteParm Activation Record

As you can see, there are four bytes on the stack associated with the `b` parameter, but only one of the four bytes contains valid data (the L.O. byte). The remaining three bytes are just padding and the procedure should ignore these bytes. In particular, you should never assume that these extra bytes contain zeros or some other consistent value. Depending on the type of parameter you pass, HLA's automatic code generation may or may not push zero bytes as the extra data on the stack.

When passing a byte parameter to a procedure, HLA will automatically emit code that pushes four bytes on the stack. Because HLA's parameter passing mechanism guarantees not to disturb any register or other values, HLA often generates more code than is actually needed to pass a byte parameter. For example, if you decide to pass the `AL` register as the byte parameter, HLA will emit code that pushes the `EAX` register onto the stack. This single push instruction is a very efficient way to pass `AL` as a four-byte parameter object. On the other hand, if you decide to pass the `AH` register as the byte parameter, pushing `EAX` won't work because this would leave the value in `AH` at offset `EBP+9` in the activation record shown in Figure 3.9. Unfortunately, the procedure expects this value at offset `EBP+8` so simply pushing `EAX` won't do the job. If you pass `AH`, `BH`, `CH`, or `DH` as a byte parameter, HLA emits code like the following:

```
sub( 4, esp );    // Make room for the parameter on the stack.
mov( ah, [esp] ); // Store AH into the L.O. byte of the parameter.
```

As you can clearly see, passing one of the "H" registers as a byte parameter is less efficient (two instructions) than passing one of the "L" registers. So you should attempt to use the "L" registers whenever possible if passing an eight-bit register as a parameter¹⁰. Note, by the way, that there is very little you can do about the difference in efficiency, even if you manually pass the parameters yourself.

If the byte parameter you decide to pass is a variable rather than a register, HLA generates decidedly worse code. For example, suppose you call `OneByteParm` as follows:

```
OneByteParm( uns8Var );
```

For this call, HLA will emit code similar to the following to push this single byte parameter:

```
push( eax );
push( eax );
mov( uns8Var, al );
mov( al, [esp+4] );
pop( eax );
```

10. Or better yet, pass the parameter directly in the register if you are writing the procedure yourself.

As you can plainly see, this is a lot of code to pass a single byte on the stack! HLA emits this much code because (1) it guarantees not to disturb any registers, and (2) it doesn't know whether *uns8Var* is the last variable in allocated memory. You can generate much better code if you don't have to enforce either of these two constraints.

If you've got a spare 32-bit register laying around (especially one of EAX, EBX, ECX or EDX) then you can pass a byte parameter on the stack using only two instructions. Move (or move with zero/sign extension) the byte value into the register and then push the register onto the stack. For the current call to *OneByteParm*, the calling sequence would look like the following:

```
mov( uns8Var, al );
push( eax );
call OneByteParm;
```

If only ESI or EDI were available, you could use code like this:

```
movzx( uns8Var, esi );
push( esi );
call OneByteParm;
```

Another trick you can use to pass the parameter with only a single push instruction is to coerce the byte variable to a double word object, i.e.,

```
push( (type dword uns8Var) );
call OneByteParm;
```

This last example is very efficient. Note that it pushes the first three bytes of whatever value happens to follow *uns8Var* in memory as the padding bytes. HLA doesn't use this technique because there is a (very tiny) chance that using this scheme will cause the program to fail. If it turns out that the *uns8Var* object is the last byte of a given page in memory and the next page of memory is unreadable, the PUSH instruction will cause a memory access exception. To be on the safe side, the HLA compiler does not use this scheme. However, if you always ensure that the actual parameter you pass in this fashion is not the last variable you declare in a static section, then you can get away with code that uses this technique. Since it is nearly impossible for the byte object to appear at the last accessible address on the stack, it is probably safe to use this technique with VAR objects.

When passing word parameters on the stack you must also ensure that you include padding bytes so that each parameter consumes an even multiple of four bytes. You can use the same techniques we use to pass bytes except, of course, there are two valid bytes of data to pass instead of one. For example, you could use either of the following two schemes to pass a word object *w* to a *OneWordParm* procedure:

```
mov( w, ax );
push( eax );
call OneWordParm;

push( (type dword w) );
call OneWordParm;
```

When passing large objects by value on the stack (e.g., records and arrays), you do not have to ensure that each element or field of the object consumes an even multiple of four bytes; all you need to do is ensure that the entire data structure consumes an even number of bytes on the stack. For example, if you have an array of 10 three-byte elements, the entire array will need two bytes of padding (10*3 is 30 bytes which is not evenly divisible by four, but 10*3 + 2 is 32 which is divisible by four). HLA does a fairly good job of passing large data objects by value to a procedure. For larger objects, you should use the HLA high level language procedure invocation syntax unless you have some special requirements. Of course, if you want efficient operation, you should try to avoid passing large data structures by value.

3.8.5.3 Accessing Reference Parameters on the Stack

Since HLA passes the address of the actual parameters for reference parameters, accessing the reference parameters within a procedure is slightly more difficult than accessing value parameters because you have to dereference the pointers to the reference parameters. Unfortunately, HLA's high level syntax for procedure declarations and invocations does not (and cannot) abstract this detail away for you. You will have to manually dereference these pointers yourself. This section describes how you do this.

Consider the following program:

```

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

    procedure RefParm( var theParameter: uns32 ); nodisplay;
    begin RefParm;

        // Add two directly to the parameter passed by
        // reference to this procedure.

        mov( theParameter, eax );
        add( 2, (type uns32 [eax]) );

        // Fetch the value of the reference parameter
        // and print it's value.

        mov( [eax], eax );
        stdout.put
        (
            "theParameter now equals ",
            (type uns32 eax),
            nl
        );

    end RefParm;

static
    p1: uns32 := 10;
    p2: uns32 := 15;

begin accessingReferenceParameters;

    RefParm( p1 );
    RefParm( p2 );

    stdout.put( "On return, p1=", p1, " and p2=", p2, nl );

end accessingReferenceParameters;

```

Program 3.7 Accessing a Reference Parameter

In this example the `RefParm` procedure has a single pass by reference parameter. Pass by reference parameters are always a pointer to the type specified by the parameter's declaration. Therefore, *theParameter* is actual an object of type "pointer to `uns32`" rather than an `uns32` value. In order to access the value associated with *theParameter*, this code has to load that double word address into a 32-bit register and access the data indirectly. The "`mov(theParameter, eax);`" instruction in the code above fetches this pointer into the

EAX register and then the procedure uses the “[eax]” addressing mode to access the actual value of *theParameter*.

Since this procedure accesses the data of the actual parameter, adding two to this data affects the values of the variables passed to the *RefParm* procedure from the main program. Of course, this should come as no surprise since this is the standard semantics for pass by reference parameters.

As you can see, accessing (small) pass by reference parameters is a little less efficient than accessing value parameters because you need extra instructions to load the address into a 32-bit pointer register. If you access reference parameters frequently, these extra instructions can really begin to add up, reducing the efficiency of your program. Furthermore, it's easy to forget to dereference a reference parameter and use the address of the value instead of the value in your calculations (this is especially true when passing double-word parameters, like the *uns32* parameter in the example above, to your procedures). Therefore, unless you really need to affect the value of the actual parameter, you should use pass by value to pass small objects to a procedure.

Passing large objects, like arrays and records, is where reference parameters become very efficient. When passing these objects by value, the calling code has to make a copy of the actual parameter; if the actual parameter is a large object, the copy process can be very inefficient. Since computing the address of a large object is just as efficient as computing the address of a small scalar object, there is no efficiency loss when passing large objects by reference. Within the procedure you must still dereference the pointer to access the object but the efficiency loss due to indirection is minimal when you contrast this with the cost of copying that large object. The following program demonstrates how to use pass by reference to initialize an array of records:

```

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

const
    NumElements := 64;

type
    Pt: record
        x:uns8;
        y:uns8;
    endrecord;

    Pts: Pt [NumElements];

procedure RefArrayParm( var ptArray: Pts ); nodisplay;
begin RefArrayParm;

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

    mov( ptArray, edx );    // Get address of parameter into EDX.

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

        // For each element of the array, set the "x" field
        // to (ecx div 8) and set the "y" field to (ecx mod 8).

        mov( cl, al );
        shr( 3, al );    // ECX div 8.
        mov( al, (type Pt [edx+ecx*2]).x ); // Note: "Pts" elements are
                                           // two bytes long.

```

```

        mov( cl, al );
        and( %111, al ); // ECX mod 8.
        mov( al, (type Pt [edx+ecx*2]).y );

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

end RefArrayParm;

static
    MyPts: Pts;

begin accessingRefArrayParameters;

    // Initialize the elements of the array.

    RefArrayParm( MyPts );

    // Display the elements of the array.

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

        stdout.put
        (
            "RefArrayParm[",
            (type uns32 ebx):2,
            "].x=",
            MyPts.x[ ebx*2 ],

            "    RefArrayParm[",
            (type uns32 ebx):2,
            "].y=",
            MyPts.y[ ebx*2 ],
            nl
        );

    endfor;

end accessingRefArrayParameters;

```

Program 3.8 Passing an Array of Records by Referencing

As you can see from this example, passing large objects by reference isn't particularly inefficient. Other than tying up the EDX register throughout the *RefArrayParm* procedure plus a single instruction to load EDX with the address of the reference parameter, the *RefArrayParm* procedure doesn't require many more instructions than the same procedure where you would pass the parameter by value.

3.8.5.4 Passing Reference Parameters on the Stack

HLA's high level syntax often makes passing reference parameters a breeze. All you need to do is specify the name of the actual parameter you wish to pass in the procedure's parameter list. HLA will automatically emit some code that will compute the address of the specified actual parameter and push this address

onto the stack. However, like the code HLA emits for value parameters, the code HLA generates to pass the address of the actual parameter on the stack may not be the most efficient that is possible. Therefore, if you want to write fast code, you may want to manually write the code to pass reference parameters to a procedure. This section discusses how to do exactly that.

Whenever you pass a static object as a reference parameter, HLA generates very efficient code to pass the address of that parameter to the procedure. As an example, consider the following code fragment:

```
procedure HasRefParm( var d:dword );
.
.
.
static
    FourBytes:dword;

var
    v: dword;
.
.
.
HasRefParm( FourBytes );
.
.
.
```

For the call to the *HasRefParm* procedure, HLA emits the following instruction sequence:

```
pushd( &FourBytes );
call HasRefParm;
```

You really aren't going to be able to do substantially better than this if you are passing your reference parameters on the stack. So if you're passing static objects as reference parameters, HLA generates fairly good code and you should stick with the high level syntax for the procedure call.

Unfortunately, when passing automatic (VAR) objects or indexed variables as reference parameters, HLA needs to compute the address of the object at run-time. This generally requires the use of the LEA instruction. Unfortunately, the LEA instruction requires the use of a 32-bit register and HLA promises not to disturb the values in any registers when it automatically generates code for you¹¹. Therefore, HLA needs to preserve the value in whatever register it uses when it computes an address via LEA to pass a parameter by reference. The following example shows you the code that HLA actually emits:

```
// Call to the HasRefParm procedure:

    HasRefParm( v );

// HLA actually emits the following code for the above call:

    push( eax );
    push( eax );
    lea( eax, v );
    mov( eax, [esp+4] );
    pop( eax );
    call HasRefParm;
```

As you can see, this is quite a bit of code, especially if you have a 32-bit register available and you don't need to preserve that register's value. Here's a better code sequence given the availability of EAX:

```
    lea( eax, v );
    push( eax );
    call HasRefParm;
```

11. This isn't entirely true. You'll see the exception in the chapter on Classes and Objects.

Remember, when passing an actual parameter by reference, you must compute the address of that object and push the address onto the stack. For simple static objects you can use the address-of operator (“&”) to easily compute the address of the object and push it onto the stack; however, for indexed and automatic objects, you will probably need to use the LEA instruction to compute the address of the object. Here are some examples that demonstrate this using the *HasRefParm* procedure from the previous examples:

```
static
    i:    int32;
    Ary:  int32[16];
    iptr: pointer to int32 := &i;

var
    v:    int32;
    AV:   int32[10];
    vptr: pointer to int32;
    .
    .
    .
    lea( eax, v );
    mov( eax, vptr );
    .
    .
    .
// HasRefParm( i );

    push( &i );           // Simple static object, so just use "&".
    call HasRefParm;

// HasRefParm( Ary[ebx] ); // Pass element of Ary by reference.

    lea( eax, Ary[ ebx*4 ] ); // Must use LEA for indexed addresses.
    push( eax );
    call HasRefParm;

// HasRefParm( *iptr ); -- Pass object pointed at by iptr

    push( iptr );           // Pass address (iptr's value) on stack.
    call HasRefParm;

// HasRefParm( v );

    lea( eax, v );           // Must use LEA to compute the address
    push( eax );             // of automatic vars passed on stack.
    call HasRefParm;

// HasRefParm( AV[ esi ] ); -- Pass element of AV by reference.

    lea( eax, AV[ esi*4 ] ); // Must use LEA to compute address of the
    push( eax );             // desired element.
    call HasRefParm;

// HasRefParm( *vptr ); -- Pass address held by vptr...

    push( vptr );           // Just pass vptr's value as the specified
    call HasRefParm;        // address.
```

3.8.5.5 Passing Formal Parameters as Actual Parameters

The examples in the previous two sections show how to pass static and automatic variables as parameters to a procedure, either by value or by reference. There is one situation that these sections don't handle properly: the case when you are passing a formal parameter in one procedure as an actual parameter to another procedure. The following simple example demonstrates the different cases that can occur for pass by value and pass by reference parameters:

```

procedure p1( val v:dword; var r:dword );
begin p1;
    .
    .
    .
end p1;

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1( r2, v2 );    // (2) Second call to p1.

end p2;
```

In the statement labelled (1) above, procedure *p2* calls procedure *p1* and passes its two formal parameters as parameters to *p1*. Note that this code passes the first parameter of both procedures by value and it passes the second parameter of both procedures by reference. Therefore, in statement (1), the program passes the parameter into *p2* by value and passes it on to *p1* by value; likewise, the program passes *r2* in by reference and it passes the value onto *p2* by reference.

Since *p2*'s caller passes *v2* in by value and *p2* passes this parameter to *p1* by value, all the code needs to do is make a copy of *v2*'s value and pass this on to *p1*. The code to do this is nothing more than a single push instruction, e.g.,

```

push( v2 );
<< code to handle r2 >>
call p1;
```

As you can see, this code is identical to passing an automatic variable by value. *Indeed, it turns out that the code you need to write to pass a value parameter to another procedure is identical to the code you would write to pass a local, automatic, variable to that other procedure.*

Passing *r2* in statement (1) above requires a little more thought. You do not take the address of *r2* using the LEA instruction as you would a value parameter or an automatic variable. When passing *r2* on through to *p1*, the author of this code probably expects the *r* formal parameter to contain the address of the variable whose address *p2*'s caller passed into *p2*. In plain English, this means that *p2* must pass the address of *r2*'s actual parameter on through to *p1*. Since the *r2* parameter is actually a double word value containing the address of the corresponding actual parameter, this means that the code must pass the dword value of *r2* on to *p1*. The complete code for statement (1) above looks like the following:

```

push( v2 );    // Pass the value passed in through v2 to p1.
push( r2 );    // Pass the address passed in through r2 to p1.
call p1;
```

The important thing to note in this example is that passing a formal reference parameter (*r2*) as an actual reference parameter (*r*) does not involve taking the address of the formal parameter (*r2*). *P2*'s caller has already done this; *p2* need only pass this address on through to *p1*.

In the second call to *p1* in the example above (2), the code swaps the actual parameters so that the call to *p1* passes *r2* by value and *v2* by reference. Specifically, *p1* expects *p2* to pass it the value of the *dword* object associated with *r2*; likewise, it expects *p2* to pass it the address of the value associated with *v2*.

To pass the value of the object associated with *r2*, your code must dereference the pointer associated with *r2* and directly pass the value. Here is the code HLA automatically generates to pass *r2* as the first parameter to *p1* in statement (2):

```
sub( 4, esp );    // Make room on stack for parameter.
push( eax );     // Preserve EAX's value.
mov( r2, eax );  // Get address of object passed in to p2.
mov( [eax], eax ); // Dereference to get the value of this object.
mov( eax, [esp+4] ); // Put value of parameter into its location on stack.
pop( eax );      // Restore original EAX value.
```

As usual, HLA generates a little more code than may be necessary because it won't destroy the value in the EAX register. You can write more efficient code if a register is available to use in this sequence. If EAX is unused, you could trim this down to the following:

```
mov( r2, eax );    // Get the pointer to the actual object.
pushd( [eax] );    // Push the value of the object onto the stack.
```

Since you can treat value parameters exactly like local (automatic) variables, you use the same code to pass *v2* by reference to *p1* as you would to pass a local variable in *p2* to *p1*. Specifically, you use the LEA instruction to compute the address of the value in the *v2*. The code HLA automatically emits for statement (2) above preserves all registers and takes the following form (same as passing an automatic variable by reference):

```
push( eax );      // Make room for the parameter.
push( eax );      // Preserve EAX's value.
lea( eax, v2 );   // Compute address of v2's value.
mov( eax, [esp+4] ); // Store away address as parameter value.
pop( eax );       // Restore EAX's value
```

Of course, if you have a register available, you can improve on this code. Here's the complete code that corresponds to statement (2) above:

```
mov( r2, eax );    // Get the pointer to the actual object.
pushd( [eax] );    // Push the value of the object onto the stack.
lea( eax, v2 );    // Push the address of V2 onto the stack.
push( eax );
call p1;
```

3.8.5.6 HLA Hybrid Parameter Passing Facilities

Like control structures, HLA provides a high level language syntax for procedure calls that is convenient to use and easy to read. However, this high level language syntax is sometimes inefficient and may not provide the capabilities you need (for example, you cannot specify an arithmetic expression as a value parameter as you can in high level languages). HLA lets you overcome these limitations by writing low-level ("pure") assembly language code. Unfortunately, the low-level code is harder to read and maintain than procedure calls that use the high level syntax. Furthermore, it's quite possible that HLA generates perfectly fine code for certain parameters and only one or two parameters present a problem. Fortunately, HLA provides a hybrid syntax for procedure calls that allows you to use both high-level and low-level syntax as appropriate for a given actual parameter. This lets you use the high level syntax where appropriate and then drop down into pure assembly language to pass those special parameters that HLA's high level language syntax cannot handle efficiently (if at all).

Within an actual parameter list (using the high level language syntax), if HLA encounters "{ " followed by a sequence of statements and a closing "}", HLA will substitute the instructions between the braces in place of the code it would normally generate for that parameter. For example, consider the following code fragment:

```

procedure HybridCall( i:uns32; j:uns32 );
begin HybridCall;
    .
    .
    .
end HybridCall;
    .
    .
    .
    HybridCall( 5, #{ mov( i, eax ); add( j, eax ); push( eax ); }# );

```

The call to *HybridCall* immediately above is equivalent to the following “pure” assembly language code:

```

pushd( 5 );
mov( i, eax );
add( j, eax );
push( eax );
call HybridCall;

```

As a second example, consider the example from the previous section:

```

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1( r2, v2 );    // (2) Second call to p1.

end p2;

```

HLA generates exceedingly mediocre code for the second call to *p1* in this example. If efficiency is important in the context of this procedure call, and you have a free register available, you might want to rewrite this code as follows:

```

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1                // (2) Second call to p1.
    (                // This code assumes EAX is free.
        #{
            mov( r2, eax );
            pushd( [eax] );
        }#,
        #{
            lea( eax, v2 );
            push( eax );
        }#
    );

end p2;

```

3.8.5.7 Mixing Register and Stack Based Parameters

You can mix register parameters and standard (stack-based) parameters in the same high level procedure declaration, e.g.,

```
procedure HasBothRegAndStack( var dest:dword in edi; count:un32 );
```

When constructing the activation record, HLA ignores the parameters you pass in registers and only processes those parameters you pass on the stack. Therefore, a call to the *HasBothRegAndStack* procedure will push only a single parameter onto the stack (*count*). It will pass the *dest* parameter in the EDI register. When this procedure returns to its caller, it will only remove four bytes of parameter data from the stack.

3.9 Procedure Pointers

The x86 CALL instruction is very similar to the JMP instruction. In particular, it allows the same three basic forms as the JMP instruction: direct calls (to a procedure name), indirect calls through a 32-bit general purpose register, and indirect calls through a double word pointer variable. The CALL instruction allows the following (low-level) syntax supporting these three types of procedure invocations:

```
call Procname;      // Direct call to procedure "Procname".
call( Reg32 );      // Indirect call to procedure whose address appears
                      // in the Reg32 general-purpose 32-bit register.
call( dwordVar );    // Indirect call to the procedure whose address appears
                      // in the dwordVar double word variable.
```

HLA treats procedure names like static objects. Therefore, you can compute the address of a procedure by using the address-of (“&”) operator along with the procedure’s name or by using the LEA instruction. For example, “&Procname” is the address of the very first instruction of the *Procname* procedure. Therefore, all three of the following code sequences wind up calling the *Procname* procedure:

```
call Procname;
.
.
.
mov( &Procname, eax );
call( eax );
.
.
.
lea( eax, Procname );
call( eax );
```

Since the address of a procedure fits in a 32-bit object, you can store such an address into a *dword* variable; in fact, you can initialize a *dword* variable with the address of a procedure using code like the following:

```
procedure p;
begin p;
end p;
.
.
.
static
ptrToP: dword := &p;
.
.
.
call( ptrToP ); // Calls the “p” procedure if ptrToP has not changed.
```

Because the use of procedure pointers occurs frequently in assembly language programs, HLA provides a special syntax for declaring procedure pointer variables and for calling procedures indirectly through such pointer variables. To declare a procedure pointer in an HLA program, you can use a variable declaration like the following:

```
static
    procPtr: procedure;
```

Note that this syntax uses the keyword `PROCEDURE` as a data type. It follows the variable name and a colon in one of the variable declaration sections (`STATIC`, `READONLY`, `STORAGE`, `DATA`, or `VAR`). This sets aside exactly four bytes of storage for the *procPtr* variable. To call the procedure whose address is held by *procPtr*, you can use either of the following two forms:

```
call( procPtr );    // Low-level syntax.
procPtr();          // High-level language syntax.
```

Note that the high level syntax for an indirect procedure call is identical to the high level syntax for a direct procedure call. HLA can figure out whether to use a direct call or an indirect call by the type of the identifier. If you've specified a variable name, HLA assumes it needs to use an indirect call; if you specify a procedure name, HLA uses a direct call.

Like all pointer objects, you should not attempt to indirectly call a procedure through a pointer variable unless you've initialized that variable with the address appropriately. There are two ways to initialize a procedure pointer variable: `STATIC`, `DATA`, and `READONLY` objects allow an initializer, or you can compute the address of a routine (as a 32-bit value) and store that 32-bit address directly into the procedure pointer at run-time. The following code fragment demonstrates both ways you can initialize a procedure pointer.

```
static
    ProcPtr: procedure := &p;    // Initialize ProcPtr with the address of p.
    .
    .
    .
    ProcPtr();                  // First invocation calls p.

    mov( &q, ProcPtr );        // Reload ProcPtr with the address of q.
    .
    .
    .
    ProcPtr();                  // This invocation calls the "q" procedure.
```

Procedure pointer variable declarations also allow the declaration of parameters. To declare a procedure pointer with parameters, you must use a declaration like the following:

```
static
    p:procedure( i:int32; c:char );
```

This declaration states that *p* is a 32-bit pointer that contains the address of a procedure having two parameters. If desired, you could also initialize this variable *p* with the address of some procedure by using a static initializer, e.g.,

```
static
    p:procedure( i:int32; c:char ) := &SomeProcedure;
```

Note that *SomeProcedure* must be a procedure whose parameter list exactly matches *p*'s parameter list (i.e., two value parameters, the first is an *int32* parameter and the second is a *char* parameter). To indirectly call this procedure, you could use either of the following sequences:

```
push( << Value for i >> );
push( << Value for c >> );
call( p );
-or-
p( <<Value for i>>, <<Value for c>> );
```

The high level language syntax has the same features and restrictions as the high level syntax for a direct procedure call. The only difference is the actual `CALL` instruction HLA emits at the end of the calling sequence.

Although all of the examples in this section have used `STATIC` variable declarations, don't get the idea that you can only declare simple procedure pointers in the `STATIC` or other variable declaration sections. You can declare procedure pointer types in the `TYPE` section. You can declare procedure pointers as fields of a `RECORD`. Assuming you create a type name for a procedure pointer in the `TYPE` section, you can even create arrays of procedure pointers. The following code fragments demonstrate some of the possibilities:

```
type
  pptr:  procedure;
  prec:  record
          p:pptr;
          // other fields...
        endrecord;
static
  p1:pptr;
  p2:pptr[2]
  p3:prec;
  .
  .
  .
  p1();
  p2[ebx*4]();
  p3.p();
```

One very important thing to keep in mind when using procedure pointers is that HLA does not (and cannot) enforce strict type checking on the pointer values you assign to a procedure pointer variable. In particular, if the parameter lists do not agree between the declarations of the pointer variable and the procedure whose address you assign to the pointer variable, the program will probably crash if you attempt to call the mismatched procedure indirectly through the pointer using the high level syntax. Like the low-level "pure" procedure calls, it is your responsibility to ensure that the proper number and types of parameters are on the stack prior to the call.

3.10 Procedural Parameters

One place where procedure pointers are quite invaluable is in parameter lists. Selecting one of several procedures to call by passing the address of some procedure, selected from a set of procedures, is not an uncommon operation. Therefore, HLA lets you declare procedure pointers as parameters.

There is nothing special about a procedure parameter declaration. It looks exactly like a procedure variable declaration except it appears within a parameter list rather than within a variable declaration section. The following are some typical procedure prototypes that demonstrate how to declare such parameters:

```
procedure p1( procparm: procedure ); forward;
procedure p2( procparm: procedure( i:int32 ) ); forward;
procedure p3( val procparm: procedure ); forward;
```

The last example above is identical to the first. It does point out, though, that you generally pass procedural parameters by value. This may seem counter-intuitive since procedure pointers are addresses and you will need to pass an address as the actual parameter; however, a pass by reference procedure parameter means something else entirely. consider the following (legal!) declaration:

```
procedure p4( var procPtr:procedure ); forward;
```

This declaration tells HLA that you are passing a procedure variable *by reference* to `p4`. The address HLA expects must be the address of a procedure pointer variable, not a procedure.

When passing a procedure pointer by value, you may specify either a procedure variable (whose value HLA passes to the actual procedure) or a procedure pointer constant. A procedure pointer constant consists of the address-of operator ("`&`") immediately followed by a procedure name. Passing procedure constants is probably the most convenient way to pass procedural parameters. For example, the following calls to the `Plot` routine might plot out the function passed as a parameter from -2π to $+2\pi$.

```
Plot( &sineFunc );
Plot( &cosFunc );
Plot( &tanFunc );
```

Note that you cannot pass a procedure as a parameter by simply specifying the procedure's name. I.e., "Plot(sineFunc);" will not work. Simply specifying the procedure name doesn't work because HLA will attempt to directly call the procedure whose name you specify (remember, a procedure name inside a parameter list invokes instruction composition). However, since you don't specify a parameter list, or at least an empty pair of parentheses, after the parameter/procedure's name, HLA generates a syntax error message. Moral of the story: don't forget to preface procedure parameter constant names with the address-of operator.

3.11 Untyped Reference Parameters

Sometimes you will want to write a procedure to which you pass a generic memory object by reference without regard to the type of that memory object. A classic example is a procedure that zeros out some data structure. Such a procedure might have the following prototype:

```
procedure ZeroMem( var mem:byte; count:uint32 );
```

This procedure would zero out *count* bytes starting at the address the first parameter specifies. The problem with this procedure prototype is that HLA will complain if you attempt to pass anything other than a byte object as the first parameter. Of course, you can overcome this problem using type coercion like the following, but if you call this procedure several times with lots of different data types, then the following coercion operator is rather tedious to use:

```
ZeroMem( (type byte MyDataObject), @size( MyDataObject ) );
```

Of course, you can always use hybrid parameter passing or manually push the parameters yourself, but these solutions are even more work than using the type coercion operation. Fortunately, HLA provides a far more convenient solution: untyped reference parameters.

Untyped reference parameters are exactly that – pass by reference parameters on which HLA doesn't bother to compare the type of the actual parameter against the type of the formal parameter. With an untyped reference parameter, the call to *ZeroMem* above would take the following form:

```
ZeroMem( MyDataObject, @size( MyDataObject ) );
```

MyDataObject could be any type and multiple calls to *ZeroMem* could pass different typed objects without any objections from HLA.

To declare an untyped reference parameter, you specify the parameter using the normal syntax except that you use the reserved word **VAR** in place of the parameter's type. This **VAR** keyword tells HLA that any variable object is legal for that parameter. Note that you must pass untyped reference parameters by reference, so the **VAR** keyword must precede the parameter's declaration as well. Here's the correct declaration for the *ZeroMem* procedure using an untyped reference parameter:

```
procedure ZeroMem( var mem:var; count:uint32 );
```

With this declaration, HLA will compute the address of whatever memory object you pass as an actual parameter to *ZeroMem* and pass this on the stack.

3.12 Iterators and the FOREACH Loop

One nifty feature HLA provides is support for *true* iterators¹². An iterator is a special type of procedure or function that you use in conjunction with the HLA FOREACH..ENDFOR loop. Combined, these two language features (iterators and the FOREACH..ENDFOR loop) provide a very powerful user-defined looping construct.

The HLA FOREACH..ENDFOR statement uses the following basic syntax:

```
foreach iteratorID( optional_parameters ) do

    << loop body >>

endfor;
```

The FOREACH statement calls the specified iterator. If the iterator *succeeds*, then the FOREACH statement executes the loop body; if the iterator *fails*, then control transfers to the first statement following the ENDFOR clause. On each iteration of the loop body, the program re-enters the iterator code and, once again, the iterator returns success or failure to determine whether to repeat the loop body.

At first glance, you might get the impression that the FOREACH loop is nothing more than a WHILE loop and an iterator is a function that returns true (success) or false (failure). However, this is not an accurate picture of how the FOREACH loop operates. First of all, the FOREACH loop does not CALL the iterator on each iteration of the loop; it *re-enters* the iterator. Specifically, control does not (necessarily) begin with the first statement of the iterator whenever control returns to the top of the FOREACH loop. The second big difference between a FOREACH/iterator loop and a WHILE/function loop is that the iterator procedure maintains its activation record in memory for the duration of the FOREACH loop. A function you would call from a WHILE loop, by contrast, builds and destroys the function's activation record on each iteration of the loop. This means that the iterator's local (automatic) variables maintain their values until the FOREACH loop terminates. This has important ramifications, especially for recursive iterator functions.

An iterator declaration looks very similar to a procedure declaration. Indeed, about the only syntactical difference is the use of the reserved word ITERATOR rather than PROCEDURE. The following is an example of a simple iterator:

```
iterator range( start:uns32; last:uns32 ); nodisplay;
begin range;

    mov( start, eax );
    while( eax <= last ) do

        push( eax );
        yield();
        pop( eax );
        inc( eax );

    endwhile;

end range;
```

The only thing special about this iterator declaration, other than the use of the ITERATOR reserved word, is that it calls a special procedure named *yield*. In a few paragraphs you'll see the purpose of the call to the *yield* procedure.

A typical FOREACH loop that calls the *range* iterator might look like the following:

12. HLA's iterators are based on the control structure by the same name from the CLU programming language. Those things that C/C++ programmers refer to as iterators are more properly called *cursors*. While it is certainly possible to write cursors in HLA, it is important to note that HLA's iterators are quite a bit more powerful than C/C++'s iterators.

```
foreach range( 1, 10 ) do

    stdout.put( "Iteration = ", (type uns32 eax), nl );

endfor;
```

Here's how the iterator and the FOREACH loop work together. Upon first encountering the FOREACH statement, the program makes an *initial call* to the *range* iterator. Except for a few extra parameters HLA pushes on the stack, this call is exactly like a standard procedure call. Upon entry into the iterator, the *start* parameter has the initial value one and the *last* parameter has the initial value ten. The iterator loads *start* into EAX and compares this against the value in *last* (ten). Since EAX's value is less than or equal to ten, the program enters the loop's body. The loop body pushes EAX's value onto the stack and then calls the *yield* procedure. The *yield* procedure transfers control to the body of the FOREACH loop that called the *range* iterator in the first place. Calling *yield* is how the iterator returns success to the FOREACH loop. Within the body of the FOREACH loop, above, the code prints out the value of the EAX register as an unsigned integer. During the first iteration of the loop, EAX contains one so the loop body prints this value.

At the bottom of the FOREACH loop, the program *re-enters* the iterator. When the FOREACH loop re-enters the iterator, it transfers control to the first statement following the call to the *yield* function. Intuitively, you can view the FOREACH loop body as a procedure that the iterator calls whenever you call the *yield* function¹³. Whenever the program encounters the ENDFOR clause, it returns to the iterator, executing the first statement beyond the *yield* call. In the current example, this pops the value of EAX off the stack (preserved before the call to *yield*), the loop increments EAX and repeats as long as EAX is less than ten.

When the *range* iterator increments EAX to 11, the WHILE loop in the iterator terminates and control falls off the bottom of the iterator. This is how an iterator returns failure to the calling FOREACH loop. At that point control transfers to the first statement following the ENDFOR in the FOREACH..ENDFOR loop.

By the way, the *range* iterator, combined with the FOREACH loop above, creates a relatively inefficient implementation of the following loop:

```
for( mov( 1, eax ); eax < 10; inc( eax ) ) do

    stdout.put( "Iteration = ", (type uns32 eax), nl );

endfor;
```

However, don't get the impression from this example that iterators are particularly inefficient. Iterators are not a good choice for something like *range*. However, there are many iterators you can write that are just as efficient as other means of loop control and computation.

An important point to remember when using iterators is that the iterator's activation record remains on the stack as long as the iterator returns success. The program only removes the activation record when the iterator fails. For example, the *range* iterator takes advantage of this fact since it refers to the value of its *last* parameter on each re-entry from the FOREACH loop. The fact that parameters and local (automatic) variables maintain their values for the duration of the FOREACH loop is very important to many algorithms that use iterators, especially recursive algorithms.

One side effect of having an iterator maintain its activation record until it fails is that the value of ESP changes considerably between the statement immediately before the FOREACH statement and the first statement in the body of the FOREACH loop. This is because the program "pushes" the activation record onto the stack upon encountering the FOREACH loop and doesn't "pop" this activation record off the stack until the FOREACH loop fails. Therefore, code like the following will not work as expected:

```
pushd( 10 );
foreach range( 1, 25 ) do
    pop( ebx );
    push( ebx );
    stdout.put( "eax=", eax, " ebx=", ebx, nl );
```

13. In fact, this is exactly how HLA implements iterators and the FOREACH loop. See the chapter on Advanced Procedures for more details.

```
endfor;
pop( ebx );
```

The problem with this code is that the FOREACH loop pushes a whole lot of data onto the stack after the PUSH instruction pushes the value 10 onto the stack. Therefore, the POP instruction inside the loop does not pop the value 10 from the stack. Instead, it pops some data pushed on the stack by the iterator (specifically, it pops the return address that transfers control to the first instruction following the *yield* call). Therefore, you cannot use the stack to transfer data into or out of a FOREACH loop¹⁴.

Another problem with the stack and the FOREACH loop occurs if you try to prematurely exit a FOREACH loop before the iterator returns failure. Whenever an iterator fails, it cleans up the stack and restores ESP to the value it had upon encountering the FOREACH statement. However, statements like BREAK, BREAKIF, EXIT, EXITIF, JMP and any other flow of control transfer instructions will not clean up the stack if they transfer control out of a FOREACH loop. For example, the following code will leave the activation record for the *range* iterator sitting on the stack:

```
foreach range( 2, 5 ) do

    jmp ExitFor;

endifor;
ExitFor:
```

Depending on the iterator and the code that calls the iterator, prematurely exiting a FOREACH loop without having the iterator return failure and leaving this junk sitting on the stack may have an adverse effect on the operation of your program. Clearly if you've pushed data onto the stack prior to the FOREACH loop, you will not be able to pop that data off unless you manually clean up the stack yourself (this involves saving the value of ESP prior to the FOREACH statement and restoring this value at the *ExitFor* label, above). Also, don't forget that prematurely exiting a FOREACH loop without letting the iterator finish may wind up grabbing some system resources that the iterator would normally free just before returning failure (e.g., calling *free* and closing files).

The chapter on Advanced Procedures will go into the details concerning the low-level implementation of iterators. Until then, keep in mind that iterators build their activation records differently than standard procedures. Until you read that chapter, you should not attempt to call an iterator directly (i.e., outside a FOREACH loop) nor should you use the "noframe" option with an iterator. See the chapter on Advanced Procedures for more details on the implementation of iterators.

3.13 Sample Programs

This section presents two sample programs. The first demonstrates the use of iterators using a fibonacci number iterator. The second demonstrates the use of procedural parameters.

3.13.1 Generating the Fibonacci Sequence Using an Iterator

The following program generates the Fibonacci sequence $f_1, f_2, f_3, \dots, f_{\text{count}}$ where *count* is a parameter. This simple example displays all the fibonacci numbers the iterator generates.

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

// Basic (recursive version) algorithm for
// the fibonacci sequence.
```

14. Not that it's a good idea to transfer data into or out of any loop using the stack. Such code tends to have lots of errors due to extra pushes or pops appearing in the program.

```

//
// int fib(int N)
// {
//     if(N<=2)
//         return 1;
//     else
//         return fib(N-1) + fib(N-2)
// }
//
// Iterator (iterative) that computes all the fibonacci
// numbers between fib(1) and fib(count).

iterator fib( count:uint32 ); nodisplay;
var
    lastVal:          uint32;
    BeforeLastVal:    uint32;

begin fib;

    if( count > 0 ) then

        mov( 0, BeforeLastVal );
        mov( 1, eax );
        mov( eax, lastVal );

        // Handle fib(1) as a special case.

        yield();
        dec( count );

        // Okay, handle fib(2)..fib(count) here.

        while( @nz ) do

            // Compute fib(n) = fib(n-1) + fib(n-2).
            // and then copy fib(n-1) {lastVal} to
            // fib(n-2) {BeforeLastVal} and store the
            // current result into lastVal so we'll
            // have the n-1 and n-2 values on the next
            // call.

            mov( lastVal, eax );
            add( BeforeLastVal, eax );
            mov( lastVal, BeforeLastVal );
            mov( eax, lastVal );

            // Yield fib(n) to the FOREACH loop.

            yield();

            // Repeat this iterator the specified number
            // of times.

            dec( count );

        endwhile;

    endif;

end fib;

```

```

static
    iteration:uns32;

begin iterDemo;

    // Display the fibonacci sequence for the first
    // ten fibonacci numbers.

    mov( 1, iteration );
    foreach fib( 10 ) do

        stdout.put( "fib(", iteration, ") = ", (type uns32 eax), nl );
        inc( iteration );

    endfor;

end iterDemo;

```

3.13.2 Outer Product Computation with Procedural Parameters

The following program generates an *addition table*, a *subtraction table*, or a *multiplication table* based on user inputs. These tables are computed using an *outer product* calculation and procedural parameters. An outer product is simply the process of computing all the values for the elements of a matrix by using the row and column indices as inputs to some function (e.g., addition, subtraction, or multiplication).

```

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

static
    size: uns32;
    ftbl: array.dArray( uns32, 2 );

    // GenerateTable-
    //
    // This function computes the "Outer Product". That is,
    // take the cartesian product of the indices into
    // the rows and columns of this array [(0,0), (0,1), ... (0,size-1),
    // (1,0), (1,1), ..., (size-1,size-1)], then feed the left and
    // right values of each coordinate to the "func" procedure passed
    // as a parameter. Whatever result the function returns, store that
    // into element (l,r) of the ftbl array.

    procedure GenerateTable( func:procedure( l:uns32; r:uns32 )); nodisplay;
    begin GenerateTable;

        push( eax );
        push( ebx );
        push( ecx );
        push( edi );

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

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

                array.index( edi, ftbl, ebx, ecx );
                func( ebx, ecx );
            endfor;
        endfor;
    end GenerateTable;

```

```

        mov( eax, [edi] );

    endfor;

endfor;

pop( edi );
pop( ecx );
pop( ebx );
pop( eax );

end GenerateTable;

// The following functions compute the various
// values used to fill the table (obviously,
// "+" = addFunc, "-" = subFunc, and "*" = mulFunc).

procedure addFunc( left:uns32; right:uns32 ); nodisplay;
begin addFunc;

    mov( left, eax );
    add( right, eax );

end addFunc;

procedure subFunc( left:uns32; right:uns32 ); nodisplay;
begin subFunc;

    mov( left, eax );
    sub( right, eax );

end subFunc;

procedure mulFunc( left:uns32; right:uns32 ); nodisplay;
begin mulFunc;

    mov( left, eax );
    intmul( right, eax );

end mulFunc;

begin funcTable;

    stdout.put( "Function table generator: " nl );
    stdout.put( "----- " nl nl );

    // Get the size of the function table from the user:

    forever

        try

            stdout.put( "Enter the size of the matrix: " );
            stdin.getu32();
            bound( eax, 1, 20 );
            unprotected break;

        exception( ex.ConversionError )

```

```

        stdout.put( "Illegal character, re-enter" nl );

    exception( ex.ValueOutOfRange )

        stdout.put( "Value out of range (1..20), please re-enter" nl );

    exception( ex.BoundInstr )

        stdout.put( "Value out of range (1..20), please re-enter" nl );

    endtry;

endfor;

// Allocate storage for the function table:

mov( eax, size );
array.daAlloc( ftbl, size, size );

// Get the function from the user:

stdout.put( "What type of table do you want to generate?" nl nl );
stdout.put( "+" Addition" nl );
stdout.put( "-" Subtraction" nl );
stdout.put( "*" Multiplication" nl );
stdout.newln();
repeat

    stdout.put( "Choice? (+, -, *): " );
    stdin.FlushInput();
    stdin.getc();

until( al in { '+', '-', '*' } );

// Fill in the entries in the table:

if( al = '+' ) then

    GenerateTable( &addFunc );

elseif( al = '-' ) then

    GenerateTable( &subFunc );

elseif( al = '*' ) then

    GenerateTable( &mulFunc );

endif;

// Display the column labels across the top:

stdout.put( nl nl "      " );
for( mov( 0, ebx ); ebx < size; inc( ebx ) ) do

    stdout.put( (type uns32 ebx):5 );

endfor;

```

```

stdout.newln();
stdout.put( "      " );
for( mov( 0, ebx ); ebx < size; inc( ebx ) ) do

    stdout.put( "-----" );

endfor;
stdout.newln();

// Display the row labels and fill in the table.
// Note that this code prints the result as int32
// rather than uns32 because the subFunc function
// returns negative values.

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

    stdout.put( (type uns32 ebx):4, ": " );
    for( mov( 0, ecx); ecx < size; inc( ecx ) ) do

        array.index( edi, ftbl, ebx, ecx );
        stdout.puti32size( [edi], 5, ' ' );

    endfor;
    stdout.newln();

endfor;

end funcTable;

```

3.14 Putting It All Together

In this chapter you saw the low level implementation of procedures and calls to procedures. You learned more about passing parameters by value and reference and you also learned a little more about local variables. This chapter discussed activations records and HLA procedure options. Finally, this chapter wraps up with a discussion of iterators and the FOREACH loop

Your journey through procedures is hardly complete, however. The next volume presents new ways to pass parameters, discusses nested procedures, and explains the low-level implementation of iterators. For more details, see the next volume in this series.

Advanced Arithmetic

Chapter Four

4.1 Chapter Overview

This chapter deals with those arithmetic operations for which assembly language is especially well suited and high level languages are, in general, poorly suited. It covers three main topics: extended precision arithmetic, arithmetic on operands whose sizes are different, and decimal arithmetic.

By far, the most extensive subject this chapter covers is multi-precision arithmetic. By the conclusion of this chapter you will know how to apply arithmetic and logical operations to integer operands of any size. If you need to work with integer values outside the range ± 2 billion (or with unsigned values beyond four billion), no sweat; this chapter will show you how to get the job done.

Operands whose sizes are not the same also present some special problems in arithmetic operations. For example, you may want to add a 128-bit unsigned integer to a 256-bit signed integer value. This chapter discusses how to convert these two operands to a compatible format so the operation may proceed.

Finally, this chapter discusses decimal arithmetic using the BCD (binary coded decimal) features of the 80x86 instruction set and the FPU. This lets you use decimal arithmetic in those few applications that absolutely require base 10 operations (rather than binary).

4.2 Multiprecision Operations

One big advantage of assembly language over high level languages is that assembly language does not limit the size of integer operations. For example, the C programming language defines a maximum of three different integer sizes: short int, int, and long int. On the PC, these are often 16 or 32 bit integers. Although the 80x86 machine instructions limit you to processing eight, sixteen, or thirty-two bit integers with a single instruction, you can always use more than one instruction to process integers of any size you desire. If you want 256 bit integer values, no problem, it's relatively easy to accomplish this in assembly language. The following sections describe how extended various arithmetic and logical operations from 16 or 32 bits to as many bits as you please.

4.2.1 Multiprecision Addition Operations

The 80x86 ADD instruction adds two eight, sixteen, or thirty-two bit numbers¹. After the execution of the add instruction, the 80x86 carry flag is set if there is an overflow out of the H.O. bit of the sum. You can use this information to do multiprecision addition operations. Consider the way you manually perform a multidigit (multiprecision) addition operation:

Step 1: Add the least significant digits together:

289		289
+456	produces	+456
----		----
		5 with carry 1.

Step 2: Add the next significant digits plus the carry:

1 (previous carry)		
289		289
+456	produces	+456
----		----

1. As usual, 32 bit arithmetic is available only on the 80386 and later processors.

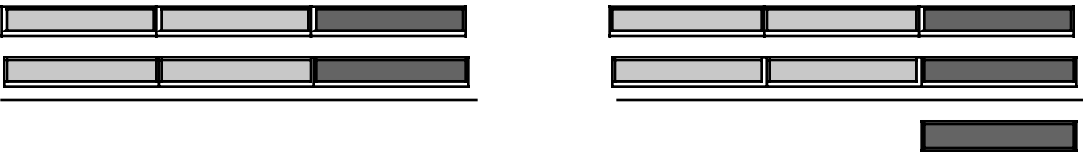
5 45 with carry 1.

Step 3: Add the most significant digits plus the carry:

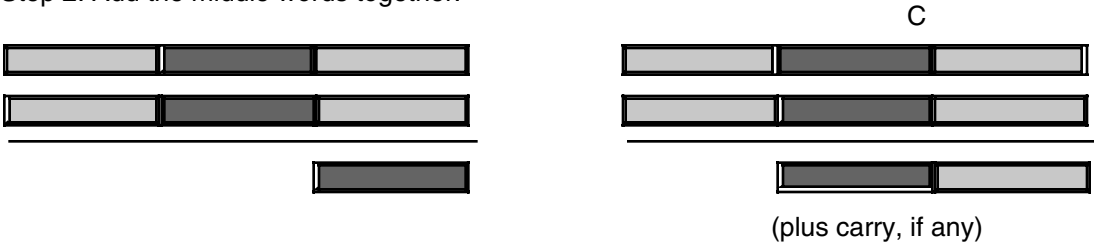
		1 (previous carry)
289		289
+456	produces	+456
----		----
45		745

The 80x86 handles extended precision arithmetic in an identical fashion, except instead of adding the numbers a digit at a time, it adds them together a byte, word, or dword at a time. Consider the three double word (96 bit) addition operation in Figure 4.1.

Step 1: Add the least significant words together:



Step 2: Add the middle words together:



Step 3: Add the most significant words together:

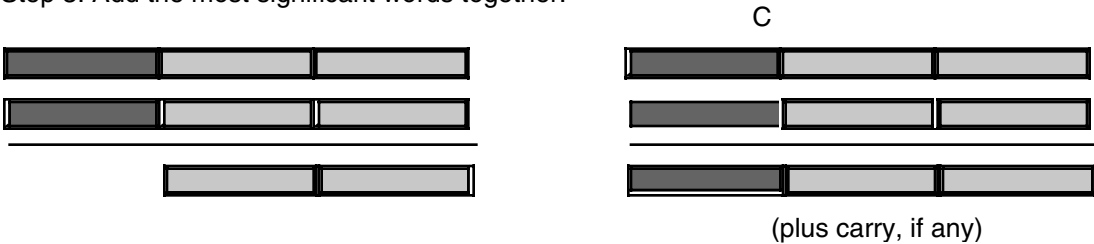


Figure 4.1 Adding Two 96-bit Objects Together

As you can see from this figure, the idea is to break up a larger operation into a sequence of smaller operations. Since the x86 processor family is capable of adding together, at most, 32 bits at a time, the operation must proceed in blocks of 32-bits or less. So the first step is to add the two L.O. double words together much as we would add the two L.O. digits of a decimal number together in the manual algorithm. There is nothing special about this operation, you can use the ADD instruction to achieve this.

The second step involves adding together the second pair of double words in the two 96-bit values. Note that in step two, the calculation must also add in the carry out of the previous addition (if any). If there was a carry out of the L.O. addition, the ADD instruction sets the carry flag to one; conversely, if there was no carry out of the L.O. addition, the earlier ADD instruction clears the carry flag. Therefore, in this second

addition, we really need to compute the sum of the two double words plus the carry out of the first instruction. Fortunately, the x86 chips provide an instruction that does exactly this: the ADC (add with carry) instruction. The ADC instruction uses the same syntax as the ADD instruction and performs almost the same operation:

```
adc( source, dest ); // dest := dest + source + C
```

As you can see, the only difference between the ADD and ADC instruction is that the ADC instruction adds in the value of the carry flag along with the source and destination operands. It also sets the flags the same way the ADD instruction does (including setting the carry flag if there is an unsigned overflow). This is exactly what we need to add together the middle two double words of our 96-bit sum.

In step three of Figure 4.1, the algorithm adds together the H.O. double words of the 96-bit value. Once again, this addition operation also requires the addition of the carry out of the sum of the middle two double words; hence the ADC instruction is needed here, as well. To sum it up, the ADD instruction adds the L.O. double words together. The ADC (add with carry) instruction adds all other double word pairs together. At the end of the extended precision addition sequence, the carry flag indicates unsigned overflow (if set), a set overflow flag indicates signed overflow, and the sign flag indicates the sign of the result. The zero flag doesn't have any real meaning at the end of the extended precision addition (it simply means that the sum of the H.O. two double words is zero, this does not indicate that the whole result is zero).

For example, suppose that you have two 64-bit values you wish to add together, defined as follows:

```
static
X: qword;
Y: qword;
```

Suppose, also, that you want to store the sum in a third variable, Z, that is likewise defined with the *qword* directive. The following x86 code will accomplish this task:

```
mov( (type dword X), eax ); // Add together the L.O. 32 bits
add( (type dword Y), eax ); // of the numbers and store the
mov( eax, (type dword Z) ); // result into the L.O. dword of Z.

mov( (type dword X[4]), eax ); // Add together (with carry) the
adc( (type dword Y[4]), eax ); // H.O. 32 bits and store the result
mov( eax, (type dword Z[4]) ); // into the H.O. dword of Z.
```

Remember, these variables are declared with the *qword* directive. Therefore the compiler will not accept an instruction of the form "mov(X, eax);" because this instruction would attempt to load a 64 bit value into a 32 bit register. Therefore this code uses the coercion operator to coerce symbols X, Y, and Z to 32 bits. The first three instructions add the L.O. double words of X and Y together and store the result at the L.O. double word of Z. The last three instructions add the H.O. double words of X and Y together, along with the carry out of the L.O. word, and store the result in the H.O. double word of Z. Remember, address expressions of the form "X[4]" access the H.O. double word of a 64 bit entity. This is due to the fact that the x86 address space addresses bytes and it takes four consecutive bytes to form a double word.

You can extend this to any number of bits by using the ADC instruction to add in the higher order words in the values. For example, to add together two 128 bit values, you could use code that looks something like the following:

```
type
tBig: dword[4]; // Storage for four dwords is 128 bits.

static
BigVal1: tBig;
BigVal2: tBig;
BigVal3: tBig;
.
.
.
mov( BigVal1[0], eax ); // Note there is no need for (type dword BigValx)
add( BigVal2[0], eax ); // because the base type of BitValx is dword.
```

```

mov( eax, BigVal3[0] );

mov( BigVal1[4], eax );
adc( BigVal2[4], eax );
mov( eax, BigVal3[4] );

mov( BigVal1[8], eax );
adc( BigVal2[8], eax );
mov( eax, BigVal3[8] );

mov( BigVal1[12], eax );
adc( BigVal2[12], eax );
mov( eax, BigVal3[12] );

```

4.2.2 Multiprecision Subtraction Operations

Like addition, the 80x86 performs multi-byte subtraction the same way you would manually, except it subtracts whole bytes, words, or double words at a time rather than decimal digits. The mechanism is similar to that for the ADD operation. You use the SUB instruction on the L.O. byte/word/double word and the SBB (subtract with borrow) instruction on the high order values. The following example demonstrates a 64 bit subtraction using the 32 bit registers on the x86:

```

static
Left:  qword;
Right: qword;
Diff:  qword;
.
.
.
mov( (type dword Left),  eax );
sub( (type dword Right), eax );
mov( eax, (type dword Diff) );

mov( (type dword Left[4]), eax );
sbb( (type dword Right[4]), eax );
mov( (type dword Diff[4]), eax );

```

The following example demonstrates a 128-bit subtraction:

```

type
tBig: dword[4]; // Storage for four dwords is 128 bits.

static
BigVal1: tBig;
BigVal2: tBig;
BigVal3: tBig;
.
.
.

// Compute BigVal3 := BigVal1 - BigVal2

mov( BigVal1[0], eax ); // Note there is no need for (type dword BigValx)
sub( BigVal2[0], eax ); // because the base type of BitValx is dword.
mov( eax, BigVal3[0] );

mov( BigVal1[4], eax );
sbb( BigVal2[4], eax );
mov( eax, BigVal3[4] );

```

```

mov( BigVal1[8], eax );
sbb( BigVal2[8], eax );
mov( eax, BigVal3[8] );

mov( BigVal1[12], eax );
sbb( BigVal2[12], eax );
mov( eax, BigVal3[12] );

```

4.2.3 Extended Precision Comparisons

Unfortunately, there isn't a "compare with borrow" instruction that can be used to perform extended precision comparisons. Since the CMP and SUB instructions perform the same operation, at least as far as the flags are concerned, you'd probably guess that you could use the SBB instruction to synthesize an extended precision comparison; however, you'd only be partly right. There is, however, a better way.

Consider the two unsigned values \$2157 and \$1293. The L.O. bytes of these two values do not affect the outcome of the comparison. Simply comparing \$21 with \$12 tells us that the first value is greater than the second. In fact, the only time you ever need to look at both bytes of these values is if the H.O. bytes are equal. In all other cases comparing the H.O. bytes tells you everything you need to know about the values. Of course, this is true for any number of bytes, not just two. The following code compares two signed 64 bit integers:

```

// This sequence transfers control to location "IsGreater" if
// QwordValue > QwordValue2. It transfers control to "IsLess" if
// QwordValue < QwordValue2. It falls through to the instruction
// following this sequence if QwordValue = QwordValue2. To test for
// inequality, change the "IsGreater" and "IsLess" operands to "NotEqual"
// in this code.

mov( (type dword QWordValue[4]), eax ); // Get H.O. dword
cmp( eax, (type dword QWordValue2[4]) );
jg IsGreater;
jl IsLess;

mov( (type dword QWordValue[0]), eax ); // If H.O. dwords were equal,
cmp( eax, (type dword QWordValue2[0]) ); // then we must compare the
jg IsGreater;                          // L.O. dwords.
jl IsLess;

// Fall through to this point if the two values were equal.

```

To compare unsigned values, simply use the JA and JB instructions in place of JG and JL.

You can easily synthesize any possible comparison from the sequence above, the following examples show how to do this. These examples demonstrate signed comparisons, substitute JA, JAE, JB, and JBE for JG, JGE, JL, and JLE (respectively) if you want unsigned comparisons.

```

static
    QW1: qword;
    QW2: qword;

const
    QW1d: text := "(type dword QW1)";
    QW2d: text := "(type dword QW2)";

// 64 bit test to see if QW1 < QW2 (signed).
// Control transfers to "IsLess" label if QW1 < QW2. Control falls
// through to the next statement if this is not true.

```

```

    mov( QW1d[4], eax );    // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg NotLess;
    jl IsLess;

    mov( QW1d[0], eax );    // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jl IsLess;
NotLess:

// 64 bit test to see if QW1 <= QW2 (signed).  Jumps to "IsLessEq" if the
// condition is true.

    mov( QW1d[4], eax );    // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg NotLessEQ;
    jl IsLessEQ;

    mov( QW1d[0], eax );    // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jle IsLessEQ;
NotLessEQ:

// 64 bit test to see if QW1 > QW2 (signed).  Jumps to "IsGtr" if this condition
// is true.

    mov( QW1d[4], eax );    // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg IsGtr;
    jl NotGtr;

    mov( QW1d[0], eax );    // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jg IsGtr;
NotGtr:

// 64 bit test to see if QW1 >= QW2 (signed).  Jumps to "IsGtrEQ" if this
// is the case.

    mov( QW1d[4], eax );    // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg IsGtrEQ;
    jl NotGtrEQ;

    mov( QW1d[0], eax );    // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jge IsGtrEQ;
NotGtrEQ:

// 64 bit test to see if QW1 = QW2 (signed or unsigned).  This code branches
// to the label "IsEqual" if QW1 = QW2.  It falls through to the next instruction
// if they are not equal.

    mov( QW1d[4], eax );    // Get H.O. dword
    cmp( eax, QW2d[4] );
    jne NotEqual;

    mov( QW1d[0], eax );    // Fall through to here if the H.O. dwords are equal.

```

```

    cmp( eax, QW2d[0] );
    je IsEqual;
NotEqual:

// 64 bit test to see if QW1 <> QW2 (signed or unsigned). This code branches
// to the label "NotEqual" if QW1 <> QW2. It falls through to the next
// instruction if they are equal.

    mov( QW1d[4], eax );    // Get H.O. dword
    cmp( eax, QW2d[4] );
    jne NotEqual;

    mov( QW1d[0], eax );    // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jne NotEqual;

// Fall through to this point if they are equal.

```

You cannot directly use the HLA high level control structures if you need to perform an extended precision comparison. However, you may use the HLA hybrid control structures and bury the appropriate comparison into this statements. Doing so will probably make your code easier to read. For example, the following *if..then..else..endif* statement checks to see if *QW1 > QW2* using an unsigned comparison:

```

if
{
    mov( QW1d[4], eax );
    cmp( eax, QW2d[4] );
    jg true;

    mov( QW1d[0], eax );
    cmp( eax, QW2d[0] );
    jng false;
}

<< code to execute if QW1 > QW2 >>

else

    << code to execute if QW1 <= QW2 >>

endif;

```

If you need to compare objects that are larger than 64 bits, it is very easy to generalize the code above. Always start the comparison with the H.O. double words of the objects and work you way down towards the L.O. double words of the objects as long as the corresponding double words are equal. The following example compares two 128-bit values to see if the first is less than or equal (unsigned) to the second:

```

type
    t128: dword[4];

static
    Big1: t128;
    Big2: t128;
    .
    .
    .
    if
    {
        mov( Big1[12], eax );
        cmp( eax, Big2[12] );
    }

```

```

    jb true;
    mov( Big1[8], eax );
    cmp( eax, Big2[8] );
    jb true;
    mov( Big1[4], eax );
    cmp( eax, Big2[4] );
    jb true;
    mov( Big1[0], eax );
    cmp( eax, Big2[0] );
    jnbe false;
}

<< Code to execute if Big1 <= Big2 >>

else

    << Code to execute if Big1 > Big2 >>

endif;

```

4.2.4 Extended Precision Multiplication

Although a 16x16 or 32x32 multiply is usually sufficient, there are times when you may want to multiply larger values together. You will use the x86 single operand MUL and IMUL instructions for extended precision multiplication.

Not surprisingly (in view of how we achieved extended precision addition using ADC and SBB), you use the same techniques to perform extended precision multiplication on the x86 that you employ when manually multiplying two values. Consider a simplified form of the way you perform multi-digit multiplication by hand:

1) Multiply the first two digits together (5*3):

```

123
 45
---
15

```

2) Multiply 5*2:

```

123
 45
---
15
10

```

3) Multiply 5*1:

```

123
 45
---
15
10
 5

```

4) Multiply 4*3:

```

123
 45
---
15
10
 5
12

```

5) Multiply 4*2:

```

123
 45
---
 15
 10
  5
 12
  8

```

6) Multiply 4*1:

```

123
 45
---
 15
 10
  5
 12
  8
  4

```

7) Add all the partial products together:

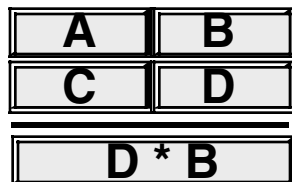
```

123
 45
---
 15
 10
  5
 12
  8
  4
-----
5535

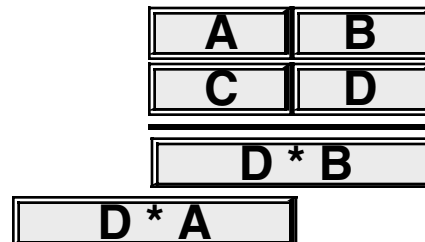
```

The 80x86 does extended precision multiplication in the same manner except that it works with bytes, words, and double words rather than digits. Figure 4.2 shows how this works

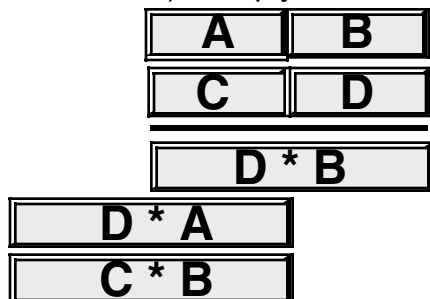
1) Multiply the L.O. words



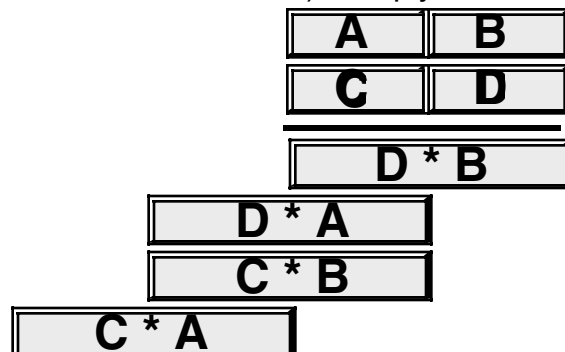
2) Multiply D * A



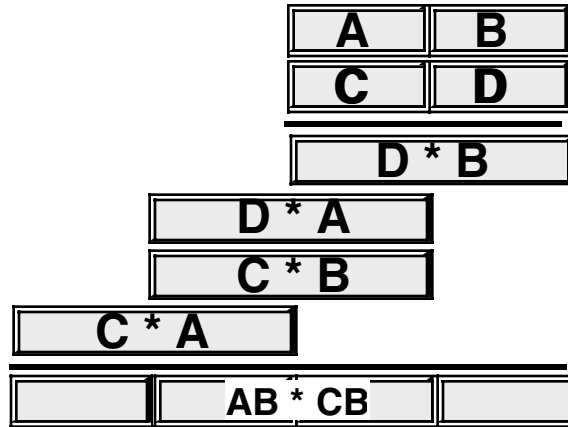
3) Multiply C times B



4) Multiply C * A



5) Compute sum of partial products

**Figure 4.2 Extended Precision Multiplication**

Probably the most important thing to remember when performing an extended precision multiplication is that you must also perform a multiple precision addition at the same time. Adding up all the partial products requires several additions that will produce the result. The following listing demonstrates the proper way to multiply two 64 bit values on a 32 bit processor:

Note: *Multiplier* and *Multiplicand* are 64 bit variables declared in the data segment via the *qword* type. *Product* is a 128 bit variable declared in the data segment via the *qword[2]* type.

```

program testMUL64;
#include( "stdlib.hhf" )

type
  t128:dword[4];

procedure MUL64( Multiplier:qword; Multiplicand:qword; var Product:t128 );
const
  mp:text := "(type dword Multiplier)";
  mc:text := "(type dword Multiplicand)";
  prd:text := "(type dword [edi])";

begin MUL64;

  mov( Product, edi );

  // Multiply the L.O. dword of Multiplier times Multiplicand.

  mov( mp, eax );
  mul( mc, eax );      // Multiply L.O. dwords.
  mov( eax, prd );    // Save L.O. dword of product.
  mov( edx, ecx );    // Save H.O. dword of partial product result.

  mov( mp, eax );
  mul( mc[4], eax );  // Multiply mp(L.O.) * mc(H.O.)
  add( ecx, eax );    // Add to the partial product.
  adc( 0, edx );      // Don't forget the carry!
  mov( eax, ebx );    // Save partial product for now.

```

```

mov( edx, ecx );

// Multiply the H.O. word of Multiplier with Multiplicand.

mov( mp[4], eax ); // Get H.O. dword of Multiplier.
mul( mc, eax );    // Multiply by L.O. word of Multiplicand.
add( ebx, eax );   // Add to the partial product.
mov( eax, prd[4] ); // Save the partial product.
adc( edx, ecx );    // Add in the carry!

mov( mp[4], eax ); // Multiply the two H.O. dwords together.
mul( mc[4], eax );
add( ecx, eax );   // Add in partial product.
adc( 0, edx );     // Don't forget the carry!
mov( eax, prd[8] ); // Save the partial product.
mov( edx, prd[12] );

end MUL64;

static
  op1: qword;
  op2: qword;
  rslt: t128;

begin testMUL64;

  // Initialize the qword values (note that static objects
  // are initialized with zero bits).

  mov( 1234, (type dword op1) );
  mov( 5678, (type dword op2) );
  MUL64( op1, op2, rslt );

  // The following only prints the L.O. qword, but
  // we know the H.O. qword is zero so this is okay.

  stdout.put( "rslt=" );
  stdout.putu64( (type qword rslt) );

end testMUL64;

```

Program 4.1 Extended Precision Multiplication

One thing you must keep in mind concerning this code, it only works for unsigned operands. To multiply two signed values you must note the signs of the operands before the multiplication, take the absolute value of the two operands, do an unsigned multiplication, and then adjust the sign of the resulting product based on the signs of the original operands. Multiplication of signed operands appears in the exercises.

This example was fairly straight-forward since it was possible to keep the partial products in various registers. If you need to multiply larger values together, you will need to maintain the partial products in temporary (memory) variables. Other than that, the algorithm that Program 4.1 uses generalizes to any number of double words.

4.2.5 Extended Precision Division

You cannot synthesize a general n-bit/m-bit division operation using the DIV and IDIV instructions. Such an operation must be performed using a sequence of shift and subtract instructions and is extremely messy. However, a less general operation, dividing an n-bit quantity by a 32 bit quantity is easily synthesized using the DIV instruction. This section presents both methods for extended precision division.

Before describing how to perform a multi-precision division operation, you should note that some operations require an extended precision division even though they may look calculable with a single DIV or IDIV instruction. Dividing a 64-bit quantity by a 32-bit quantity is easy, as long as the resulting quotient fits into 32 bits. The DIV and IDIV instructions will handle this directly. However, if the quotient does not fit into 32 bits then you have to handle this problem as an extended precision division. The following sequence will accomplish this for you:

```
static
dividend: dword[2] := [$1234, 4]; // = $4_0000_1234.
divisor:  dword := 2;             // dividend/divisor = $2_0000_091A
quotient: dword[2];
remainder:dword;
.
.
.
mov( divisor, ebx );
mov( dividend[4], eax );
xor( edx, edx );                // Zero extend for unsigned division.
div( ebx, edx:eax );
mov( eax, quotient[4] );        // Save H.O. dword of the quotient (2).
mov( dividend[0], eax );        // Note that this code does *NOT* zero extend
div( ebx, edx:eax );            // EAX into EDX before this DIV instr.
mov( eax, quotient[0] );        // Save L.O. dword of the quotient ($91a).
mov( edx, remainder );         // Save away the remainder.
```

Since it is perfectly legal to divide a value by one, it is certainly possible that the resulting quotient after a division could require as many bits as the dividend. That is why the *quotient* variable in this example is the same size (64 bits) as the *dividend* variable. Regardless of the size of the dividend and divisor operands, the remainder is always no larger than the size of the division operation (32 bits in this case). Hence the *remainder* variable in this example is just a double word.

Before analyzing this code to see how it works, let's take a brief look at why a single 64/32 division will *not* work for this particular example even though the DIV instruction does indeed calculate the result for a 64/32 division. The naive approach, assuming that the x86 is capable of this operation, would look something like the following:

```
// This code does *NOT* work!

mov( dividend[0], eax ); // Get dividend into edx:eax
mov( dividend[4], edx );
div( divisor, edx:eax ); // Divide edx:eax by divisor.
```

Although this code is syntactically correct and will compile, if you attempt to run this code it will raise an *ex.DivideError*² exception. The reason, if you'll remember how the DIV instruction works, is that the quotient must fit into 32 bits; since the quotient turns out to be \$2_0000_091A, it will not fit into the EAX register, hence the resulting exception.

Now let's take another look at the former code that correctly computes the 64/32 quotient. This code begins by computing the 32/32 quotient of *dividend[4]/divisor*. The quotient from this division (2) becomes the H.O. double word of the final quotient. The remainder from this division (0) becomes the extension in EDX for the second half of the division operation. The second half divides *edx:dividend[0]* by *divisor* to

2. Windows may translate this to an *ex.IntoInstr* exception.

produce the L.O. double word of the quotient and the remainder from the division. Note that the code does not zero extend EAX into EDX prior to the second DIV instruction. EDX already contains valid bits and this code must not disturb them.

The 64/32 division operation above is actually just a special case of the more general division operation that lets you divide an arbitrary sized value by a 32-bit divisor. To achieve this, you begin by moving the H.O. double word of the dividend into EAX and zero extending this into EDX. Next, you divide this value by the divisor. Then, without modifying EDX along the way, you store away the partial quotients, load EAX with the next lower double word in the dividend, and divide it by the divisor. You repeat this operation until you've processed all the double words in the dividend. At that time the EDX register will contain the remainder. The following program demonstrates how to divide a 128 bit quantity by a 32 bit divisor, producing a 128 bit quotient and a 32 bit remainder:

```

program testDiv128;
#include( "stdlib.hhf" )

type
    t128:dword[4];

procedure div128
(
    Dividend:  t128;
    Divisor:   dword;
    var QuotAdrs: t128;
    var Remainder: dword
); nodisplay;

const
    Quotient: text := "(type dword [edi])";

begin div128;

    push( eax );
    push( edx );
    push( edi );

    mov( QuotAdrs, edi );      // Pointer to quotient storage.

    mov( Dividend[12], eax );  // Begin division with the H.O. dword.
    xor( edx, edx );           // Zero extend into EDX.
    div( Divisor, edx:eax );    // Divide H.O. dword.
    mov( eax, Quotient[12] );   // Store away H.O. dword of quotient.

    mov( Dividend[8], eax );   // Get dword #2 from the dividend
    div( Divisor, edx:eax );    // Continue the division.
    mov( eax, Quotient[8] );    // Store away dword #2 of the quotient.

    mov( Dividend[4], eax );   // Get dword #1 from the dividend.
    div( Divisor, edx:eax );    // Continue the division.
    mov( eax, Quotient[4] );    // Store away dword #1 of the quotient.

    mov( Dividend[0], eax );   // Get the L.O. dword of the dividend.
    div( Divisor, edx:eax );    // Finish the division.
    mov( eax, Quotient[0] );    // Store away the L.O. dword of the quotient.

    mov( Remainder, edi );     // Get the pointer to the remainder's value.
    mov( edx, [edi] );         // Store away the remainder value.

    pop( edi );

```

```

    pop( edx );
    pop( eax );

end div128;

static
    op1:    t128    := [$2222_2221, $4444_4444, $6666_6666, $8888_8888];
    op2:    dword   := 2;
    quo:    t128;
    rmndr:  dword;

begin testDiv128;

    div128( op1, op2, quo, rmndr );

    stdout.put
    (
        nl
        nl
        "After the division: " nl
        nl
        "Quotient = $",
        quo[12], "_",
        quo[8], "_",
        quo[4], "_",
        quo[0], nl

        "Remainder = ", (type uns32 rmndr )
    );

end testDiv128;

```

Program 4.2 Unsigned 128/32 Bit Extended Precision Division

You can extend this code to any number of bits by simply adding additional MOV / DIV / MOV instructions to the sequence. Like the extended multiplication the previous section presents, this extended precision division algorithm works only for unsigned operands. If you need to divide two signed quantities, you must note their signs, take their absolute values, do the unsigned division, and then set the sign of the result based on the signs of the operands.

If you need to use a divisor larger than 32 bits you're going to have to implement the division using a shift and subtract strategy. Unfortunately, such algorithms are very slow. In this section we'll develop two division algorithms that operate on an arbitrary number of bits. The first is slow but easier to understand, the second is quite a bit faster (in general).

As for multiplication, the best way to understand how the computer performs division is to study how you were taught to perform long division by hand. Consider the operation 3456/12 and the steps you would take to manually perform this operation:

$\begin{array}{r} 12 \overline{) 3456} \\ \underline{24} \\ 105 \end{array}$ <p>(1) 12 goes into 34 two times.</p>	$\begin{array}{r} 2 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \end{array}$ <p>(2) Subtract 24 from 35 and drop down the 105.</p>
$\begin{array}{r} 28 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 9 \end{array}$ <p>(3) 12 goes into 105 eight times.</p>	$\begin{array}{r} 28 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 9 \end{array}$ <p>(4) Subtract 96 from 105 and drop down the 96.</p>
$\begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 9 \\ \underline{96} \\ 0 \end{array}$ <p>(5) 12 goes into 96 exactly eight times.</p>	$\begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 9 \\ \underline{96} \\ 0 \end{array}$ <p>(6) Therefore, 12 goes into 3456 exactly 288 times.</p>

Figure 4.3 Manual Digit-by-digit Division Operation

This algorithm is actually easier in binary since at each step you do not have to guess how many times 12 goes into the remainder nor do you have to multiply 12 by your guess to obtain the amount to subtract. At each step in the binary algorithm the divisor goes into the remainder exactly zero or one times. As an example, consider the division of 27 (11011) by three (11):

$\begin{array}{r} 11 \overline{) 11011} \\ \underline{11} \\ 00000 \end{array}$	11 goes into 11 one time.
$\begin{array}{r} 1 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \end{array}$	Subtract out the 11 and bring down the zero.
$\begin{array}{r} 1 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ \underline{00} \\ 00 \end{array}$	11 goes into 00 zero times.
$\begin{array}{r} 10 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ \underline{00} \\ 01 \end{array}$	Subtract out the zero and bring down the one.

$$\begin{array}{r}
 10 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00}
 \end{array}
 \quad \begin{array}{l}
 11 \text{ goes into } 01 \text{ zero times.}
 \end{array}$$

$$\begin{array}{r}
 100 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00} \\
 11
 \end{array}
 \quad \begin{array}{l}
 \text{Subtract out the zero and bring down the one.}
 \end{array}$$

$$\begin{array}{r}
 100 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00} \\
 11 \\
 \underline{11} \\
 00
 \end{array}
 \quad \begin{array}{l}
 11 \text{ goes into } 11 \text{ one time.}
 \end{array}$$

$$\begin{array}{r}
 1001 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00} \\
 11 \\
 \underline{11} \\
 00
 \end{array}
 \quad \begin{array}{l}
 \text{This produces the final result} \\
 \text{of } 1001.
 \end{array}$$

Figure 4.4 Longhand Division in Binary

There is a novel way to implement this binary division algorithm that computes the quotient and the remainder at the same time. The algorithm is the following:

```

Quotient := Dividend;
Remainder := 0;
for i:= 1 to NumberBits do

    Remainder:Quotient := Remainder:Quotient SHL 1;
    if Remainder >= Divisor then

        Remainder := Remainder - Divisor;
        Quotient := Quotient + 1;

    endif

```

endfor

NumberBits is the number of bits in the *Remainder*, *Quotient*, *Divisor*, and *Dividend* variables. Note that the "Quotient := Quotient + 1;" statement sets the L.O. bit of *Quotient* to one since this algorithm previously shifts *Quotient* one bit to the left. The following program implements this algorithm

```

program testDiv128b;
#include( "stdlib.hhf" )

type
    t128:dword[4];

// div128-
//
// This procedure does a general 128/128 division operation
// using the following algorithm:
// (all variables are assumed to be 128 bit objects)
//
// Quotient := Dividend;
// Remainder := 0;
// for i:= 1 to NumberBits do
//
//     Remainder:Quotient := Remainder:Quotient SHL 1;
//     if Remainder >= Divisor then
//
//         Remainder := Remainder - Divisor;
//         Quotient := Quotient + 1;
//
//     endif
// endfor
//

procedure div128
(
    Dividend:  t128;
    Divisor:   t128;
    var QuotAdrs:  t128;
    var RmndrAdrs: t128
); nodisplay;

const
    Quotient: text := "Dividend"; // Use the Dividend as the Quotient.

var
    Remainder: t128;

begin div128;

    push( eax );
    push( ecx );
    push( edi );

    mov( 0, eax ); // Set the remainder to zero.
    mov( eax, Remainder[0] );
    mov( eax, Remainder[4] );
    mov( eax, Remainder[8] );
    mov( eax, Remainder[12] );

```

```

mov( 128, ecx );           // Count off 128 bits in ECX.
repeat

    // Compute Remainder:Quotient := Remainder:Quotient SHL 1:

    shl( 1, Dividend[0] ); // See the section on extended
    rcl( 1, Dividend[4] ); // precision shifts to see how
    rcl( 1, Dividend[8] ); // this code shifts 256 bits to
    rcl( 1, Dividend[12] ); // the left by one bit.
    rcl( 1, Remainder[0] );
    rcl( 1, Remainder[4] );
    rcl( 1, Remainder[8] );
    rcl( 1, Remainder[12] );

    // Do a 128-bit comparison to see if the remainder
    // is greater than or equal to the divisor.

    if
    {
        mov( Remainder[12], eax );
        cmp( eax, Divisor[12] );
        ja true;
        jb false;

        mov( Remainder[8], eax );
        cmp( eax, Divisor[8] );
        ja true;
        jb false;

        mov( Remainder[4], eax );
        cmp( eax, Divisor[4] );
        ja true;
        jb false;

        mov( Remainder[0], eax );
        cmp( eax, Divisor[0] );
        jb false;
    }

    // Remainder := Remainder - Divisor

    mov( Divisor[0], eax );
    sub( eax, Remainder[0] );

    mov( Divisor[4], eax );
    sbb( eax, Remainder[4] );

    mov( Divisor[8], eax );
    sbb( eax, Remainder[8] );

    mov( Divisor[12], eax );
    sbb( eax, Remainder[12] );

    // Quotient := Quotient + 1;

    add( 1, Quotient[0] );
    adc( 0, Quotient[4] );
    adc( 0, Quotient[8] );
    adc( 0, Quotient[12] );

endif;

```

```

        dec( ecx );

until( @z );

// Okay, copy the quotient (left in the Dividend variable)
// and the remainder to their return locations.

mov( QuotAdrs, edi );
mov( Quotient[0], eax );
mov( eax, [edi] );
mov( Quotient[4], eax );
mov( eax, [edi+4] );
mov( Quotient[8], eax );
mov( eax, [edi+8] );
mov( Quotient[12], eax );
mov( eax, [edi+12] );

mov( RmndrAdrs, edi );
mov( Remainder[0], eax );
mov( eax, [edi] );
mov( Remainder[4], eax );
mov( eax, [edi+4] );
mov( Remainder[8], eax );
mov( eax, [edi+8] );
mov( Remainder[12], eax );
mov( eax, [edi+12] );

pop( edi );
pop( ecx );
pop( eax );

end div128;

// Some simple code to test out the division operation:

static
    op1:    t128    := [$2222_2221, $4444_4444, $6666_6666, $8888_8888];
    op2:    t128    := [2, 0, 0, 0];
    quo:    t128;
    rmndr:  t128;

begin testDiv128b;

    div128( op1, op2, quo, rmndr );

    stdout.put
    (
        nl
        nl
        "After the division: " nl
        nl
        "Quotient = $",
        quo[12], "_",
        quo[8], "_",
        quo[4], "_",
        quo[0], nl
    )

```

```

        "Remainder = ", (type uns32 rmndr )
    );

end testDiv128b;

```

Program 4.3 Extended Precision Division

This code looks simple but there are a few problems with it. First, it does not check for division by zero (it will produce the value \$FFFF_FFFF_FFFF_FFFF if you attempt to divide by zero), it only handles unsigned values, and it is very slow. Handling division by zero is very simple, just check the divisor against zero prior to running this code and return an appropriate error code if the divisor is zero. Dealing with signed values is the same as the earlier division algorithm, this problem appears as a programming exercise. The performance of this algorithm, however, leaves a lot to be desired. It's around an order of magnitude or two worse than the DIV/IDIV instructions on the x86 and they are among the slowest instructions on the CPU.

There is a technique you can use to boost the performance of this division by a fair amount: check to see if the divisor variable uses only 32 bits. Often, even though the divisor is a 128 bit variable, the value itself fits just fine into 32 bits (i.e., the H.O. double words of Divisor are zero). In this special case, that occurs frequently, you can use the DIV instruction which is much faster.

4.2.6 Extended Precision NEG Operations

Although there are several ways to negate an extended precision value, the shortest way for smaller values (96 bits or less) is to use a combination of NEG and SBB instructions. This technique uses the fact that NEG subtracts its operand from zero. In particular, it sets the flags the same way the SUB instruction would if you subtracted the destination value from zero. This code takes the following form (assuming you want to negate the 64-bit value in EDX:EAX):

```

neg( edx );
neg( eax );
sbb( 0, edx );

```

The SBB instruction decrements EDX if there is a borrow out of the L.O. word of the negation operation (which always occurs unless EAX is zero).

To extend this operation to additional bytes, words, or double words is easy; all you have to do is start with the H.O. memory location of the object you want to negate and work towards the L.O. byte. The following code computes a 128 bit negation:

```

static
Value: dword[4];
.
.
.
neg( Value[12] );      // Negate the H.O. double word.
neg( Value[8] );       // Neg previous dword in memory.
sbb( 0, Value[12] );   // Adjust H.O. dword.

neg( Value[4] );       // Negate the second dword in the object.
sbb( 0, Value[8] );    // Adjust third dword in object.
sbb( 0, Value[12] );   // Adjust the H.O. dword.

neg( Value );          // Negate the L.O. dword.
sbb( 0, Value[4] );    // Adjust second dword in object.
sbb( 0, Value[8] );    // Adjust third dword in object.
sbb( 0, Value[12] );   // Adjust the H.O. dword.

```

Unfortunately, this code tends to get really large and slow since you need to propagate the carry through all the H.O. words after each negate operation. A simpler way to negate larger values is to simply subtract that value from zero:

```
static
    Value: dword[5];    // 160-bit value.
    .
    .
    .
    mov( 0, eax );
    sub( Value, eax );
    mov( eax, Value );

    mov( 0, eax );
    sbb( Value[4], eax );
    mov( eax, Value[4] );

    mov( 0, eax );
    sbb( Value[8], eax );
    mov( eax, Value[8] );

    mov( 0, eax );
    sbb( Value[12], eax );
    mov( eax, Value[12] );

    mov( 0, eax );
    sbb( Value[16], eax );
    mov( eax, Value[16] );
```

4.2.7 Extended Precision AND Operations

Performing an n-word AND operation is very easy – simply AND the corresponding words between the two operands, saving the result. For example, to perform the AND operation where all three operands are 64 bits long, you could use the following code:

```
mov( (type dword source1), eax );
and( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
and( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );
```

This technique easily extends to any number of words, all you need to is logically AND the corresponding bytes, words, or double words together in the operands. Note that this sequence sets the flags according to the value of the last AND operation. If you AND the H.O. double words last, this sets all but the zero flag correctly. If you need to test the zero flag after this sequence, you will need to logically OR the two resulting double words together (or otherwise compare them both against zero).

4.2.8 Extended Precision OR Operations

Multi-word logical OR operations are performed in the same way as multi-word AND operations. You simply OR the corresponding words in the two operand together. For example, to logically OR two 96 bit values, use the following code:

```
mov( (type dword source1), eax );
or( (type dword source2), eax );
```

```

mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
or( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );

mov( (type dword source1[8]), eax );
or( (type dword source2[8]), eax );
mov( eax, (type dword dest[8]) );

```

As for the previous example, this does not set the zero flag properly for the entire operation. If you need to test the zero flag after a multiprecision OR, you must logically OR the resulting double words together.

4.2.9 Extended Precision XOR Operations

Extended precision XOR operations are performed in a manner identical to AND/OR – simply XOR the corresponding words in the two operands to obtain the extended precision result. The following code sequence operates on two 64 bit operands, computes their exclusive-or, and stores the result into a 64 bit variable.

```

mov( (type dword source1), eax );
xor( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
xor( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );

```

The comment about the zero flag in the previous two sections applies here.

4.2.10 Extended Precision NOT Operations

The NOT instruction inverts all the bits in the specified operand. An extended precision NOT is performed by simply executing the NOT instruction on all the affected operands. For example, to perform a 64 bit NOT operation on the value in (edx:eax), all you need to do is execute the instructions:

```

not( eax );
not( edx );

```

Keep in mind that if you execute the NOT instruction twice, you wind up with the original value. Also note that exclusive-ORing a value with all ones (\$FF, \$FFFF, or \$FFFF_FFFF) performs the same operation as the NOT instruction.

4.2.11 Extended Precision Shift Operations

Extended precision shift operations require a shift and a rotate instruction. Consider what must happen to implement a 64 bit SHL using 32 bit operations:

- 1) A zero must be shifted into bit zero.
- 2) Bits zero through 30 are shifted into the next higher bit.
- 3) Bit 31 is shifted into bit 32.
- 4) Bits 32 through 62 must be shifted into the next higher bit.
- 5) Bit 63 is shifted into the carry flag.

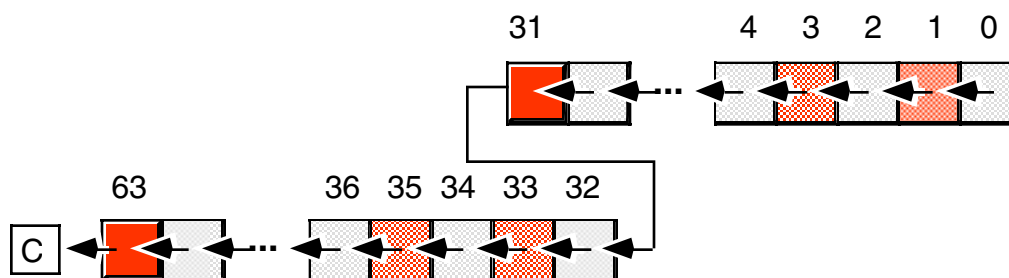


Figure 4.5 64-bit Shift Left Operation

The two instructions you can use to implement this 32 bit shift are SHL and RCL. For example, to shift the 64 bit quantity in (EDX:EAX) one position to the left, you'd use the instructions:

```
shl( 1, eax );
rcl( 1, eax );
```

Note that you can only shift an extended precision value one bit at a time. You cannot shift an extended precision operand several bits using the CL register. Nor can you specify a constant value greater than one using this technique.

To understand how this instruction sequence works, consider the operation of these instructions on an individual basis. The SHL instruction shifts a zero into bit zero of the 64 bit operand and shifts bit 31 into the carry flag. The RCL instruction then shifts the carry flag into bit 32 and then shifts bit 63 into the carry flag. The result is exactly what we want.

To perform a shift left on an operand larger than 64 bits you simply add additional RCL instructions. An extended precision shift left operation always starts with the least significant word and each succeeding RCL instruction operates on the next most significant word. For example, to perform a 96 bit shift left operation on a memory location you could use the following instructions:

```
shl( 1, (type dword Operand[0]) );
rcl( 1, (type dword Operand[4]) );
rcl( 1, (type dword Operand[8]) );
```

If you need to shift your data by two or more bits, you can either repeat the above sequence the desired number of times (for a constant number of shifts) or you can place the instructions in a loop to repeat them some number of times. For example, the following code shifts the 96 bit value *Operand* to the left the number of bits specified in ECX:

```
ShiftLoop:
    shl( 1, (type dword Operand[0]) );
    rcl( 1, (type dword Operand[4]) );
    rcl( 1, (type dword Operand[8]) );
    dec( ecx );
    jnz ShiftLoop;
```

You implement SHR and SAR in a similar way, except you must start at the H.O. word of the operand and work your way down to the L.O. word:

```
// Double precision SAR:

sar( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );

// Double precision SHR:
```

```
shr( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );
```

There is one major difference between the extended precision shifts described here and their 8/16/32 bit counterparts – the extended precision shifts set the flags differently than the single precision operations. This is because the rotate instructions affect the flags differently than the shift instructions. Fortunately, the carry is the flag most often tested after a shift operation and the extended precision shift operations (i.e., rotate instructions) properly set this flag.

The SHLD and SHRD instructions let you efficiently implement multiprecision shifts of several bits. These instructions have the following syntax:

```
shld( constant, Operand1, Operand2 );
shld( cl, Operand1, Operand2 );
shlr( constant, Operand1, Operand2 );
shlr( cl, Operand1, Operand2 );
```

The SHLD instruction does the following:

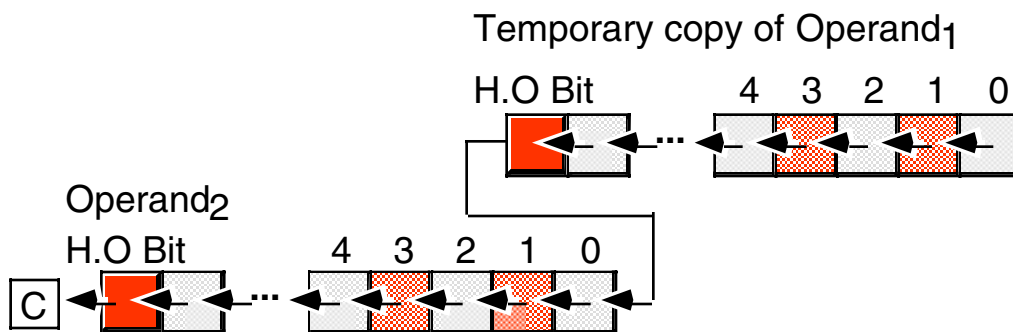


Figure 4.6 SHLD Operation

Operand₁ must be a 16 or 32 bit register. *Operand₂* can be a register or a memory location. Both operands must be the same size. The immediate operand can be a value in the range zero through n-1, where n is the number of bits in the two operands; it specifies the number of bits to shift.

The SHLD instruction shifts bits in *Operand₂* to the left. The H.O. bits shift into the carry flag and the H.O. bits of *Operand₁* shift into the L.O. bits of *Operand₂*. Note that this instruction does not modify the value of *Operand₁*, it uses a temporary copy of *Operand₁* during the shift. The immediate operand specifies the number of bits to shift. If the count is n, then SHLD shifts bit n-1 into the carry flag. It also shifts the H.O. n bits of *Operand₁* into the L.O. n bits of *Operand₂*. The SHLD instruction sets the flag bits as follows:

- If the shift count is zero, the SHLD instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the *Operand₂*.
- If the shift count is one, the overflow flag will contain one if the sign bit of *Operand₂* changes during the shift. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

The SHRD instruction is similar to SHLD except, of course, it shifts its bits right rather than left. To get a clear picture of the SHRD instruction, consider Figure 4.7

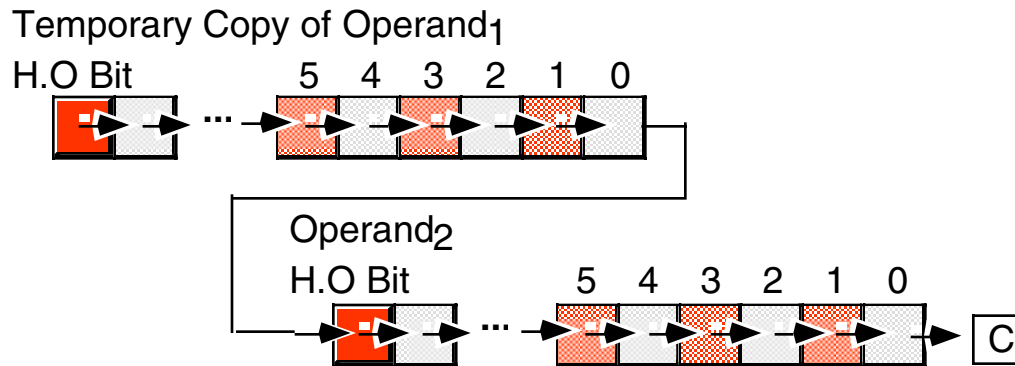


Figure 4.7 SHRD Operation

The SHRD instruction sets the flag bits as follows:

- If the shift count is zero, the SHRD instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the *Operand2*.
- If the shift count is one, the overflow flag will contain one if the H.O. bit of *Operand2* changes. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

Consider the following code sequence:

```
static
ShiftMe: dword[3] := [ $1234, $5678, $9012 ];
.
.
.
mov( ShiftMe[4], eax )
shld( 6, eax, ShiftMe[8] );
mov( ShiftMe[0], eax );
shld( 6, eax, ShiftMe[4] );
shl( 6, ShiftMe[0] );
```

The first SHLD instruction above shifts the bits from *ShiftMe+4* into *ShiftMe+8* without affecting the value in *ShiftMe+4*. The second SHLD instruction shifts the bits from *SHIFTME* into *SHIFTME+4*. Finally, the SHL instruction shifts the L.O. double word the appropriate amount. There are two important things to note about this code. First, unlike the other extended precision shift left operations, this sequence works from the H.O. double word down to the L.O. double word. Second, the carry flag does not contain the carry out of the H.O. shift operation. If you need to preserve the carry flag at that point, you will need to push the flags after the first SHLD instruction and pop the flags after the SHL instruction.

You can do an extended precision shift right operation using the SHRD instruction. It works almost the same way as the code sequence above except you work from the L.O. double word to the H.O. double word. The solution is left as an exercise at the end of this chapter.

4.2.12 Extended Precision Rotate Operations

The RCL and RCR operations extend in a manner almost identical to that for SHL and SHR. For example, to perform 96 bit RCL and RCR operations, use the following instructions:

```
rcl( 1, (type dword Operand[0]) );
```

```

rcl( 1, (type dword Operand[4]) );
rcl( 1, (type dword Operand[8]) );

rcr( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );

```

The only difference between this code and the code for the extended precision shift operations is that the first instruction is a RCL or RCR rather than a SHL or SHR instruction.

Performing an extended precision ROL or ROR instruction isn't quite as simple an operation. You can use the BT, SHLD, and SHRD instructions to implement an extended precision ROL or ROR instruction. The following code shows how to use the SHLD instruction to do an extended precision ROL:

```

// Compute ROL( 4, EDX:EAX );

mov( edx, ebx );
shld, 4, eax, edx );
shld( 4, ebx, eax );
bt( 0, eax );          // Set carry flag, if desired.

```

An extended precision ROR instruction is similar; just keep in mind that you work on the L.O. end of the object first and the H.O. end last.

4.2.13 Extended Precision I/O

Once you have the ability to compute using extended precision arithmetic, the next problem is how do you get those extended precision values into your program and how do you display those extended precision values to the user? HLA's Standard Library provides routines for unsigned decimal, signed decimal, and hexadecimal I/O for values that are eight, 16, 32, or 64 bits in length. So as long as you're working with values whose size is less than or equal to 64 bits in length, you can use the Standard Library code. If you need to input or output values that are greater than 64 bits in length, you will need to write your own procedures to handle the operation. This section discusses the strategies you will need to write such routines.

The examples in this section work specifically with 128-bit values. The algorithms are perfectly general and extend to any number of bits (indeed, the 128-bit algorithms in this section are really nothing more than an extension of the algorithms the HLA Standard Library uses for 64-bit values). If you need a set of 128-bit unsigned I/O routines, you will probably be able to use the following code as-is. If you need to handle larger values, simple modifications to the following code is all that should be necessary.

The following examples all assume a common data type for 128-bit values. The HLA type declaration for this data type is one of the following depending on the type of value

```

type
bits128: dword[4];
uns128: bits128;
int128: bits128;

```

4.2.13.1 Extended Precision Hexadecimal Output

Extended precision hexadecimal output is very easy. All you have to do is output each double word component of the extended precision value from the H.O. double word to the L.O. double word using a call to the *stdout.putdw* routine. The following procedure does exactly this to output a *bits128* value:

```

procedure putb128( b128: bits128 ); nodisplay;
begin putb128;

    stdout.putdw( b128[12] );

```

```

stdout.putdw( b128[8] );
stdout.putdw( b128[4] );
stdout.putdw( b128[0] );

end putb128;

```

Since HLA provides the *stdout.putqw* procedure, you can shorten the code above by calling *stdout.putqw* just twice:

```

procedure putb128( b128: bits128 ); nodisplay;
begin putb128;

    stdout.putqw( (type qword b128[8]) );
    stdout.putqw( (type qword b128[0]) );

end putb128;

```

Note that this code outputs the two quad words with the H.O. quad word output first and L.O. quad word output second.

4.2.13.2 Extended Precision Unsigned Decimal Output

Decimal output is a little more complicated than hexadecimal output because the H.O. bits of a binary number affect the L.O. digits of the decimal representation (this was not true for hexadecimal values which is why hexadecimal output is so easy). Therefore, we will have to create the decimal representation for a binary number by extracting one decimal digit at a time from the number.

The most commonly employed solution for unsigned decimal output is to successively divide the value by ten until the result becomes zero. The remainder after the first division is a value in the range 0..9 and this value corresponds to the L.O. digit of the decimal number. Successive divisions by ten (and their corresponding remainder) extract successive digits to the left in the number.

Iterative solutions to this problem generally allocate storage for a string of characters large enough to hold the entire number. Then the code extracts the decimal digits in a loop and places them in the string one by one. At the end of the conversion process, the routine prints the characters in the string in reverse order (remember, the divide algorithm extracts the L.O. digits first and the H.O. digits last, the opposite of the way you need to print them).

In this section, we will employ a recursive solution because it is a little more elegant. The recursive solution begins by dividing the value by 10 and saving the remainder in a local variable. If the quotient was not zero, the routine recursively calls itself to print any leading digits first. On return from the recursive call (which prints all the leading digits), the recursive algorithm prints the digit associated with the remainder to complete the operation. Here's how the operation works when printing the decimal value "123":

- (1) Divide 123 by 10. Quotient is 12, remainder is 3.
- (2) Save the remainder (3) in a local variable and recursively call the routine with the quotient.
- (3) [Recursive Entry 1] Divide 12 by 10. Quotient is 1, remainder is 2.
- (4) Save the remainder (2) in a local variable and recursively call the routine with the quotient.
- (5) [Recursive Entry 2] Divide 1 by 10. Quotient is 0, remainder is 1.
- (6) Save the remainder (1) in a local variable. Since the Quotient is zero, don't call the routine recursively.
- (7) Output the remainder value saved in the local variable (1). Return to the caller (Recursive Entry 1).
- (8) [Return to Recursive Entry 1] Output the remainder value saved in the local variable in recursive entry 1 (2). Return to the caller (original invocation of the procedure).
- (9) [Original invocation] Output the remainder value saved in the local variable in the original call (3). Return to the original caller of the output routine.

The only operation that requires extended precision calculation through this entire algorithm is the "divide by 10" requirement. Everything else is simple and straight-forward. We are in luck with this algorithm, since we are dividing an extended precision value by a value that easily fits into a double word, we can use the fast (and easy) extended precision division algorithm that uses the DIV instruction (see "Extended Precision Division" on page 836). The following program implements a 128-bit decimal output routine utilizing this technique.

```

program out128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    uns128: dword[4];

// DivideBy10-
//
// Divides "divisor" by 10 using fast
// extended precision division algorithm
// that employs the DIV instruction.
//
// Returns quotient in "quotient"
// Returns remainder in eax.
// Trashes EBX, EDX, and EDI.

procedure DivideBy10( dividend:uns128; var quotient:uns128 ); nodisplay;
begin DivideBy10;

    mov( quotient, edi );
    xor( edx, edx );
    mov( dividend[12], eax );
    mov( 10, ebx );
    div( ebx, edx:eax );
    mov( eax, [edi+12] );

    mov( dividend[8], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+8] );

    mov( dividend[4], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+4] );

    mov( dividend[0], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+0] );
    mov( edx, eax );

end DivideBy10;

// Recursive version of putu128.
// A separate "shell" procedure calls this so that
// this code does not have to preserve all the registers
// it uses (and DivideBy10 uses) on each recursive call.

```

```

procedure recursivePutul28( b128:uns128 ); nodisplay;
var
    remainder: byte;

begin recursivePutul28;

    // Divide by ten and get the remainder (the char to print).

    DivideBy10( b128, b128 );
    mov( al, remainder );          // Save away the remainder (0..9).

    // If the quotient (left in b128) is not zero, recursively
    // call this routine to print the H.O. digits.

    mov( b128[0], eax );          // If we logically OR all the dwords
    or( b128[4], eax );           // together, the result is zero if and
    or( b128[8], eax );           // only if the entire number is zero.
    or( b128[12], eax );
    if( @nz ) then

        recursivePutul28( b128 );

    endif;

    // Okay, now print the current digit.

    mov( remainder, al );
    or( '0', al );                // Converts 0..9 -> '0..'9'.
    stdout.putc( al );

end recursivePutul28;

// Non-recursive shell to the above routine so we don't bother
// saving all the registers on each recursive call.

procedure putul28( b128:uns128 ); nodisplay;
begin putul28;

    push( eax );
    push( ebx );
    push( edx );
    push( edi );

    recursivePutul28( b128 );

    pop( edi );
    pop( edx );
    pop( ebx );
    pop( eax );

end putul28;

// Code to test the routines above:

static
    b0: uns128 := [0, 0, 0, 0];          // decimal = 0
    b1: uns128 := [1234567890, 0, 0, 0]; // decimal = 1234567890

```

```

b2: uns128 := [$8000_0000, 0, 0, 0];    // decimal = 2147483648
b3: uns128 := [0, 1, 0, 0 ];           // decimal = 4294967296

// Largest uns128 value
// (decimal=340,282,366,920,938,463,463,374,607,431,768,211,455) :

b4: uns128 := [$FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF ];

begin out128;

  stdout.put( "b0 = " );
  putul28( b0 );
  stdout.newln();

  stdout.put( "b1 = " );
  putul28( b1 );
  stdout.newln();

  stdout.put( "b2 = " );
  putul28( b2 );
  stdout.newln();

  stdout.put( "b3 = " );
  putul28( b3 );
  stdout.newln();

  stdout.put( "b4 = " );
  putul28( b4 );
  stdout.newln();

end out128;

```

Program 4.4 128-bit Extended Precision Decimal Output Routine

4.2.13.3 Extended Precision Signed Decimal Output

Once you have an extended precision unsigned decimal output routine, writing an extended precision signed decimal output routine is very easy. The basic algorithm takes the following form:

- Check the sign of the number. If it is positive, call the unsigned output routine to print it.
- If the number is negative, print a minus sign. Then negate the number and call the unsigned output routine to print it.

To check the sign of an extended precision integer, of course, you simply test the H.O. bit of the number. To negate a large value, the best solution is to probably subtract that value from zero. Here's a quick version of *puti128* that uses the *putul28* routine from the previous section.

```

procedure puti128( i128: int128 ); nodisplay;
begin puti128;

  if( (type int32 i128[12]) < 0 ) then

    stdout.put( '-' );

    // Extended Precision Negation:

    push( eax );

```

```

        mov( 0, eax );
        sub( i128[0], eax );
        mov( eax, i128[0] );

        mov( 0, eax );
        sbb( i128[4], eax );
        mov( eax, i128[4] );

        mov( 0, eax );
        sbb( i128[8], eax );
        mov( eax, i128[8] );

        mov( 0, eax );
        sbb( i128[12], eax );
        mov( eax, i128[12] );
        pop( eax );

    endif;
    putu128( (type uns128 i128));

end puti128;

```

4.2.13.4 Extended Precision Formatted I/O

The code in the previous two sections prints signed and unsigned integers using the minimum number of necessary print positions. To create nicely formatted tables of values you will need the equivalent of a *puti128Size* or *putu128Size* routine. Once you have the "unformatted" versions of these routines, implementing the formatted versions is very easy.

The first step is to write an "i128Size" and a "u128Size" routine that computes the minimum number of digits needed to display the value. The algorithm to accomplish this is very similar to the numeric output routines. In fact, the only difference is that you initialize a counter to zero upon entry into the routine (e.g., the non-recursive shell routine) and you increment this counter rather than outputting a digit on each recursive call. (Don't forget to increment the counter inside "i128Size" if the number is negative; you must allow for the output of the minus sign.) After the calculation is complete, these routines should return the size of the operand in the EAX register.

Once you have the "i128Size" and "u128Size" routines, writing the formatted output routines is very easy. Upon initial entry into *puti128Size* or *putu128Size*, these routines call the corresponding "size" routine to determine the number of print positions for the number to display. If the value that the "size" routine returns is greater than the absolute value of the minimum size parameter (passed into *puti128Size* or *putu128Size*) all you need to do is call the put routine to print the value, no other formatting is necessary. If the absolute value of the parameter size is greater than the value *i128Size* or *u128Size* returns, then the program must compute the difference between these two values and print that many spaces (or other filler character) before printing the number (if the parameter size value is positive) or after printing the number (if the parameter size value is negative). The actual implementation of these two routines is left as an exercise at the end of the volume. If you have any further questions about how to do this, you can take a look at the HLA Standard Library code for routines like *stdout.putu32Size*.

4.2.13.5 Extended Precision Input Routines

There are a couple of fundamental differences between the extended precision output routines and the extended precision input routines. First of all, numeric output generally occurs without possibility of error³;

3. Technically speaking, this isn't entirely true. It is possible for a device error (e.g., disk full) to occur. The likelihood of this is so low that we can effectively ignore this possibility.

numeric input, on the other hand, must handle the very real possibility of an input error such as illegal characters and numeric overflow. Also, HLA's Standard Library and run-time system encourages a slightly different approach to input conversion. This section discusses those issues that differentiate input conversion from output conversion.

Perhaps the biggest difference between input and output conversion is the fact that output conversion is *unbracketed*. That is, when converting a numeric value to a string of characters for output, the output routine does not concern itself with character preceding the output string nor does it concern itself with the characters following the numeric value in the output stream. Numeric output routines convert their data to a string and print that string without considering the context (i.e., the characters before and after the string representation of the numeric value). Numeric input routines cannot be so cavalier; the contextual information surrounding the numeric string is very important.

A typical numeric input operation consists of reading a string of characters from the user and then translating this string of characters into an internal numeric representation. For example, a statement like `"stdin.get(i32);"` typically reads a line of text from the user and converts a sequence of digits appearing at the beginning of that line of text into a 32-bit signed integer (assuming *i32* is an *int32* object). Note, however, that the *stdin.get* routine skips over certain characters in the string that may appear before the actual numeric characters. For example, *stdin.get* automatically skips any leading spaces in the string. Likewise, the input string may contain additional data beyond the end of the numeric input (for example, it is possible to read two integer values from the same input line), therefore the input conversion routine must somehow determine where the numeric data ends in the input stream. Fortunately, HLA provides a simple mechanism that lets you easily determine the start and end of the input data: the *Delimiters* character set.

The *Delimiters* character set is a variable, internal to HLA, that contains the set of legal character that may precede or follow a legal numeric value. By default, this character set includes the end of string marker (a zero byte), a tab character, a line feed character, a carriage return character, a space, a comma, a colon, and a semicolon. Therefore, HLA's numeric input routines will automatically ignore any characters in this set that occur on input before a numeric string. Likewise, characters from this set may legally follow a numeric string on input (conversely, if any non-delimiter character follows the numeric string, HLA will raise an *ex.ConversionError* exception).

The *Delimiters* character set is a private variable inside the HLA Standard Library. Although you do not have direct access to this object, the HLA Standard Library does provide two accessor functions, *conv.setDelimiters* and *conv.getDelimiters* that let you access and modify the value of this character set. These two functions have the following prototypes (found in the "conv.hhf" header file):

```
procedure conv.setDelimiters( Delims:cset );
procedure conv.getDelimiters( var Delims:cset );
```

The *conv.SetDelimiters* procedure will copy the value of the *Delims* parameter into the internal *Delimiters* character set. Therefore, you can use this procedure to change the character set if you want to use a different set of delimiters for numeric input. The *conv.getDelimiters* call returns a copy of the internal *Delimiters* character set in the variable you pass as a parameter to the *conv.getDelimiters* procedure. We will use the value returned by *conv.getDelimiters* to determine the end of numeric input when writing our own extended precision numeric input routines.

When reading a numeric value from the user, the first step will be to get a copy of the *Delimiters* character set. The second step is to read and discard input characters from the user as long as those characters are members of the *Delimiters* character set. Once a character is found that is not in the *Delimiters* set, the input routine must check this character and verify that it is a legal numeric character. If not, the program should raise an *ex.IllegalChar* exception if the character's value is outside the range \$00..\$7f or it should raise the *ex.ConversionError* exception if the character is not a legal numeric character. Once the routine encounters a numeric character, it should continue reading characters as long as they valid numeric characters; while reading the characters the conversion routine should be translating them to the internal representation of the numeric data. If, during conversion, an overflow occurs, the procedure should raise the *ex.ValueOutOfRange* exception.

Conversion to numeric representation should end when the procedure encounters the first delimiter character at the end of the string of digits. However, it is very important that the procedure does not consume the delimiter character that ends the string. That is, the following is incorrect:

```

static
    Delimiters: cset;
    .
    .
    .
conv.getDelimiters( Delimiters );

// Skip over leading delimiters in the string:

while( stdin.getc() in Delimiters ) do /* getc did the work */ endwhile;
while( al in {'0'..'9'}) do

    // Convert character in AL to numeric representation and
    // accumulate result...

    stdin.getc();

endwhile;
if( al not in Delimiters ) then

    raise( ex.ConversionError );

endif;

```

The first WHILE loop reads a sequence of delimiter characters. When this first WHILE loop ends, the character in AL is not a delimiter character. So far, so good. The second WHILE loop processes a sequence of decimal digits. First, it checks the character read in the previous WHILE loop to see if it is a decimal digit; if so, it processes that digit and reads the next character. This process continues until the call to *stdin.getc* (at the bottom of the loop) reads a non-digit character. After the second WHILE loop, the program checks the last character read to ensure that it is a legal delimiter character for a numeric input value.

The problem with this algorithm is that it consumes the delimiter character after the numeric string. For example, the colon symbol is a legal delimiter in the default *Delimiters* character set. If the user types the input "123:456" and executes the code above, this code will properly convert "123" to the numeric value one hundred twenty-three. However, the very next character read from the input stream will be the character "4" not the colon character (":"). While this may be acceptable in certain circumstances, Most programmers expect numeric input routines to consume only leading delimiter characters and the numeric digit characters. They do not expect the input routine to consume any trailing delimiter characters (e.g., many programs will read the next character and expect a colon as input if presented with the string "123:456"). Since *stdin.getc* consumes an input character, and there is no way to "put the character back" onto the input stream, some other way of reading input characters from the user, that doesn't consume those characters, is needed⁴.

The HLA Standard Library comes to the rescue by providing the *stdin.peekc* function. Like *stdin.getc*, the *stdin.peekc* routine reads the next input character from HLA's internal buffer. There are two major differences between *stdin.peekc* and *stdin.getc*. First, *stdin.peekc* will not force the input of a new line of text from the user if the current input line is empty (or you've already read all the text from the input line). Instead, *stdin.peekc* simply returns zero in the AL register to indicate that there are no more characters on the input line. Since #0 is (by default) a legal delimiter character for numeric values, and the end of line is certainly a legal way to terminate numeric input, this works out rather well. The second difference between *stdin.getc* and *stdin.peekc* is that *stdin.peekc* does not consume the character read from the input buffer. If you call *stdin.peekc* several times in a row, it will always return the same character; likewise, if you call *stdin.getc* immediately after *stdin.peekc*, the call to *stdin.getc* will generally return the same character as returned by *stdin.peekc* (the only exception being the end of line condition). So although we cannot put characters back onto the input stream after we've read them with *stdin.getc*, we can peek ahead at the next

4. The HLA Standard Library routines actually buffer up input lines in a string and process characters out of the string. This makes it easy to "peek" ahead one character when looking for a delimiter to end the input value. Your code can also do this, however, the code in this chapter will use a different approach.

character on the input stream and base our logic on that character's value. A corrected version of the previous algorithm might be the following:

```
static
  Delimiters: cset;
  .
  .
  .
conv.getDelimiters( Delimiters );

// Skip over leading delimiters in the string:

while( stdin.peekc() in Delimiters ) do

  // If at the end of the input buffer, we must explicitly read a
  // new line of text from the user.  stdin.peekc does not do this
  // for us.

  if( al = #0 ) then

    stdin.ReadLn();

  else

    stdin.getc(); // Remove delimiter from the input stream.

  endif;

endwhile;
while( stdin.peekc in {'0'..'9'}) do

  stdin.getc(); // Remove the input character from the input stream.

  // Convert character in AL to numeric representation and
  // accumulate result...

endwhile;
if( al not in Delimiters ) then

  raise( ex.ConversionError );

endif;
```

Note that the call to *stdin.peekc* in the second WHILE does not consume the delimiter character when the expression evaluates false. Hence, the delimiter character will be the next character read after this algorithm finishes.

The only remaining comment to make about numeric input is to point out that the HLA Standard Library input routines allow arbitrary underscores to appear within a numeric string. The input routines ignore these underscore characters. This allows the user to input strings like "FFFF_F012" and "1_023_596" which are a little more readable than "FFFFF012" or "1023596". To allow underscores (or any other symbol you choose) within a numeric input routine is quite simple; just modify the second WHILE loop above as follows:

```
while( stdin.peekc in {'0'..'9', '_'}) do

  stdin.getc(); // Read the character from the input stream.

  // Ignore underscores while processing numeric input.

  if( al <> '_' ) then
```

```

        // Convert character in AL to numeric representation and
        // accumulate result...

    endif;

endwhile;

```

4.2.13.6 Extended Precision Hexadecimal Input

As was the case for numeric output, hexadecimal input is the easiest numeric input routine to write. The basic algorithm for hexadecimal string to numeric conversion is the following:

- Initialize the extended precision value to zero.
- For each input character that is a valid hexadecimal digit, do the following:
 - Convert the hexadecimal character to a value in the range 0..15 (\$0..\$F).
 - If the H.O. four bits of the extended precision value are non-zero, raise an exception.
 - Multiply the current extended precision value by 16 (i.e., shift left four bits).
 - Add the converted hexadecimal digit value to the accumulator.
- Check the last input character to ensure it is a valid delimiter. Raise an exception if it is not.

The following program implements this extended precision hexadecimal input routine for 128-bit values.

```

program Xin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    b128: dword[4];

procedure getb128( var inValue:b128 ); nodisplay;
const
    HexChars := {'0'..'9', 'a'..'f', 'A'..'F', '_' };
var
    Delimiters: cset;
    LocalValue: b128;

begin getb128;

    push( eax );
    push( ebx );

    // Get a copy of the HLA standard numeric input delimiters:

    conv.getDelimiters( Delimiters );

    // Initialize the numeric input value to zero:

    xor( eax, eax );
    mov( eax, LocalValue[0] );
    mov( eax, LocalValue[4] );
    mov( eax, LocalValue[8] );

```

```

mov( eax, LocalValue[12] );

// By default, #0 is a member of the HLA Delimiters
// character set. However, someone may have called
// conv.setDelimiters and removed this character
// from the internal Delimiters character set. This
// algorithm depends upon #0 being in the Delimiters
// character set, so let's add that character in
// at this point just to be sure.

cs.unionChar( #0, Delimiters );

// If we're at the end of the current input
// line (or the program has yet to read any input),
// for the input of an actual character.

if( stdin.peekc() = #0 ) then

    stdin.ReadLn();

endif;

// Skip the delimiters found on input. This code is
// somewhat convoluted because stdin.peekc does not
// force the input of a new line of text if the current
// input buffer is empty. We have to force that input
// ourselves in the event the input buffer is empty.

while( stdin.peekc() in Delimiters ) do

    // If we're at the end of the line, read a new line
    // of text from the user; otherwise, remove the
    // delimiter character from the input stream.

    if( al = #0 ) then

        stdin.ReadLn(); // Force a new input line.

    else

        stdin.getc(); // Remove the delimiter from the input buffer.

    endif;

endwhile;

// Read the hexadecimal input characters and convert
// them to the internal representation:

while( stdin.peekc() in HexChars ) do

    // Actually read the character to remove it from the
    // input buffer.

    stdin.getc();

    // Ignore underscores, process everything else.

```

```

if( al <> '_' ) then

    if( al in '0'..'9' ) then

        and( $f, al ); // '0'..'9' -> 0..9

    else

        and( $f, al ); // 'a'/'A'..'f'/'F' -> 1..6
        add( 9, al ); // 1..6 -> 10..15

    endif;

    // Conversion algorithm is the following:
    //
    // (1) LocalValue := LocalValue * 16.
    // (2) LocalValue := LocalValue + al
    //
    // Note that "* 16" is easily accomplished by
    // shifting LocalValue to the left four bits.
    //
    // Overflow occurs if the H.O. four bits of LocalValue
    // contain a non-zero value prior to this operation.

    // First, check for overflow:

    test( $F0, (type byte LocalValue[15]));
    if( @nz ) then

        raise( ex.ValueOutOfRange );

    endif;

    // Now multiply LocalValue by 16 and add in
    // the current hexadecimal digit (in EAX).

    mov( LocalValue[8], ebx );
    shld( 4, ebx, LocalValue[12] );
    mov( LocalValue[4], ebx );
    shld( 4, ebx, LocalValue[8] );
    mov( LocalValue[0], ebx );
    shld( 4, ebx, LocalValue[4] );
    shl( 4, ebx );
    add( eax, ebx );
    mov( ebx, LocalValue[0] );

    endif;

endwhile;

// Okay, we've encountered a non-hexadecimal character.
// Let's make sure it's a valid delimiter character.
// Raise the ex.ConversionError exception if it's invalid.

if( al not in Delimiters ) then

    raise( ex.ConversionError );

endif;

// Okay, this conversion has been a success. Let's store

```

```

    // away the converted value into the output parameter.

    mov( inValue, ebx );
    mov( LocalValue[0], eax );
    mov( eax, [ebx] );

    mov( LocalValue[4], eax );
    mov( eax, [ebx+4] );

    mov( LocalValue[8], eax );
    mov( eax, [ebx+8] );

    mov( LocalValue[12], eax );
    mov( eax, [ebx+12] );

    pop( ebx );
    pop( eax );

end getb128;

// Code to test the routines above:

static
    b1:b128;

begin Xin128;

    stdout.put( "Input a 128-bit hexadecimal value: " );
    getb128( b1 );
    stdout.put
    (
        "The value is: $",
        b1[12], ' ',
        b1[8], ' ',
        b1[4], ' ',
        b1[0],
        nl
    );

end Xin128;

```

Program 4.5 Extended Precision Hexadecimal Input

Extending this code to handle objects that are not 128 bits long is very easy. There are only three changes necessary: you must zero out the whole object at the beginning of the `getb128` routine; when checking for overflow (the `"test($F, (type byte LocalValue[15]);"` instruction) you must test the H.O. four bits of the new object you're processing; and you must modify the code that multiplies `LocalValue` by 16 (via `SHLD`) so that it multiplies your object by 16 (i.e., shifts it to the left four bits).

4.2.13.7 Extended Precision Unsigned Decimal Input

The algorithm for extended precision unsigned decimal input is nearly identical to that for hexadecimal input. In fact, the only difference (beyond only accepting decimal digits) is that you multiply the extended

precision value by 10 rather than 16 for each input character (in general, the algorithm is the same for any base; just multiply the accumulating value by the input base). The following code demonstrates how to write a 128-bit unsigned decimal input routine.

```

program Uin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    u128: dword[4];

procedure getu128( var inValue:u128 ); nodisplay;
var
    Delimiters: cset;
    LocalValue: u128;
    PartialSum: u128;

begin getu128;

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

    // Get a copy of the HLA standard numeric input delimiters:

    conv.getDelimiters( Delimiters );

    // Initialize the numeric input value to zero:

    xor( eax, eax );
    mov( eax, LocalValue[0] );
    mov( eax, LocalValue[4] );
    mov( eax, LocalValue[8] );
    mov( eax, LocalValue[12] );

    // By default, #0 is a member of the HLA Delimiters
    // character set. However, someone may have called
    // conv.setDelimiters and removed this character
    // from the internal Delimiters character set. This
    // algorithm depends upon #0 being in the Delimiters
    // character set, so let's add that character in
    // at this point just to be sure.

    cs.unionChar( #0, Delimiters );

    // If we're at the end of the current input
    // line (or the program has yet to read any input),
    // for the input of an actual character.

    if( stdin.peekc() = #0 ) then

        stdin.ReadLn();

```

```

endif;

// Skip the delimiters found on input. This code is
// somewhat convoluted because stdin.peekc does not
// force the input of a new line of text if the current
// input buffer is empty. We have to force that input
// ourselves in the event the input buffer is empty.

while( stdin.peekc() in Delimiters ) do

    // If we're at the end of the line, read a new line
    // of text from the user; otherwise, remove the
    // delimiter character from the input stream.

    if( al = #0 ) then

        stdin.ReadLn(); // Force a new input line.

    else

        stdin.getc(); // Remove the delimiter from the input buffer.

    endif;

endwhile;

// Read the decimal input characters and convert
// them to the internal representation:

while( stdin.peekc() in '0'..'9' ) do

    // Actually read the character to remove it from the
    // input buffer.

    stdin.getc();

    // Ignore underscores, process everything else.

    if( al <> '_' ) then

        and( $f, al ); // '0'..'9' -> 0..9
        mov( eax, PartialSum[0] ); // Save to add in later.

        // Conversion algorithm is the following:
        //
        // (1) LocalValue := LocalValue * 10.
        // (2) LocalValue := LocalValue + al
        //
        // First, multiply LocalValue by 10:

        mov( 10, eax );
        mul( LocalValue[0], eax );
        mov( eax, LocalValue[0] );
        mov( edx, PartialSum[4] );

        mov( 10, eax );
        mul( LocalValue[4], eax );
        mov( eax, LocalValue[4] );
        mov( edx, PartialSum[8] );
    endif;
endwhile;

```

```

    mov( 10, eax );
    mul( LocalValue[8], eax );
    mov( eax, LocalValue[8] );
    mov( edx, PartialSum[12] );

    mov( 10, eax );
    mul( LocalValue[12], eax );
    mov( eax, LocalValue[12] );

    // Check for overflow. This occurs if EDX
    // contains a none zero value.

    if( edx /* <> 0 */ ) then

        raise( ex.ValueOutOfRange );

    endif;

    // Add in the partial sums (including the
    // most recently converted character).

    mov( PartialSum[0], eax );
    add( eax, LocalValue[0] );

    mov( PartialSum[4], eax );
    adc( eax, LocalValue[4] );

    mov( PartialSum[8], eax );
    adc( eax, LocalValue[8] );

    mov( PartialSum[12], eax );
    adc( eax, LocalValue[12] );

    // Another check for overflow. If there
    // was a carry out of the extended precision
    // addition above, we've got overflow.

    if( @c ) then

        raise( ex.ValueOutOfRange );

    endif;

endif;

endwhile;

// Okay, we've encountered a non-decimal character.
// Let's make sure it's a valid delimiter character.
// Raise the ex.ConversionError exception if it's invalid.

if( al not in Delimiters ) then

    raise( ex.ConversionError );

endif;

// Okay, this conversion has been a success. Let's store
// away the converted value into the output parameter.

```

```

    mov( inValue, ebx );
    mov( LocalValue[0], eax );
    mov( eax, [ebx] );

    mov( LocalValue[4], eax );
    mov( eax, [ebx+4] );

    mov( LocalValue[8], eax );
    mov( eax, [ebx+8] );

    mov( LocalValue[12], eax );
    mov( eax, [ebx+12] );

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

end getul28;

// Code to test the routines above:

static
    b1:ul28;

begin Uin128;

    stdout.put( "Input a 128-bit decimal value: " );
    getul28( b1 );
    stdout.put
    (
        "The value is: $",
        b1[12], '-',
        b1[8], '-',
        b1[4], '-',
        b1[0],
        nl
    );

end Uin128;

```

Program 4.6 Extended Precision Unsigned Decimal Input

As for hexadecimal input, extending this decimal input to some number of bits beyond 128 is fairly easy. All you need do is modify the code that zeros out the LocalValue variable and the code that multiplies LocalValue by ten (overflow checking is done in this same code, so there are only two spots in this code that require modification).

4.2.13.8 Extended Precision Signed Decimal Input

Once you have an unsigned decimal input routine, writing a signed decimal input routine is easy. The following algorithm describes how to accomplish this:

- Consume any delimiter characters at the beginning of the input stream.

- If the next input character is a minus sign, consume this character and set a flag noting that the number is negative.
- Call the unsigned decimal input routine to convert the rest of the string to an integer.
- Check the return result to make sure it's H.O. bit is clear. Raise the `ex.ValueOutOfRangeException` exception if the H.O. bit of the result is set.
- If the sign flag was set in step two above, negate the result.

The actual code is left as a programming exercise at the end of this volume.

4.3 Operating on Different Sized Operands

Occasionally you may need to compute some value on a pair of operands that are not the same size. For example, you may need to add a word and a double word together or subtract a byte value from a word value. The solution is simple: just extend the smaller operand to the size of the larger operand and then do the operation on two similarly sized operands. For signed operands, you would sign extend the smaller operand to the same size as the larger operand; for unsigned values, you zero extend the smaller operand. This works for any operation, although the following examples demonstrate this for the addition operation.

To extend the smaller operand to the size of the larger operand, use a sign extension or zero extension operation (depending upon whether you're adding signed or unsigned values). Once you've extended the smaller value to the size of the larger, the addition can proceed. Consider the following code that adds a byte value to a word value:

```
static
    var1: byte;
    var2: word;
    .
    .
    .
// Unsigned addition:

    movzx( var1, ax );
    add( var2, ax );

// Signed addition:

    movsx( var1, ax );
    add( var2, ax );
```

In both cases, the byte variable was loaded into the AL register, extended to 16 bits, and then added to the word operand. This code works out really well if you can choose the order of the operations (e.g., adding the eight bit value to the sixteen bit value). Sometimes, you cannot specify the order of the operations. Perhaps the sixteen bit value is already in the AX register and you want to add an eight bit value to it. For unsigned addition, you could use the following code:

```
    mov( var2, ax );      // Load 16 bit value into AX
    .                    // Do some other operations leaving
    .                    // a 16-bit quantity in AX.
    add( var1, al );      // Add in the eight-bit value
    adc( 0, ah );         // Add carry into the H.O. word.
```

The first ADD instruction in this example adds the byte at *var1* to the L.O. byte of the value in the accumulator. The ADC instruction above adds the carry out of the L.O. byte into the H.O. byte of the accumulator. Care must be taken to ensure that this ADC instruction is present. If you leave it out, you may not get the correct result.

Adding an eight bit signed operand to a sixteen bit signed value is a little more difficult. Unfortunately, you cannot add an immediate value (as above) to the H.O. word of AX. This is because the H.O. extension byte can be either \$00 or \$FF. If a register is available, the best thing to do is the following:

```
    mov( ax, bx );        // BX is the available register.
```

```
movsx( var1, ax );
add( bx, ax );
```

If an extra register is not available, you might try the following code:

```
push( ax );           // Save word value.
movsx( var1, ax );    // Sign extend 8-bit operand to 16 bits.
add( [esp], ax );     // Add in previous word value
add( 2, esp );        // Pop junk from stack
```

Another alternative is to store the 16 bit value in the accumulator into a memory location and then proceed as before:

```
mov( ax, temp );
movsx( var1, ax );
add( temp, ax );
```

All the examples above added a byte value to a word value. By zero or sign extending the smaller operand to the size of the larger operand, you can easily add any two different sized variables together.

As a last example, consider adding an eight bit signed value to a quadword (64 bit) value:

```
static
QVal:qword;
BVal:int8;
.
.
.
movsx( BVal, eax );
cdq();
add( (type dword QVal), eax );
adc( (type dword QVal[4]), edx );
```

4.4 Decimal Arithmetic

The 80x86 CPUs use the binary numbering system for their native internal representation. The binary numbering system is, by far, the most common numbering system in use in computer systems today. In days long since past, however, there were computer systems that were based on the decimal (base 10) numbering system rather than the binary numbering system. Consequently, their arithmetic system was decimal based rather than binary. Such computer systems were very popular in systems targeted for business/commercial systems⁵. Although systems designers have discovered that binary arithmetic is almost always better than decimal arithmetic for general calculations, the myth still persists that decimal arithmetic is better for money calculations than binary arithmetic. Therefore, many software systems still specify the use of decimal arithmetic in their calculations (not to mention that there is lots of legacy code out there whose algorithms are only stable if they use decimal arithmetic). Therefore, despite the fact that decimal arithmetic is generally inferior to binary arithmetic, the need for decimal arithmetic still persists.

Of course, the 80x86 is not a decimal computer; therefore we have to play tricks in order to represent decimal numbers using the native binary format. The most common technique, even employed by most so-called decimal computers, is to use the *binary coded decimal*, or BCD representation. The BCD representation (see “Nibbles” on page 46) uses four bits to represent the 10 possible decimal digits. The binary value of those four bits is equal to the corresponding decimal value in the range 0..9. Of course, with four bits we can actually represent 16 different values. The BCD format ignores the remaining six bit combinations.

5. In fact, until the release of the IBM 360 in the middle 1960's, most scientific computer systems were binary based while most commercial/business systems were decimal based. IBM pushed their system\360 as a single purpose solution for both business and scientific applications. Indeed, the model designation (360) was derived from the 360 degrees on a compass so as to suggest that the system\360 was suitable for computations "at all points of the compass" (i.e., business and scientific).

Table 1: Binary Code Decimal (BCD) Representation

BCD Representation	Decimal Equivalent
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	Illegal
1011	Illegal
1100	Illegal
1101	Illegal
1110	Illegal
1111	Illegal

Since each BCD digit requires four bits, we can represent a two-digit BCD value with a single byte. This means that we can represent the decimal values in the range 0..99 using a single byte (versus 0..255 if we treat the value as an unsigned binary number). Clearly it takes a bit more memory to represent the same value in BCD as it does to represent the same value in binary. For example, with a 32-bit value you can represent BCD values in the range 0..99,999,999 (eight significant digits) but you can represent values in the range 0..4,294,967,295 (better than nine significant digits) using the binary representation.

Note only does the BCD format waste memory on a binary computer (since it uses more bits to represent a given integer value), but decimal arithmetic is slower. For these reasons, you should avoid the use of decimal arithmetic unless it is absolutely mandated for a given application.

Binary coded decimal representation does offer one big advantage over binary representation: it is fairly trivial to convert between the string representation of a decimal number and the BCD representation. This feature is particularly beneficial when working with fractional values since fixed and floating point binary representations cannot exactly represent many commonly used values between zero and one (e.g., $1/10$). Therefore, BCD operations can be efficient when reading from a BCD device, doing a simple arithmetic operation (e.g., a single addition) and then writing the BCD value to some other device.

4.4.1 Literal BCD Constants

HLA does not provide, nor do you need, a special literal BCD constant. Since BCD is just a special form of hexadecimal notation that does not allow the values \$A..\$F, you can easily create BCD constants using HLA's hexadecimal notation. Of course, you must take care not to include the symbols 'A'..'F' in a

BCD constant since they are illegal BCD values. As an example, consider the following MOV instruction that copies the BCD value '99' into the AL register:

```
mov( $99, al );
```

The important thing to keep in mind is that you must not use HLA literal decimal constants for BCD values. That is, "mov(95, al);" does not load the BCD representation for ninety-five into the AL register. Instead, it loads \$5F into AL and that's an illegal BCD value. Any computations you attempt with illegal BCD values will produce garbage results. Always remember that, even though it seems counter-intuitive, you use hexadecimal literal constants to represent literal BCD values.

4.4.2 The 80x86 DAA and DAS Instructions

The integer unit on the 80x86 does not directly support BCD arithmetic. Instead, the 80x86 requires that you perform the computation using binary arithmetic and use some auxiliary instructions to convert the binary result to BCD. To support packed BCD addition and subtraction with two digits per byte, the 80x86 provides two instructions: decimal adjust after addition (DAA) and decimal adjust after subtraction (DAS). You would execute these two instructions immediately after an ADD/ADC or SUB/SBB instruction to correct the binary result in the AL register.

Two add a pair of two-digit (i.e., single-byte) BCD values together, you would use the following sequence:

```
mov( bcd_1, al );    // Assume that bcd1 and bcd2 both contain
add( bcd_2, al );    // value BCD values.
daa();
```

The first two instructions above add the two byte values together using standard binary arithmetic. This may not produce a correct BCD result. For example, if *bcd_1* contains \$9 and *bcd_2* contains \$1, then the first two instructions above will produce the binary sum \$A instead of the correct BCD result \$10. The DAA instruction corrects this invalid result. It checks to see if there was a carry out of the low order BCD digit and adjusts the value (by adding six to it) if there was an overflow. After adjusting for overflow out of the L.O. digit, the DAA instruction repeats this process for the H.O. digit. DAA sets the carry flag if there was a (decimal) carry out of the H.O. digit of the operation.

The DAA instruction only operates on the AL register. It will not adjust (properly) for a decimal addition if you attempt to add a value to AX, EAX, or any other register. Specifically note that DAA limits you to adding two decimal digits (a single byte) at a time. This means that for the purposes of computing decimal sums, you have to treat the 80x86 as though it were an eight-bit processor, capable of adding only eight bits at a time. If you wish to add more than two digits together, you must treat this as a multiprecision operation. For example, to add four decimal digits together (using DAA), you must execute a sequence like the following:

```
// Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"

mov( bcd_1[0], al );
add( bcd_2[0], al );
daa();
mov( al, bcd_3[0] );
mov( bcd_1[1], al );
adc( bcd_2[1], al );
daa();
mov( al, bcd_3[1], al );

// Carry is set at this point if there was unsigned overflow.
```

Since a binary addition of a word requires only three instructions, you can see why decimal arithmetic is so expensive⁶.

The DAS (decimal adjust after subtraction) adjusts the decimal result after a binary SUB or SBB instruction. You use it the same way you use the DAA instruction. Examples:

```
// Two-digit (one byte) decimal subtraction:

mov( bcd_1, al );    // Assume that bcd1 and bcd2 both contain
sub( bcd_2, al );    // value BCD values.
das();

// Four-digit (two-byte) decimal subtraction.
// Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"

mov( bcd_1[0], al );
sub( bcd_2[0], al );
das();
mov( al, bcd_3[0] );
mov( bcd_1[1], al );
sbb( bcd_2[1], al );
das();
mov( al, bcd_3[1], al );

// Carry is set at this point if there was unsigned overflow.
```

Unfortunately, the 80x86 only provides support for addition and subtraction of packed BCD values using the DAA and DAS instructions. It does not support multiplication, division, or any other arithmetic operations. Because decimal arithmetic using these instructions is so limited, you'll rarely see any programs use these instructions.

4.4.3 The 80x86 AAA, AAS, AAM, and AAD Instructions

In addition to the *packed decimal* instructions (DAA and DAS), the 80x86 CPUs support four *unpacked decimal* adjustment instructions. Unpacked decimal numbers store only one digit per eight-bit byte. As you can imagine, this data representation scheme wastes a considerable amount of memory. However, the unpacked decimal adjustment instructions support the multiplication and division operations, so they are marginally more useful.

The instruction mnemonics AAA, AAS, AAM, and AAD stand for "ASCII adjust for Addition, Subtraction, Multiplication, and Division" (respectively). Despite their name, these instructions do not process ASCII characters. Instead, they support an unpacked decimal value in AL whose L.O. four bits contain the decimal digit and the H.O. four bits contain zero. Note, though, that you can easily convert an ASCII decimal digit character to an unpacked decimal number by simply ANDing AL with the value \$0F.

The AAA instruction adjusts the result of a binary addition of two unpacked decimal numbers. If the addition of those two values exceeds 10, then AAA will subtract 10 from AL and increment AH by one (as well as set the carry flag). AAA assumes that the two values you add together were legal unpacked decimal values. Other than the fact that AAA works with only one decimal digit at a time (rather than two), you use it the same way you use the DAA instruction. Of course, if you need to add together a string of decimal digits, using unpacked decimal arithmetic will require twice as many operations and, therefore, twice the execution time.

You use the AAS instruction the same way you use the DAS instruction except, of course, it operates on unpacked decimal values rather than packed decimal values. As for AAA, AAS will require twice the number of operations to add the same number of decimal digits as the DAS instruction. If you're wondering why anyone would want to use the AAA or AAS instructions, keep in mind that the unpacked format supports multiplication and division, while the packed format does not. Since packing and unpacking the data is usually more expensive than working on the data a digit at a time, the AAA and AAS instruction are more efficient if you have to work with unpacked data (because of the need for multiplication and division).

6. You'll also soon see that it's rare to find decimal arithmetic done this way. So it hardly matters.

The AAM instruction modifies the result in the AX register to produce a correct unpacked decimal result after multiplying two unpacked decimal digits using the MUL instruction. Because the largest product you may obtain is 81 (9*9 produces the largest possible product of two single digit values), the result will fit in the AL register. AAM unpacks the binary result by dividing it by 10, leaving the quotient (H.O. digit) in AH and the remainder (L.O. digit) in AL. Note that AAM leaves the quotient and remainder in different registers than a standard eight-bit DIV operation.

Technically, you do not have to use the AAM instruction immediately after a multiply. AAM simply divides AL by ten and leaves the quotient and remainder in AH and AL (respectively). If you have need of this particular operation, you may use the AAM instruction for this purpose (indeed, that's about the only use for AAM in most programs these days).

If you need to multiply more than two unpacked decimal digits together using MUL and AAM, you will need to devise a multiprecision multiplication that uses the manual algorithm from earlier in this chapter. Since that is a lot of work, this section will not present that algorithm. If you need a multiprecision decimal multiplication, see the next section; it presents a better solution.

The AAD instruction, as you might expect, adjusts a value for unpacked decimal division. The unusual thing about this instruction is that you must execute it *before* a DIV operation. It assumes that AL contains the least significant digit of a two-digit value and AH contains the most significant digit of a two-digit unpacked decimal value. It converts these two numbers to binary so that a standard DIV instruction will produce the correct unpacked decimal result. Like AAM, this instruction is nearly useless for its intended purpose as extended precision operations (e.g., division of more than one or two digits) are extremely inefficient. However, this instruction is actually quite useful in its own right. It computes $AX = AH * 10 + AL$ (assuming that AH and AL contain single digit decimal values). You can use this instruction to easily convert a two-character string containing the ASCII representation of a value in the range 0..99 to a binary value. E.g.,

```
mov( '9', al );
mov( '9', ah );    // "99" is in AH:AL.
and( $0F0F, ax );  // Convert from ASCII to unpacked decimal.
aad();             // After this, AX contains 99.
```

The decimal and ASCII adjust instructions provide an extremely poor implementation of decimal arithmetic. To better support decimal arithmetic on 80x86 systems, Intel incorporated decimal operations into the FPU. The next section discusses how to use the FPU for this purpose. However, even with FPU support, decimal arithmetic is inefficient and less precise than binary arithmetic. Therefore, you should carefully consider whether you really need to use decimal arithmetic before incorporating it into your programs.

4.4.4 Packed Decimal Arithmetic Using the FPU

To improve the performance of applications that rely on decimal arithmetic, Intel incorporated support for decimal arithmetic directly into the FPU. Unlike the packed and unpacked decimal formats of the previous sections, the FPU easily supports values with up to 18 decimal digits of precision, all at FPU speeds. Furthermore, all the arithmetic capabilities of the FPU (e.g., transcendental operations) are available in addition to addition, subtraction, multiplication, and division. Assuming you can live with *only* 18 digits of precision and a few other restrictions, decimal arithmetic on the FPU is the right way to go if you must use decimal arithmetic in your programs.

The first fact you must note when using the FPU is that it doesn't really support decimal arithmetic. Instead, the FPU provides two instructions, FBLD and FBSTP, that convert between packed decimal and binary floating point formats when moving data to and from the FPU. The FBLD (float/BCD load) instruction loads an 80-bit packed BCD value onto the top of the FPU stack after converting that BCD value to the IEEE binary floating point format. Likewise, the FBSTP (float/BCD store and pop) instruction pops the floating point value off the top of stack, converts it to a packed BCD value, and stores the BCD value into the destination memory location.

Once you load a packed BCD value into the FPU, it is no longer BCD. It's just a floating point value. This presents the first restriction on the use of the FPU as a decimal integer processor: calculations are done

using binary arithmetic. If you have an algorithm that absolutely positively depends upon the use of decimal arithmetic, it may fail if you use the FPU to implement it⁷.

The second limitation is that the FPU supports only one BCD data type: a ten-byte 18-digit packed decimal value. It will not support smaller values nor will it support larger values. Since 18 digits is usually sufficient and memory is cheap, this isn't a big restriction.

A third consideration is that the conversion between packed BCD and the floating point format is not a cheap operation. The FBLD and FBSTP instructions can be quite slow (more than two orders of magnitude slower than FLD and FSTP, for example). Therefore, these instructions can be costly if you're doing simple additions or subtractions; the cost of conversion far outweighs the time spent adding the values a byte at a time using the DAA and DAS instructions (multiplication and division, however, are going to be faster on the FPU).

You may be wondering why the FPU's packed decimal format only supports 18 digits. After all, with ten bytes it should be possible to represent 20 BCD digits. As it turns out, the FPU's packed decimal format uses the first nine bytes to hold the packed BCD value in a standard packed decimal format (the first byte contains the two L.O. digits and the ninth byte holds the H.O. two digits). The H.O. bit of the tenth byte holds the sign bit and the FPU ignores the remaining bits in the tenth byte. If you're wondering why Intel didn't squeeze in one more digit (i.e., use the L.O. four bits of the tenth byte to allow for 19 digits of precision), just keep in mind that doing so would create some possible BCD values that the FPU could not exactly represent in the native floating point format. Hence the limitation to 18 digits.

The FPU uses a one's complement notation for negative BCD values. That is, the sign bit contains a one if the number is negative or zero and it contains a zero if the number is positive or zero (like the binary one's complement format, there are two distinct representations for zero).

HLA's *tbyte* type is the standard data type you would use to define packed BCD variables. The FBLD and FBSTP instructions require a *tbyte* operand. Unfortunately, the current version of HLA does not let you (directly) provide an initializer for a *tbyte* variable. There are several solutions to this problem, the next chapter will present some of them. One solution is to put any *tbyte* objects you need to initialize in the DATA section of your program. For example, consider the following code fragment:

```
data
    tbyteObject: tbyte;
                byte $21, $43, $65, 0, 0, 0, 0, 0, 0, 0;
```

The *tbyteObject* declaration tells HLA that this is a *tbyte* object but does not explicitly set aside any space for the variable (see "The Data and Static Sections" on page 159). The following BYTE directive sets aside ten bytes of storage and initializes these ten bytes with the value \$654321 (remember that the 80x86 organizes data from the L.O. byte to the H.O. byte in memory). While this scheme is inelegant, it will get the job done. The next chapter on Macros and the Compile-Time Language will discuss a better way to initialize *tbyte* and *qword* data.

Because the FPU converts packed decimal values to the internal floating point format, you can mix packed decimal, floating point, and (binary) integer formats in the same calculation. The following program demonstrate how you might achieve this:

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

data
    tb: tbyte;
        byte $21,$43,$65,0,0,0,0,0,0,0;

begin MixedArithmetic;

    fbld( tb );        // BCD value
```

7. An example of such an algorithm might be a multiplication by ten by shifting the number one digit to the left. However, such operations are not possible within the FPU itself, so algorithms that misbehave inside the FPU are actually quite rare.

```

    fmul( 2.0 );    // Real value
    fiadd( 1 );    // Integer value
    fbstp( tb );
    stdout.put( "bcd value is " );
    stdout.puttb( tb );
    stdout.newln();

end MixedArithmetic;

```

Program 4.7 Mixed Mode FPU Arithmetic

The FPU treats packed decimal values as integer values. Therefore, if your calculations produce fractional results, the FBSTP instruction will round the result according to the current FPU rounding mode. If you need to work with fractional values, you need to stick with floating point results.

4.5 Sample Program

The sample program for this chapter demonstrates BCD I/O. The following program provides two procedures, BCDin and BCDout. These two procedures read an 18-digit BCD value from the user (with possible leading minus sign) and write a BCD value to the standard output device.

```

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

// The following is equivalent to TBYTE except it
// lets us easily gain access to the individual
// components of a BCD value.

type
    bcd:record

        LO8:    dword;
        MID8:   dword;
        HO2:    byte;
        Sign:   byte;

    endrecord;

// BCDin-
//
// This function reads a BCD value from the standard input
// device. The number can be up to 18 decimal digits long
// and may contain a leading minus sign.
//
// This procedure stores the BCD value in the variable passed
// by reference as a parameter to this routine.

procedure BCDin( var input:tbyte ); nodisplay;
var

```

```

bcdVal:      bcd;
delimiters: cset;

begin BCDin;

    push( eax );
    push( ebx );

    // Get a copy of the input delimiter characters and
    // make sure that #0 is a member of this set.

    conv.getDelimiters( delimiters );
    cs.unionChar( #0, delimiters );

    // Skip over any leading delimiter characters in the text:

    while( stdin.peekc() in delimiters ) do

        // If we're at the end of an input line, read a new
        // line of text from the user, otherwise remove the
        // delimiter character from the input stream.

        if( stdin.peekc() = #0 ) then

            stdin.ReadLn(); // Get a new line of input text.

        else

            stdin.getc();    // Remove the delimiter.

        endif;

    endwhile;

    // Initialize our input accumulator to zero:

    xor( eax, eax );
    mov( eax, bcdVal.LO8 );
    mov( eax, bcdVal.MID8 );
    mov( al, bcdVal.HO2 );
    mov( al, bcdVal.Sign );

    // If the first character is a minus sign, then eat it and
    // set the sign bit to one.

    if( stdin.peekc() = '-' ) then

        stdin.getc();          // Eat the sign character.
        mov( $80, bcdVal.Sign ); // Make this number negative.

    endif;

    // We must have at least one decimal digit in this number:

    if( stdin.peekc() not in '0'..'9' ) then

        raise( ex.ConversionError );

    endif;

```

```

// Okay, read in up to 18 decimal digits:

while( stdin.peekc() in '0'..'9' ) do

    stdin.getc();    // Read this decimal digit.
    shl( 4, al );    // Move digit to H.O. bits of AL

    mov( 4, ebx );
    repeat

        // Cheesy way to SHL bcdVal by four bits and
        // merge in the new character.

        shl( 1, al );
        rcl( 1, bcdVal.LO8 );
        rcl( 1, bcdVal.MID8 );
        rcl( 1, bcdVal.HO2 );

        // If the user has entered more than 18
        // decimal digits, the carry will be set
        // after the RCL above.  Test that here.

        if( @c ) then

            raise( ex.ValueOutOfRange );

        endif;
        dec( ebx );

    until( @z );

endwhile;

// Be sure that the number ends with a proper delimiter:

if( stdin.peekc() not in delimiters ) then

    raise( ex.ConversionError );

endif;

// Okay, store the ten-byte input result into
// the location specified by the parameter.

mov( input, ebx );
mov( bcdVal.LO8, eax );
mov( eax, [ebx] );
mov( bcdVal.MID8, eax );
mov( eax, [ebx+4] );
mov( (type word bcdVal.HO2), ax ); // Grabs "Sign" too.
mov( ax, [ebx+8] );

pop( ebx );
pop( eax );

end BCDin;

```

```

// BCDout-
//
// The converse of the above. Prints the string representation
// of the packed BCD value to the standard output device.

procedure BCDout( output:tbyte ); nodisplay;
var
    q:qword;

begin BCDout;

    // This code cheats *big time*.
    // It converts the BCD value to a 64-bit integer
    // and then calls the stdout.puti64 routine to
    // actually print the number. In theory, this is
    // a whole lot slower than converting the BCD value
    // to ASCII and printing the ASCII chars, however,
    // I/O is so much slower than the conversion that
    // no one will notice the extra time.

    fbld( output );
    fistp( q );
    stdout.puti64( q );

end BCDout;

static
    tb1:    tbyte;
    tb2:    tbyte;
    tbRslt: tbyte;

begin bcdIO;

    stdout.put( "Enter a BCD value: " );
    BCDin( tb1 );
    stdout.put( "Enter a second BCD value: " );
    BCDin( tb2 );

    fbld( tb1 );
    fbld( tb2 );
    fadd();
    fbstp( tbRslt );

    stdout.put( "The sum of " );
    BCDout( tb1 );
    stdout.put( " + " );
    BCDout( tb2 );
    stdout.put( " is " );
    BCDout( tbRslt );
    stdout.newln();

end bcdIO;

```

Program 4.8 BCD I/O Sample Program

4.6 Putting It All Together

Extended precision arithmetic is one of those activities where assembly language truly shines. It's much easier to perform extended precision arithmetic in assembly language than in most high level languages; it's far more efficient to do it in assembly language, as well. Extended precision arithmetic was, perhaps, the most important subject that this chapter teaches.

Although extended precision arithmetic and logical calculations are important, what good are extended precision calculations if you can't get the extended precision values in and out of the machine? Therefore, this chapter devotes a good chunk of its space to describing how to write your own extended precision I/O routines. Between the calculations and the I/O this chapter describes, you're set to perform those really hairy calculations you've always dreamed of!

Although decimal arithmetic is nowhere near as prominent as it once was, the need for decimal arithmetic does arise on occasion. Therefore, this chapter spends some time discussing BCD arithmetic on the 80x86.

Bit Manipulation

Chapter Five

5.1 Chapter Overview

Manipulating bits in memory is, perhaps, the thing that assembly language is most famous for. Indeed, one of the reasons people claim that the “C” programming language is a “medium-level” language rather than a high level language is because of the vast array of bit manipulation operators that it provides. Even with this wide array of bit manipulation operations, the C programming language doesn’t provide as complete a set of bit manipulation operations as assembly language.

This chapter will discuss how to manipulate strings of bits in memory and registers using 80x86 assembly language. This chapter begins with a review of the bit manipulation instructions covered thus far in this text and it also introduces a few new instructions. This chapter reviews information on packing and unpacking bit strings in memory since this is the basis for many bit manipulation operations. Finally, this chapter discusses several bit-centric algorithms and their implementation in assembly language.

5.2 What is Bit Data, Anyway?

Before describing how to manipulate bits, it might not be a bad idea to define exactly what this text means by “bit data.” Most readers problem assume that “bit manipulation programs” twiddle individual bits in memory. While programs that do this are definitely “bit manipulation programs,” we’re not going to limit this title to just those programs. For our purposes, bit manipulation refers to working with data types that consist of strings of bits that are non-contiguous or are not an even multiple of eight bits long. Generally, such bit objects will not represent numeric integers, although we will not place this restriction on our bit strings.

A *bit string* is some contiguous sequence of one or more bits (this term even applies if the bit string’s length is an even multiple of eight bits). Note that a bit string does not have to start or end at any special point. For example, a bit string could start in bit seven of one byte in memory and continue through to bit six of the next byte in memory. Likewise, a bit string could begin in bit 30 of EAX, consume the upper two bits of EAX, and then continue from bit zero through bit 17 of EBX. In memory, the bits must be physically contiguous (i.e., the bit numbers are always increasing except when crossing a byte boundary, and at byte boundaries the byte number increases by one). In registers, if a bit string crosses a register boundary, the application defines the continuation register but the bit string always continues in bit zero of that second register.

A *bit set* is a collection of bits, not necessarily contiguous (though it may be), within some larger data structure. For example, bits 0..3, 7, 12, 24, and 31 from some double word object forms a set of bits. Usually, we will limit bit sets to some reasonably sized *container object* (that is, the data structure that encapsulates the bit set), but the definition doesn’t specifically limit the size. Normally, we will deal with bit sets that are part of an object no more than about 32 or 64 bits in size. Note that bit strings are special cases of bit sets.

A *bit run* is a sequence of bits with all the same value. A *run of zeros* is a bit string containing all zeros, a *run of ones* is a bit string containing all ones. The *first set bit* in a bit string is the bit position of the first set bit in a bit string, i.e., the first set bit following a run of zeros (assuming of course, that such a bit even exists). A similar definition exists for the *first clear bit*. The *last set bit* is the last bit position in a bit string containing a one; afterwards, the remainder of the string forms an uninterrupted run of zeros. A similar definition exists for the *last clear bit*.

A *bit offset* is the number of bits from some boundary position (usually a byte boundary) to the specified bit. As noted in Volume One, we’ll number the bits starting from zero at the boundary location. If the offset is less than 32, then the bit offset is the same as the bit number in a byte, word, or double word value.

A *mask* is a sequence of bits that we’ll use to manipulate certain bits in another value. For example, the bit string `%0000_1111_0000`, when used with the AND instruction, can mask away (clear) all the bits except

bits four through seven. Likewise, if you use this same value with the OR instruction, it can force bits four through seven to ones in the destination operand. The term “mask” comes from the use of these bit strings with the AND instruction; in those situations the one and zero bits behave like masking tape when you’re painting something; they pass through certain bits unchanged while masking out the other bits.

Armed with these definitions, we’re ready to start manipulating some bits!

5.3 Instructions That Manipulate Bits

Bit manipulation generally consists of six activities: setting bits, clearing bits, inverting bits, testing and comparing bits, extracting bits from a bit string, and inserting bits into a bit string. By now you should be familiar with most of the instructions we’ll use to perform these operations; their introduction started way back in the earliest chapters of Volume One. Nevertheless, it’s worthwhile to review the old instructions here as well as present the few bit manipulation instructions we’ve yet to consider.

The most basic bit manipulation instructions are the AND, OR, XOR, NOT, TEST, and shift and rotate instructions. Indeed, on the earliest 80x86 processors, these were the only instructions available for bit manipulation. The following paragraphs review these instructions, concentrating on how you could use them to manipulate bits in memory or registers.

The AND instruction provides the ability to strip away unwanted bits from some bit sequence, replacing the unwanted bits with zeros. This instruction is especially useful for isolating a bit string or a bit set that is merged with other, unrelated data (or, at least, data that is not part of the bit string or bit set). For example, suppose that a bit string consumes bit positions 12 through 24 of the EAX register, we can isolate this bit string by setting all other bits in EAX to zero by using the following instruction:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
```

Most programs use the AND instruction to clear bits that are not part of the desired bit string. In theory, you could use the OR instruction to mask all unwanted bits to ones rather than zeros, but later comparisons and operations are often easier if the unneeded bit positions contain zero.

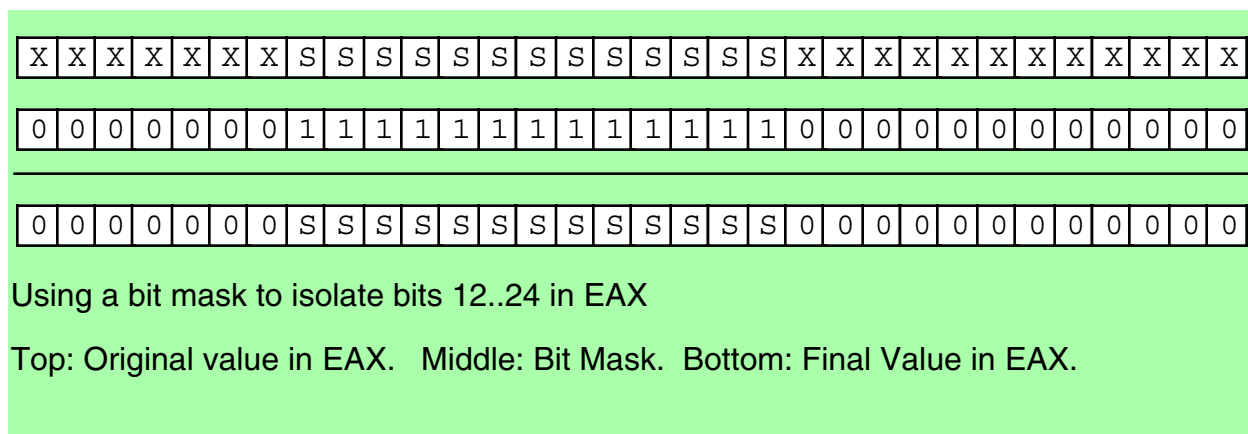


Figure 5.1 Isolating a Bit String Using the AND Instruction

Once you’ve cleared the unneeded bits in a set of bits, you can often operate on the bit set in place. For example, to see if the string of bits in positions 12 through 24 of EAX contain \$12F3 you could use the following code:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, %1_0010_1111_0011_0000_0000_0000 );
```

Here's another solution, using constant expressions, that's a little easier to digest:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, $12F3 << 12 ); // "<12" shifts $12F3 to the left 12 bits.
```

Most of the time, however, you'll want (or need) the bit string aligned with bit zero in EAX prior to any operations you would want to perform. Of course, you can use the SHR instruction to properly align the value after you've masked it:

```

and( %1_1111_1111_1111_0000_0000_0000, eax );
shr( 12, eax );
cmp( eax, $12F3 );
<< Other operations that requires the bit string at bit #0 >>

```

Now that the bit string is aligned with bit zero, the constants and other values you use in conjunction with this value are easier to deal with.

You can also use the OR instruction to mask unwanted bits around a set of bits. However, the OR instruction does not let you clear bits, it allows you to set bits to ones. In some instances setting all the bits around your bit set may be desirable; most software, however, is easier to write if you clear the surrounding bits rather than set them.

The OR instruction is especially useful for inserting a bit set into some other bit string. To do this, there are several steps you must go through:

- Clear all the bits surrounding your bit set in the source operand.
- Clear all the bits in the destination operand where you wish to insert the bit set.
- OR the bit set and destination operand together.

For example, suppose you have a value in bits 0..12 of EAX that you wish to insert into bits 12..24 of EBX without affecting any of the other bits in EBX. You would begin by stripping out bits 13 and above from EAX; then you would strip out bits 12..24 in EBX. Next, you would shift the bits in EAX so the bit string occupies bits 12..24 of EAX. Finally, you would OR the value in EAX into EBX (see Figure 5.2):

```
and( $1FFF, eax );      // Strip all but bits 0..12 from EAX
and( $1FF_F000, ebx );  // Clear bits 12..24 in EBX.
shl( 12, eax );          // Move bits 0..12 to 12..24 in EAX.
or( eax, ebx );          // Merge the bits into EBX.
```

EBX:

[illegible]

EAX :

[illegible]

Step One: Strip the unneeded bits from EAX (the “U” bits)

EBX:

[illegible]

EAX:

[illegible]

Step Two: Mask out the destination bit field in EBX.

Notice the use of the compile-time not operator (“!”) to invert the bit mask in order to clear the bit positions in EBX where the code inserts the bits from EAX. This saves having to create another constant in the program that has to be changed anytime you modify the *BitMask* constant. Having to maintain two separate symbols whose values are dependent on one another is not a good thing in a program.

Of course, in addition to merging one bit set with another, the OR instruction is also useful for forcing bits to one in a bit string. By setting various bits in a source operand to one you can force the corresponding bits in the destination operand to one by using the OR instruction.

The XOR instruction, as you may recall, gives you the ability to invert selected bits belonging to a bit set. Although the need to invert bits isn’t as common as the need to set or clear them, the XOR instruction still sees considerable use in bit manipulation programs. Of course, if you want to invert all the bits in some destination operand, the NOT instruction is probably more appropriate than the XOR instruction; however, to invert selected bits while not affecting others, the XOR is the way to go.

One interesting fact about XOR’s operation is that it lets you manipulate known data in just about any way imaginable. For example, if you know that a field contains %1010 you can force that field to zero by XORing it with %1010. Similarly, you can force it to %1111 by XORing it with %0101. Although this might seem like a waste, since you can easily force this four-bit string to zero or all ones using AND or OR, the XOR instruction has two advantages: (1) you are not limited to forcing the field to all zeros or all ones; you can actually set these bits to any of the 16 valid combination via XOR; (2) if you need to manipulate other bits in the destination operand at the same time, AND/OR may not be able to accommodate you. For example, suppose that you know that one field contains %1010 that you want to force to zero and another field contains %1000 and you wish to increment that field by one (i.e., set the field to %1001). You cannot accomplish both operations with a single AND or OR instruction, but you can do this with a single XOR instruction; just XOR the first field with %1010 and the second field with %0001. Remember, however, that this trick only works if you know the current value of a bit set within the destination operand. Of course, while you’re adjusting the values of bit fields containing known values, you can invert bits in other fields simultaneously.

In addition to setting, clearing, and inverting bits in some destination operand, the AND, OR, and XOR instructions also affect various condition codes in the FLAGS register. These instructions affect the flags as follows:

- These instructions always clear the carry and overflow flags.
- These instructions set the sign flag if the result has a one in the H.O. bit; they clear it otherwise. I.e., these instructions copy the H.O. bit of the result into the sign flag.
- These instructions set/clear the zero flag depending on whether the result is zero.
- These instructions set the parity flag if there are an even number of set bits in the L.O. byte of the destination operand, they clear the parity flag if there are an odd number of one bits in the L.O. byte of the destination operand.

The first thing to note is that these instructions always clear the carry and overflow flags. This means that you cannot expect the system to preserve the state of these two flags across the execution of these instructions. A very common mistake in many assembly language programs is the assumption that these instructions do not affect the carry flag. Many people will execute an instruction that sets/clears the carry flag, execute an AND/OR/XOR instruction, and then attempt to test the state of the carry from the previous instruction. This simply will not work.

One of the more interesting aspects to these instructions is that they copy the H.O. bit of their result into the sign flag. This means that you can easily test the setting of the H.O. bit of the result by testing the sign flag (using SETS/SETNS, JS/JNS, or by using the @S/@NS flags in a boolean expression). For this reason, many assembly language programmers will often place an important boolean variable in the H.O. bit of some operand so they can easily test the state of that bit using the sign flag after a logical operation.

We haven’t talked much about the parity flag in this text. Indeed, earlier volumes have done little more than acknowledge its existence. We’re not going to get into a big discussion of this flag and what you use it for since the primary purpose for this flag has been taken over by hardware¹. However, since this is a chap-

1. Serial communications chips and other communication hardware that uses parity for error checking normally computes the parity in hardware, you don’t have to use software for this purpose.

ter on bit manipulation and parity computation is a bit manipulation operation, it seems only fitting to provide a brief discussion of the parity flag at this time.

Parity is a very simple error detection scheme originally employed by telegraphs and other serial communication schemes. The idea was to count the number of set bits in a character and include an extra bit in the transmission to indicate whether that character contained an even or odd number of set bits. The receiving end of the transmission would also count the bits and verify that the extra “parity” bit indicated a successful transmission. We’re not going to explore the information theory aspects of this error checking scheme at this point other than to point out that the purpose of the parity flag is to help compute the value of this extra bit.

The 80x86 AND, OR, and XOR instructions set the parity bit if the L.O. byte of their operand contains an even number of set bits. An important fact bears repeating here: the parity flag only reflects the number of set bits in the L.O. byte of the destination operand; it does not include the H.O. bytes in a word, double word, or other sized operand. The instruction set only uses the L.O. byte to compute the parity because communication programs that use parity are typically character-oriented transmission systems (there are better error checking schemes if you transmit more than eight bits at a time).

Although the need to know whether the L.O. (or only) byte of some computation has an even or odd number of set bits isn’t common in modern programs, it does come in useful once in a great while. Since this is, intrinsically, a bit operation, it’s worthwhile to mention the use of this flag and how the AND/OR/XOR instructions affect this flag.

The zero flag setting is one of the more important results the AND/OR/XOR instructions produce. Indeed, programs reference this flag so often after the AND instruction that Intel added a separate instruction, TEST, whose main purpose was to logically AND two results and set the flags without otherwise affecting either instruction operand.

There are three main uses of the zero flag after the execution of an AND or TEST instruction: (1) checking to see if a particular bit in an operand is set; (2) checking to see if at least one of several bits in a bit set is one; and (3) checking to see if an operand is zero. Use (1) is actually a special case of (2) where the bit set contains only a single bit. We’ll explore each of these uses in the following paragraphs.

A common use for the AND instruction, and also the original reason for the inclusion of the TEST instruction in the 80x86 instruction set, is to test to see if a particular bit is set in a given operand. To perform this type of test, you would normally AND/TEST a constant value containing a single set bit with the operand you wish to test. These clears all the other bits in the second operand leaving a zero in the bit position under test (the bit position with the single set bit in the constant operand) if the operand contains a zero in that position and leaving a one if the operand contains a one in that position. Since all of the other bits in the result are zero, the entire result will be zero if that particular bit is zero, the entire result will be non-zero if that bit position contains a one. The 80x86 reflects this status in the zero flag (Z=1 indicates a zero bit, Z=0 indicates a one bit). The following instruction sequence demonstrates how to test to see if bit four is set in EAX:

```
test( %1_000, eax ); // Check bit #4 to see if it is 0/1
if( @nz ) then

    << Do this if the bit is set >>

else

    << Do this if the bit is clear >>

endif;
```

You can also use the AND/TEST instructions to see if any one of several bits is set. Simply supply a constant that has one bits in all the positions you want to test (and zeros everywhere else). AND such a value with an unknown quantity will produce a non-zero value if one or more of the bits in the operand under test contain a one. The following example tests to see if the value in EAX contains a one in bit positions one, two, four, and seven:

```
test( %1001_0010, eax );
if( @nz ) then // at least one of the bits is set.
```

```

    << do whatever needs to be done if one of the bits is set >>

endif;

```

Note that you cannot use a single AND or TEST instruction to see if all the corresponding bits in the bit set are equal to one. To accomplish this, you must first mask out the bits that are not in the set and then compare the result against the mask itself. If the result is equal to the mask, then all the bits in the bit set contain ones. You must use the AND instruction for this operation as the TEST instruction does not mask out any bits. The following example checks to see if all the bits corresponding to a value this code calls *bitMask* are equal to one:

```

    and( bitMask, eax );
    cmp( eax, bitMask );
    if( @e ) then

        << All the bit positions in EAX corresponding to the set >>
        << bits in bitMask are equal to one if we get here.      >>

endif;

```

Of course, once we stick the CMP instruction in there, we don't really have to check to see if all the bits in the bit set contain ones. We can check for any combination of values by specifying the appropriate value as the operand to the CMP instruction.

Note that the TEST/AND instructions will only set the zero flag in the above code sequences if all the bits in EAX (or other destination operand) have zeros in the positions where ones appear in the constant operand. This suggests another way to check for all ones in the bit set: invert the value in EAX prior to using the AND or TEST instruction. Then if the zero flag is set, you know that there were all ones in the (original) bit set, e.g.,

```

    not( eax );
    test( bitMask, eax );
    if( @z ) then

        << At this point, EAX contained all ones in the bit positions >>
        << occupied by ones in the bitMask constant.                >>

endif;

```

The paragraphs above all suggest that the *bitMask* (i.e., source operand) is a constant. This was for purposes of example only. In fact, you can use a variable or other register here, if you prefer. Simply load that variable or register with the appropriate bit mask before you execute the TEST, AND, or CMP instructions in the examples above.

Another set of instructions we've already seen that we can use to manipulate bits are the bit test instructions. These instructions include BT (bit test), BTS (bit test and set), BTC (bit test and complement), and BTR (bit test and reset). We've used these instructions to manipulate bits in HLA character set variables, we can also use them to manipulate bits in general. The BTx instructions allow the following syntactical forms:

```

    btx( BitNumber, BitsToTest );

    btx( reg16, reg16 );
    btx( reg32, reg32 );
    btx( constant, reg16 );
    btx( constant, reg32 );

    btx( reg16, mem16 );
    btx( reg32, mem32 );
    btx( constant, mem16 );
    btx( constant, mem32 );

```

The BT instruction's first operand is a bit number that specifies which bit to check in the second operand. If the second operand is a register, then the first operand must contain a value between zero and the size of the register (in bits) minus one; since the 80x86's largest registers are 32 bits, this value have the maximum value 31 (for 32-bit registers). If the second operand is a memory location, then the bit count is not limited to values in the range 0..31. If the first operand is a constant, it can be any eight-bit value in the range 0..255. If the first operand is a register, it has no limitation.

The BT instruction copies the specified bit from the second operand into the carry flag. For example, the "bt(8, eax);" instruction copies bit number eight of the EAX register into the carry flag. You can test the carry flag after this instruction to determine whether bit eight was set or clear in EAX.

In general, the BT instruction is, perhaps, not the best instruction for testing individual bits in a register. The TEST (or AND) instruction is a bit more efficient. These latter two instructions are Intel "RISC Core" instructions while the BT instruction is a "Complex" instruction. Therefore, you will often get better performance using TEST or AND rather than BT. If you want to test bits in memory operands (especially in bit arrays), then the BT instruction is probably a reasonable way to go.

The BTS, BTC, and BTR instructions manipulate the bit they test while they are testing it. These instructions are rather slow and you should avoid them if performance is your primary concern. In a later volume when we discuss semaphores you will see the true purpose for these instructions. Until then, if performance (versus convenience) is an issue, you should always try two different algorithms, one that uses these instructions, one that uses AND/OR instructions, and measure the performance difference; then choose the best of the two different approaches.

The shift and rotate instructions are another group of instructions you can use to manipulate and test bits. Of course, all of these instructions move the H.O. (left shift/rotate) or L.O. (right shift/rotate) bits into the carry flag. Therefore, you can test the carry flag after you execute one of these instructions to determine the original setting of the operand's H.O. or L.O. bit (depending on the direction). Of course, the shift and rotate instructions are invaluable for aligning bit strings, packing, and unpacking data. Volume One has several examples of this and, of course, some earlier examples in the section also use the shift instructions for this purpose.

5.4 The Carry Flag as a Bit Accumulator

The BTx, shift, and rotate instructions all set or clear the carry flag depending on the operation and/or selected bit. Since these instructions place their "bit result" in the carry flag, it is often convenient to think of the carry flag as a one-bit register or accumulator for bit operations. In this section we will explore some of the operations possible with this bit result in the carry flag.

Instructions that will be useful for manipulating bit results in the carry flag are those that use the carry flag as some sort of input value. The following is a sampling of such instructions:

- ADC, SBB
- RCL, RCR
- CMC (we'll throw in CLC and STC even though they don't use the carry as input)
- JC, JNC
- SETC, SETNC

The ADC and SBB instructions add or subtract their operands along with the carry flag. So if you've computed some bit result into the carry flag, you can figure that result into an addition or subtraction using these instructions. This isn't a common operation, but it is available if it's useful to you.

To merge a bit result in the carry flag, you most often use the rotate through carry instructions (RCL and RCR). These instructions, of course, move the carry flag into the L.O. or H.O. bits of their destination operand. These instructions are very useful for packing a set of bit results into a byte, word, or double word value.

The CMC (complement carry) instruction lets you easily invert the result of some bit operation. You can also use the CLC and STC instructions to initialize the carry flag prior to some string of bit operations involving the carry flag.

Of course, instructions that test the carry flag are going to be very popular after a calculation that leaves a bit result in the carry flag. The JC, JNC, SETC, and SETNC instructions are quite useful here. You can also use the HLA @C and @NC operands in a boolean expression to test the result in the carry flag.

If you have a sequence of bit calculations and you would like to test to see if the calculations produce a specific sequence of one-bit results, the easiest way to do this is to clear a register or memory location and use the RCL or RCR instructions to shift each result into that location. Once the bit operations are complete, then you can compare the register or memory location against a constant value to see if you've obtained a particular result. If you want to test a sequence of results involving conjunction and disjunction (i.e., strings of results involving ANDs and ORs) then you could use the SETC and SETNC instruction to set a register to zero or one and then use the AND/OR instructions to merge the results.

5.5 Packing and Unpacking Bit Strings

A common bit operation is inserting a bit string into an operand or extracting a bit string from an operand. Previous chapters in this text have provided simple examples of packing and unpacking such data, now it is time to formally describe how to do this.

For the purposes of the current discussion, we will assume that we're dealing with bit strings; that is, a contiguous sequence of bits. A little later in this chapter we'll take a look at how to extract and insert bit sets in an operand. Another simplification we'll make is that the bit string completely fits within a byte, word, or double word operand. Large bit strings that cross object boundaries require additional processing; a discussion of bit strings that cross double word boundaries appears later in this section.

A bit string has two attributes that we must consider when packing and unpacking that bit string in a larger operand: a starting bit position and a length. The starting bit position is the bit number of the L.O. bit of the bit string in the larger operand. The length, of course, is the number of bits in the operand. To insert (pack) data into a destination operand we will assume that we start with a bit string of the appropriate length that is right-justified (i.e., starts in bit position zero) in an operand and is zero extended to eight, sixteen, or thirty-two bits. The task is to insert this data at the appropriate starting position in some other operand that is eight, sixteen, or thirty-bits wide. There is no guarantee that the destination bit positions contain any particular value.

The first two steps (which can occur in any order) is to clear out the corresponding bits in the destination operand and shift (a copy of) the bit string so that the L.O. bit begins at the appropriate bit position. After completing these two steps, the third step is to OR the shifted result with the destination operand. This inserts the bit string into the destination operand.

Destination:

X	X	X	X	X	X	X	D	D	D	D	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	0	0	0	0	Y	Y	Y	Y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step One: Insert YYYY into the positions occupied by DDDD in the destination operand.
Begin by shifting the source operand to the left five bits.

Destination:

X	X	X	X	X	X	X	X	D	D	D	D	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step Two: Clear out the destination bits using the AND instruction.

Destination:

X	X	X	X	X	X	X	X	0	0	0	0	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step Three: OR the two values together

Destination:

X	X	X	X	X	X	X	X	Y	Y	Y	Y	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Final result appears in the destination operand.

Figure 5.3 Inserting a Bit String Into a Destination Operand

It only takes three instructions to insert a bit string of known length into a destination operand. The following three instructions demonstrate how to handle the insertion operation in Figure 5.3; These instructions assume that the source operand is in BX and the destination operand is AX:

```
shl( 5, bx );
and( %111111000011111, ax );
or( bx, ax );
```

If the length and the starting position aren't known when you're writing the program (that is, you have to calculate them at run time), then bit string insertion is a little more difficult. However, with the use of a lookup table it's still an easy operation to accomplish. Let's assume that we have two eight-bit values: a starting bit position for the field we're inserting and a non-zero eight-bit length value. Also assume that the source operand is in EBX and the destination operand is EAX. The code to insert one operand into another could take the following form:

readonly

```
// The index into the following table specifies the length of the bit string
// at each position:
```

```
MaskByLen: dword[ 32 ] :=
```

```

[
    0, $1, $3, $7, $f, $1f, $3f, $7f,
    $ff, $1ff, $3fff, $7fff, $fff, $1fff, $3fff, $7fff, $ffff,
    $1_ffff, $3_ffff, $7_ffff, $f_ffff,
    $1f_ffff, $3f_ffff, $7f_ffff, $ff_ffff,
    $1ff_ffff, $3ff_ffff, $7ff_ffff, $fff_ffff,
    $1fff_ffff, $3fff_ffff, $7fff_ffff, $ffff_ffff
];
.
.
.
movzx( Length, edx );
mov( MaskByLen[ edx*4 ], edx );
mov( StartingPosition, cl );
shl( cl, edx );
not( edx );
shl( cl, ebx );
and( edx, eax );
or( ebx, eax );

```

Each entry in the *MaskByLen* table contains the number of one bits specified by the index into the table. Using the *Length* value as an index into this table fetches a value that has as many one bits as the *Length* value. The code above fetches an appropriate mask, shifts it to the left so that the L.O. bit of this run of ones matches the starting position of the field into which we want to insert the data, then it inverts the mask and uses the inverted value to clear the appropriate bits in the destination operand.

To extract a bit string from a larger operand is just as easy as inserting a bit string into some larger operand. All you've got to do is mask out the unwanted bits and then shift the result until the L.O. bit of the bit string is in bit zero of the destination operand. For example, to extract the four-bit field starting at bit position five in EBX and leave the result in EAX, you could use the following code:

```

mov( ebx, eax );           // Copy data to destination.
and( %1_1110_0000, ebx ); // Strip unwanted bits.
shr( 5, eax );             // Right justify to bit position zero.

```

If you do not know the bit string's length and starting position when you're writing the program, you can still extract the desired bit string. The code is very similar to insertion (though a tiny bit simpler). Assuming you have the *Length* and *StartingPosition* values we used when inserting a bit string, you can extract the corresponding bit string using the following code (assuming source=EBX and dest=EAX):

```

movzx( Length, edx );
mov( MaskByLen[ edx*4 ], edx );
mov( StartingPosition, cl );
mov( ebx, eax );
shr( cl, eax );
and( edx, eax );

```

The examples up to this point all assume that the bit string appears completely within a double word (or smaller) object. This will always be the case if the bit string is less than or equal to 24 bits in length. However, if the length of the bit string plus its starting position (mod eight) within an object is greater than 32, then the bit string will cross a double word boundary within the object. To extract such bit strings requires up to three operations: one operation to extract the start of the bit string (up to the first double word boundary), an operation that copies whole double words (assuming the bit string is so long that it consumes several double words), and a final operation that copies left-over bits in the last double word at the end of the bit string. The actual implementation of this operation is left as an exercise at the end of this volume.

5.6 Coalescing Bit Sets and Distributing Bit Strings

Inserting and extract bit sets is little different than inserting and extract bit strings if the “shape” of the bit set you’re inserting (or resulting bit set you’re extracting) is the same as the bit set in the main object. The “shape” of a bit set is the distribution of the bits in the set, ignoring the starting bit position of the set. So a bit set that includes bits zero, four, five, six, and seven has the same shape as a bit set that includes bits 12, 16, 17, 18, and 19 since the distribution of the bits is the same. The code to insert or extract this bit set is nearly identical to that of the previous section; the only difference is the mask value you use. For example, to insert this bit set starting at bit number zero in EAX into the corresponding bit set starting at position 12 in EBX, you could use the following code:

```
and( %1111_0001_0000_0000_0000, ebx ); // Mask out destination bits.
shl( 12, eax );                          // Move source bits into posn.
or( eax, ebx );                          // Merge the bit set into EBX.
```

However, suppose you have five bits in bit positions zero through four in EAX and you want to merge them into bits 12, 16, 17, 18, and 19 in EBX. Somehow you’ve got to distribute the bits in EAX prior to logically ORing the values into EBX. Given the fact that this particular bit set has only two runs of one bits, the process is somewhat simplified, the following code achieves this in a somewhat sneaky fashion:

```
and( %1111_0001_0000_0000_0000, ebx );
shl( 3, eax ); // Spread out the bits: 1-4 goes to 4-7 and 0 to 3.
btr( 3, eax ); // Bit 3->carry and then clear bit 3
rcl( 12, eax ); // Shift in carry and put bits into final position
or( eax, ebx ); // Merge the bit set into EBX.
```

This trick with the BTR (bit test and reset) instruction worked well because we only had one bit out of place in the original source operand. Alas, had the bits all been in the wrong location relative to one another, this scheme might not have worked quite as well. We’ll see a more general solution in just a moment.

Extracting this bit set and collecting (“coalescing”) the bits into a bit string is not quite as easy. However, there are still some sneaky tricks we can pull. Consider the following code that extracts the bit set from EBX and places the result into bits 0..4 of EAX:

```
mov( ebx, eax );
and( %1111_0001_0000_0000_0000, eax ); // Strip unwanted bits.
shr( 5, eax );                          // Put bit 12 into bit 7, etc.
shr( 3, ah );                           // Move bits 11..14 to 8..11.
shr( 7, eax );                           // Move down to bit zero.
```

This code moves (original) bit 12 into bit position seven, the H.O. bit of AL. At the same time it moves bits 16..19 down to bits 11..14 (bits 3..6 of AH). Then the code shifts the bits 3..6 in AH down to bit zero. This positions the H.O. bits of the bit set so that they are adjacent to the bit left in AL. Finally, the code shifts all the bits down to bit zero. Again, this is not a general solution, but it shows a clever way to attack this problem if you think about it carefully.

The problem with the coalescing and distribution algorithms above is that they are not general. They apply only to their specific bit sets. In general, specific solutions are going to provide the most efficient solution. A generalized solution (perhaps that lets you specify a mask and the code distributes or coalesces the bits accordingly) is going to be a bit more difficult. The following code demonstrates how to distribute the bits in a bit string according to the values in a bit mask:

```
// EAX- Originally contains some value into which we insert bits from EBX.
// EBX- L.O. bits contain the values to insert into EAX.
// EDX- bitmap with ones indicating the bit positions in EAX to insert.
// CL- Scratchpad register.

mov( 32, cl ); // Count # of bits we rotate.
jmp DistLoop;

CopyToEAX: rcr( 1, ebx ); // Don't use SHR here, must preserve Z-flag.
```

```

        rcr( 1, eax );
        jz  Done;
DistLoop: dec( cl );
        shr( 1, edx );
        jc CopyToEAX;
        ror( 1, eax );    // Keep current bit in EAX.
        jnz DistLoop;

Done:    ror( cl, eax );    // Reposition remaining bits.

```

In the code above, if we load EDX with %1100_1001 then this code will copy bits 0..3 to bits 0, 3, 6, and 7 in EAX. Notice the short circuit test that checks to see if we've exhausted the values in EDX (by checking for a zero in EDX). Note that the rotate instructions do not affect the zero flag while the shift instructions do. Hence the SHR instruction above will set the zero flag when there are no more bits to distribute (i.e., when EDX becomes zero).

The general algorithm for coalescing bits is a tad more efficient than distribution. Here's the code that will extract bits from EBX via the bit mask in EDX and leave the result in EAX:

```

// EAX- Destination register.
// EBX- Source register.
// EDX- Bitmap with ones representing bits to copy to EAX.
// EBX and EDX are not preserved.

        sub( eax, eax );    // Clear destination register.
        jmp ShiftLoop;

ShiftInEAX: rcl( 1, ebx );    // Up here we need to copy a bit from
        rcl( 1, eax );    // EBX to EAX.
ShiftLoop: shl( 1, edx );    // Check mask to see if we need to copy a bit.
        jc ShiftInEAX;    // If carry set, go copy the bit.
        rcl( 1, ebx );    // Current bit is uninteresting, skip it.
        jnz ShiftLoop;    // Repeat as long as there are bits in EDX.

```

This sequence takes advantage of one sneaky trait of the shift and rotate instructions: the shift instructions affect the zero flag while the rotate instructions do not. Therefore, the “shl(1, edx);” instruction sets the zero flag when EDX becomes zero (after the shift). If the carry flag was also set, the code will make one additional pass through the loop in order to shift a bit into EAX, but the next time the code shifts EDX one bit to the left, EDX is still zero and so the carry will be clear. On this iteration, the code falls out of the loop.

Another way to coalesce bits is via table lookup. By grabbing a byte of data at a time (so your tables don't get too large) you can use that byte's value as an index into a lookup table that coalesces all the bits down to bit zero. Finally, you can merge the bits at the low end of each byte together. This might produce a more efficient coalescing algorithm in certain cases. The implementation is left to the reader...

5.7 Packed Arrays of Bit Strings

Although it is far more efficient to create arrays whose elements' have an integral number of bytes, it is quite possible to create arrays of elements whose size is not a multiple of eight bits. The drawback is that calculating the “address” of an array element and manipulating that array element involves a lot of extra work. In this section we'll take a look at a few examples of packing and unpacking array elements in an array whose elements are an arbitrary number of bits long.

Before proceeding, it's probably worthwhile to discuss why you would want to bother with arrays of bit objects. The answer is simple: space. If an object only consumes three bits, you can get 2.67 times as many elements into the same space if you pack the data rather than allocating a whole byte for each object. For

very large arrays, this can be a substantial savings. Of course, the cost of this space savings is speed: you've got to execute extra instructions to pack and unpack the data, thus slowing down access to the data.

The calculation for locating the bit offset of an array element in a large block of bits is almost identical to the standard array access; it is

$$\text{Element_Address_in_bits} = \text{Base_address_in_bits} + \text{index} * \text{element_size_in_bits}$$

Once you calculate the element's address in bits, you need to convert it to a byte address (since we have to use byte addresses when accessing memory) and extract the specified element. Because the base address of an array element (almost) always starts on a byte boundary, we can use the following equations to simplify this task:

$$\begin{aligned} \text{Byte_of_1st_bit} &= \text{Base_Address} + (\text{index} * \text{element_size_in_bits}) / 8 \\ \text{Offset_to_1st_bit} &= (\text{index} * \text{element_size_in_bits}) \% 8 \quad (\text{note } \% = \text{MOD}) \end{aligned}$$

For example, suppose we have an array of 200 three-bit objects that we declare as follows:

```
static
AO3Bobjects: byte[ (200*3)/8 + 1 ]; // "+1" handles trucation.
```

The constant expression in the dimension above reserves space for enough bytes to hold 600 bits (200 elements, each three bits long). As the comment notes, the expression adds an extra byte at the end to ensure we don't lose any odd bits (that won't happen in this example since 600 is evenly divisible by 8, but in general you can't count on this; one extra byte usually won't hurt things).

Now suppose you want to access the i^{th} three-bit element of this array. You can extract these bits by using the following code:

```
// Extract the ith group of three bits in AO3Bobjects and leave this value
// in EAX.

sub( ecx, ecx );      // Put of i/8 remainder here.
mov( i, eax );        // Get the index into the array.
shrd( 3, eax, ecx );  // EAX/8 -> EAX and EAX mod 8 -> ECX (H.O. bits)
rol( 3, ecx );        // Put remainder into L.O. three bits of ECX.

// Okay, fetch the word containing the three bits we want to extract.
// We have to fetch a word because the last bit or two could wind up
// crossing the byte boundary (i.e., bit offset six and seven in the
// byte).

mov( AO3Bobjects[eax], eax );
shr( cl, eax );       // Move bits down to bit zero.
and( %111, eax );     // Remove the other bits.
```

Inserting an element into the array is a bit more difficult. In addition to computing the base address and bit offset of the array element, you've also got to create a mask to clear out the bits in the destination where you're going to insert the new data. The following code inserts the L.O. three bits of EAX into the i^{th} element of the AO3Bobjects array.

// Insert the L.O. three bits of AX into the ith element of AO3Bobjects:

```
readonly
Masks: word[8] :=
[
    !%0000_0111, !%0000_1110, !%0001_1100, !%0011_1000,
    !%0111_0000, !%1110_0000, !%1_1100_0000, !%11_1000_0000
];
.
.
.
sub( ecx, ecx );      // Put remainder here.
mov( i, ebx );        // Get the index into the array.
```

```

shrd( 3, ebx, ecx ); // i/8 -> EBX, i % 8 -> ECX.
rol( 3, ecx );

and( %111, ax ); // Clear unneeded bits from AX.
mov( Masks[ecx], dx ); // Mask to clear out our array element.
and( AO3Bobjects[ ebx ], dx ); // Grab the bits and clear those
// we're inserting.
shl( cl, ax ); // Put our three bits in their proper location.
or( ax, dx ); // Merge bits into destination.
mov( dx, AO3Bobjects[ ebx ] ); // Store back into memory.

```

Notice the use of a lookup table to generate the masks needed to clear out the appropriate position in the array. Each element of this array contains all ones except for three zeros in the position we need to clear for a given bit offset (note the use of the “!” operator to invert the constants in the table).

5.8 Searching for a Bit

A very common bit operation is to locate the end of some run of bits. A very common special case of this operation is to locate the first (or last) set or clear bit in a 16- or 32-bit value. In this section we’ll explore ways to accomplish this.

Before describing how to search for the first or last bit of a given value, perhaps it’s wise to discuss exactly what the terms “first” and “last” mean in this context. The term “first set bit” means the first bit in a value, scanning from bit zero towards the high order bit, that contains a one. A similar definition exists for the “first clear bit.” The “last set bit” is the first bit in a value, scanning from the high order bit towards bit zero, that contains a one. A similar definition exists for the last clear bit.

One obvious way to scan for the first or last bit is to use a shift instruction in a loop and count the number of iterations before you shift out a one (or zero) into the carry flag. The number of iterations specifies the position. Here’s some sample code that checks for the first set bit in EAX and returns that bit position in ECX:

```

mov( -32, ecx ); // Count off the bit positions in ECX.
TstLp: shr( 1, eax ); // Check to see if current bit position contains
jc Done // a one; exit loop if it does.
inc( ecx ); // Bump up our bit counter by one.
jnz TstLp; // Exit if we execute this loop 32 times.

Done: add( 32, cl ); // Adjust loop counter so it holds the bit posn.

// At this point, ECX contains the bit position of the first set bit.
// ECX contains 32 if EAX originally contained zero (no set bits).

```

The only thing tricky about this code is the fact that it runs the loop counter from -32 to zero rather than 32 down to zero. This makes it slightly easier to calculate the bit position once the loop terminates.

The drawback to this particular loop is that it’s expensive. This loop repeats as many as 32 times depending on the original value in EAX. If the value’s you’re checking often have lots of zeros in the L.O. bits of EAX, this code runs rather slow.

Searching for the first (or last) set bit is such a common operation that Intel added a couple of instructions on the 80386 specifically to accelerate this process. These instructions are BSF (bit scan forward) and BSR (bit scan reverse). Their syntax is as follows:

```

bsr( source, destReg );
bsf( source, destReg );

```

The source and destinations operands must be the same size and they must both be 16- or 32-bit objects. The destination operand has to be a register, the source operand can be a register or a memory location.

The BSF instruction scans for the first set bit (starting from bit position zero) in the source operand. The BSR instruction scans for the last set bit in the source operand by scanning from the H.O. bit towards the

L.O. bit. If these instructions find a bit that is set in the source operand then they clear the zero flag and put the bit position into the destination register. If the source register contains zero (i.e., there are no set bits) then these instructions set the zero flag and leave an indeterminate value in the destination register. Note that you should test the zero flag immediately after the execution of these instructions to validate the destination register's value. Examples:

```
mov( SomeValue, ebx );      // Value whose bits we want to check.
bsf( ebx, eax );           // Put position of first set bit in EAX.
jz NoBitsSet;              // Branch if SomeValue contains zero.
mov( eax, FirstBit );      // Save location of first set bit.
.
.
.
```

You use the BSR instruction in an identical fashion except that it computes the bit position of the last set bit in an operand (that is, the first set bit it finds when scanning from the H.O. bit towards the L.O. bit).

The 80x86 CPUs do not provide instructions to locate the first bit containing a zero. However, you can easily scan for a zero bit by first inverting the source operand (or a copy of the source operand if you must preserve the source operand's value). If you invert the source operand, then the first "1" bit you find corresponds to the first zero bit in the original operand value.

The BSF and BSR instructions are complex instructions (i.e., they are not a part of the 80x86 "RISC core" instruction set). Therefore, these instructions are necessarily as fast as other instructions. Indeed, in some circumstances it may be faster to locate the first set bit using discrete instructions. However, since the execution time of these instructions varies widely from CPU to CPU, you should first test the performance of these instructions prior to using them in time critical code.

Note that the BSF and BSR instructions do not affect the source operand. A common operation is to extract the first (or last) set bit you find in some operand. That is, you might want to clear the bit once you find it. If the source operand is a register (or you can easily move it into a register) then you can use the BTR (or BTC) instruction to clear the bit once you've found it. Here's some code that achieves this result:

```
bsf( eax, ecx );           // Locate first set bit in EAX.
if( @nz ) then             // If we found a bit, clear it.

    btr( ecx, eax );        // Clear the bit we just found.

endif;
```

At the end of this sequence, the zero flag indicates whether we found a bit (note that BTR does not affect the zero flag). Alternately, you could add an ELSE section to the IF statement above that handles the case when the source operand (EAX) contains zero at the beginning of this instruction sequence.

Since the BSF and BSR instructions only support 16- and 32-bit operands, you will have to compute the first bit position of an eight-bit operand a little differently. There are a couple of reasonable approaches. First, of course, you can usually zero extend an eight-bit operand to 16 or 32 bits and then use the BSF or BSR instructions on this operand. Another alternative is to create a lookup table where each entry in the table contains the number of bits in the value you use as an index into the table; then you can use the XLAT instruction to "compute" the first bit position in the value (note that you will have to handle the value zero as a special case). Another solution is to use the shift algorithm appearing at the beginning of this section; for an eight-bit operand, this is not an entirely inefficient solution.

One interesting use of the BSF and BSR instructions is to "fill in" a character set with all the values from the lowest-valued character in the set through the highest-valued character. For example, suppose a character set contains the values {'A', 'M', 'a'..'n', 'z'}; if we filled in the gaps in this character set we would have the values {'A'..'z'}. To compute this new set we can use BSF to determine the ASCII code of the first character in the set and BSR to determine the ASCII code of the last character in the set. After doing this, we can feed those two ASCII codes to the *cs.rangeChar* function to compute the new set.

You can also use the BSF and BSR instructions to determine the size of a run of bits, assuming that you have a single run of bits in your operand. Simply locate the first and last bits in the run (as above) and the

compute the difference (plus one) of the two values. Of course, this scheme is only valid if there are no intervening zeros between the first and last set bits in the value.

5.9 Counting Bits

The last example in the previous section demonstrates a specific case of a very general problem: counting bits. Unfortunately, that example has a severe limitation: it only counts a single run of one bits appearing in the source operand. This section discusses a more general solution to this problem.

Hardly a week goes by that someone doesn't ask how to count the number of bits in a register operand on one of the Internet news groups. This is a common request, undoubtedly, because many assembly language course instructors assign this task a project to their students as a way to teach them about the shift and rotate instructions. Undoubtedly, the solution these instructor expect is something like the following:

```
// BitCount1:
//
// Counts the bits in the EAX register, returning the count in EBX.

        mov( 32, cl );      // Count the 32 bits in EAX.
        sub( ebx, ebx );    // Accumulate the count here.
CntLoop: shr( 1, eax );      // Shift next bit out of EAX and into Carry.
        adc( 0, bl );        // Add the carry into the EBX register.
        dec( cl );          // Repeat 32 times.
        jnz CntLoop
```

The “trick” worth noting here is that this code uses the ADC instruction to add the value of the carry flag into the BL register. Since the count is going to be less than 32, the result will fit comfortably into BL. This code uses “adc(0, bl);” rather than “adc(0, ebx);” because the former instruction is smaller.

Tricky code or not, this instruction sequence is not particularly fast. As you can tell with just a small amount of analysis, the loop above always executes 32 times, so this code sequence executes 130 instructions (four instructions per iteration plus two extra instructions). One might ask if there is a more efficient solution, the answer is yes. The following code, taken from the AMD Athlon optimization guide, provides a faster solution (see the comments for a description of the algorithm):

```
// bitCount-
//
// Counts the number of "1" bits in a dword value.
// This function returns the dword count value in EAX.

procedure bits.cnt( BitsToCnt:dword ); nodisplay;

const
    EveryOtherBit      := $5555_5555;
    EveryAlternatePair := $3333_3333;
    EvenNibbles        := $0f0f_0f0f;

begin cnt;

    push( edx );
    mov( BitsToCnt, eax );
    mov( eax, edx );

    // Compute sum of each pair of bits
    // in EAX. The algorithm treats
    // each pair of bits in EAX as a two
    // bit number and calculates the
    // number of bits as follows (description
    // is for bits zero and one, it generalizes
    // to each pair):
```

```

//
// EDX =   BIT1  BIT0
// EAX =       0  BIT1
//
// EDX-EAX =   00 if both bits were zero.
//             01 if Bit0=1 and Bit1=0.
//             01 if Bit0=0 and Bit1=1.
//             10 if Bit0=1 and Bit1=1.
//
// Note that the result is left in EDX.

shr( 1, eax );
and( EveryOtherBit, eax );
sub( eax, edx );

// Now sum up the groups of two bits to
// produces sums of four bits. This works
// as follows:
//
// EDX = bits 2,3, 6,7, 10,11, 14,15, ..., 30,31
//       in bit positions 0,1, 4,5, ..., 28,29 with
//       zeros in the other positions.
//
// EAX = bits 0,1, 4,5, 8,9, ... 28,29 with zeros
//       in the other positions.
//
// EDX+EAX produces the sums of these pairs of bits.
// The sums consume bits 0,1,2, 4,5,6, 8,9,10, ... 28,29,30
// in EAX with the remaining bits all containing zero.

mov( edx, eax );
shr( 2, edx );
and( EveryAlternatePair, eax );
and( EveryAlternatePair, edx );
add( edx, eax );

// Now compute the sums of the even and odd nibbles in the
// number. Since bits 3, 7, 11, etc. in EAX all contain
// zero from the above calculation, we don't need to AND
// anything first, just shift and add the two values.
// This computes the sum of the bits in the four bytes
// as four separate value in EAX (AL contains number of
// bits in original AL, AH contains number of bits in
// original AH, etc.)

mov( eax, edx );
shr( 4, eax );
add( edx, eax );
and( EvenNibbles, eax );

// Now for the tricky part.
// We want to compute the sum of the four bytes
// and return the result in EAX. The following
// multiplication achieves this. It works
// as follows:
// (1) the $01 component leaves bits 24..31
//     in bits 24..31.
//
// (2) the $100 component adds bits 17..23
//     into bits 24..31.
//

```

```

// (3) the $1_0000 component adds bits 8..15
//     into bits 24..31.
//
// (4) the $1000_0000 component adds bits 0..7
//     into bits 24..31.
//
// Bits 0..23 are filled with garbage, but bits
// 24..31 contain the actual sum of the bits
// in EAX's original value. The SHR instruction
// moves this value into bits 0..7 and zeroes
// out the H.O. bits of EAX.

intmul( $0101_0101, eax );
shr( 24, eax );

pop( edx );

end cnt;

```

5.10 Reversing a Bit String

Another common programming project instructions assign, and a useful function in its own right, is a program that reverses the bits in an operand. That is, it swaps the L.O. bit with the H.O. bit, bit #1 with the next-to-H.O. bit, etc. The typical solution an instructor probably expects for this assignment is the following:

```

// Reverse the 32-bits in EAX, leaving the result in EBX:

mov( 32, cl );
RvsLoop: shr( 1, eax );    // Move current bit in EAX to the carry flag.
          rcl( 1, ebx );   // Shift the bit back into EBX, backwards.
          dec( cl );
          jnz RvsLoop

```

As with the previous examples, this code suffers from the fact that it repeats the loop 32 times for a grand total of 129 instructions. By unrolling the loop you can get it down to 64 instructions, but this is still somewhat expensive.

As usual, the best solution to an optimization problem is often a better algorithm rather than attempting to tweak your code by trying to choose faster instructions to speed up some code. However, a little intelligence goes a long way when manipulating bits. In the last section, for example, we were able to speed up counting the bits in a string by substituting a more complex algorithm for the simplistic “shift and count” algorithm. In the example above, we are once again faced with a very simple algorithm with a loop that repeats for one bit in each number. The question is: “Can we discover an algorithm that doesn’t execute 129 instructions to reverse the bits in a 32-bit register?” The answer is “yes” and the trick is to do as much work as possible in parallel.

Suppose that all we wanted to do was swap the even and odd bits in a 32-bit value. We can easily swap the even and odd bits in EAX using the following code:

```

mov( eax, edx );    // Make a copy of the odd bits in the data.
shr( 1, eax );      // Move the even bits to the odd positions.
and( $5555_5555, edx ); // Isolate the odd bits by clearing even bits.
and( $5555_5555, eax ); // Isolate the even bits (in odd posn now).
shl( 1, edx );      // Move the odd bits to the even positions.
or( edx, eax );     // Merge the bits and complete the swap.

```

Of course, swapping the even and odd bits, while somewhat interesting, does not solve our larger problem of reversing all the bits in the number. But it does take us part of the way there. For example, if after executing the code sequence above, we swap adjacent pairs of bits, then we've managed to swap the bits in all the nibbles in the 32-bit value. Swapping adjacent pairs of bits is done in a manner very similar to the above, the code is

```
mov( eax, edx );      // Make a copy of the odd numbered bit pairs.
shr( 2, eax );        // Move the even bit pairs to the odd posn.
and( $3333_3333, edx ); // Isolate the odd pairs by clearing even pairs.
and( $3333_3333, eax ); // Isolate the even pairs (in odd posn now).
shl( 2, edx );        // Move the odd pairs to the even positions.
or( edx, eax );       // Merge the bits and complete the swap.
```

After completing the sequence above we swap the adjacent nibbles in the 32-bit register. Again, the only difference is the bit mask and the length of the shifts. Here's the code:

```
mov( eax, edx );      // Make a copy of the odd numbered nibbles.
shr( 4, eax );        // Move the even nibbles to the odd position.
and( $0f0f_0f0f, edx ); // Isolate the odd nibbles.
and( $0f0f_0f0f, eax ); // Isolate the even nibbles (in odd posn now).
shl( 4, edx );        // Move the odd pairs to the even positions.
or( edx, eax );       // Merge the bits and complete the swap.
```

You can probably see the pattern developing and can figure out that in the next two steps we've got to swap the bytes and then the words in this object. You can use code like the above, but there is a better way – use the BSWAP instruction. The BSWAP (byte swap) instruction uses the following syntax:

```
bswap( reg32 );
```

This instruction swaps bytes zero and three and it swaps bytes one and two in the specified 32-bit register. The principle use of this instruction is to convert data between the so-called “little endian” and “big-endian” data formats². Although we don't specifically need this instruction for this purpose here, the BSWAP instruction does swap the bytes and words in a 32-bit object exactly the way we want them when reversing bits, so rather than sticking in another 12 instructions to swap the bytes and then the words, we can simply use a “bswap(eax);” instruction to complete the job after the instructions above. The final code sequence is

```
mov( eax, edx );      // Make a copy of the odd bits in the data.
shr( 1, eax );        // Move the even bits to the odd positions.
and( $5555_5555, edx ); // Isolate the odd bits by clearing even bits.
and( $5555_5555, eax ); // Isolate the even bits (in odd posn now).
shl( 1, edx );        // Move the odd bits to the even positions.
or( edx, eax );       // Merge the bits and complete the swap.

mov( eax, edx );      // Make a copy of the odd numbered bit pairs.
shr( 2, eax );        // Move the even bit pairs to the odd posn.
and( $3333_3333, edx ); // Isolate the odd pairs by clearing even pairs.
and( $3333_3333, eax ); // Isolate the even pairs (in odd posn now).
shl( 2, edx );        // Move the odd pairs to the even positions.
or( edx, eax );       // Merge the bits and complete the swap.

mov( eax, edx );      // Make a copy of the odd numbered nibbles.
shr( 4, eax );        // Move the even nibbles to the odd position.
and( $0f0f_0f0f, edx ); // Isolate the odd nibbles.
and( $0f0f_0f0f, eax ); // Isolate the even nibbles (in odd posn now).
shl( 4, edx );        // Move the odd pairs to the even positions.
or( edx, eax );       // Merge the bits and complete the swap.
```

2. In the little endian system, which the native 80x86 format, the L.O. byte of an object appears at the lowest address in memory. In the big endian system, which various RISC processors use, the H.O. byte of an object appears at the lowest address in memory. The BSWAP instruction converts between these two data formats.

```
bswap( eax );           // Swap the bytes and words.
```

This algorithm only requires 13 instructions and it executes much faster than the bit shifting loop appearing earlier. Of course, this sequence does consume a bit more memory, so if you're trying to save memory rather than clock cycles, the loop is probably a better solution.

5.11 Merging Bit Strings

Another common bit string operation is producing a single bit string by merging, or interleaving, bits from two different sources. The following example code sequence creates a 32-bit string by merging alternate bits from two 16-bit strings:

```
// Merge two 16-bit strings into a single 32-bit string.
// AX - Source for even numbered bits.
// BX - Source for odd numbered bits.
// CL - Scratch register.
// EDX- Destination register.

MergeLp:  mov( 16, cl );
          shrd( 1, eax, edx ); // Shift a bit from EAX into EDX.
          shrd( 1, ebx, edx ); // Shift a bit from EBX into EDX.
          dec( cl );
          jne MergeLp;
```

This particular example merged two 16-bit values together, alternating their bits in the result value. For a faster implementation of this code, unrolling the loop is probably your best bet since this eliminates half the instructions that execute on each iteration of the loop above.

With a few slight modifications, we could also have merged four eight-bit values together, or we could have generated the result using other bit sequences; for example, the following code copies bits 0..5 from EAX, then bits 0..4 from EBX, then bits 6..11 from EAX, then bits 5..15 from EBX, and finally bits 12..15 from EAX:

```
shrd( 6, eax, edx );
shrd( 5, ebx, edx );
shrd( 6, eax, edx );
shrd( 11, ebx, edx );
shrd( 4, eax, edx );
```

5.12 Extracting Bit Strings

Of course, we can easily accomplish the converse of merging two bit streams; i.e., we can extract and distribute bits in a bit string among multiple destinations. The following code takes the 32-bit value in EAX and distributes alternate bits among the BX and DX registers:

```
ExtractLp:  mov( 16, cl );           // Count off the number of loop iterations.
            shrd( 1, eax, ebx ); // Extract even bits to (E)BX.
            shrd( 1, eax, edx ); // Extract odd bits to (E)DX.
            dec( cl );           // Repeat 16 times.
            jnz ExtractLp;
            shr( 16, ebx );      // Need to move the results from the H.O.
            shr( 16, edx );      // bytes of EBX/EDX to the L.O. bytes.
```

This sequence executes 51 instructions. This isn't terrible, but we can probably do a little better by using a better algorithm that extracts bits in parallel. Employing the technique we used to reverse bits in a register, we can come up with the following algorithm that relocates all the even bits to the L.O. word of EAX and all the odd bits to the H.O. word of EAX.

```

// Swap bits at positions (1,2), (5,6), (9,10), (13,14), (17,18),
// (21,22), (25,26), and (29, 30).

    mov( eax, edx );
    and( $9999_9999, eax );      // Mask out the bits we'll keep for now.
    mov( edx, ecx );
    shr( 1, edx );               // Move 1st bits in tuple above to the
    and( $2222_2222, ecx );      // correct position and mask out the
    and( $2222_2222, edx );      // unneeded bits.
    shl( 1, ecx );               // Move 2nd bits in tuples above.
    or( edx, ecx );              // Merge all the bits back together.
    or( ecx, eax );

// Swap bit pairs at positions ((2,3), (4,5)), ((10,11), (12,13)), etc.

    mov( eax, edx );
    and( $c3c3_c3c3, eax );      // The bits we'll leave alone.
    mov( edx, ecx );
    shr( 2, edx );
    and( $0c0c_0c0c, ecx );
    and( $0c0c_0c0c, edx );
    shl( 2, ecx );
    or( edx, ecx );
    or( ecx, eax );

// Swap nibbles at nibble positions (1,2), (5,6), (9,10), etc.

    mov( eax, edx );
    and( $f00f_f00f, eax );
    mov( edx, ecx );
    shr(4, edx );
    and( $0f0f_0f0f, ecx );
    and( $0f0f_0f0f, edx );
    shl( 4, ecx );
    or( edx, ecx );
    or( ecx, eax );

// Swap bits at positions 1 and 2.

    ror( 8, eax );
    xchg( al, ah );
    rol( 8, eax );

```

This sequence require 30 instructions. At first blush it looks like a winner since the original loop executes 64 instructions. However, this code isn't quite as good as it looks. After all, if we're willing to write this much code, why not unroll the loop above 16 times to obtain the following 32 instructions:

```

shrd( 1, eax, ebx );
shrd( 1, eax, edx );
shrd( 1, eax, ebx );
shrd( 1, eax, edx );
shrd( 1, eax, ebx );
shrd( 1, eax, edx );
shrd( 1, eax, ebx );
shrd( 1, eax, edx );
shrd( 1, eax, ebx );
shrd( 1, eax, edx );
shrd( 1, eax, ebx );
shrd( 1, eax, edx );
shrd( 1, eax, ebx );
shrd( 1, eax, edx );
shrd( 1, eax, ebx );
shrd( 1, eax, edx );

```


Matched:

```
<< If we get to this point, we matched the bit string. We can >>
<< compute the position in the original source as 28-cl. >>
```

Done:

Bit string scanning is a special case of string matching. String matching is a well studied problem in Computer Science and many of the algorithms you can use for string matching are applicable to bit string matching as well. Such algorithms are a bit beyond the scope of this chapter, but to give you a preview of how this works, you compute some function (like XOR or SUB) between the pattern and the current source bits and use the result as an index into a lookup table to determine how many bits you can skip. Such algorithms let you skip several bits rather than only shifting once per each iteration of the scanning loop (as is done by the algorithm above). For more details on string scanning and their possible application to bit string matching, see the appropriate chapter in the volume on Advanced String Handling.

5.14 The HLA Standard Library Bits Module

The HLA Standard Library provides a “bits” module that provides several bit related functions, including built-in functions for many of the algorithms we’ve studied in this chapter. This section will describe these functions available in the HLA Standard Library.

```
procedure bits.cnt( b:dword ); returns( "EAX" );
```

This procedure returns the number of one bits present in the “b” parameter. It returns the count in the EAX register. To count the number of zero bits in the parameter value, invert the value of the parameter before passing it to *bits.cnt*. If you want to count the number of bits in a 16-bit operand, simply zero extend it to 32 bits prior to calling this function. Here are a couple of examples:

```
// Compute the number of bits in a 16-bit register:
```

```
    pushw( 0 );
    push( ax );
    call bits.cnt;
```

```
// If you prefer to use a higher-level syntax, try the following:
```

```
    bits.cnt( #{ pushw(0); push(ax); }# );
```

```
// Compute the number of bits in a 16-bit memory location:
```

```
    pushw( 0 );
    push( mem16 );
    bits.cnt;
```

If you want to compute the number of bits in an eight-bit operand it’s probably faster to write a simple loop that rotates all the bits in the source operand and adds the carry into the accumulating sum. Of course, if performance isn’t an issue, you can zero extend the byte to 32 bits and call the *bits.cnt* procedure.

```
procedure bits.distribute( source:dword; mask:dword; dest:dword );
    returns( "EAX" );
```

This function takes the L.O. *n* bits of *source*, where *n* is the number of “1” bits in *mask*, and inserts these bits into *dest* at the bit positions specified by the “1” bits in *mask* (i.e., the same as the distribute algorithm appearing earlier in this chapter). This function does not change the bits in *dest* that correspond to the zeros in the *mask* value. This function does not affect the value of the actual *dest* parameter, instead, it returns the new value in the EAX register.

```
procedure bits.coalesce( source:dword; mask:dword );
  returns( "EAX" );
```

This function is the converse of *bits.distribute*. It extracts all the bits in source whose corresponding positions in mask contain a one. This function coalesces (right justifies) these bits in the L.O. bit positions of the result and returns the result in EAX.

```
procedure bits.extract( var d:dword ); returns( "EAX" ); // Really a macro.
```

This function extracts the first set bit in *d* searching from bit #0 and returns the index of this bit in the EAX register; the function will also return the zero flag clear in this case. This function also clears that bit in the operand. If *d* contains zero, then this function returns the zero flag set and EAX will contain -1.

Note that HLA actually implements this function as a macro, not a procedure (see the chapter on Macros for details). This means that you can pass any double word operand as a parameter (i.e., a memory or a register operand). However, the results are undefined if you pass EAX as the parameter (since this function returns the bit number in EAX).

```
procedure bits.reverse32( d:dword ); returns( "EAX" );
procedure bits.reverse16( w:word ); returns( "AX" );
procedure bits.reverse8( b:byte ); returns( "AL" );
```

These three routines return their parameter value with the its bits reversed in the accumulator register (AL/AX/EAX). Call the routine appropriate for your data size.

```
procedure bits.merge32( even:dword; odd:dword ); returns( "EDX:EAX" );
procedure bits.merge16( even:word; odd:word ); returns( "EAX" );
procedure bits.merge8( even:byte; odd:byte ); returns( "AX" );
```

These routines merge two streams of bits to produce a value whose size is the combination of the two parameters. The bits from the “even” parameter occupy the even bits in the result, the bits from the “odd” parameter occupy the odd bits in the result. Notice that these functions return 16, 32, or 64 bits based on byte, word, and double word parameter values.

```
procedure bits.nibbles32( d:dword ); returns( "EDX:EAX" );
procedure bits.nibbles16( w:word ); returns( "EAX" );
procedure bits.nibbles8( b:byte ); returns( "AX" );
```

These routines extract each nibble from the parameter and place those nibbles into individual bytes. The *bits.nibbles8* function extracts the two nibbles from the *b* parameter and places the L.O. nibble in AL and the H.O. nibble in AH. The *bits.nibbles16* function extracts the four nibbles in *w* and places them in each of the four bytes of EAX. You can use the BSWAP or ROx instructions to gain access to the nibbles in the H.O. word of EAX. The *bits.nibbles32* function extracts the eight nibbles in EAX and distributes them through the eight bytes in EDX:EAX. Nibble zero winds up in AL and nibble seven winds up in the H.O. byte of EDX. Again, you can use BSWAP or the rotate instructions to access the upper bytes of EAX and EDX.

5.15 Putting It All Together

Bit manipulation is one area where assembly language really shines. Not only is bit manipulation far more efficient in assembly language than in high level languages, but it’s often easier as well. Although the need to manipulate bits is not an everyday requirement, bit manipulation is still a very important problem area. In this chapter we’ve explored several ways to manipulate data as bits. Although this chapter only

begins to cover the possibilities, it should give you some ideas for developing your own bit manipulation algorithms for use in your applications.

The String Instructions

Chapter Six

6.1 Chapter Overview

A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes, words, or (on 80386 and later processors) double words. The 80x86 microprocessor family supports several instructions specifically designed to cope with strings. This chapter explores some of the uses of these string instructions.

The 80x86 CPUs can process three types of strings: byte strings, word strings, and double word strings. They can move strings, compare strings, search for a specific value within a string, initialize a string to a fixed value, and do other primitive operations on strings. The 80x86's string instructions are also useful for manipulating arrays, tables, and records. You can easily assign or compare such data structures using the string instructions. Using string instructions may speed up your array manipulation code considerably.

6.2 The 80x86 String Instructions

All members of the 80x86 family support five different string instructions: `MOVSw`, `CMPSw`, `SCASw`, `LODSw`, and `STOSw`¹. ($x = B, W, \text{ or } D$ for byte, word, or double word, respectively. This text will generally drop the x suffix when talking about these string instructions in a general sense.) They are the string primitives since you can build most other string operations from these five instructions. How you use these five instructions is the topic of the next several sections.

For `MOVSw`:

```
movsb();
movsw();
movsd();
```

For `CMPSw`:

```
cmpsb(); // Note: repz is a synonym for repe
cmpsw();
cmpsd();

cmpsb(); // Note: repnz is a synonym for repne.
cmpsw();
cmpsd();
```

For `SCASw`:

```
scasb(); // Note: repz is a synonym for repe
scasw();
scasd();

scasb(); // Note: repnz is a synonym for repne.
scasw();
scasd();
```

For `STOSw`:

```
stosb();
stosw();
stosd();
```

1. The 80x86 processor support two additional string instructions, `INS` and `OUTS` which input strings of data from an input port or output strings of data to an output port. We will not consider these instructions since they are privileged instructions and you cannot execute them in a standard Win32 application.

```

For LODS:
    lodsb();
    lodsw();
    lodsd();

```

6.2.1 How the String Instructions Operate

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the MOVS instruction moves a sequence of bytes from one memory location to another. The CMPS instruction compares two blocks of memory. The SCAS instruction scans a block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the MOVS instruction to copy a string, you need a source address, a destination address, and a count (the number of string elements to move).

Unlike other instructions which operate on memory, the string instructions don't have any explicit operands. The operands for the string instructions include

- the ESI (source index) register,
- the EDI (destination index) register,
- the ECX (count) register,
- the AL/AX/EAX register, and
- the direction flag in the FLAGS register.

For example, one variant of the MOVS (move string) instruction copies a string from the source address specified by ESI to the destination address specified by EDI, of length ECX. Likewise, the CMPS instruction compares the string pointed at by ESI, of length ECX, to the string pointed at by EDI.

Not all instructions have source and destination operands (only MOVS and CMPS support them). For example, the SCAS instruction (scan a string) compares the value in the accumulator (AL, AX, or EAX) to values in memory.

6.2.2 The REP/REPE/REPZ and REPNZ/REPNE Prefixes

The string instructions, by themselves, do not operate on strings of data. The MOVS instruction, for example, will move a single byte, word, or double word. When executed by itself, the MOVS instruction ignores the value in the ECX register. The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

```

For MOVS:
    rep.movsb();
    rep.movsw();
    rep.movsd();

For CMPS:
    repe.cmpsb();    // Note: repz is a synonym for repe.
    repe.cmpsw();
    repe.cmpsd();

    repne.cmpsb();   // Note: repnz is a synonym for repne.
    repne.cmpsw();
    repne.cmpsd();

For SCAS:
    repe.scasb();    // Note: repz is a synonym for repe.
    repe.scasw();
    repe.scasd();

```

```
repne.scasb();    // Note: repnz is a synonym for repne.
repne.scasw();
repne.scasd();
```

```
For STOS:
rep.stosb();
rep.stosw();
rep.stosd();
```

You don't normally use the repeat prefixes with the LODS instruction.

When specifying the repeat prefix before a string instruction, the string instruction repeats ECX times². Without the repeat prefix, the instruction operates only on a single byte, word, or double word.

You can use repeat prefixes to process entire strings with a single instruction. You can use the string instructions, without the repeat prefix, as string primitive operations to synthesize more powerful string operations.

6.2.3 The Direction Flag

Besides the ESI, EDI, ECX, and AL/AX/EAX registers, one other register controls the 80x86's string instructions – the flags register. Specifically, the *direction flag* in the flags register controls how the CPU processes strings.

If the direction flag is clear, the CPU increments ESI and EDI after operating upon each string element. For example, if the direction flag is clear, then executing MOVS will move the byte, word, or double word at ESI to EDI and will increment ESI and EDI by one, two, or four. When specifying the REP prefix before this instruction, the CPU increments ESI and EDI for each element in the string. At completion, the ESI and EDI registers will be pointing at the first item beyond the string.

If the direction flag is set, then the 80x86 decrements ESI and EDI after processing each string element. After a repeated string operation, the ESI and EDI registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the CLD (clear direction flag) and STD (set direction flag) instructions. When using these instructions inside a procedure, keep in mind that they modify the machine state. Therefore, you may need to save the direction flag during the execution of that procedure. The following example exhibits the kinds of problems you might encounter:

```
procedure Str2; nodisplay;
begin Str2;

    std();
    <Do some string operations>
    .
    .
    .
end Str2;
    .
    .
    .
    cld();
    <do some operations>
    Str2();
    <do some string operations requiring D=0>
```

2. Except for the cmps instruction which repeats *at most* the number of times specified in the cx register.

This code will not work properly. The calling code assumes that the direction flag is clear after *Str2* returns. However, this isn't true. Therefore, the string operations executed after the call to *Str2* will not function properly.

There are a couple of ways to handle this problem. The first, and probably the most obvious, is always to insert the CLD or STD instructions immediately before executing a sequence of one or more string instructions. The other alternative is to save and restore the direction flag using the PUSHFD and POPFD instructions. Using these two techniques, the code above would look like this:

Always issuing CLD or STD before a string instruction:

```
procedure Str2; nodisplay;
begin Str2;

    std();
    <Do some string operations>
    .
    .
    .
end Str2;
    .
    .
    .
    cld();
    <do some operations>
    Str2();
    cld();
    <do some string operations requiring D=0>
```

Saving and restoring the flags register:

```
procedure Str2; nodisplay;
begin Str2;

    pushfd();
    std();
    <Do some string operations>
    .
    .
    .
    popfd();
end Str2;
    .
    .
    .
    cld();
    <do some operations>
    Str2();
    <do some string operations requiring D=0>
```

If you use the PUSHFD and POPFD instructions to save and restore the flags register, keep in mind that you're saving and restoring all the flags. Therefore, such subroutines cannot return any information in the flags. For example, you will not be able to return an error condition in the carry flag if you use PUSHFD and POPFD.

6.2.4 The MOVS Instruction

The MOVS instruction uses the following syntax:

```
movsb()
```

```

movsw()
movsd()
rep.movsb()
rep.movsw()
rep.movsd()

```

The MOVSB (move string, bytes) instruction fetches the byte at address ESI, stores it at address EDI and then increments or decrements the ESI and EDI registers by one. If the REP prefix is present, the CPU checks ECX to see if it contains zero. If not, then it moves the byte from ESI to EDI and decrements the ECX register. This process repeats until ECX becomes zero.

The MOVSW (move string, words) instruction fetches the word at address ESI, stores it at address EDI and then increments or decrements ESI and EDI by two. If there is a REP prefix, then the CPU repeats this procedure as many times as specified in ECX.

The MOVSD instruction operates in a similar fashion on double words. Incrementing or decrementing ESI and EDI by four for each data movement.

When you use the *rep* prefix, the MOVSB instruction moves the number of bytes you specify in the ECX register. The following code segment copies 384 bytes from *CharArray1* to *CharArray2*:

```

CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1 );
lea( edi, CharArray2 );
mov( 384, ecx );
rep.movsb();

```

If you substitute MOVSW for MOVSB, then the code above will move 384 words (768 bytes) rather than 384 bytes:

```

WordArray1: word[ 384 ];
WordArray2: word[ 384 ];
.
.
.
cld();
lea( esi, WordArray1 );
lea( edi, WordArray2 );
mov( 384, ecx );
rep.movsw();

```

Remember, the ECX register contains the element count, not the byte count. When using the MOVSW instruction, the CPU moves the number of words specified in the ECX register. Similarly, MOVSD moves the number of double words you specify in the ECX register, not the number of bytes.

If you've set the direction flag before executing a MOVSB/MOVSW/MOVSD instruction, the CPU decrements the ESI and EDI registers after moving each string element. This means that the ESI and EDI registers must point at the end of their respective strings before issuing a MOVSB, MOVSW, or MOVSD instruction. For example,

```

CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1[383] );
lea( edi, CharArray2[383] );
mov( 384, ecx );

```

```
rep.movsb();
```

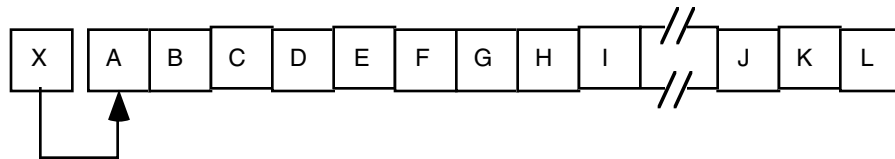
Although there are times when processing a string from tail to head is useful (see the CMPS description in the next section), generally you'll process strings in the forward direction since it's more straightforward to do so. There is one class of string operations where being able to process strings in both directions is absolutely mandatory: processing strings when the source and destination blocks overlap. Consider what happens in the following code:

```
CharArray1: byte;
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1 );
lea( edi, CharArray2 );
mov( 384, ecx );
rep.movsb();
```

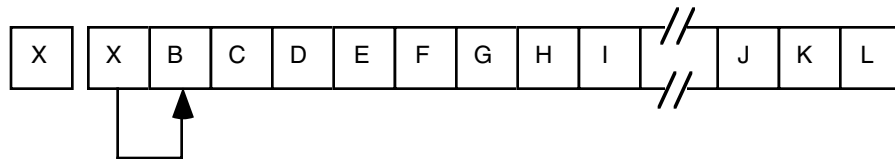
This sequence of instructions treats *CharArray1* and *CharArray2* as a pair of 384 byte strings. However, the last 383 bytes in the *CharArray1* array overlap the first 383 bytes in the *CharArray2* array. Let's trace the operation of this code byte by byte.

When the CPU executes the MOVSB instruction, it copies the byte at ESI (*CharArray1*) to the byte pointed at by EDI (*CharArray2*). Then it increments ESI and EDI, decrements ECX by one, and repeats this process. Now the ESI register points at *CharArray1+1* (which is the address of *CharArray2*) and the EDI register points at *CharArray2+1*. The MOVSB instruction copies the byte pointed at by ESI to the byte pointed at by EDI. However, this is the byte originally copied from location *CharArray1*. So the MOVSB instruction copies the value originally in location *CharArray1* to both locations *CharArray2* and *CharArray2+1*. Again, the CPU increments ESI and EDI, decrements ECX, and repeats this operation. Now the movsb instruction copies the byte from location *CharArray1+2* (*CharArray2+1*) to location *CharArray2+2*. But once again, this is the value that originally appeared in location *CharArray1*. Each repetition of the loop copies the next element in *CharArray1[0]* to the next available location in the *CharArray2* array. Pictorially, it looks something like that shown in Figure 6.1.

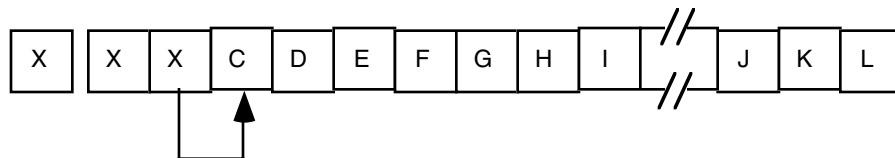
1st move operation:



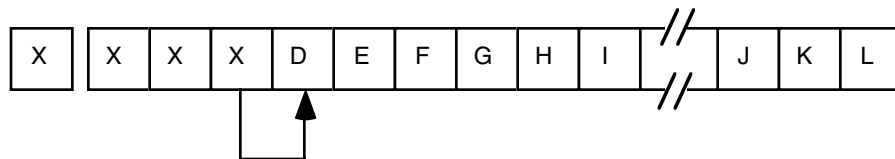
2nd move operation:



3rd move operation:



4th move operation:



nth move operation:

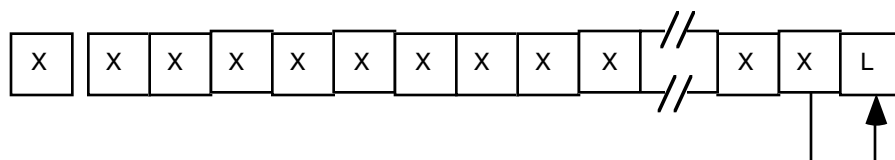


Figure 6.1 Copying Data Between Two Overlapping Arrays (forward direction)

The end result is that the MOVSB instruction replicates *X* throughout the string. The MOVSB instruction copies the source operand into the memory location which will become the source operand for the very next move operation, which causes the replication.

If you really want to move one array into another when they overlap, you should move each element of the source string to the destination string starting at the end of the two strings as shown in Figure 6.2.

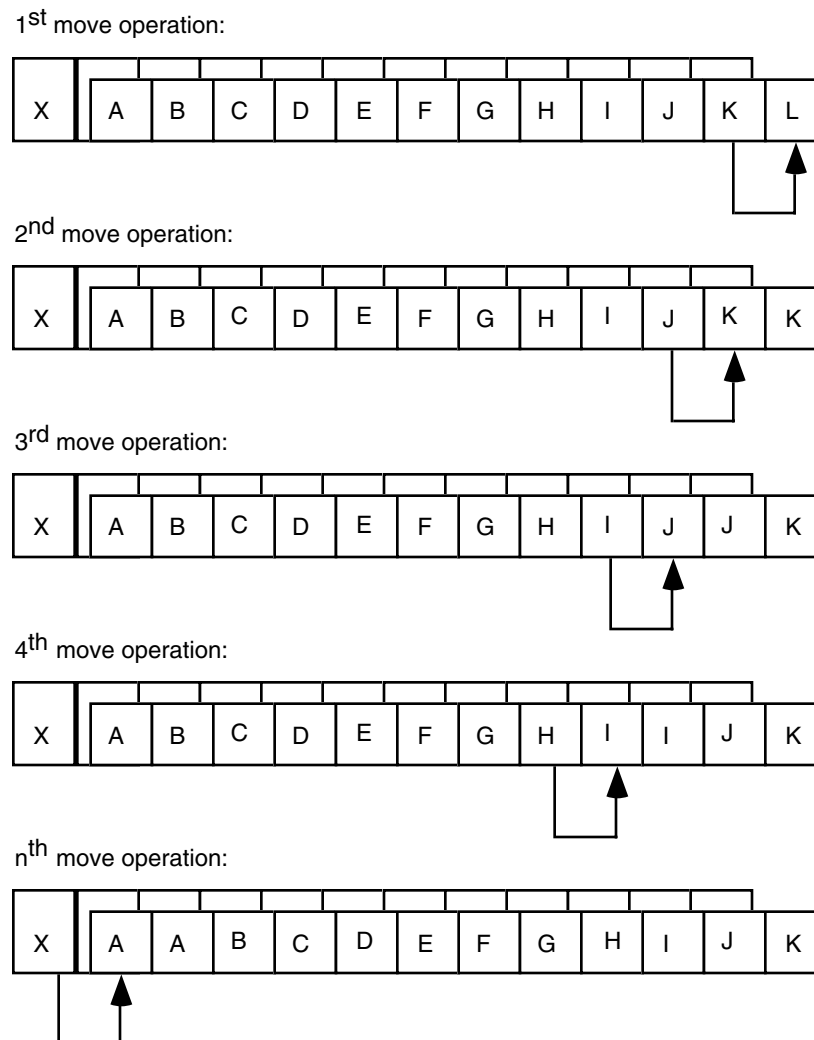


Figure 6.2 Using a Backwards Copy to Copy Data in Overlapping Arrays

Setting the direction flag and pointing ESI and EDI at the end of the strings will allow you to (correctly) move one string to another when the two strings overlap and the source string begins at a lower address than the destination string. If the two strings overlap and the source string begins at a higher address than the destination string, then clear the direction flag and point ESI and EDI at the beginning of the two strings.

If the two strings do not overlap, then you can use either technique to move the strings around in memory. Generally, operating with the direction flag clear is the easiest, so that makes the most sense in this case.

You shouldn't use the MOVSB instruction to fill an array with a single byte, word, or double word value. Another string instruction, STOS, is much better for this purpose. However, for arrays whose elements are larger than four bytes, you can use the MOVSB instruction to initialize the entire array to the content of the first element. See the questions for additional information.

The MOVSB instruction is far more efficient when copying double words than it is copying bytes or words. In fact, it typically takes the same amount of time to copy a byte using MOVSB as it does to copy a double word using MOVSD³. Therefore, if you are moving a large number of bytes from one array to another, the copy operation will be faster if you can use the MOVSD instruction rather than the MOVSB

3. This is true for MOVSW, as well.

instruction. Of course, if the number of bytes you wish to move is an even multiple of four, this is a trivial change; just divide the number of bytes to copy by four, load this value into ECX, and then use the MOVSB instruction. If the number of bytes is not evenly divisible by four, then you can use the MOVSD instruction to copy all but the last one, two, or three bytes of the array (that is, the remainder after you divide the byte count by four). For example, if you want to efficiently move 4099 bytes, you can do so with the following instruction sequence:

```
lea( esi, Source );
lea( edi, Destination );
mov( 1024, ecx );           // Copy 1024 dwords = 4096 bytes
rep.movsd();
movsw();                   // Copy bytes 4097 and 4098.
movsb();                   // Copy the last byte.
```

Using this technique to copy data never requires more than three MOVSB instructions since you can copy one, two, or three bytes with no more than two MOVSB and MOVSW instructions. The scheme above is most efficient if the two arrays are aligned on double word boundaries. If not, you might want to move the MOVSB or MOVSW instruction (or both) before the MOVSD so that the MOVSD instruction works with dword-aligned data (see Chapter Three for an explanation of the performance benefits of double word aligned data).

If you do not know the size of the block you are copying until the program executes, you can still use code like the following to improve the performance of a block move of bytes:

```
lea( esi, Source );
lea( edi, Dest );
mov( Length, ecx );
shr( 2, ecx );             // divide by four.
if( @nz ) then             // Only execute MOVSD if four or more bytes.

    rep.movsd();           // Copy the dwords.

endif;
mov( Length, ecx );
and( %11, ecx );           // Compute (Length mod 4).
if( @nz ) then             // Only execute MOVSB if #bytes/4 <> 0.

    rep.movsb();           // Copy the remaining one, two, or three bytes.

endif;
```

On most computer systems, the MOVSD instruction provides about the fastest way to copy bulk data from one location to another. While there are, arguably, faster ways to copy the data on certain CPUs, ultimately the memory bus performance is the limiting factor and the CPUs are generally much faster than the memory bus. Therefore, unless you have a special system, writing fancy code to improve memory to memory transfers is probably a waste of time.

6.2.5 The CMPS Instruction

The CMPS instruction compares two strings. The CPU compares the string referenced by EDI to the string pointed at by ESI. CX contains the length of the two strings (when using the REPE or REPNE prefix). Like the MOVS instruction, HLA allows several different forms of this instruction:

```
cmpsb();
cmpsw();
cmpsd();

repe.cmpsb();
repe.cmpsw();
repe.cmpsd();
```

```
repne.cmpsb();
repne.cmpsw();
repne.cmpsd();
```

Like the MOVS instruction you specify the actual operand addresses in the ESI and EDI registers.

Without a repeat prefix, the CMPS instruction subtracts the value at location EDI from the value at ESI and updates the flags. Other than updating the flags, the CPU doesn't use the difference produced by this subtraction. After comparing the two locations, CMPS increments or decrements the ESI and EDI registers by one, two, or four (for CMPSB/CMPSW/CMPSD, respectively). CMPS increments the ESI and EDI registers if the direction flag is clear and decrements them otherwise.

Of course, you will not tap the real power of the CMPS instruction using it to compare single bytes, words, or double words in memory. This instruction shines when you use it to compare whole strings. With CMPS, you can compare consecutive elements in a string until you find a match or until consecutive elements do not match.

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match. Consider the following strings:

“String1”

“String1”

The only way to determine that these two strings are equal is to compare each character in the first string to the corresponding character in the second. After all, the second string could have been “String2” which definitely is not equal to “String1”. Of course, once you encounter a character in the destination string which doesn't equal the corresponding character in the source string, the comparison can stop. You needn't compare any other characters in the two strings.

The REPE prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and ECX is greater than zero. We could compare the two strings above using the following 80x86 assembly language code:

```
cld();
mov( AdrsString1, esi );
mov( AdrsString2, edi );
mov( 7, ecx );
repe.cmpsb();
```

After the execution of the CMPSB instruction, you can test the flags using the standard conditional jump instructions. This lets you check for equality, inequality, less than, greater than, etc.

Character strings are usually compared using *lexicographical ordering*. In lexicographical ordering, the least significant element of a string carries the most weight. This is in direct contrast to standard integer comparisons where the most significant portion of the number carries the most weight. Furthermore, the length of a string affects the comparison only if the two strings are identical up to the length of the shorter string. For example, “Zebra” is less than “Zebras”, because it is the shorter of the two strings, however, “Zebra” is greater than “AAAAAAAAAH!” even though it is shorter. Lexicographical comparisons compare corresponding elements until encountering a character which doesn't match, or until encountering the end of the shorter string. If a pair of corresponding characters do not match, then this algorithm compares the two strings based on that single character. If the two strings match up to the length of the shorter string, we must compare their length. The two strings are equal if and only if their lengths are equal and each corresponding pair of characters in the two strings is identical. Lexicographical ordering is the standard alphabetical ordering you've grown up with.

For character strings, use the CMPS instruction in the following manner:

- The direction flag must be cleared before comparing the strings.
- Use the CMPSB instruction to compare the strings on a byte by byte basis. Even if the strings contain an even number of characters, you cannot use the CMPSW or CMPSD instructions. They do not compare strings in lexicographical order.
- You must load the ECX register with the length of the smaller string.

- Use the REPE prefix.
- The ESI and EDI registers must point at the very first character in the two strings you want to compare.

After the execution of the CMPS instruction, if the two strings were equal, their lengths must be compared in order to finish the comparison. The following code compares a couple of character strings:

```
mov( AdrsStr1, esi );
mov( AdrsStr2, edi );
mov( LengthSrc, ecx );
if( ecx > LengthDest ) then // Put the length of the shorter string in ECX.

    mov( LengthDest, ecx );

endif;
repe.cmpsb();
if( @z ) then // If equal to the length of the shorter string, cmp lengths.

    mov( LengthSrc, ecx );
    cmp( ecx, LengthDest );

endif;
```

If you're using bytes to hold the string lengths, you should adjust this code appropriately (i.e., use a MOVZX instruction to load the lengths into ECX). Of course, HLA strings use a double word to hold the current length value, so this isn't an issue when using HLA strings.

You can also use the CMPS instruction to compare multi-word integer values (that is, extended precision integer values). Because of the amount of setup required for a string comparison, this isn't practical for integer values less than six or eight double words in length, but for large integer values, it's an excellent way to compare such values. Unlike character strings, we cannot compare integer strings using a lexicographical ordering. When comparing strings, we compare the characters from the least significant byte to the most significant byte. When comparing integers, we must compare the values from the most significant byte (or word/double word) down to the least significant byte, word or double word. So, to compare two 32-byte (256-bit) integer values, use the following code on the 80x86:

```
std();
lea( esi, SourceInteger[28] );
lea( edi, DestInteger[28] );
mov( 8, ecx );
rep.cmpsd();
```

This code compares the integers from their most significant word down to the least significant word. The CMPSD instruction finishes when the two values are unequal or upon decrementing ECX to zero (implying that the two values are equal). Once again, the flags provide the result of the comparison.

The REPNE prefix will instruct the CMPS instruction to compare successive string elements as long as they do not match. The 80x86 flags are of little use after the execution of this instruction. Either the ECX register is zero (in which case the two strings are totally different), or it contains the number of elements compared in the two strings until a match. While this form of the CMPS instruction isn't particularly useful for comparing strings, it is useful for locating the first pair of matching items in a couple of byte, word, or double word arrays. In general, though, you'll rarely use the REPNE prefix with CMPS.

One last thing to keep in mind with using the CMPS instruction – the value in the ECX register determines the number of elements to process, not the number of bytes. Therefore, when using CMPSW, ECX specifies the number of words to compare. This, of course, is twice the number of bytes to compare. Likewise, for CMPSD, ECX contains the number of double words to process.

6.2.6 The SCAS Instruction

The CMPS instruction compares two strings against one another. You do not use it to search for a particular element within a string. For example, you could not use the CMPS instruction to quickly scan for a zero throughout some other string. You can use the SCAS (scan string) instruction for this task.

Unlike the MOVS and CMPS instructions, the SCAS instruction only requires a destination string (pointed at by EDI) rather than both a source and destination string. The source operand is the value in the AL (SCASB), AX (SCASW), or EAX (SCASD) register. The SCAS instruction compares the value in the accumulator (AL, AX, or EAX) against the value pointed at by EDI and then increments (or decrements) EDI by one, two, or four. The CPU sets the flags according to the result of the comparison. While this might be useful on occasion, SCAS is a lot more useful when using the REPE and REPNE prefixes.

With the REPE prefix (repeat while equal), SCAS scans the string searching for an element which does not match the value in the accumulator. When using the REPNE prefix (repeat while not equal), SCAS scans the string searching for the first string element which is equal to the value in the accumulator.

You're probably wondering "why do these prefixes do exactly the opposite of what they ought to do?" The paragraphs above haven't quite phrased the operation of the SCAS instruction properly. When using the REPE prefix with SCAS, the 80x86 scans through the string while the value in the accumulator is equal to the string operand. This is equivalent to searching through the string for the first element which does not match the value in the accumulator. The SCAS instruction with REPNE scans through the string while the accumulator is not equal to the string operand. Of course, this form searches for the first value in the string which matches the value in the accumulator register. The SCAS instructions take the following forms:

```
scasb()
scasw()
scasd()

repe.scasb()
repe.scasw()
repe.scasd()

repne.scasb()
repne.scasw()
repne.scasd()
```

Like the CMPS and MOVS instructions, the value in the ECX register specifies the number of elements to process, not bytes, when using a repeat prefix.

6.2.7 The STOS Instruction

The STOS instruction stores the value in the accumulator at the location specified by EDI. After storing the value, the CPU increments or decrements EDI depending upon the state of the direction flag. Although the STOS instruction has many uses, its primary use is to initialize arrays and strings to a constant value. For example, if you have a 256-byte array you want to clear out with zeros, use the following code:

```
cld();
lea( edi, DestArray );
mov( 64, ecx );      // 64 double words = 256 bytes.
xor( eax, eax );     // Zero out EAX.
rep.stosd();
```

This code writes 64 double words rather than 256 bytes because a single STOSD operation is faster than four STOSB operations.

The STOS instructions take four forms. They are

```
stosb();
stosw();
```

```

stosd() ;

rep.stosb() ;
rep.stosw() ;
rep.stosd() ;

```

The STOSB instruction stores the value in the AL register into the specified memory location(s), the STOSW instruction stores the AX register into the specified memory location(s) and the STOSD instruction stores EAX into the specified location(s).

Keep in mind that the STOS instruction is useful only for initializing a byte, word, or double word array to a constant value. If you need to initialize an array to different values, you cannot use the STOS instruction. See the exercises for additional details.

6.2.8 The LODS Instruction

The LODS instruction is unique among the string instructions. You will probably never use a repeat prefix with this instruction. The LODS instruction copies the byte, word, or double word pointed at by ESI into the AL, AX, or EAX register, after which it increments or decrements the ESI register by one, two, or four. Repeating this instruction via the repeat prefix would serve no purpose whatsoever since the accumulator register will be overwritten each time the LODS instruction repeats. At the end of the repeat operation, the accumulator will contain the last value read from memory.

Instead, use the LODS instruction to fetch bytes (LODSB), words (LODSW), or double words (LODSD) from memory for further processing. By using the STOS instruction, you can synthesize powerful string operations.

Like the STOS instruction, the LODS instructions take four forms:

```

lodsb() ;
lodsw() ;
lodsd() ;

rep.lodsb() ;
rep.lodsw() ;
rep.lodsd() ;

```

As mentioned earlier, you'll rarely, if ever, use the REP prefixes with these instructions⁴. The 80x86 increments or decrements ESI by one, two, or four depending on the direction flag and whether you're using the LODSB, LODSW, or LODSD instruction.

6.2.9 Building Complex String Functions from LODS and STOS

The 80x86 supports only five different string instructions: MOVS, CMPS, SCAS, LODS, and STOS⁵. These certainly aren't the only string operations you'll ever want to use. However, you can use the LODS and STOS instructions to easily generate any particular string operation you like. For example, suppose you wanted a string operation that converts all the upper case characters in a string to lower case. You could use the following code:

```

mov( StringAddress, esi ); // Load string address into ESI.
mov( esi, edi );           // Also point EDI here.
mov( (type str.strrec [esi]).length, ecx );

repeat

```

4. They appear here simply because they are allowed. They're not useful, but they are allowed.

5. Not counting INS and OUTS which we're ignoring here.

```
lodsb();           // Get the next character in the string.
if( al in 'A'..'Z' ) then

    or( $20, al );  // Convert upper case character to lower case.

endif;
stosb();           // Store converted character back into string.
dec( ecx );

until( ecx == 0 );
```

Since the LODS and STOS instructions use the accumulator as an intermediary, you can use any accumulator operation to quickly manipulate string elements.

6.3 Putting It All Together

In this chapter we took a quick look at the 80x86's string instructions. We studied their implementation and saw how to use them. These instructions are quite useful for synthesizing character set functions (see the source code for the HLA Standard Library string module for examples). We also saw how to use these instructions for non-character string purpose such as moving large blocks of memory (i.e., assigning one array to another) and comparing large integer values. For more information on the use of these instructions, please see the volume on Advanced String Handling.

The HLA Compile-Time Language

Chapter Seven

6.1 Chapter Overview

Now we come to the fun part. For the past nine chapters this text has been molding and conforming you to deal with the HLA language and assembly language programming in general. In this chapter you get to turn the tables; you'll learn how to force HLA to conform to your desires. This chapter will teach you how to extend the HLA language using HLA's *compile-time language*. By the time you are through with this chapter, you should have a healthy appreciation for the power of the HLA compile-time language. You will be able to write short compile-time programs. You will also be able to add new statements, of your own choosing, to the HLA language.

6.2 Introduction to the Compile-Time Language (CTL)

HLA is actually two languages rolled into a single program. The *run-time language* is the standard 80x86/HLA assembly language you've been reading about for the past nine chapters. This is called the run-time language because the programs you write execute when you run the executable file. HLA contains an interpreter for a second language, the HLA Compile-Time Language (or CTL) that executes programs while HLA is compiling a program. The source code for the CTL program is embedded in an HLA assembly language source file; that is, HLA source files contain instructions for both the HLA CTL and the run-time program. HLA executes the CTL program during compilation. Once HLA completes compilation, the CTL program terminates; the CTL application is not a part of the run-time executable that HLA emits, although the CTL application can *write* part of the run-time program for you and, in fact, this is the major purpose of the CTL.

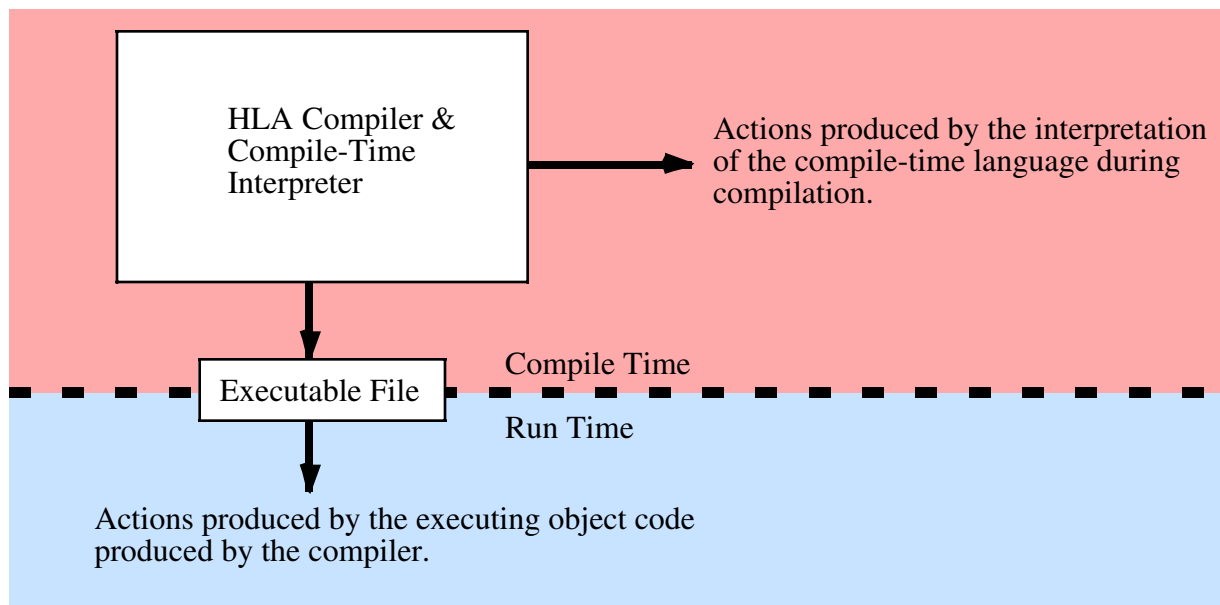


Figure 6.1 Compile-Time vs. Run-Time Execution

It may seem confusing to have two separate languages built into the same compiler. Perhaps you're even questioning why anyone would need a compile time language. To understand the benefits of a compile time language, consider the following statement that you should be very comfortable with at this point:

```
stdout.put( "i32=", i32, " strVar=", strVar, " charVar=", charVar, nl );
```

This statement is neither a statement in the HLA language nor a call to some HLA Standard Library procedure. Instead, *stdout.put* is actually a statement in a CTL application provided by the HLA Standard Library. The *stdout.put* "application" processes a list of objects (the parameter list) and makes calls to various other Standard Library procedures; it chooses the procedure to call based on the type of the object it is currently processing. For example, the *stdout.put* "application" above will emit the following statements to the run-time executable:

```
stdout.puts( "i32=" );
stdout.puti32( i32 );
stdout.puts( " strVar=" );
stdout.puts( strVar );
stdout.puts( " charVar=" );
stdout.putc( charVar );
stdout.newln();
```

Clearly the *stdout.put* statement is much easier to read and write than the sequence of statements that *stdout.put* emits in response to its parameter list. This is one of the more powerful capabilities of the HLA programming language: the ability to modify the language to simplify common programming tasks. Printing lots of different data objects in a sequential fashion is a common task; the *stdout.put* "application" greatly simplifies this process.

The HLA Standard Library is *loaded* with lots of HLA CTL examples. In addition to standard library usage, the HLA CTL is quite adept at handling "one-off" or "one-use" applications. A classic example is filling in the data for a lookup table. In Chapter Six this text noted that it is possible to construct look-up tables using the HLA CTL (see "Tables" on page 625 and "Generating Tables" on page 629). Not only is this possible, but it is often far less work to use the HLA CTL to construct these look-up tables. This chapter abounds with examples of exactly this application of the CTL.

Although the CTL itself is relatively inefficient, it does maximize the use of that one precious commodity of which there is so little available: your time. By learning how to use the HLA CTL, and applying it properly, you can develop assembly language applications as rapidly as high level language applications (even faster since HLA's CTL lets you create *very* high level language constructs).

6.3 The #PRINT and #ERROR Statements

Chapter One of this textbook began with the typical first program most people write when learning a new language; the "Hello World" program. It is only fitting for this chapter to present that same program when discussing the second language of this text. So here it is, the basic "Hello World" program written in the HLA compile time language:

```
program ctlHelloWorld;
begin ctlHelloWorld;

    #print( "Hello, World of HLA/CTL" )

end ctlHelloWorld;
```

Program 6.1 The CTL "Hello World" Program

The only CTL statement in this program is the `#print` statement. The remaining lines are needed just to keep the compiler happy (we could reduced the overhead to two lines by using a `UNIT` rather than a `PROGRAM` declaration).

The `#PRINT` statement displays the textual representation of its argument list during the compilation of an HLA program. Therefore, if you compile the program above with the command `"hla ctlHW.hla"` the HLA compiler will immediately print, before returning control to the command line, the text:

```
Hello, World of HLA/CTL
```

Note that there is a big difference between the following two statements in an HLA source file:

```
#print( "Hello World" )
stdout.puts( "Hello World" nl );
```

The first statement prints "Hello World" (and a newline) during the compilation process. This first statement does not have any effect on the executable program. The second line doesn't affect the compilation process (other than the emission of code to the executable file). However, when you run the executable file, the second statement prints the string "Hello World" followed by a new line sequence.

The HLA/CTL `#PRINT` statement uses the following basic syntax:

```
#print( list_of_comma_separated_constants )
```

Note that a semicolon does not terminate this statement. Semicolons terminate run-time statements, they generally do not terminate compile-time statements (there is one big exception, as you will see a little later).

The `#PRINT` statement must have at least one operand; if multiple operands appear in the parameter list, you must separate each operand with a comma (just like `stdout.put`). If a particular operand is not a string constant, HLA will translate that constant to its corresponding string representation and print that string. Example:

```
#print( "A string Constant ", 45, ' ', 54.9, ' ', true )
```

You may specify named symbolic constants and constant expressions. However, all `#PRINT` operands must be constants (either literal constants or constants you define in the `CONST` or `VAL` sections) and those constants must be defined before you use them in the `#PRINT` statement. Example:

```
const
    pi := 3.14159;
    charConst := 'c';

#print( "PI = ", pi, " CharVal=", CharConst )
```

The HLA `#PRINT` statement is particularly invaluable for debugging CTL programs (since there is no debugger available for CTL code). This statement is also useful for displaying the progress of the compilation and displaying assumptions and default actions that take place during compilation. Other than displaying the text associated with the `#PRINT` parameter list, the `#PRINT` statement does not have any affect on the compilation of the program.

The `#ERROR` statement allows a single string constant operand. Like `#PRINT` this statement will display the string to the console during compilation. However, the `#ERROR` statement treats the string as a error message and displays the string as part of an HLA error diagnostic. Further, the `#ERROR` statement increments the error count and this will cause HLA to stop the compilation (without assembly or linking) at the conclusion of the source file. You would normally use the `#ERROR` statement to display an error message during compilation if your CTL code discovers something that prevents it from creating valid code. Example:

```
#error( "Statement must have exactly one operand" )
```

Like the `#PRINT` statement, the `#ERROR` statement does not end with a semicolon. Although `#ERROR` only allows a string operand, it's very easy to print other values by using the string (constant) concatenation operator and several of the HLA built-in compile-time functions (see "Compile-Time Constants and Variables" on page 922 and "Compile-Time Functions" on page 925) for more details).

6.4 Compile-Time Constants and Variables

Just as the run-time language supports constants and variables, so does the compile-time language. You declare compile-time constants in the CONST section, the same as for the run-time language. You declare compile-time variables in the VAL section. Objects you declare in the VAL sections are constants as far as the run-time language is concerned, but remember that you can change the value of an object you declare in the VAL section throughout the source file. Hence the term "compile-time variable." See "HLA Constant and Value Declarations" on page 381 for more details.

The CTL assignment statement ("?:") computes the value of the constant expression to the right of the assignment operator (":=") and stores the result into the VAL object name appearing immediately to the left of the assignment operator¹. The following example is a rework of the example above; this example, however, may appear anywhere in your HLA source file, not just in the VAL section of the program.

```
?ConstToPrint := 25;
#print( "ConstToPrint = ", ConstToPrint )
?ConstToPrint := ConstToPrint + 5;
#print( "Now ConstToPrint = ", ConstToPrint )
```

Note that HLA's CTL ignores the distinction between the different sizes of numeric objects. HLA always reserves storage for the largest possible object of a given type, so HLA merges the following types:

```
byte, word, dword -> dword
uns8, uns16, uns32 -> uns32
int8, int16, int32 -> int32
real32, real64, real80 -> real80
```

For most practical applications of the CTL, this shouldn't make a difference in the operation of the program.

6.5 Compile-Time Expressions and Operators

As the previous section states, the HLA CTL supports constant expressions in the CTL assignment statement. Unlike the run-time language (where you have to translate algebraic notation into a sequence of machine instructions), the HLA CTL allows a full set of arithmetic operations using familiar expression syntax. This gives the HLA CTL considerable power, especially when combined with the built-in compile-time functions the next section discusses.

HLA's CTL supports the following operators in compile-time expressions:

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
- (unary)	numeric	Negates the specific numeric value (int, uns, real).
	cset	Returns the complement of the specified character set.
! (unary)	integer	Inverts all the bits in the operand (bitwise not).
	boolean	Boolean NOT of the operand.
*	numericL * numericR	Multiplies the two operands.
	csetL * csetR	Computes the intersection of the two sets

1. If the identifier to the left of the assignment operator is undefined, HLA will automatically declare this object at the current scope level.

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
div	integerL div integerR	Computes the quotient of the two integer (int/uns/dword) operands.
mod	integerL mod integerR	Computes the remainder of the division of the two integer (int/uns/dword) operands.
/	numericL / numericR	Computes the real quotient of the two numeric operands. Returns a real result even if both operands are integers.
<<	integerL << integerR	Shifts integerL operand to the left the number of bits specified by the integerR operand.
>>	integerL >> integerR	Shifts integerL operand to the right the number of bits specified by the integerR operand.
+	numericL + numericR	Adds the two numeric operands.
	csetL + csetR	Computes the union of the two sets.
	strL + strR	Concatenates the two strings.
-	numericL - numericR	Computes the difference between numericL and numericR.
	csetL - csetR	Computes the set difference of csetL-csetR.
= or ==	numericL = numericR	Returns true if the two operands have the same value.
	csetL = csetR	Returns true if the two sets are equal.
	strL = strR	Returns true if the two strings/chars are equal.
	typeL = typeR	Returns true if the two values are equal. They must be the same type.
<> or !=	typeL <> typeR (Same as =)	Returns false if the two (compatible) operands are not equal to one another.
<	numericL < numericR	Returns true if numericL is less than numericR.
	csetL < csetR	Returns true if csetL is a proper subset of csetR.
	strL < strR	Returns true if strL is less than strR
	booleanL < booleanR	Returns true if left operand is less than right operand (note: false < true).
	enumL < enumR	Returns true if enumL appears in the same enum list as enumR and enumL appears first.
<=	Same as <	Returns true if the left operand is less than or equal to the right operand. For character sets, this means that the left operand is a subset of the right operand.
>	Same as <	Returns true if the left operand is greater than the right operand. For character sets, this means that the left operand is a proper superset of the right operand.

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
>=	Same as <=	Returns true if the left operand is greater than or equal to the right operand. For character sets, this means that the left operand is a superset of the right operand.
& or &&	integerL & integerR	Computes the bitwise AND of the two operands.
	booleanL & booleanR	Computes the logical AND of the two operands.
or	integerL integerR	Computes the bitwise OR of the two operands.
	booleanL booleanR	Computes the logical OR of the two operands.
^	integerL ^ integerR	Computes the bitwise XOR of the two operands.
	booleanL ^ booleanR	Computes the logical XOR of the two operands. Note that this is equivalent to "booleanL <> booleanR".
in	charL in csetR	Returns true if charL is a member of csetR.

a. Numeric is {intXX, unsXX, byte, word, dword, and realXX} values. Cset is a character set operand. Type integer is { intXX, unsXX, byte, word, dword }. Type str is any string or character value. "TYPE" indicates an arbitrary HLA type. Other types specify an explicit HLA data type.

Table 2: Operator Precedence and Associativity

Associativity	Precedence (Highest to Lowest)	Operator
Right-to-left	6	!
		-
Left to right	5	*
		div
		mod
		/
		>>
		<<
Left to right	4	+
		-

Table 2: Operator Precedence and Associativity

Associativity	Precedence (Highest to Lowest)	Operator
Left to right	3	= or ==
		<> or !=
		<
		<=
		>
		>=
Left to right	2	& or &&
		or
		^
Nonassociative	1	in

Of course, you can always override the default precedence and associativity of an operator by using parentheses in an expression.

6.6 Compile-Time Functions

HLA provides a wide range of compile-time functions you can use. These functions compute values during compilation the same way a high level language function computes values at run-time. The HLA compile-time language includes a wide variety of numeric, string, and symbol table functions that help you write sophisticated compile-time programs.

Most of the names of the built-in compile-time functions begin with the special symbol "@" and have names like `@sin` or `@length`. The use of these special identifiers prevents conflicts with common names you might want to use in your own programs (like `length`). The remaining compile-time functions (those that do not begin with "@") are typically data conversion functions that use type names like `int8` and `real64`. You can even create your own compile-time functions using macros (see "Macros" on page 939).

HLA organizes the compile-time functions into various classes depending on the type of operation. For example, there are functions that convert constants from one form to another (e.g., string to integer conversion), there are many useful string functions, and HLA provides a full set of compile-time numeric functions.

The complete list of HLA compile-time functions is too lengthy to present here. Instead, a complete description of each of the compile-time objects and functions appears in Appendix H (see "HLA Compile-Time Functions" on page 1325); this section will highlight a few of the functions in order to demonstrate their use. Later sections in this chapter, as well as future chapters, will make extensive use of the various compile-time functions.

Perhaps the most important concept to understand about the compile-time functions is that they are equivalent to constants in your assembly language code (i.e., the run-time program). For example, the compile-time function invocation `"@sin(3.1415265358979328)"` is roughly equivalent to specifying "0.0" at that point in your program². A function invocation like `"@sin(x)"` is legal only if `x` is a constant with a previous declaration at the point of the function call in the source file. In particular, `x` cannot be a run-time vari-

2. Actually, since `@sin`'s parameter in this example is not exactly π , you will get a small positive number instead of zero as the function result, but in theory you should get zero.

able or other object whose value exists at run-time rather than compile-time. Since HLA replaces compile-time function calls with their constant result, you may ask why you should even bother with compile time functions. After all, it's probably more convenient to type "0.0" than it is to type "@sin(3.1415265358979328)" in your program. However, compile-time functions are really handy for generating lookup tables (see "Generating Tables" on page 629) and other mathematical results that may change whenever you change a CONST value in your program. Later sections in this chapter will explore these ideas farther.

6.6.1 Type Conversion Compile-time Functions

One set of commonly used compile-time functions are the type conversion functions. These functions take a single parameter of one type and convert that information to some specified type. These functions use several of the HLA built-in data type names as the function names. Functions in this category are

- boolean
- int8, int16, and int32
- uns8, uns16, and uns32
- byte, word, dword (these are effectively equivalent to uns8, uns16, and uns32)
- real32, real64, and real80
- char
- string
- cset
- text

These functions accept a single constant expression parameter and, if at all reasonable, convert that expression's value to the type specified by the type name. For example, the following function call returns the value -128 since it converts the string constant to the corresponding integer value:

```
int8( "-128" )
```

Certain conversions don't make sense or have restrictions associated with them. For example, the *boolean* function will accept a string parameter, but that string must be "true" or "false" or the function will generate a compile-time error. Likewise, the numeric conversion functions (e.g., *int8*) allow a string operand but the string operand must represent a legal numeric value. Some conversions (e.g., *int8* with a character set parameter) simply don't make sense and are always illegal.

One of the most useful functions in this category is the *string* function. This function accepts nearly all constant expression types and it generates a string that represents the parameter's data. For example, the invocation "string(128)" produces the string "128" as the return result. This function is real handy when you have a value that you wish to use where HLA requires a string. For example, the #ERROR compile-time statement only allows a single string operand. You can use the string function and the string concatenation operator ("+") to easily get around this limitation, e.g.,

```
#error( "Value ( " + string( Value ) + " ) is out of range" )
```

6.6.2 Numeric Compile-Time Functions

The functions in this category perform standard mathematical operations at compile time. These functions are real handy for generating lookup tables and "parameterizing" your source code by recalculating functions on constants defined at the beginning of your program. Functions in this category include the following:

- @abs
- @ceil, @floor
- @sin, @cos, @tan
- @exp, @log, @log10
- @min, @max

- @random, @randomize
- @sqrt

See Appendix H for more details on these functions.

6.6.3 Character Classification Compile-Time Functions

The functions in this group all return a boolean result. They test a character (or all the characters in a string) to see if it belongs to a certain class of characters. The functions in this category include

- @isAlpha, @isAlphanum
- @isDigit, @isxDigit
- @isLower, @isUpper
- @isSpace

In addition to these character classification functions, the HLA language provides a set of pattern matching functions that you can also use to classify character and string data. See the appropriate sections a little later for the discussion of these routines.

6.6.4 Compile-Time String Functions

The functions in this category operate on string parameters. Most return a string result although a few (e.g., @length and @index) return integer results. These functions do not directly affect the values of their parameters; instead, they return an appropriate result that you can assign back to the parameter if you wish to do so.

- @delete, @insert
- @index, @rindex
- @length
- @lowercase, @uppercase
- @strbrk, @strspan
- @strset
- @substr, @tokenize, @trim

For specific details concerning these functions and their parameters and their types, see Appendix H. Combined with the pattern matching functions, the string handling functions give you the ability to extend the HLA language by processing textual data that appears in your source files. Later sections appearing in this chapter will discuss ways to do this.

The @length function deserves a special discussion because it is probably the most popular function in this category. It returns an *uns32* constant specifying the number of characters found in its string parameter. The syntax is the following:

```
@length( string_expression )
```

Where *string_expression* represents any compile-time string expression. As noted above, this function returns the length, in characters, of the specified expression.

6.6.5 Compile-Time Pattern Matching Functions

HLA provides a very rich set of string/pattern matching functions that let you test a string to see if it begins with certain types of characters or strings. Along with the string processing functions, the pattern matching functions let you extend the HLA language and provide several other benefits as well. There are far too many pattern matching functions to list here (see Appendix H for complete details). However, a few examples will demonstrate the power and convenience of these routines.

The pattern matching functions all return a boolean true/false result. If a function returns true, we say that the function *succeeds* in matching its operand. If the function returns false, then we say it *fails* to match its operand. An important feature of the pattern matching functions is that they do not have to match the entire string you supply as a parameter, these patterns will (usually) succeed as long as they match a prefix of the string parameter. The `@matchStr` function is a good example, the following function invocation always returns true:

```
@matchStr( "Hello World", "Hello" )
```

The first parameter of all the pattern matching functions ("Hello World" in this example) is the string to match. The matching functions will attempt to match the characters at the beginning of the string with the other parameters supplied for that particular function. In the `@matchStr` example above, the function succeeds if the first parameter begins with the string specified as the second parameter (which it does). The fact that the "Hello World" string contains additional characters beyond "Hello" is irrelevant; it only needs to begin with the string "Hello" is doesn't require equality with "Hello".

Most of the compile-time pattern matching functions support two optional parameters. The functions store additional data into the VAL objects specified by these two parameters if the function is successful (conversely, if the function fails, it does not modify these objects). The first parameter is where the function stores the *remainder*. The remainder after the execution of a pattern matching function is those characters that follow the matched characters in the string. In the example above, the remainder would be " World". If you wanted to capture this remainder data, you would add a third parameter to the `@matchStr` function invocation:

```
@matchStr( "Hello World", "Hello", World )
```

This function invocation would leave " World" sitting in the *World* VAL object. Note that *World* must be pre-declared in the VAL section (or via the "?" statement) prior to the invocation of this function.

By using the conjunction operator ("&") you can combine several pattern matching functions into a single expression, e.g.,

```
@matchStr( "Hello There World", "Hello ", tw ) & @matchStr( tw, "There ", World )
```

This full expression returns true and leaves "World" sitting in the *World* variable. It also leaves "There World" sitting in *tw*, although *tw* is probably a temporary object whose value has no meaning beyond this expression. Of course, the above could be more efficiently implemented as follows:

```
@matchStr( "Hello There World", "Hello There", World )
```

However, keep in mind that you can combine different pattern matching functions using conjunction, they needn't all be calls to `@matchStr`.

The second optional parameter to most pattern matching functions holds a copy of the text that the function matched. E.g., the following call to `@matchStr` returns "Hello" in the Hello VAL object³

```
@matchStr( "Hello World", "Hello", World, Hello )
```

For more information on these pattern matching functions please see Appendix H. The chapter on Domain Specific Languages (see "Domain Specific Embedded Languages" on page 973) and several other sections in this chapter will also make further use of these functions.

6.6.6 Compile-Time Symbol Information

During compilation HLA maintains an internal database known as the *symbol table*. The symbol table contains lots of useful information concerning all the identifiers you've defined up to a given point in the program. In order to generate machine code output, HLA needs to query this database to determine how to

3. Strictly speaking, this example is rather contrived since we generally know the string that `@matchStr` matches. However, for other pattern matching functions this is not the case.

treat certain symbols. In your compile-time programs, it is often necessary to query the symbol table to determine how to handle an identifier or expression in your code. The HLA compile-time symbol information functions handle this task.

Many of the compile-time symbol information functions are well beyond the scope of this chapter (and, in some cases, beyond the scope of this text). This chapter will present a few of the functions and later chapters will add to this list. For a complete list of the compile-time symbol table functions, see Appendix H. The functions we will consider in this chapter include the following:

- @size
- @defined
- @typeName
- @elements
- @elementSize

Without question, the @size function is probably the most important function in this group. Indeed, previous chapters have made use of this function already. The @size function accepts a single HLA identifier or constant expression as a parameter. It returns the size, in bytes, of the data type of that object (or expression). If you supply an identifier, it can be a constant, type, or variable identifier. HLA returns the size of the type of the object. As you've seen in previous chapters, this function is invaluable for allocating storage via *malloc* and allocating arrays.

Another very useful function in this group is the @defined function. This function accepts a single HLA identifier as a parameter, e.g.,

```
@defined( MyIdentifier )
```

This function returns true if the identifier is defined at that point in the program, it returns false otherwise.

The @typeName function returns a string specifying the type name of the identifier or expression you supply as a parameter. For example, if *i32* is an *int32* object, then "@typeName(i32)" returns the string "int32". This function is useful for testing the types of objects you are processing in your compile-time programs.

The @elements function requires an array identifier or expression. It returns the total number of array elements as the function result. Note that for multi-dimensional arrays this function returns the product of all the array dimensions⁴.

The @elementSize function returns the size, in bytes, of an element of an array whose name you pass as a parameter. This function is extremely valuable for computing indices into an array (i.e., this function computes the *element_size* component of the array index calculation, see "Accessing Elements of a Single Dimension Array" on page 447).

6.6.7 Compile-Time Expression Classification Functions

The HLA compile-time language provides functions that will classify some arbitrary text and determine if that text is a constant expression, a register, a memory operand, a type identifier, and more. Some of the more common functions are

- @isConst
- @isReg, @isReg8, @isReg16, @isReg32, @isFReg
- @isMem
- @isType

Except for @isType, which requires an HLA identifier as a parameter, these functions all take some arbitrary text as their parameter. These functions return true or false depending upon whether that parameter satisfies the function requirements (e.g., @isConst returns true if its parameter is a constant identifier or expression). The @isType function returns true if its parameter is a type identifier.

4. There is an @dim function that returns an array specifying the bounds on each dimension of a multidimensional array. See the appendices for more details if you're interested in this function.

The HLA compile-time language includes several other classification functions that are beyond the scope of this chapter. See Appendix H for details on those functions.

6.6.8 Miscellaneous Compile-Time Functions

The HLA compile-time language contains several additional functions that don't fall into one of the categories above. Some of the more useful miscellaneous functions include

- @odd
- @lineNumber
- @text

The @odd function takes an ordinal value (i.e., non-real numeric or character) as a parameter and returns true if the value is odd, false if it is even. The @lineNumber function requires no parameters, it returns the current line number in the source file. This function is quite useful for debugging compile-time (and run-time!) programs.

The @text function is probably the most useful function in this group. It requires a single string parameter. It expands that string as text in place of the @text function call. This function is quite useful in conjunction with the compile-time string processing functions. You can build an instruction (or a portion of an instruction) using the string manipulation functions and then convert that string to program source code using the @text function. The following is a trivial example of this function in operation:

```
?id1:string := "eax";
?id2:string := "i32";
@text( "mov( " + id1 + ", " + id2 + ");" )
```

The sequence above compiles to

```
mov( eax, i32 );
```

6.6.9 Predefined Compile-Time Variables

In addition to functions, HLA also includes several predefined compile-time variables. The use of most of HLA's compile time variables is beyond the scope of this text. However, the following you've already seen:

- @bound
- @into

Volume Three (see "Some Additional Instructions: INTMUL, BOUND, INTO" on page 377) discusses the use of these objects to control the emission of the INTO and BOUND instructions. These two boolean pseudo-variables determine whether HLA will compile the BOUND (@bound) and INTO (@into) instructions or treat them as comments. By default, these two variables contain true and HLA will compile these instructions to machine code. However, if you set these values to false, using one or both of the following statements then HLA will not compile the associated statement:

```
?@bound := false;
?@into := false;
```

If you set @bound to false, then HLA treats BOUND instructions as though they were comments. If you set @into to false, then HLA treats INTO instructions as comments. You can control the emission of these statements throughout your program by selectively setting these pseudo-variables to true or false at different points in your code.

6.6.10 Compile-Time Type Conversions of TEXT Objects

Once you create a text constant in your program, it's difficult to manipulate that object. The following example demonstrates a programmer's desire to change the definition of a text symbol within a program:

```
val
  t:text := "stdout.put";
  .
  .
  .
  ?t:text := "fileio.put";
```

The basic idea in this example is that *t* expands to "stdout.put" in the first half of the code and it expands to "fileio.put" in the second half of the program. Unfortunately, this simple example will not work. The problem is that HLA will expand a text symbol in place almost anywhere it finds that symbol. This includes occurrences of *t* within the "?" statement above. Therefore, the code above expands to the following (incorrect) text:

```
val
  t:text := "stdout.put";
  .
  .
  .
  ?stdout.put:text := "fileio.put";
```

HLA doesn't know how to deal with the "?" statement above, so it generates a syntax error.

Other times you may not want HLA to expand a text object. Your code may want to further process the string data held by the text object. HLA provides a couple of operators that deal with these two problems:

- @string:identifier
- @toString:identifier

The @string:identifier operator consists of @string, immediately followed by a colon and a text identifier (with no interleaving spaces or other characters). HLA returns a string constant corresponding to the text data associated with the text object. In other words, this operator lets you treat a text object as though it were a string constant within an expression.

Unfortunately, the @string operator converts a text object to a string constant, not a string identifier. Therefore, you cannot say something like

```
?@string:t := "Hello"
```

This doesn't work because @string:t replaces itself with the string constant associated with the text object *t*. Given the former assignment to *t*, this statement expands to

```
? "stdout.put" := "Hello";
```

This statement is still illegal.

The @toString:identifier operator comes to the rescue in this case. The @toString operator requires a text object as the associated identifier. It converts this text object to a string object (still maintaining the same string data) and then returns the identifier. Since the identifier is now a string object, you can assign a value to it (and change its type to something else, e.g., *text*, if that's what you need). To achieve the original goal, therefore, you'd use code like the following:

```
val
  t:text := "stdout.put";
  .
  .
  .
  ?@toString:t : text := "fileio.put";
```

6.7 Conditional Compilation (Compile-Time Decisions)

HLA's compile-time language provides an IF statement, #IF, that lets you make various decisions at compile-time. The #IF statement has two main purposes: the traditional use of #IF is to support *conditional compilation* (or *conditional assembly*) allowing you to include or exclude code during a compilation depending on the status of various symbols or constant values in your program. The second use of this statement is to support the standard IF statement decision making process in the HLA compile-time language. This section will discuss these two uses for the HLA #IF statement.

The simplest form of the HLA compile-time #IF statement uses the following syntax:

```
#if( constant_boolean_expression )

    << text >>

#endif
```

Note that you do not place semicolons after the #ENDIF clause. If you place a semicolon after the #ENDIF, it becomes part of the source code and this would be identical to inserting that semicolon immediately before the next text item in the program.

At compile-time, HLA evaluates the expression in the parentheses after the #IF. This must be a constant expression and its type must be boolean. If the expression evaluates true, then HLA continues processing the text in the source file as though the #IF statement were not present. However, if the expression evaluates false, then HLA treats all the text between the #IF and the corresponding #ENDIF clause as though it were a comment (i.e., it ignores this text).

```
#if( constant_boolean_expression )
```

HLA compiles this code if the expression is true. Else HLA treats this code like a comment.

```
#endif
```

Figure 6.1 Operation of HLA Compile-Time #IF Statement

Keep in mind that HLA's constant expressions support a full expression syntax like you'd find in a high level language like C or Pascal. The #IF expression syntax is not limited as are expressions in the HLA IF statement. Therefore, it is perfectly reasonable to write fancy expressions like the following:

```
#if( @length( someStrConst ) < 10 && ( MaxItems < 100 || MinItems < 10 ) )

    << text >>

#endif
```

Keep in mind that the items in a compile-time expression must all be CONST or VAL identifiers or an HLA compile-time function call (with appropriate parameters). In particular, remember that HLA evaluates these expressions at compile-time so they cannot contain run-time variables⁵. Also note that HLA's compile time language uses complete boolean evaluation, so any side effects that occur in the expression may produce undesired results.

5. Except, of course, as parameters to certain HLA compile-time functions like @size or @typeName.

The HLA `#IF` statement supports optional `#ELSEIF` and `#ELSE` clauses that behave in the intuitive fashion. The complete syntax for the `#IF` statement looks like the following:

```
#if( constant_boolean_expression1 )

    << text1 >>

#elseif( constant_boolean_expression2 )

    << text2 >>

#else

    << text3 >>

#endif
```

If the first boolean expression evaluates true then HLA processes the text up to the `#ELSEIF` clause. It then skips all text (i.e., treats it like a comment) until it encounters the `#ENDIF` clause. HLA continues processing the text after the `#ENDIF` clause in the normal fashion.

If the first boolean expression above evaluates false, then HLA skips all the text until it encounters a `#ELSEIF`, `#ELSE`, or `#ENDIF` clause. If it encounters a `#ELSEIF` clause (as above), then HLA evaluates the boolean expression associated with that clause. If it evaluates true, then HLA processes the text between the `#ELSEIF` and the `#ELSE` clauses (or to the `#ENDIF` clause if the `#ELSE` clause is not present). If, during the processing of this text, HLA encounters another `#ELSEIF` or, as above, a `#ELSE` clause, then HLA ignores all further text until it finds the corresponding `#ENDIF`.

If both the first and second boolean expressions in the example above evaluate false, then HLA skips their associated text and begins processing the text in the `#ELSE` clause. As you can see, the `#IF` statement behaves in a relatively intuitive fashion once you understand how HLA "executes" the body of these statements (that is, it processes the text or treats it as a comment depending on the state of the boolean expression). Of course, you can create a nearly infinite variety of different `#IF` statement sequences by including zero or more `#ELSEIF` clauses and optionally supplying the `#ELSE` clause. Since the construction is identical to the HLA `IF..THEN..ELSEIF..ELSE..ENDIF` statement, there is no need to elaborate further here.

A very traditional use of conditional compilation is to develop software that you can easily configure for several different environments. For example, the `FCOMIP` instruction makes floating point comparisons very easy but this instruction is available only on Pentium Pro and later processors. If you want to use this instruction on the processors that support it, and fall back to the standard floating point comparison on the older processors you would normally have to write two versions of the program - one with the `FCOMIP` instruction and one with the traditional floating point comparison sequence. Unfortunately, maintaining two different source files (one for newer processors and one for older processors) is very difficult. Most engineers prefer to use conditional compilation to embed the separate sequences in the same source file. The following example demonstrates how to do this.

```
const
    PentProOrLater: boolean := false; // Set true to use FCOMIxx instrs.
    .
    .
    .
    #if( PentProOrLater )

        fcomip();           // Compare st1 to st0 and set flags.

    #else

        fcomp();           // Compare st1 to st0.
        fstsw( ax );       // Move the FPU condition code bits
        sahf();            // into the FLAGS register.
```

```
#endif
```

As currently written, this code fragment will compile the three instruction sequence in the `#ELSE` clause and ignore the code between the `#IF` and `#ELSE` clauses (because the constant *PentProOrLater* is false). By changing the value of *PentProOrLater* to true, you can tell HLA to compile the single `FCOMIP` instruction rather than the three-instruction sequence. Of course, you can use the *PentProOrLater* constant in other `#IF` statements throughout your program to control how HLA compiles your code.

Note that conditional compilation does not let you create a single *executable* that runs efficiently on all processors. When using this technique you will still have to create two executable programs (one for Pentium Pro and later processors, one for the earlier processors) by compiling your source file twice; during the first compilation you must set the *PentProOrLater* constant to false, during the second compilation you must set this constant to true. Although you must create two separate executables, you need only maintain a single source file.

If you are familiar with conditional compilation in other languages, such as the C/C++ language, you may be wondering if HLA supports a statement like C's `"#ifdef"` statement. The answer is no, it does not. However, you can use the HLA compile-time function `@DEFINED` to easily test to see if a symbol has been defined earlier in the source file. Consider the following modification to the above code that uses this technique:

```
const
    // Note: uncomment the following line if you are compiling this
    // code for a Pentium Pro or later CPU.

    // PentProOrLater :=0; // Value and type are irrelevant
    .
    .
    .
    #if( @defined( PentProOrLater ) )

        fcomip();          // Compare st1 to st0 and set flags.

    #else

        fcomp();           // Compare st1 to st0.
        fstsw( ax );       // Move the FPU condition code bits
        sahf();            // into the FLAGS register.

    #endif
```

Another common use of conditional compilation is to introduce debugging and testing code into your programs. A typical debugging technique that many HLA programmers use is to insert "print" statements at strategic points throughout their code in order to trace through their code and display important values at various checkpoints. A big problem with this technique is that they must remove the debugging code prior to completing the project. The software's customer (or a student's instructor) probably doesn't want to see debugging output in the middle of a report the program produces. Therefore, programmers who use this technique tend to insert code temporarily and then remove the code once they run the program and determine what is wrong. There are at least two problems with this technique:

- Programmers often forget to remove some debugging statements and this creates defects in the final program, and
- After removing a debugging statement, these programmers often discover that they need that same statement to debug some different problem at a later time. Hence they are constantly inserting, removing, and inserting the same statements over and over again.

Conditional compilation can provide a solution to this problem. By defining a symbol (say, *debug*) to control debug output in your program, you can easily activate or deactivate *all* debugging output by simply modifying a single line of source code. The following code fragment demonstrates this:

```
const
```

```

debug: boolean := false;    // Set to true to activate debug output.
.
.
.
#if( debug )

    stdout.put( "At line ", @lineNumber, " i=", i, nl );

#endif

```

As long as you surround all debugging output statements with a `#IF` statement like the one above, you don't have to worry about debug output accidentally appearing in your final application. By setting the *debug* symbol to false you can automatically disable all such output. Likewise, you don't have to remove all your debugging statements from your programs once they've served their immediate purpose. By using conditional compilation, you can leave these statements in your code because they are so easy to deactivate. Later, if you decide you need to view this same debugging information during a program run, you won't have to reenter the debugging statement - you simply reactivate it by setting the *debug* symbol to true.

We will return to this issue of inserting debugging code into your programs in the section on macros.

Although program configuration and debugging control are two of the more common, traditional, uses for conditional compilation, don't forget that the `#IF` statement provides the basic conditional statement in the HLA compile-time language. You will use the `#IF` statement in your compile-time programs the same way you would use an `IF` statement in HLA or some other language. Later sections in this chapter will present lots of examples of using the `#IF` statement in this capacity.

6.8 Repetitive Compilation (Compile-Time Loops)

HLA's `#WHILE..#ENDWHILE` statement provides a compile-time loop construct. The `#WHILE` statement tells HLA to repetitively process the same sequence of statements during compilation. This is very handy for constructing data tables (see "Constructing Data Tables at Compile Time" on page 966) as well as providing a traditional looping structure for compile-time programs. Although you will not employ the `#WHILE` statement anywhere near as often as the `#IF` statement, this compile-time control structure is very important when writing advanced HLA programs.

The `#WHILE` statement uses the following syntax:

```

#while( constant_boolean_expression )

    << text >>

#endwhile

```

When HLA encounters the `#WHILE` statement during compilation, it will evaluate the constant boolean expression. If the expression evaluates false, then HLA will skip over the text between the `#WHILE` and the `#ENDWHILE` clause (the behavior is similar to the `#IF` statement if the expression evaluates false). If the expression evaluates true, then HLA will process the statements between the `#WHILE` and `#ENDWHILE` clauses and then "jump back" to the start of the `#WHILE` statement in the source file and repeat this process.

```
#while( constant_boolean_expression )
```

HLA repetitively compiles this code as long as the expression is true. It effectively inserts multiple copies of this statement sequence into your source file (the exact number of copies depends on the value of the loop control expression).

```
#endwhile
```

Figure 6.1 HLA Compile-Time #WHILE Statement Operation

To understand how this process works, consider the following program:

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

static
  ary: uns32[5] := [ 2, 3, 5, 8, 13 ];

begin ctWhile;

  ?i := 0;
  #while( i < 5 )

    stdout.put( "array[ ", i, " ] = ", ary[i*4], nl );
    ?i := i + 1;

  #endwhile

end ctWhile;
```

Program 6.2 #WHILE..#ENDWHILE Demonstration

As you can probably surmise, the output from this program is the following:

```
array[ 0 ] = 2
array[ 1 ] = 3
array[ 2 ] = 4
array[ 3 ] = 5
array[ 4 ] = 13
```

What is not quite obvious is how this program generates this output. Remember, the #WHILE..#ENDWHILE construct is a compile-time language feature, not a run-time control construct. Therefore, the #WHILE loop above repeats five times during *compilation*. On each repetition of the loop, the HLA compiler processes the statements between the #WHILE and #ENDWHILE clauses. Therefore, the program above is really equivalent to the following:

```

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

static
    ary: uns32[5] := [ 2, 3, 5, 8, 13 ];

begin ctWhile;

    stdout.put( "array[ ", 0, " ] = ", ary[0*4], nl );
    stdout.put( "array[ ", 1, " ] = ", ary[1*4], nl );
    stdout.put( "array[ ", 2, " ] = ", ary[2*4], nl );
    stdout.put( "array[ ", 3, " ] = ", ary[3*4], nl );
    stdout.put( "array[ ", 4, " ] = ", ary[4*4], nl );

end ctWhile;

```

Program 6.3 Program Equivalent to the Code in Program 6.2

As you can see, the #WHILE statement is very convenient for constructing repetitive code sequences. This is especially invaluable for unrolling loops. Additional uses of the #WHILE loop appear in later sections of this chapter.

6.9 Putting It All Together

The HLA compile-time language provides considerable power. With the compile-time language you can automate the generation of tables, selectively compile code for different environments, easily unroll loops to improve performance, and check the validity of code you're writing. Combined with macros and other features that HLA provides, the compile-time language is probably the premier feature of the HLA language – no other assembler provides comparable features. For more information about the HLA compile time language, be sure to read the next chapter on macros.

Macros

Chapter Eight

7.1 Chapter Overview

Now we come to the fun part. For the past nine chapters this text has been molding and conforming you to deal with the HLA language and assembly language programming in general. In this chapter you get to turn the tables; you'll learn how to force HLA to conform to your desires. This chapter will teach you how to extend the HLA language using HLA's *compile-time language*. By the time you are through with this chapter, you should have a healthy appreciation for the power of the HLA compile-time language. You will be able to write short compile-time programs. You will also be able to add new statements, of your own choosing, to the HLA language.

7.2 Macros (Compile-Time Procedures)

Macros are symbols that a language processor replaces with other text during compilation. Macros are great devices for replacing long repetitive sequences of text with much shorter sequences of text. In addition to the traditional role that macros play (e.g., "#define" in C/C++), HLA's macros also serve as the equivalent of a compile-time language procedure or function. Therefore, macros are very important in HLA's compile-time language; just as important as functions and procedures are in other high level languages.

Although macros are nothing new, HLA's implementation of macros far exceeds the macro processing capabilities of other programming languages. The following sections explore HLA's macro processing facilities and the relationship between macros and other HLA CTL control constructs.

7.2.1 Standard Macros

HLA supports a straight-forward macro facility that lets you define macros in a manner that is similar to declaring a procedure. A typical, simple, macro declaration takes the following form:

```
macro macroname;

    << macro body >>

endmacro;
```

Although macro and procedure declarations are similar, there are several immediate differences between the two that are obvious from this example. First, of course, macro declarations use the reserved word `MACRO` rather than `PROCEDURE`. Second, you do not begin the body of the macro with a "`BEGIN macroname;`" clause. This is because macros don't have a declaration section like procedures so there is no need for a keyword that separates the macro declarations from the macro body. Finally, you will note that macros end with the "`ENDMACRO;`" clause rather than "`END macroname;`" The following is a concrete example of a macro declaration:

```
macro neg64;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

endmacro;
```

Execution of this macro's code will compute the two's complement of the 64-bit value in EDX:EAX (see "Extended Precision NEG Operations" on page 844).

To execute the code associated with *neg64*, you simply specify the macro's name at the point you want to execute these instructions, e.g.,

```
mov( (type dword i64), eax );
mov( (type dword i64+4), edx );
neg64;
```

Note that you do *not* follow the macro's name with a pair of empty parentheses as you would a procedure call (the reason for this will become clear a little later).

Other than the lack of parentheses following *neg64*'s invocation¹ this looks just like a procedure call. You could implement this simple macro as a procedure using the following procedure declaration:

```
procedure neg64p;
begin neg64p;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

end neg64p;
```

Note that the following two statements will both negate the value in EDX:EAX:

```
neg64;           neg64p();
```

The difference between these two (i.e., the macro invocation versus the procedure call) is the fact that macros expand their text in-line whereas a procedure call emits a call to the associate procedure elsewhere in the text. That is, HLA replaces the invocation "neg64;" directly with the following text:

```
neg( edx );
neg( eax );
sbb( 0, edx );
```

On the other hand, HLA replaces the procedure call "neg64p();" with the single call instruction:

```
call neg64p;
```

Presumably, you've defined the *neg64p* procedure earlier in the program.

You should make the choice of macro versus procedure call on the basis of efficiency. Macros are slightly faster than procedure calls because you don't execute the CALL and corresponding RET instructions. On the other hand, the use of macros can make your program larger because a macro invocation expands to the text in the body of the macro on each invocation. Procedure calls jump to a single instance of the procedure's body. Therefore, if the macro body is large and you invoke the macro several times throughout your program, it will make your final executable much larger. Also, if the body of your macro executes more than a few simple instructions, the overhead of a CALL/RET sequence has little impact on the overall execution time of the code, so the execution time savings are nearly negligible. On the other hand, if the body of a procedure is very short (like the *neg64* example above), you'll discover that the macro implementation is much faster and doesn't expand the size of your program by much. Therefore, a good rule of thumb is

- ❑ **Use macros for short, time-critical program units. Use procedures for longer blocks of code and when execution time is not critical.**

Macros have many other disadvantages over procedures. Macros cannot have local (automatic) variables, macro parameters work differently than procedure parameters, macros don't support (run-time) recur-

1. To differentiate macros and procedures, this text will use the term *invocation* when describing the use of a macro and *call* when describing the use of a procedure.

sion, and macros are a little more difficult to debug than procedures (just to name a few disadvantages). Therefore, you shouldn't really use macros as a substitute for procedures except in some rare situations.

7.2.2 Macro Parameters

Like procedures, macros let you define parameters that let you supply different data on each macro invocation. This lets you write generic macros whose behavior can vary depending on the parameters you supply. By processing these macro parameters at compile-time, you can write very sophisticated macros.

Macro parameter declaration syntax is very straight-forward. You simply supply a list of parameter names within parentheses in a macro declaration:

```
macro neg64( reg32HO, reg32LO );

    neg( reg32HO );
    neg( reg32LO );
    sbb( 0, reg32HO );

endmacro;
```

Note that you do not associate a data type with a macro parameter like you do procedural parameters. This is because HLA macros are always *text* objects. The next section will explain the exact mechanism HLA uses to substitute an actual parameter for a formal parameter.

When you invoke a macro, you simply supply the actual parameters the same way you would for a procedure call:

```
neg64( edx, eax );
```

Note that a macro invocation that requires parameters expects you to enclose the parameter list within parentheses.

7.2.2.1 Standard Macro Parameter Expansion

As the previous section explains, HLA automatically associates the type *text* with macro parameters. This means that during a macro expansion, HLA substitutes the text you supply as the actual parameter everywhere the formal parameter name appears. The semantics of "pass by textual substitution" are a little different than "pass by value" or "pass by reference" so it is worthwhile exploring those differences here.

Consider the following macro invocations, using the *neg64* macro from the previous section:

```
neg64( edx, eax );
neg64( ebx, ecx );
```

These two invocations expand into the following code:

```
// neg64(edx, eax );

neg( edx );
neg( eax );
sbb( 0, edx );

// neg64( ebx, ecx );

neg( ebx );
neg( ecx );
sbb( 0, ebx );
```

Note that macro invocations do not make a local copy of the parameters (as pass by value does) nor do they pass the address of the actual parameter to the macro. Instead, a macro invocation of the form "neg64(edx, eax);" is equivalent to the following:

```
?reg32HO: text := "edx";
?reg32LO: text := "eax";

neg( reg32HO );
neg( reg32LO );
sbb( 0, reg32HO );
```

Of course, the text objects immediately expand their string values in-line, producing the former expansion for "neg64(edx, eax);".

Note that macro parameters are not limited to memory, register, or constant operands as are instruction or procedure operands. Any text is fine as long as its expansion is legal wherever you use the formal parameter. Similarly, formal parameters may appear anywhere in the macro body, not just where memory, register, or constant operands are legal. Consider the following macro declaration and sample invocations:

```
macro chkError( instr, jump, target );

    instr;
    jump target;

endmacro;

chkError( cmp( eax, 0 ), jnl, RangeError );    // Example 1
...
chkError( test( 1, bl ), jnz, ParityError );    // Example 2

// Example 1 expands to

    cmp( eax, 0 );
    jnl RangeError;

// Example 2 expands to

    test( 1, bl );
    jnz ParityError;
```

In general, HLA assumes that all text between commas constitutes a single macro parameter. If HLA encounters any opening "bracketing" symbols (left parentheses, left braces, or left brackets) then it will include all text up to the appropriate closing symbol, ignoring any commas that may appear within the bracketing symbols. This is why the *chkError* invocations above treat "cmp(eax, 0)" and "test(1, bl)" as single parameters rather than as a pair of parameters. Of course, HLA does not consider commas (and bracketing symbols) within a string constant as the end of an actual parameter. So the following macro and invocation is perfectly legal:

```
macro print( strToPrint );

    stdout.out( strToPrint );

endmacro;

.
.
.
print( "Hello, world!" );
```

HLA treats the string "Hello, world!" as a single parameter since the comma appears inside a literal string constant, just as your intuition suggests.

If you are unfamiliar with textual macro parameter expansion in other languages, you should be aware that there are some problems you can run into when HLA expands your actual macro parameters. Consider the following macro declaration and invocation:

```
macro Echo2nTimes( n, theStr );

    ?echoCnt: uns32 := 0;
    #while( echoCnt < n*2 )

        #print( theStr )
        ?echoCnt := echoCnt + 1;

    #endwhile

endmacro;

.
.
.
Echo2nTimes( 3+1, "Hello" );
```

This example displays "Hello" five times during compilation rather than the eight times you might intuitively expect. This is because the #WHILE statement above expands to

```
#while( echoCnt < 3+1*2 )
```

The actual parameter for *n* is "3+1", since HLA expands this text directly in place of *n*, you get the text above. Of course, at compile time HLA computes "3+1*2" as the value five rather than as the value eight (which you would get had HLA passed this parameter by value rather than by textual substitution).

The common solution to this problem, when passing numeric parameters that may contain compile-time expressions, is to surround the formal parameter in the macro with parentheses. E.g., you would rewrite the macro above as follows:

```
macro Echo2nTimes( n, theStr );

    ?echoCnt: uns32 := 0;
    #while( echoCnt < (n)*2 )

        #print( theStr )
        ?echoCnt := echoCnt + 1;

    #endwhile

endmacro;
```

The previous invocation would expand to the following code:

```
?echoCnt: uns32 := 0;
#while( echoCnt < (3+1)*2 )

    #print( theStr )
    ?echoCnt := echoCnt + 1;

#endwhile
```

This version of the macro produces the intuitive result.

If the number of actual parameters does not match the number of formal parameters, HLA will generate a diagnostic message during compilation. Like procedures, the number of actual parameters must agree with the number of formal parameters. If you would like to have optional macro parameters, then keep reading...

7.2.2.2 Macros with a Variable Number of Parameters

You may have noticed by now that some HLA macros don't require a fixed number of parameters. For example, the *stdout.put* macro in the HLA Standard Library allows one or more actual parameters. HLA uses a special array syntax to tell the compiler that you wish to allow a variable number of parameters in a macro parameter list. If you follow the last macro parameter in the formal parameter list with "[]" then HLA will allow a variable number of actual parameters (zero or more) in place of that formal parameter. E.g.,

```
macro varParms( varying[] );
```

```
<< macro body >>
```

```
endmacro;
```

```
.
```

```
.
```

```
.
```

```
varying( 1 );
```

```
varying( 1, 2 );
```

```
varying( 1, 2, 3 );
```

```
varying();
```

Note, especially, the last example above. If a macro has any formal parameters, you must supply parentheses with the macro list after the macro invocation. This is true even if you supply zero actual parameters to a macro with a varying parameter list. Keep in mind this important difference between a macro with no parameters and a macro with a varying parameter list but no actual parameters.

When HLA encounters a formal macro parameter with the "[]" suffix (which must be the last parameter in the formal parameter list), HLA creates a constant string array and initializes that array with the text associated with the remaining actual parameters in the macro invocation. You can determine the number of actual parameters assigned to this array using the *@ELEMENTS* compile-time function. For example, "*@elements(varying)*" will return some value, zero or greater, that specifies the total number of parameters associated with that parameter. The following declaration for *varParms* demonstrates how you might use this:

```
macro varParms( varying[] );
```

```
    ?vpCnt := 0;
```

```
    #while( vpCnt < @elements( varying ) )
```

```
        #print( varying[ vpCnt ] )
```

```
        ?vpCnt := vpCnt + 1;
```

```
    #endwhile
```

```
endmacro;
```

```
.
```

```
.
```

```
.
```

```
varying( 1 );           // Prints "1" during compilation.
```

```
varying( 1, 2 );        // Prints "1" and "2" on separate lines.
```

```
varying( 1, 2, 3 );     // Prints "1", "2", and "3" on separate lines.
```

```
varying();              // Doesn't print anything.
```

Since HLA doesn't allow arrays of *text* objects, the varying parameter must be an array of strings. This, unfortunately, means you must treat the varying parameters differently than you handle standard macro parameters. If you want some element of the varying string array to expand as text within the macro body, you can always use the *@TEXT* function to achieve this. Conversely, if you want to use a non-varying formal parameter as a string object, you can always use the *@STRING:name* operator. The following example demonstrates this:

```

macro ReqAndOpt( Required, optional[] );

    ?@text( optional[0] ) := @string:ReqAndOpt;
    #print( @text( optional[0] ))

endmacro;

.
.
.
ReqAndOpt( i, j );

// The macro invocation above expands to

    ?@text( "j" ) := @string:i;
    #print( "j" )

// The above further expands to

    j := "i";
    #print( j )

// The above simply prints "i" during compilation.

```

Of course, it would be a good idea in a macro like the above to verify that there are at least two parameters before attempting to reference element zero of the *optional* parameter. You can easily do this as follows:

```

macro ReqAndOpt( Required, optional[] );

    #if( @elements( optional ) > 0 )

        ?@text( optional[0] ) := @string:ReqAndOpt;
        #print( @text( optional[0] ))

    #else

        #error( "ReqAndOpt must have at least two parameters" )

    #endif

endmacro;

```

7.2.2.3 Required Versus Optional Macro Parameters

As noted in the previous section, HLA requires exactly one actual parameter for each non-varying formal macro parameter. If there is no varying macro parameter (and there can be at most one) then the number of actual parameters must exactly match the number of formal parameters. If a varying formal parameter is present, then there must be at least as many actual macro parameters as there are non-varying (or required) formal macro parameters. If there is a single, varying, actual parameter, then a macro invocation may have zero or more actual parameters.

There is one big difference between a macro invocation of a macro with no parameters and a macro invocation of a macro with a single, varying, parameter that has no actual parameters: the macro with the varying parameter list must have an empty set of parentheses after it while the macro invocation of the macro without any parameters does not allow this. You can use this fact to your advantage if you wish to write a macro that doesn't have any parameters but you want to follow the macro invocation with "()" so that it matches the syntax of a procedure call with no parameters. Consider the following macro:

```

macro neg64( JustForTheParens[] );

    #if( @elements( JustForTheParens ) = 0 )

        neg( edx );
        neg( eax );
        sbb( 0, edx );

    #else

        #error( "Unexpected operand(s)" )

    #endif

endmacro;

```

The macro above allows invocations of the form "neg64();" using the same syntax you would use for a procedure call. This feature is useful if you want the syntax of your parameterless macro invocations to match the syntax of a parameterless procedure call. It's not a bad idea to do this, just in the off chance you need to convert the macro to a procedure at some point (or vice versa, for that matter).

If, for some reason, it is more convenient to operate on your macro parameters as *string* objects rather than *text* objects, you can specify a single varying parameter for the macro and then use #IF and @ELEMENTS to enforce the number of required actual parameters.

7.2.2.4 The "#(" and ")#" Macro Parameter Brackets

Once in a (really) great while, you may want to include arbitrary commas (i.e., outside a bracketing pair) within a macro parameter. Or, perhaps, you might want to include other text as part of a macro expansion that HLA would normally process before storing away the text as the value for the formal parameter². The "#(" and ")#" bracketing symbols tell HLA to collect all text, except for surrounding whitespace, between these two symbols and treat that text as a single parameter. Consider the following macro:

```

macro PrintName( theParm );

    ?theName := @string:theParm;
    #print( theName )

endmacro;

```

Normally, this macro will simply print the text of the actual parameter you pass to it. So were you to invoke the macro with "PrintName(j);" HLA would simply print "j" during compilation. This occurs because HLA associates the parameter data ("j") with the string value for the text object *theParm*. The macro converts this text data to a string, puts the string data in *theName*, and then prints this string.

Now consider the following statements:

```

?tt:text := "j";
PrintName( tt );

```

This macro invocation will also print "j". The reason is that HLA expands text constants immediately upon encountering them. So after this expansion, the invocation above is equivalent to

```

PrintName( j );

```

So this macro invocation prints "j" for the same reason the last example did.

2. For example, HLA will normally expand all *text* objects prior to the creation of the data for the formal parameter. You might not want this expansion to occur.

What if you want the macro to print "tt" rather than "j"? Unfortunately, HLA's *eager* evaluation of the text constant gets in the way here. However, if you bracket "tt" with the "#(" and ")#" brackets, you can instruct HLA to *defer* the expansion of this text constant until it actually expands the macro. I.e., the following macro invocation prints "tt" during compilation:

```
PrintName( #( tt )# );
```

Note that HLA allows any amount of arbitrary text within the "#(" and ")#" brackets. This can include commas and other arbitrary text. The following macro invocation prints "Hello, World!" during compilation:

```
PrintName( #( Hello, World! )# );
```

Normally, HLA would complain about the mismatched number of parameters since the comma would suggest that there are two parameters here. However, the deferred evaluation brackets tell HLA to consider all the text between the "#(" and ")#" symbols as a single parameter.

7.2.2.5 Eager vs. Deferred Macro Parameter Evaluation

HLA uses two schemes to process macro parameters. As you saw in the previous section, HLA uses *eager* evaluation when processing text constants appearing in a macro parameter list. You can force *deferred* evaluation of the text constant by surrounding the text with the "#(" and ")#" brackets. For other types of operands, HLA uses deferred macro parameter evaluation. This section discusses the difference between these two forms and describes how to force eager evaluation if necessary.

Eager evaluation occurs while HLA is collecting the text associated with each macro parameter. For example, if "T" is a text constant containing the string "U" and "M" is a macro, then when HLA encounters "M(T)" it will first expand "T" to "U". Then HLA processes the macro invocation "M(U)" as though you had supplied the text "U" as the parameter to begin with.

Deferred evaluation of macro parameters means that HLA does not process the parameter(s), but rather passes the text unchanged to the macro. Any expansion of the text associated with macro parameters occurs within the macro itself. For example, if M and N are both macros accepting a single parameter, then the invocation "M(N(0))" defers the evaluation of "N(0)" until HLA processes the macro body. It does not evaluate "N(0)" first and pass this expansion on to the macro. The following program demonstrates eager and deferred evaluation:

```
// EgrDfrd.HLA
//
// This program demonstrates the difference
// between deferred and eager macro parameter
// processing.

program EagerVsDeferredEvaluation;

macro ToDefer( tdParm );

    #print( "ToDefer: ", @string:tdParm )

endmacro;

macro testEVD( theParm );

    #print( "testEVD:", @string:theParm, " " )
    @string:tdParm

endmacro;
```

```

const

    txt:text := "Hello";
    Hello:string := "there";

begin EagerVsDeferredEvaluation;

    testEVD( Hello );           // Deferred evaluation.
    testEVD( txt );             // Eager evaluation.
    testEVD( ToDefer( World ) ); //Deferred evaluation.

end EagerVsDeferredEvaluation;

```

Program 7.1 Eager vs. Deferred Macro Parameter Evaluation

Note that the macro *testEVD* outputs the text associated with the formal parameter as a string during compilation. When you compile Program 7.1 it produces the following output:

```

testEVD: 'Hello'
testEVD: 'Hello'
testEVD: 'ToDefer( World )'

```

The first line prints 'Hello' because this is the text supplied as a parameter for the first call to *testEVD*. Since this is a string constant, not a text constant, HLA uses deferred evaluation. This means that it passes the text appearing between the parentheses unchanged to the *testEVD* macro. That text is "Hello" hence the same output as the parameter text.

The second *testEVD* invocation prints 'Hello'. This is because the macro parameter, *txt*, is a text object. HLA eagerly processes text constants before invoking the macro. Therefore, HLA translates "testEVD(txt)" to "testEVD(Hello)" prior to invoking the macro. Since the macro parameter text is now "Hello", that's what HLA prints during compilation while processing this macro.

The third invocation of *testEVD* above is semantically identical to the first. It is present just to demonstrate that HLA defers processing macros just like it defers the processing of everything else except text constants.

Although the code in Program 7.1 does not actually evaluate the *ToDefer* macro invocation, this is only because the body of *testEVD* does not directly use the parameter. Instead, it converts *theParm* to a string and prints its value. Had this code actually referred to *theParm* in an expression (or as a statement), then HLA would have invoked *ToDefer* and let it do its job. Consider the following modification to the above program:

```

// Deferred.HLA
//
// This program demonstrates deferred macro parameter
// processing.

program DeferredEvaluation;

macro ToDefer( tdParm );

    @string:tdParm

endmacro;

macro testEVD( theParm );

```

```

    #print( "Hello ", theParm )

endmacro;

begin DeferredEvaluation;

    testEVD( ToDefer( World ) );

end DeferredEvaluation;

```

Program 7.2 Deferred Macro Parameter Expansion

The macro invocation "testEVD(ToDefer(World));" defers the evaluation of its parameter. Therefore, the actual parameter *theParm* is a text object containing the string "ToDefer(World)". Inside the testEVD macro, HLA encounters theParm and expands it to this string, i.e.,

```
#print( "Hello ", theParm )
```

expands to

```
#print( "Hello ", ToDefer( World ) )
```

When HLA processes the #PRINT statement, it eagerly processes all parameters. Therefore, HLA expands the statement above to

```
#print( "Hello ", "World" )
```

since "ToDefer(World)" expands to *@string:tdParm* and that expands to "World".

Most of the time, the choice between deferred and eager evaluation produces the same result. In Program 7.2, for example, it doesn't matter whether the *ToDefer* macro expansion is eager (thus passing the string "World" as the parameter to *testEVD*) or deferred. Either mechanism produces the same output.

There are situations where deferred evaluation is not interchangeable with eager evaluation. The following program demonstrates a problem that can occur when you use deferred evaluation rather than eager evaluation. In this example the program attempts to pass the current line number in the source file as a parameter to a macro. This does not work because HLA expands (and evaluates) the @LINENUMBER function call inside the macro on every invocation. Therefore, this program always prints the same line number (eight) regardless of the invocation line number:

```

// DeferredFails.HLA
//
// This program a situation where deferred
// evaluation fails to work properly.

program DeferredFails;

macro printAt( where );

    #print( "at line ", where )

endmacro;

begin DeferredFails;

    printAt( @linenumber );
    printAt( @lineNumber );

```

```
end DeferredFails;
```

Program 7.3 An Example Where Deferred Evaluation Fails to Work Properly

Intuitively, this program should print:

```
at line 14
at line 15
```

Unfortunately, because of deferred evaluation, the two `printAt` invocations simply pass the text "`@linenumber`" as the actual parameter value rather than the string representing the line numbers of these two statements in the program. Since the formal parameter always expands to `@LINENUMBER` on the same line (line eight), this program always prints the same line number regardless of the line number of the macro invocation.

If you need an eager evaluation of a macro parameter there are three ways to achieve this. First of all, of course, you can specify a *text* object as a macro parameter and HLA will immediately expand that object prior to passing it as the macro parameter. The second option is to use the `@TEXT` function (with a string parameter). HLA will also immediately process this object, expanding it to the appropriate text, prior to processing that text as a macro parameter. The third option is to use the `@EVAL` pseudo-function. Within a macro invocation's parameter list, the `@EVAL` function instructs HLA to evaluate the `@EVAL` parameter prior to passing the text to the macro. Therefore, you can correct the problem in Program 7.3 by using the following code (which properly prints at "at line 14" and "at line 15"):

```
// EvalSucceeds.HLA
//
// This program a situation where deferred
// evaluation fails to work properly.

program EvalSucceeds;

macro printAt( where );

    #print( "at line ", where )

endmacro;

begin EvalSucceeds;

    printAt( @eval( @linenumber ) );
    printAt( @eval( @lineNumber ) );

end EvalSucceeds;
```

Program 7.4 Demonstration of @EVAL Compile-time Function

In addition to immediately processing built-in compiler functions like `@LINENUMBER`, the `@EVAL` pseudo-function will also invoke any macros appearing in the `@EVAL` parameter. `@EVAL` usually leaves other values unchanged.

7.2.3 Local Symbols in a Macro

Consider the following macro declaration:

```
macro JZC( target );

    jnz NotTarget;
    jc  target;
    NotTarget:

endmacro;
```

The purpose of this macro is to simulate an instruction that jumps to the specified target location if the zero flag is set *and* the carry flag is set. Conversely, if either the zero flag is clear or the carry flag is clear this macro transfers control to the instruction immediately following the macro invocation.

There is a serious problem with this macro. Consider what happens if you use this macro more than once in your program:

```
JZC( Dest1 );
.
.
.
JZC( Dest2 );
.
.
.
```

The macro invocations above expand to the following code:

```
    jnz NotTarget;
    jc  Dest1;
NotTarget:
.
.
.
    jnz NotTarget;
    jc  Dest2;
NotTarget:
.
.
.
```

The problem with the expansion of these two macro invocations is that they both emit the same label, *NotTarget*, during macro expansion. When HLA processes this code it will complain about a duplicate symbol definition. Therefore, you must take care when defining symbols inside a macro because multiple invocations of that macro may lead to multiple definitions of that symbol.

HLA's solution to this problem is to allow the use of *local symbols* within a macro. Local macro symbols are unique to a specific invocation of a macro. For example, had *NotTarget* been a local symbol in the *JZC* macro invocations above, the program would have compiled properly since HLA treats each occurrence of *NotTarget* as a unique symbol.

HLA does not automatically make internal macro symbol definitions local to that macro³. Instead, you must explicitly tell HLA which symbols must be local. You do this in a macro declaration using the following generic syntax:

```
macro macroname ( optional_parameters ) : optional_list_of_local_names ;
    << macro body >>
endmacro;
```

The list of local names is a sequence of one or more HLA identifiers separated by commas. Whenever HLA encounters this name in a particular macro invocation it automatically substitutes some unique name for that identifier. For each macro invocation, HLA substitutes a different name for the local symbol.

3. Sometimes you actually want the symbols to be global.

You can correct the problem with the *JZC* macro by using the following macro code:

```
macro JZC( target ):NotTarget;

    jnz NotTarget;
    jc  target;
    NotTarget:

endmacro;
```

Now whenever HLA processes this macro it will automatically associate a unique symbol with each occurrence of *NotTarget*. This will prevent the duplicate symbol error that occurs if you do not declare *NotTarget* as a local symbol.

HLA implements local symbols by substituting a symbol of the form "*_nnnn_*" (where *nnnn* is a four-digit hexadecimal number) wherever the local symbol appears in a macro invocation. For example, a macro invocation of the form "JZC(SomeLabel);" might expand to

```
    jnz _010A_;
    jc  SomeLabel;
_010A_:
```

For each local symbol appearing within a macro expansion HLA will generate a unique temporary identifier by simply incrementing this numeric value for each new local symbol it needs. As long as you do not explicitly create labels of the form "*_nnnn_*" (where *nnnn* is a hexadecimal value) there will never be a conflict in your program. HLA explicitly reserves all symbols that begin and end with a single underscore for its own private use (and for use by the HLA Standard Library). As long as you honor this restriction, there should be no conflicts between HLA local symbol generation and labels in your own programs since all HLA-generated symbols begin and end with a single underscore.

HLA implements local symbols by effectively converting that local symbol to a text constant that expands to the unique symbol HLA generates for the local label. That is, HLA effectively treats local symbol declarations as indicated by the following example:

```
macro JZC( target );
    ?NotTarget:text := "_010A_";

    jnz NotTarget;
    jc  target;
    NotTarget:

endmacro;
```

Whenever HLA expands this macro it will substitute "*_010A_*" for each occurrence of *NotTarget* it encounters in the expansion. This analogy isn't perfect because the text symbol *NotTarget* in this example is still accessible after the macro expansion whereas this is not the case when defining local symbols within a macro. But this does give you an idea of how HLA implements local symbols.

One important consequence of HLA's implementation of local symbols within a macro is that HLA will produce some puzzling error messages if an error occurs on a line that uses a local symbol. Consider the following (incorrect) macro declaration:

```
macro LoopZC( TopOfLoop ): ExitLocation;

    jnz ExitLocation;
    jc  TopOfLoop;

endmacro;
```

Note that in this example the macro does not define the *ExitLocation* symbol even though there is a jump (JNZ) to this label. If you attempt to compile this program, HLA will complain about an undefined statement label and it will state that the symbol is something like "*_010A_*" rather than *ExitLocation*.

Locating the exact source of this problem can be challenging since HLA cannot report this error until the end of the procedure or program in which *LoopZC* appears (long after you've invoked the macro). If you have lots of macros with lots of local symbols, locating the exact problem is going to be a lot of work; your only option is to carefully analyze the macros you do call (perhaps by commenting them out of your program one by one until the error goes away) to discover the source of the problem. Once you determine the offending macro, the next step is to determine which local symbol is the culprit (if the macro contains more than one local symbol). Because tracking down bugs associated with local symbols can be tough, you should be especially careful when using local symbols within a macro.

Because local symbols are effectively text constants, don't forget that HLA eagerly processes any local symbols you pass as parameters to other macros. To see this effect, consider the following sample program:

```
// LocalDemo.HLA
//
// This program demonstrates the effect
// of passing a local macro symbol as a
// parameter to another macro. Remember,
// local macro symbols are text constants
// so HLA eager evaluates them when they
// appear as macro parameters.

program LocalExpansionDemo;

macro printIt( what );

    #print( @string:what )
    #print( what )

endmacro;

macro LocalDemo:local;

    ?local:string := "localStr";

    printIt( local );           // Eager evaluation, passes "_nnnn".
    printIt( #( local )# )     // Force deferred evaluation, passes "local".

endmacro;

begin LocalExpansionDemo;

    LocalDemo;

end LocalExpansionDemo;
```

Program 7.5 Local Macro Symbols as Macro Parameters

Inside *LocalDemo* HLA associates the unique symbol "_0001_" (or something similar) with the local symbol *local*. Next, HLA defines "_0001_" to be a string constant and associates the text "localStr" with this constant.

The first *printIt* macro invocation expands to "printIt(_0001_)" because HLA eagerly processes text constants in macro parameter lists (remember, local symbols are, effectively, text constants). Therefore, *printIt*'s *what* parameter contains the text "_0001_" for this first invocation. Therefore, the first #PRINT statement prints this textual data ("_0001_") and the second print statement prints the value associated with "_0001_" which is "localStr".

The second *printIt* macro invocation inside the *LocalDemo* macro explicitly forces HLA to use deferred evaluation since it surrounds *local* with the `"#("` and `")#"` bracketing symbols. Therefore, HLA associates the text "local" with *printIt*'s formal parameter rather than the expansion `"_0001_"`. Inside *printIt*, the first `#PRINT` statement displays the text associated with the *what* parameter (which is "local" at this point). The second `#PRINT` statement expands *what* to produce "local". Since *local* is a currently defined text constant (defined within *LocalDemo* that invokes *printIt*), HLA expands this text constant to produce `"_0001_"`. Since `"_0001_"` is a string constant, HLA prints the specified string ("localStr") during compilation. The complete output during compilation is

```
_0001_
localStr
local
localStr
```

Discussing the expansion of local symbols may seem like a lot of unnecessary detail. However, as your macros become more complex you may run into difficulties with your code based on the way HLA expands local symbols. Hence it is necessary to have a good grasp on how HLA processes these symbols.

Quick tip: if you ever need to generate a unique label in your program, you can use HLA local symbol facility to achieve this. Normally, you can only reference HLA's local symbols within the macro that defines the symbol. However, you can convert that local symbol to a string and process that string in your program as the following simple program demonstrates:

```
// UniqueSymbols.HLA
//
// This program demonstrates how to generate
// unique symbols in a program.

program UniqueSymsDemo;

macro unique:theSym;

    @string:theSym

endmacro;

begin UniqueSymsDemo;

    ?label:text := unique;

    jmp label;

label:

    ?@toString:label :text := unique;
    jmp label;

label:

end UniqueSymsDemo;
```

Program 7.6 A Macro That Generates Unique Symbols for a Program

The first instance of *label:* in this program expands to `"_0001_:"` while the second instance of *label:* in this program expands to `"_0003_:"`. Of course, reusing symbols in this manner is horrible programming style (it's very confusing), but there are some cases you'll encounter when writing advanced macros where

you will want to generate a unique symbol for use in your program. The *unique* macro in this program demonstrates exactly how to do this.

7.2.4 Macros as Compile-Time Procedures

Although programmers typically use macros to expand to some sequence of machine instructions, there is absolutely no requirement that a macro body contain any executable instructions. Indeed, many macros contain only compile-time language statements (e.g., #IF, #WHILE, "?" assignments, etc.). By placing only compile-time language statements in the body of a macro, you can effectively write compile-time procedures and functions using macros.

The *unique* macro from the previous section is a good example of a compile-time function that returns a string result. Consider, again, the definition of this macro:

```
macro unique:theSym;

    @string:theSym

endmacro;
```

Whenever your code references this macro, HLA replaces the macro invocation with the text "@string:theSym" which, of course, expands to some string like "_021F_". Therefore, you can think of this macro as a compile-time function that returns a string result.

Be careful that you don't take the function analogy too far. Remember, macros always expand to their body text at the point of invocation. Some expansions may not be legal at any arbitrary point in your programs. Fortunately, most compile-time statements are legal anywhere whitespace is legal in your programs. Therefore, macros generally behave as you would expect functions or procedures to behave during the execution of your compile-time programs.

Of course, the only difference between a procedure and a function is that a function returns some explicit value while procedures simply do some activity. There is no special syntax for specifying a compile-time function return value. As the example above indicates, simply specifying the value you wish to return as a statement in the macro body suffices. A compile-time procedure, on the other hand, would not contain any non-compile-time language statements that expand into some sort of data during macro invocation.

7.2.5 Multi-part (Context-Free) Macros

HLA's macro facilities, as described up to this point, are not particularly amazing. Indeed, most assemblers provide macro facilities very similar to those this chapter presents up to this point. Earlier, this chapter made the claim that HLA's macro facilities are quite a bit more powerful than those found in other assembly languages (or any programming language for that matter). Part of this power comes from the synergy that exists between the HLA compile-time language and HLA's macros. However, the one feature that sets HLA's macro facilities apart from all others is HLA's ability to handle multi-part, or context-free⁴, macros. This section describes this powerful feature.

The best way to introduce HLA's context-free macro facilities is via an example. Suppose you wanted to create a macro to define a new high level language statement in HLA (a very common use for macros). Let's say you wanted to create a statement like the following:

```
nLoop( 10 )

    << body >>
```

4. The term "context-free" is an automata theory term used to describe constructs, like programming language control structures, that allow nesting.

```
endloop;
```

The basic idea is that this code would execute the body of the loop ten times (or however many times the *nLoop* parameter specifies). A typical low-level implementation of this control structure might take the following form:

```
    mov( 10, ecx );
UniqueLabel:

    << body >>

    dec( ecx );
    jne UniqueLabel;
```

Clearly it will require two macros (*nLoop* and *endloop*) to implement this control structure. The first attempt a beginner might try is doomed to failure:

```
macro nLoop( cnt );
    mov( cnt, ecx );
UniqueLabel:

endmacro;

macro endloop;
    dec( ecx );
    jne UniqueLabel;
endmacro;
```

You've already seen the problem with this pair of macros: they use a global target label. Any attempt to use the *nLoop* macro more than once will result in a duplicate symbol error. Previously, we utilized HLA's local symbol facilities to overcome this problem. However, that approach will not work here because local symbols are local to a specific macro invocation; unfortunately, the *endloop* macro needs to reference *UniqueLabel* inside the *nLoop* invocation, so *UniqueLabel* cannot be a local symbol in this example.

A quick and dirty solution might be to take advantage of the trick employed by the *unique* macro appearing in previous sections. By utilizing a global text constant, you can share the label information across two macros using an implementation like the following:

```
macro nLoop( cnt ):UniqueLabel;

    ?nLoop_target:string := @string:UniqueLabel;
    mov( cnt, ecx );
    UniqueLabel:

endmacro;

macro endloop;

    dec( ecx );
    jnz @text( nLoop_target );

endmacro;
```

Using this definition, you can have multiple calls to the *nLoop* and *endloop* macros and HLA will not generate a duplicate symbol error:

```
nLoop( 10 )

    stdout.put( "Loop counter = ", ecx, nl );

endloop;
```

```

nLoop( 5 )

    stdout.put( "Second Loop Counter = ", ecx, nl );

endloop;

```

The macro invocations above produce something like the following (reasonably correct) expansion:

```

mov( 10, ecx );
_023A_:          //UniqueLabel, first invocation

    stdout.put( "Loop counter = ", ecx, nl );

    dec( ecx );
    jne _023A_;    // Expansion of nLoop_target becomes _023A_.

    mov( 5, ecx );
_023B_:          // UniqueLabel, second invocation.

    stdout.put( "Second Loop Counter = ", ecx, nl );

    dec( ecx );
    jnz _023B_;    // Expansion of nLoop_target becomes _023B_.

```

This scheme looks like it's working properly. However, this implementation suffers from a big drawback- it fails if you attempt to nest the *nLoop..endloop* control structure:

```

nLoop( 10 )

    push( ecx ); // Must preserve outer loop counter.
    nLoop( 5 )

        stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    endloop;
    pop( ecx ); // Restore outer loop counter.

endloop;

```

You would expect to see this code print its message 50 times. However, the macro invocations above produce code like the following:

```

mov( 10, ecx );
_0321_:          //UniqueLabel, first invocation

    push( ecx );
    mov( 5, ecx );
_0322_:

    stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    dec( ecx );
    jne _0322_;    // Expansion of nLoop_target becomes _0322_.
    pop( ecx );
    dec( ecx );
    jne _0322_;    // Expansion of nLoop_target incorrectly becomes _0322_.

```

Note that the last JNE should jump to label "_0321_" rather than "_0322_". Unfortunately, the nested invocation of the *nLoop* macro overwrites the value of the global string constant *nLoop_target* thus the last JNE transfers control to the wrong label.

It is possible to correct this problem using an array of strings and another compile-time constant to create a *stack* of labels. By pushing and popping these labels as you encounter *nLoop* and *endloop* you can emit the correct code. However, this is a lot of work, is very inelegant, and you must repeat this process for every nestable control structure you dream up. In other words, it's a total kludge. Fortunately, HLA provides a better solution: multi-part macros.

Multi-part macros let you define a set of macros that work together. The *nLoop* and the *endloop* macros in this section are a good example of a pair of macros that work intimately together. By defining *nLoop* and *endloop* within a multi-part macro definition, the problems with communicating the target label between the two macros goes away because multi-part macros share parameters and local symbols. This provides a much more elegant solution to this problem than using global constants to hold target label information.

As its name suggests, a multi-part macro consists of a sequence of statements containing two matched macro names (e.g., *nLoop* and *endloop*). Multi-part macro invocations always consist of at least two macro invocations: a *beginning* invocation (e.g., *nLoop*) and a *terminating* invocation (e.g., *endloop*). Some number of unrelated (to the macro bodies) instructions may appear between the two invocations. To declare a multi-part macro, you use the following syntax:

```
macro beginningMacro (optional_parameters) : optional_local_symbols;

    << beginningMacro body >>

terminator terminatingMacro (optional_parameters) : optional_local_symbols;

    << terminatingMacro body >>

endmacro;
```

The presence of the TERMINATOR section in the macro declaration tells HLA that this is a multi-part macro declaration. It also ends the macro declaration of the beginning macro and begins the declaration of the terminating macro (i.e., the invocation of *beginningMacro* does not emit the code associated with the TERMINATOR macro). As you would expect, parameters and local symbols are optional in both declarations and the associated glue characters (parentheses and colons) are not present if the parameters and local symbol lists are not present.

Now let's look at the multi-part macro declaration for the *nLoop..endloop* macro pair:

```
macro nLoop( cnt ):TopOfLoop;

    mov( cnt, ecx );
    TopOfLoop:

terminator endloop;

    dec( ecx );
    jne TopOfLoop;

endmacro;
```

As you can see in this example, the definition of the *nLoop..endloop* control structure is much simpler when using multi-part macros; better still, multi-part macro declarations work even if you nest the invocations.

The most notable thing in this particular macro declaration is that the *endloop* macro has access to *nLoop*'s parameters and local symbols (in this example the *endloop* macro does not reference *cnt*, but it could if this was necessary). This makes communication between the two macros trivial.

Multi-part macro invocations must always occur in pairs. If the beginning macro appears in the text, the terminating macro must follow at some point. A terminating macro may never appear in the source file without a previous, matching, instance of the beginning macro. These semantics are identical to many of the

HLA high level control structures; i.e., you cannot have an `ENDIF` without having a corresponding `IF` clause earlier in the source file.

When you nest multi-part macro invocations, HLA "magically" keeps track of local symbols and always emits the appropriate local label value. The nested macros appearing earlier are no problem for multi-part macros:

```
nLoop( 10 )

    push( ecx ); // Must preserve outer loop counter.
    nLoop( 5 )

        stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    endloop;
    pop( ecx ); // Restore outer loop counter.

endloop;
```

The above code properly compiles to something like:

```
        mov( 10, ecx );
_01FE_:

        push( ecx );
        mov( 5, ecx );
_01FF_:

        stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

        dec( ecx );
        jne _01FF_;

        pop( ecx );

        dec( ecx );
        jne _01FE_;    // Note the correct label here.
```

In addition to terminating macros, HLA's multi-part macro facilities also provide an option for introducing additional macro declarations associated with the beginning/terminating macro pair: **KEYWORD** macros. **KEYWORD** macros are macros that are active only between a specific beginning and terminating macro pair. The classic use for **KEYWORD** macros is to allow the introduction of context-sensitive keywords into the macro (context-sensitive, in this case, meaning that the terms are only active within the context of the body of statements between the beginning and terminating macros). Classic examples of statements that could employ these types of macros include the **BREAK** and **CONTINUE** statements within a loop body and the **CASE** clause within a **SWITCH..ENDSWITCH** statement.

The syntax for a multi-part macro declaration that includes one or more **KEYWORD** macros is the following:

```
macro beginningMacro( optional_parameters ): optional_local_labels;

    << beginningMacro Body >>

keyword keywordMacro( optional_parameters ): optional_local_labels;

    << keywordMacro Body >>

terminator terminatingMacro( optional_parameters ): optional_local_labels;

    << terminatingMacro Body >>

endmacro;
```

If a KEYWORD macro is present in a macro declaration there must also be a terminating macro declaration. You cannot have a KEYWORD macro without a corresponding TERMINATOR macro. The TERMINATOR macro declaration is always last in a multi-part macro declaration.

The syntax example above specifies only a single KEYWORD macro. HLA, however, allows zero or more KEYWORD macro declarations in a multi-part macro. The HLA SWITCH statement, for example, defines two KEYWORD macros, *case* and *default*.

KEYWORD and TERMINATOR macros may refer to the parameters and local symbols defined in the beginning macro, but they may not refer to locals and parameters in other KEYWORD macros. Parameters and local symbols in KEYWORD macro declarations are local to that specific macro. If you really need to communicate information between KEYWORD and TERMINATOR macros, define some local symbols in the beginning macro and assign these local symbols the parameter (or local symbol) values in the affected KEYWORD macro. Then refer to this beginning macro local symbol in other parts of the macro. The following is a trivial example of this:

```
macro ShareParameter:parmValue;

    << beginning macro body >>

keyword ParmToShare( p );

    ?parmValue:text := @string:p;

    << keyword macro body >>

terminator UsesSharedParm;

    mov( parmValue, ecx );

    << terminator macro body >>

endmacro;
```

By assigning *ParmToShare*'s parameter value to the beginning macro's *parmValue* local symbol, this code makes the value of *p* accessible by the *UsesSharedParm* terminating macro.

This section only touches on the capabilities of HLA's multi-part macro facilities. Additional examples appear later in this chapter in the section on Domain Specific Embedded Languages (see "Domain Specific Embedded Languages" on page 973). This text will make use of HLA's multi-part macros in later chapters as well. For more information on multi-part macros, see these sections in this text or check out the HLA documentation.

7.2.6 Simulating Function Overloading with Macros

The C++ language supports a nifty feature known as *function overloading*. Function overloading lets you write several different functions or procedures that all have the same name. The difference between these functions is the types of their parameters or the number of parameters. A procedure declaration is said to be unique if it has a different number of parameters than other functions with the same name or if the types of its parameters differs from another function with the same name. HLA does not directly support procedure overloading but you can use macros to achieve the same result. This section explains how to use HLA's macros and the compile-time language to achieve function/procedure overloading.

One good use for procedure overloading is to reduce the number of standard library routines you must remember how to use. For example, the HLA Standard Library provides four different "puti" routines that output an integer value: *stdout.puti64*, *stdout.puti32*, *stdout.puti16*, and *stdout.puti8*. The different routines, as their name suggests, output integer values according to the size of their integer parameter. In the C++ language (or another other language supporting procedure/function overloading) the engineer designing the input routines would probably have chosen to name them all *stdout.puti* and leave it up to the compiler to

select the appropriate one based on the operand size⁵. The following macro demonstrates how to do this in HLA using the compile-time language to figure out the size of the parameter operand:

```
// Puti.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "puti" macro that calls stdout.puti8, stdout.puti16,
// stdout.puti32, or stdout.puti64 depending on the size of the operand.

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

// puti-
//
// Automatically decides whether we have a 64, 32, 16, or 8-bit
// operand and calls the appropriate stdout.putiX routine to
// output this value.

macro puti( operand );

    // If we have an eight-byte operand, call puti64:

    #if( @size( operand ) = 8 )

        stdout.puti64( operand );

    // If we have a four-byte operand, call puti32:

    #elseif( @size( operand ) = 4 )

        stdout.puti32( operand );

    // If we have a two-byte operand, call puti16:

    #elseif( @size( operand ) = 2 )

        stdout.puti16( operand );

    // If we have a one-byte operand, call puti8:

    #elseif( @size( operand ) = 1 )

        stdout.puti8( operand );

    // If it's not an eight, four, two, or one-byte operand,
    // then print an error message:

    #else
```

5. By the way, the HLA Standard Library does this as well. Although it doesn't provide *stdout.puti*, it does provide *stdout.put* that will choose an appropriate output routine based upon the parameter's type. This is a bit more flexible than a *puti* routine.

```

        #error( "Expected a 64, 32, 16, or 8-bit operand" )

    #endif

endmacro;

// Some sample variable declarations so we can test the macro above.

static
    i8:      int8      := -8;
    i16:     int16     := -16;
    i32:     int32     := -32;
    i64:     qword;

begin putiDemo;

    // Initialize i64 since we can't do this in the static section.

    mov( -64, (type dword i64 ) );
    mov( $FFFF_FFFF, (type dword i64[4]) );

    // Demo the puti macro:

    puti( i8 );  stdout.newln();
    puti( i16 ); stdout.newln();
    puti( i32 ); stdout.newln();
    puti( i64 ); stdout.newln();

end putiDemo;

```

Program 7.7 Simple Procedure Overloading Based on Operand Size

The example above simply tests the size of the operand to determine which output routine to use. You can use other HLA compile-time functions, like @TYPENAME, to do more sophisticated processing. Consider the following program that demonstrates a macro that overloads stdout.puti32, stdout.putu32, and stdout.putdw depending on the type of the operand:

```

// put32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "put32" macro that calls stdout.puti32, stdout.putu32,
// or stdout.putdw depending on the type of the operand.

program put32Demo;
#include( "stdlib.hhf" )

// put32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

```

```

macro put32( operand );

    // If we have an int32 operand, call puti32:
    #if( @typename( operand ) = "int32" )

        stdout.puti32( operand );

    // If we have an uns32 operand, call putu32:
    #elseif( @typename( operand ) = "uns32" )

        stdout.putu32( operand );

    // If we have a dword operand, call putidw:
    #elseif( @typename( operand ) = "dword" )

        stdout.putdw( operand );

    // If it's not a 32-bit integer value, report an error:
    #else

        #error( "Expected an int32, uns32, or dword operand" )

    #endif

endmacro;

// Some sample variable declarations so we can test the macro above.
static
    i32:    int32    := -32;
    u32:    uns32    := 32;
    d32:    dword    := $32;

begin put32Demo;

    // Demo the put32 macro:

    put32( d32 );  stdout.newln();
    put32( u32 );  stdout.newln();
    put32( i32 );  stdout.newln();

end put32Demo;

```

Program 7.8 Procedure Overloading Based on Operand Type

You can easily extend the macro above to output eight and sixteen-bit operands as well as 32-bit operands. That is left as an exercise.

The number of actual parameters is another way to resolve which overloaded procedure to call. If you specify a variable number of macro parameters (using the "[]" syntax, see “Macros with a Variable Number of Parameters” on page 944) you can use the @ELEMENTS compile-time function to determine exactly how many parameters are present and call the appropriate routine. The following sample program uses this trick to determine whether it should call *stdout.puti32* or *stdout.puti32size*:

```
// puti32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "puti32" macro that calls stdout.puti32 or stdout.puti32size
// depending on the number of parameters present.

program puti32Demo;
#include( "stdlib.hhf" )

// puti32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

macro puti32( operand[] );

    // If we have a single operand, call stdout.puti32:

    #if( @elements( operand ) = 1 )

        stdout.puti32( @text(operand[0]) );

    // If we have two operands, call stdout.puti32size and
    // supply a default value of ' ' for the padding character:

    #elseif( @elements( operand ) = 2 )

        stdout.puti32size( @text(operand[0]), @text(operand[1]), ' ' );

    // If we have three parameters, then pass all three of them
    // along to puti32size:

    #elseif( @elements( operand ) = 3 )

        stdout.puti32size
        (
            @text(operand[0]),
            @text(operand[1]),
            @text(operand[2])
        );

    // If we don't have one, two, or three operands, report an error:

    #else

        #error( "Expected one, two, or three operands" )

    #endif
endmacro

```

```

#endif

endmacro;

// A sample variable declaration so we can test the macro above.

static
    i32:    int32    := -32;

begin puti32Demo;

    // Demo the put32 macro:

    puti32( i32 );  stdout.newln();
    puti32( i32, 5 );  stdout.newln();
    puti32( i32, 5, '*' );  stdout.newln();

end puti32Demo;

```

Program 7.9 Using the Number of Parameters to Resolve Overloaded Procedures

All the examples up to this point provide procedure overloading for Standard Library routines (specifically, the integer output routines). Of course, you are not limited to overloading procedures in the HLA Standard Library. You can create your own overloaded procedures as well. All you've got to do is write a set of procedures, all with unique names, and then use a single macro to decide which routine to actually call based on the macro's parameters. Rather than call the individual routines, invoke the common macro and let it decide which procedure to actually call.

Although not strictly an example of procedure overloading, a related use for macros is to rearrange parameters in a procedure call. For example, some languages (most notably C/C++) expect you to pass their parameters on the stack in the reverse order of how you declare them in the parameter list. Since HLA expects the caller to push the parameters in the order they appear in the parameter list, HLA's order is the opposite of C/C++'s. Were you to write a procedure in HLA that you call from C/C++, you would have to reverse the order of the parameters. Similarly, if you call a C/C++ function from HLA, your HLA external declaration must specify the parameters in the reverse order that they appear in the C/C++ declaration. This is often very confusing. You can avoid this problem by simply specifying a macro that declares the (macro) parameters in the correct order and whose body calls the C/C++ function swapping the order of the parameters in the actual call. This allows you to use the same calling sequence (from the programmer's perspective) for both C/C++ and HLA calls to the same function.

7.3 Writing Compile-Time "Programs"

The HLA compile-time language provides a powerful facility with which to write "programs" that execute while HLA is compiling your assembly language programs. Although it is possible to write some general purpose programs using the HLA compile-time language, the real purpose of the HLA compile-time language is to allow you to write short programs *that write other programs*. In particular, the primary purpose of the HLA compile-time language is to automate the creation of large or complex assembly language sequences. The following subsections provide some simple examples of such compile-time programs.

7.3.1 Constructing Data Tables at Compile Time

Earlier, this text suggested that you could write programs to generate large, complex, lookup tables for your assembly language programs (see “Generating Tables” on page 629). That chapter provided examples in HLA but suggested that writing a separate program was unnecessary. This is true, you can generate most look-up tables you’ll need using nothing more than the HLA compile-time language facilities. Indeed, filling in table entries is one of the principle uses of the HLA compile-time language. In this section we will take a look at using the HLA compile-time language to construct data tables during compilation.

In the section on generating tables, this text gave an example of an HLA program that writes a text file containing a lookup table for the trigonometric *sine* function. The table contains 360 entries with the index into the table specifying an angle in degrees. Each *int32* entry in the table contained the value $\sin(\text{angle}) \times 1000$ where *angle* is equal to the index into the table. The section on generating tables suggested running this program and then including the text output from that program into the actual program that used the resulting table. You can avoid much of this work by using the compile-time language. The following HLA program includes a short compile-time code fragment that constructs this table of sines directly.

```
// demoSines.hla
//
// This program demonstrates how to create a lookup table
// of sine values using the HLA compile-time language.

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

const
    pi :real80 := 3.1415926535897;

readonly
    sines: int32[ 360 ] :=
    [
        // The following compile-time program generates
        // 359 entries (out of 360). For each entry
        // it computes the sine of the index into the
        // table and multiplies this result by 1000
        // in order to get a reasonable integer value.

        ?angle := 0;
        #while( angle < 359 )

            // Note: HLA's @sin function expects angles
            // in radians. radians = degrees*pi/180.
            // the "int32" function truncates its result,
            // so this function adds 1/2 as a weak attempt
            // to round the value up.

            int32( @sin( angle * pi / 180.0 ) * 1000 + 0.5 ),
            ?angle := angle + 1;

        #endwhile

        // Here's the 360th entry in the table. This code
        // handles the last entry specially because a comma
        // does not follow this entry in the table.

        int32( @sin( 359 * pi / 180.0 ) * 1000 + 0.5 )
    ];
```

```

begin demoSines;

    // Simple demo program that displays all the values in the table.

    for( mov( 0, ebx); ebx<360; inc( ebx )) do

        mov( sines[ ebx*4 ], eax );
        stdout.put
        (
            "sin( ",
            (type uns32 ebx ),
            " )*1000 = ",
            (type int32 eax ),
            nl
        );

    endfor;

end demoSines;

```

Program 7.10 Generating a SINE Lookup Table with the Compile-time Language

Another common use for the compile-time language is to build ASCII character lookup tables for use by the XLAT instruction at run-time. Common examples include lookup tables for alphabetic case manipulation. The following program demonstrates how to construct an upper case conversion table and a lower case conversion table⁶. Note the use of a macro as a compile-time procedure to reduce the complexity of the table generating code:

```

// demoCase.hla
//
// This program demonstrates how to create a lookup table
// of alphabetic case conversion values using the HLA
// compile-time language.

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

const
    pi :real80 := 3.1415926535897;

// emitCharRange-
//
// This macro emits a set of character entries
// for an array of characters. It emits a list
// of values (with a comma suffix on each value)
// from the starting value up to, but not including,
// the ending value.

macro emitCharRange( start, last ): index;

    ?index:uns8 := start;

```

6. Note that on modern processors, using a lookup table is probably not the most efficient way to convert between alphabetic cases. However, this is just an example of filling in the table using the compile-time language. The principles are correct even if the code is not exactly the best it could be.

```

        #while( index < last )

            char( index ),
            ?index := index + 1;

        #endwhile

endmacro;

readonly

// toUC:
// The entries in this table contain the value of the index
// into the table except for indicies #$61..#$7A (those entries
// whose indicies are the ASCII codes for the lower case
// characters). Those particular table entries contain the
// codes for the corresponding upper case alphabetic characters.
// If you use an ASCII character as an index into this table and
// fetch the specified byte at that location, you will effectively
// translate lower case characters to upper case characters and
// leave all other characters unaffected.
// The following compile-time program generates
//
// 255 entries (out of 256). For each entry
// it computes toupper( index ) where index is
// the character whose ASCII code is an index
// into the table.

toUC:  char[ 256 ] :=
    [
        // Emit all the characters through lower case 'a':

        emitCharRange( 0, uns8('a') )

        // Okay, we've generated all the entries up to
        // the start of the lower case characters. Output
        // Upper Case characters in place of the lower
        // case characters here.

        emitCharRange( uns8('A'), uns8('Z') + 1 )

        // Okay, emit the non-alphabetic characters
        // through to byte code #$FE:

        emitCharRange( uns8('z') + 1, $FF )

        // Here's the last entry in the table. This code
        // handles the last entry specially because a comma
        // does not follow this entry in the table.

        #$FF
    ];

// The following table is very similar to the one above.
// You would use this one, however, to translate upper case
// characters to lower case while leaving everything else alone.
// See the comments in the previous table for more details.

Tolc:  char[ 256 ] :=

```

```

    [
        emitCharRange( 0, uns8('A') )
        emitCharRange( uns8('a'), uns8('z') + 1 )
        emitCharRange( uns8('Z') + 1, $FF )

        # $FF
    ];

begin demoCase;

    // Display the printable ASCII characters and the corresponding
    // conversions:

    for( mov( uns32( ' ' ), eax ); eax <= $FF; inc( eax ) ) do

        mov( toUC[ eax ], bl );
        mov( TOlc[ eax ], bh );
        stdout.put
        (
            "toupper( '",
            (type char al),
            "' ) = '",
            (type char bl),
            "'      tolower( '",
            (type char al),
            "' ) = '",
            (type char bh),
            "'",
            nl
        );

    endfor;

end demoCase;

```

Program 7.11 Generating Case Conversion Tables with the Compile-Time Language

One important thing to note about this sample is the fact that a semicolon does not follow the *emitCharRange* macro invocations. Macro invocations do not require a closing semicolon. Often, it is legal to go ahead and add one to the end of the macro invocation because HLA is normally very forgiving about having extra semicolons inserted into the code. In this case, however, the extra semicolons are illegal because they would appear between adjacent entries in the *TOlc* and *toUC* tables. Keep in mind that macro invocations don't require a semicolon, especially when using macro invocations as compile-time procedures.

7.3.2 Unrolling Loops

In the chapter on Low-Level Control Structures (see “Unraveling Loops” on page 774) this text points out that you can unravel loops to improve the performance of certain assembly language programs. One problem with unravelling, or unrolling, loops is that you may need to do a lot of extra typing, especially if many iterations are necessary. Fortunately, HLA's compile-time language facilities, especially the *#WHILE* loop, comes to the rescue. With a small amount of extra typing plus one copy of the loop body, you can unroll a loop as many times as you please.

If you simply want to repeat the same exact code sequence some number of times, unrolling the code is especially trivial. All you've got to do is wrap an HLA *#WHILE..#ENDWHILE* loop around the sequence

and count down a VAL object the specified number of times. For example, if you wanted to print "Hello World" ten times, you could encode this as follows:

```
?count := 0;
#while( count < 10 )

    stdout.put( "Hello World", nl );
    ?count := count + 1;

#endwhile
```

Although the code above looks very similar to a WHILE (or FOR) loop you could write in your program, remember the fundamental difference: the code above simply consists of ten straight *stdout.put* calls in the program. Were you to encode this using a FOR loop, there would be only one call to *stdout.put* and lots of additional logic to loop back and execute that single call ten times.

Unrolling loops becomes slightly more complicated if any instructions in that loop refer to the value of a loop control variable or other value that changes with each iteration of the loop. A typical example is a loop that zeros the elements of an integer array:

```
mov( 0, eax );
for( mov( 0, ebx ); ebx < 20; inc( ebx ) ) do

    mov( eax, array[ ebx*4 ] );

endfor;
```

In this code fragment the loop uses the value of the loop control variable (in EBX) to index into *array*. Simply copying "mov(eax, array[ebx*4]);" twenty times is not the proper way to unroll this loop. You must substitute an appropriate constant index in the range 0..76 (the corresponding loop indices, times four) in place of "EBX*4" in this example. Correctly unrolling this loop should produce the following code sequence:

```
mov( eax, array[ 0*4 ] );
mov( eax, array[ 1*4 ] );
mov( eax, array[ 2*4 ] );
mov( eax, array[ 3*4 ] );
mov( eax, array[ 4*4 ] );
mov( eax, array[ 5*4 ] );
mov( eax, array[ 6*4 ] );
mov( eax, array[ 7*4 ] );
mov( eax, array[ 8*4 ] );
mov( eax, array[ 9*4 ] );
mov( eax, array[ 10*4 ] );
mov( eax, array[ 11*4 ] );
mov( eax, array[ 12*4 ] );
mov( eax, array[ 13*4 ] );
mov( eax, array[ 14*4 ] );
mov( eax, array[ 15*4 ] );
mov( eax, array[ 16*4 ] );
mov( eax, array[ 17*4 ] );
mov( eax, array[ 18*4 ] );
mov( eax, array[ 19*4 ] );
```

You can do this more efficiently using the following compile-time code sequence:

```
?iteration := 0;
#while( iteration < 20 )

    mov( eax, array[ iteration*4 ] );
    ?iteration := iteration+1;

#endwhile
```

If the statements in a loop make use of the loop control variable's value, it is only possible to unroll such loops if those values are known at compile time. You cannot unroll loops when user input (or other run-time information) controls the number of iterations.

7.4 Using Macros in Different Source Files

Unlike procedures, macros do not have a fixed piece of code at some address in memory. Therefore, you cannot create "external" macros and link them with other modules in your program. However, it is very easy to share macros with different source files – just put the macros you wish to reuse in a header file and include that file using the `#include` directive. You can make the macro will be available to any source file you choose using this simple trick.

7.5 Putting It All Together

This chapter has barely touched on the capabilities of the HLA macro processor and compile-time language. The HLA language has one of the most powerful macro processors around. None of the other 80x86 assemblers even come close to HLA's capabilities with regard to macros. Indeed, if you could say just one thing about HLA in relation to other assemblers, it would have to be that HLA's macro facilities are, by far, the best.

The combination of the HLA compile-time language and the macro processor give HLA users the ability to extend the HLA language in many ways. In the chapter on Domain Specific Languages, you'll get the opportunity to see how to create your own specialized languages using HLA's macro facilities.

Even if you don't do exotic things like creating your own languages, HLA's macro facilities and compile-time language are really great for automating code generation in your programs. The HLA Standard Library, for example, makes heavy use of HLA's macro facilities; "procedures" like *stdout.put* and *stdin.get* would be very difficult (if not impossible) to create without the power of HLA macro facilities and the compile-time language. For some good examples of the possible complexity one can achieve with HLA's macros, you should scan through the `#include` files in the HLA Standard Library and look at some of the macros appearing therein.

This chapter serves as a basic introduction to HLA's macro facilities. As you use macros in your own programs you will gain even more insight into their power. So by all means, use macros as much as you can – they can help reduce the effort needed to develop programs.

Domain Specific Embedded Languages Chapter Nine

9.1 Chapter Overview

Now we come to the fun part. For the past nine chapters this text has been molding and conforming you to deal with the HLA language and assembly language programming in general. In this chapter you get to turn the tables; you'll learn how to force HLA to conform to your desires. This chapter will teach you how to extend the HLA language using HLA's *compile-time language*. By the time you are through with this chapter, you should have a healthy appreciation for the power of the HLA compile-time language. You will be able to write short compile-time programs. You will also be able to add new statements, of your own choosing, to the HLA language.

9.2 Introduction to DSELs in HLA

One of the most interesting features of the HLA language is its ability to support *Domain Specific Embedded Languages* (or DSELs, for short, which you pronounce as D-cells). A domain specific language is a language designed with a specific purpose in mind. Applications written in an appropriate domains specific language (DSL) are often much shorter and much easier to write than that same application written in a general purpose language (like C/C++ or Pascal). Unfortunately, writing a compiler for a DSL is considerable work. Since most DSLs are so specific that few programs are ever written in them, it is generally cost-prohibitive to create a DSL for a given application. This economic fact has led to the popularity of domain specific *embedded* languages. The difference between a DSL and a DSEL is the fact that you don't write a new compiler for DSEL; instead, you provide some tools for use by an existing language translator to let the user extend the language as necessary for the specific application. This allows the language designer to use the features of the existing (i.e., *embedding*) language without having to write the translator for these features in the DSEL. The HLA language incorporates lots of features that let you extend the language to handle your own particular needs. This section discusses how to use these features to extend HLA as you choose.

As you probably suspect by now, the HLA compile-time language is the principle tool at your disposal for creating DSELs. HLA's multi-part macros let you easily create high level language-like control structures. If you need some new control structure that HLA does not directly support, it's generally an easy task to write a macro to implement that control structure. If you need something special, something that HLA's multi-part macros won't directly support, then you can write code in the HLA compile-time language to process portions of your source file as though they were simply string data. By using the compile-time string handling functions you can process the source code in just about any way you can imagine. While many such techniques are well beyond the scope of this text, it's reassuring to know that HLA can handle just about anything you want to do, even once you become an advanced assembly language programmer.

The following sections will demonstrate how to extend the HLA language using the compile-time language facilities. Don't get the idea that these simple examples push the limits of HLA's capabilities, they don't. You can accomplish quite a bit more with the HLA compile-time language; these examples must be fairly simple because of the assumed general knowledge level of the audience for this text.

9.2.1 Implementing the Standard HLA Control Structures

HLA supports a wide set of high level language-like control structures. These statements are not true assembly language statements, they are high level language statements that HLA compiles into the corresponding low-level machine instructions. They are general control statements, not "domain specific" (which is why HLA includes them) but they are quite typical of the types of statements one can add to HLA in order to extend the language. In this section we will look at how you could implement many of HLA's high-level control structures using the compile-time language. Although there is no real need to implement these state-

ments in this manner, their example should provide a template for implementing other types of control structures in HLA.

The following sections show how to implement the FOREVER..ENDFOR, WHILE..ENDWHILE, and IF..ELSEIF..ELSE..ENDIF statements. The REPEAT..UNTIL and BEGIN..EXIT..EXITIF..END statements appear as exercises at the end of the chapter. The remaining high level language control structures (e.g., TRY..ENDTRY) are a little too complex to present at this point.

Because words like "if" and "while" are reserved by HLA, the following examples will use macro identifiers like "_if" and "_while". This will let us create recognizable statements using standard HLA identifiers (i.e., no conflicts with reserved words).

9.2.1.1 The FOREVER Loop

The FOREVER loop is probably the easiest control structure to implement. After all, the basic FOREVER loop simply consists of a label and a JMP instruction. So the first pass at implementing _FOREVER.._ENDFOR might look like the following:

```
macro _forever: topOfLoop;
    topOfLoop:

terminator _endfor;
    jmp topOfLoop;

endmacro;
```

Unfortunately, there is a big problem with this simple implementation: you'll probably want the ability to exit the loop via break and breakif statements and you might want the equivalent of a continue and continueif statement as well. If you attempt to use the standard BREAK, BREAKIF, CONTINUE, and CONTINUEIF statements inside this *_forever* loop implementation, you'll quickly discover that they do not work. These statements are valid only inside an HLA loop and the *_forever* macro above is not an HLA loop. Of course, we could easily solve this problem by defining _FOREVER thusly:

```
macro _forever;
    forever

terminator _endfor;
    endfor;

endmacro;
```

Now you can use BREAK, BREAKIF, CONTINUE, and CONTINUEIF inside the *_forever.._endfor* statement. However, this solution is ridiculous. The purpose of this section is to show you how you could create this statement were it not present in the HLA language. Simply renaming FOREVER to *_forever* is not an interesting solution.

Probably the best way to implement these additional statements is via KEYWORD macros within the *_forever* macro. Not only is this easy to do, but it has the added benefit of not allowing the use of these statements outside a *_forever* loop.

Implementing a *_continue* statement is very easy. Continue must transfer control to the first statement at the top of the loop. Therefore, the *_continue* KEYWORD macro will simply expand to a single JMP instruction that transfers control to the *topOfLoop* label. The complete implementation is the following:

```
keyword _continue;
    jmp topOfLoop;
```

Implementing *_continueif* is a little bit more difficult because this statement must evaluate a boolean expression and decide whether it must jump to the *topOfLoop* label. Fortunately, the HLA JT (jump if true) pseudo-instruction makes this a relatively trivial task. The JT pseudo-instruction is passed a boolean expres-

sion (the same as allowed by CONTINUEIF) and transfers control to the corresponding target label if the result of the expression evaluation is true. The `_continueif` implementation is nearly trivial with JT:

```
keyword _continueif( ciExpr );
    JT( ciExpr ) topOfLoop;
```

You will implement the `_break` and `_breakif` KEYWORD macros in a similar fashion. The only difference is that you must add a new label just beyond the JMP in the `_endfor` macro and the break statements should jump to this local label. The following program provides a complete implementation of the `_forever.._endfor` loop as well as a sample test program for the `_forever` loop.

```

/*****
/*
/* foreverMac.hla
/*
/* This program demonstrates how to use HLA's
/* "context-free" macros, along with the JT
/* "medium-level" instruction to create
/* the FOREVER..ENDFOR, BREAK, BREAKIF,
/* CONTINUE, and CONTINUEIF control statements.
/*
/*
*****/

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

// Emulate the FOREVER..ENDFOR loop here, plus the
// corresponding CONTINUE, CONTINUEIF, BREAK, and
// BREAKIF statements.

macro _forever:foreverLbl, foreverbrk;

    // Target label for the top of the
    // loop. This is also the destination
    // for the _continue and _continueif
    // macros.

    foreverLbl:

// The _continue and _continueif statements
// transfer control to the label above whenever
// they appear in a _forever.._endfor statement.
// (Of course, _continueif only transfers control
// if the corresponding boolean expression evaluates
// true.)

keyword _continue;
    jmp foreverLbl;

keyword _continueif( cifExpr );
    jt( cifExpr ) foreverLbl;

// the _break and _breakif macros transfer
// control to the "foreverbrk" label which
// is at the bottom of the loop.
```

```

keyword _break;
    jmp foreverbrk;

keyword _breakif( bifExpr );
    jt( bifExpr ) foreverbrk;

// At the bottom of the _forever.._endfor
// loop this code must jump back to the
// label at the top of the loop. The
// _endfor terminating macro must also supply
// the target label for the _break and _breakif
// keyword macros:

terminator _endfor;
    jmp foreverLbl;
    foreverbrk:

endmacro;

begin foreverDemo;

    // A simple main program that demonstrates the use of the
    // statements above.

    mov( 0, ebx );
    _forever

        stdout.put( "Top of loop, ebx = ", (type uns32 ebx), nl );
        inc( ebx );

        // On first iteration, skip all further statements.

        _continueif( ebx = 1 );

        // On fourth iteration, stop.

        _breakif( ebx = 4 );

        _continue; // Always jumps to top of loop.
        _break;    // Never executes, just demonstrates use.

    _endfor;

end foreverDemo;

```

Program 9.1 Macro Implementation of the FOREVER..ENDFOR Loop

9.2.1.2 The WHILE Loop

Once the FOREVER..ENDFOR loop is behind us, implementing other control structures like the WHILE..ENDWHILE loop is fairly easy. Indeed, the only notable thing about implementing the

`_while..endwhile` macros is that the code should implement this control structure as a REPEAT..UNTIL statement for efficiency reasons. The implementation appearing in this section takes a rather lazy approach to implementing the DO reserved word. The following code uses a KEYWORD macro to implement a "`_do`" clause, but it does not enforce the (proper) use of this keyword. Instead, the code simply ignores the `_do` clause wherever it appears between the `_while` and `_endwhile`. Perhaps it would have been better to check for the presence of this statement (not too difficult to do) and verify that it immediately follows the `_while` clause and associated expression (somewhat difficult to do), but this just seems like a lot of work to check for the presence of an irrelevant keyword. So this implementation simply ignores the `_do`. The complete implementation appears in Program 9.2:

```

/*****
/*
/* whileMacs.hla
/*
/* This program demonstrates how to use HLA's
/* "context-free" macros, along with the JT and
/* JF "medium-level" instructions to create
/* the basic WHILE statement.
/*
/*
*****/

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

// Emulate the while..endwhile loop here.
//
// Note that this code implements the WHILE
// loop as a REPEAT..UNTIL loop for efficiency
// (though it inserts an extra jump so the
// semantics remain the same as the WHILE loop).

macro _while( whlexpr ): repeatwhl, whltest, brkwhl;

    // Transfer control to the bottom of the loop
    // where the termination test takes place.

    jmp whltest;

    // Emit a label so we can jump back to the
    // top of the loop.

    repeatwhl:

    // Ignore the "_do" clause. Note that this
    // macro should really check to make sure
    // that "_do" follows the "_while" clause.
    // But it's not semantically important so
    // this code takes the lazy way out.

    keyword _do;

    // If we encounter "_break" inside this
    // loop, transfer control to the first statement
    // beyond the loop.

    keyword _break;

```

```

        jmp brkwhl;

// Ditto for "_breakif" except, of course, we
// only exit the loop if the corresponding
// boolean expression evaluates true.

keyword _breakif( biwExpr );
    jt( biwExpr ) brkwhl;

// The "_continue" and "_continueif" statements
// should transfer control directly to the point
// where this loop tests for termination.

keyword _continue;
    jmp whltest;

keyword _continueif( ciwExpr );
    jt( ciwExpr ) whltest;

// The "_endwhile" clause does most of the work.
// First, it must emit the target label used by the
// "_while", "_continue", and "_continueif" clauses
// above. Then it must emit the code that tests the
// loop termination condition and transfers control
// to the top of the loop (the "repeatwhl" label)
// if the expression evaluates false. Finally,
// this code must emit the "brkwhl" label the "_break"
// and "_breakif" statements reference.

terminator _endwhile;

    whltest:
    jt( whlexpr ) repeatwhl;
    brkwhl:

endmacro;

begin whileDemo;

// Quick demo of the _while statement.
// Note that the _breakif in the nested
// _while statement only skips the
// inner-most _while, just as you should expect.

mov( 0, eax );
_while( eax < 10 ) _do

    stdout.put( "eax in loop = ", eax, " ebx=" );
    inc( eax );
    mov( 0, ebx );
    _while( ebx < 4 ) _do

        stdout.puti32( ebx );
        _breakif( ebx = 3 );
        stdout.put( " ", " );
        inc( ebx );

```

```

        _endwhile;
        stdout.newln();

        _continueif( eax = 5 );
        _breakif( eax = 8 );
        _continue;
        _break;

    _endwhile

end whileDemo;

```

Program 9.2 Macro Implementation of the WHILE..ENDWHILE Loop

9.2.1.3 The IF Statement

Simulating the HLA IF..THEN..ELSEIF..ELSE..ENDIF statement using macros is a little bit more involved than the simulation of FOREVER or WHILE. The semantics of the ELSEIF and ELSE clauses complicate the code generation and require careful thought. While it is easy to write KEYWORD macros for *_elseif* and *_else*, ensuring that these statements generate correct (and efficient) code is another matter altogether.

The basic *_if.._endif* statement, without the *_elseif* and *_else* clauses, is very easy to implement (even easier than the *_while.._endwhile* loop of the previous section). The complete implementation is

```

macro _if( ifExpr ): onFalse;

    jf( ifExpr ) onFalse;

keyword _then; // Just ignore _then.

terminator _endif;

    onFalse:

endmacro;

```

This macro generates code that tests the boolean expression you supply as a macro parameter. If the expression evaluates false, the code this macro emits immediately jumps to the point just beyond the *_endif* terminating macro. So this is a simple and elegant implementation of the IF..ENDIF statement, assuming you don't need an ELSE or ELSEIF clause.

Adding an ELSE clause to this statement introduces some difficulties. First of all, we need some way to emit the target label of the JF pseudo-instruction in the *_else* section if it is present and we need to emit this label in the terminator section if the *_else* section is not present.

A related problem is that the code after the *_if* clause must end with a JMP instruction that skips the *_else* section if it is present. This JMP must transfer control to the same location as the current *onFalse* label.

Another problem that occurs when we use KEYWORD macros to implement the *_else* clause, is that we need some mechanism in place to ensure that at most one invocation of the *_else* macro appears in a given *_if.._endif* sequence.

We can easily solve these problems by introducing a compile-time variable (i.e., VAL object) into the macro. We will use this variable to indicate whether we've seen an *_else* section. This variable will tell us if we have more than one *_else* clause (which is an error) and it will tell us if we need to emit the *onFalse* label in the *_endif* macro. A reasonable implementation might be the following:

```

macro _if( ifExpr ): onFalse, ifDone, hasElse;

    ?hasElse := False; // Haven't seen an _else clause yet.

    jf( ifExpr ) onFalse;

keyword _then; // Just ignore _then.

keyword _else;

    // Check to see if this _if statement already has an _else clause:

    #if( hasElse )

        #error( "Only one _else clause is legal in an _if statement" )

    #endif

    ?hasElse := true; //Let the world know we've see an _else clause.

    // Since we've just encountered the _else clause, we've just finished
    // processing the statements in the _if section. The first thing we
    // need to do is emit a JMP instruction that will skip around the
    // _else statements (so the _if section doesn't fall in to the
    // _else code).

    jmp ifDone;

    // Okay, emit the onFalse label here so a false expression will transfer
    // control to the _else statements:

    onFalse:

terminator _endif;

    // If there was no _else section, we must emit the onFalse label
    // so that the former JF instruction has a proper destination.
    // If an _else section was present, we cannot emit this label
    // (since the _else code has already done so) but we must emit
    // the ifDone label.

    #if( hasElse )

        ifdone:

    #else

        onFalse:

    #endif

endmacro;

```

Adding the *_elseif* clause to the *_if.._endif* statement complicates things considerably. The problem is that *_elseif* can appear zero or more times in an *_if* statement and each occurrence needs to generate a unique *onFalse* label. Worse, if at least one *_elseif* clause appears in the sequence, then the JF instruction in the *_if* clause must transfer control to the first *_elseif*, not to the *_else* clause. Also, the last *_elseif* clause must transfer control to the *_else* clause (or to the first statement beyond the *_endif* clause) if its expression evaluates false. A straight-forward implementation just isn't going to work here.

A clever solution is to create a string variable that contains the name of the previous JF target label. Whenever you encounter an *_elseif* or an *_else* clause you simply emit this string to the source file as the target label. Then the only trick is "how do we generate a unique label whenever we need one?". Well, let's suppose that we have a string that is unique on each invocation of the *_if* macro. This being the case, we can generate a (source file wide) unique string by concatenating a counter value to the end of this base string. Each time we need a unique string, we simply bump the value of the counter up by one and create a new string. Consider the following macro:

```
macro genLabel ( base, number );

    @text( base + string( number ) );

endmacro;
```

If the *base* parameter is a string value holding a valid HLA identifier and the *number* parameter is an integer numeric operand, then this macro will emit a valid HLA identifier that consists of the *base* string followed by a string representing the numeric constant. For example, 'genLabel("Hello", 52)' emits the label *Hello52*. Since we can easily create an *uns32* VAL object inside our *_if* macro and increment this each time we need a unique label, the only problem is to generate a unique base string on each invocation of the *_if* macro. Fortunately, HLA already does this for us.

Remember, HLA converts all local macro symbols to a unique identifier of the form "_xxxx_" where xxxx represents some four-digit hexadecimal value. Since local symbols are really nothing more than text constants initialized with these unique identifier strings, it's very easy to obtain a unique string in a macro invocation- just declare a local symbol (or use an existing local symbol) and apply the @STRING: operator to it to extract the unique name as a string. The following example demonstrates how to do this:

```
macro uniqueIDs: counter, base;

    ?counter := 0;           // Increment this for each unique symbol you need.
    ?base := @string:base;   // base holds the base name to use.
    .
    .
    .

    // Generate a unique label at this point:

    genLabel( base, counter ): // Notice the colon. We're defining a
    ?counter := counter + 1;   // label at this point!
    .
    .
    .
    genLabel( base, counter ):
    ?counter := counter + 1;
    .
    .
    .
    etc.

endmacro;
```

Once we have the capability to generate a sequence of unique labels throughout a macro, implementing the *_elseif* clause simply becomes the task of emitting the last referenced label at the beginning of each *_elseif* (or *_else*) clause and jumping if false to the next unique label in the series. Program 9.3 implements the *_if..then.._elseif.._else.._endif* statement using exactly this technique.

```

/*****
/*                                     */
/* IFmacs.hla                         */
/*                                     */

```

```

/*                                     */
/* This program demonstrates how to use HLA's      */
/* "context-free" macros, along with the JT and    */
/* JF "medium-level" instructions to create       */
/* an IF statement.                               */
/*                                     */
/* *****/
/*****/

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

    // genlabel-
    //
    // This macro creates an HLA-compatible
    // identifier of the form "_xxxx_n" where
    // "_xxxx_" is the string associated with
    // the "base" parameter and "n" represents
    // some numeric value that the caller. The
    // combination of the base and the n values
    // will produce a unique label in the
    // program if base's string is unique for
    // each invocation of the "_if" macro.

macro genLabel( base, number );

    @text( base + string( number ) )

endmacro;

// Emulate the if..elseif..else..endif statement here.

macro _if( ifexpr ):elseLbl, ifDone, hasElse, base;

    // This macro must create a unique ID string
    // in base. One sneaky way to do this is
    // to use the converted name HLA generates
    // for the "base" object (this is generally
    // a string of the form "_xxxx_" where "xxxx"
    // is a four-digit hexadecimal value).

    ?base := @string:base;

    // This macro may need to generate a large set
    // of different labels (one for each _elseif
    // clause). This macro uses the elseLbl
    // value, along with the value of "base" above,
    // to generate these unique labels.

    ?elseLbl := 0;

    // hasElse determines if we have an _else clause
    // present in this statement. This macro uses
    // this value to determine if it must emit a
    // final else label when it encounters _endif.

    ?hasElse := false;

    // For an IF statement, we must evaluate the
    // boolean expression and jump to the current
    // else label if the expression evaluates false.

```

```

    jf( ifexpr ) genLabel( base, elseLbl );

// Just ignore the _then keyword.
// A slightly better implementation would require
// this keyword, the current implementation lets
// you write an "_if" clause without the "_then"
// clause. For that matter, the current implementation
// lets you arbitrarily sprinkle "_then" clauses
// throughout the "_if" statement; we will ignore
// this for this example.

keyword _then;

// Handle the "_elseif" clause here.

keyword _elseif(elsex);

    // _elseif clauses are illegal after
    // an _else clause in the statement.
    // Enforce that here.

    #if( hasElse )

        #error( "Unexpected '_elseif' clause" )

    #endif

    // We've just finished the "_if" clause
    // or a previous "_elseif" clause. So
    // the first thing we have to do is jump
    // to the code just beyond this "_if"
    // statement.

    jmp ifDone;

    // Okay, this is where the previous "_if" or
    // "_elseif" statement must jump if its boolean
    // expression evaluates false. Emit the target
    // label. Next, because we're about to jump
    // to our own target label, bump up the elseLbl
    // value by one to prevent jumping back to the
    // label we're about to emit. Finally, emit
    // the code that tests the boolean expression and
    // transfers control to the next _elseif or _else
    // clause if the result is false.

    genLabel( base, elseLbl ):
        ?elseLbl := elseLbl+1;
        jf(elsex) genLabel( base, elseLbl );

keyword _else;

    // Only allow a single "_else" clause in this
    // "_if" statement:

    #if( hasElse )

```

```

        #error( "Unexpected '_else' clause" )

    #endif

    // As above, we've just finished the previous "_if"
    // or "_elseif" clause, so jump directly to the end
    // of the "_if" statement.

    jmp ifDone;

    // Okay, emit the current 'else' label so that
    // the failure of the previous "_if" or "_elseif"
    // test will transfer control here. Also set
    // 'hasElse' to true to catch additional "_elseif"
    // and "_else" clauses.

    genLabel( base, elseLbl ):
        ?hasElse := true;

terminator _endif;

    // At the end of the _if statement we must emit the
    // destination label that the _if and _elseif sections
    // jump to. Also, if there was no _else section, this
    // code has to emit the last deployed else label.

    ifDone:
    #if( !hasElse )

        genLabel( base, elseLbl ):

    #endif

endmacro;

begin IFDemo;

    // Quick demo of the use of the above statements.

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

        _if( eax = 0 ) _then

            stdout.put( "in _if statement" nl );

        _elseif( eax = 1 ) _then

            stdout.put( "in first _elseif clause" nl );

        _elseif( eax = 2 ) _then

            stdout.put( "in second _elseif clause" nl );

        _else

            stdout.put( "in _else clause" nl );
            _if( eax > 3 ) _then

                stdout.put( "in second _if statement" nl );

```

```

        _endif;

    _endif;

endfor;

end IFDemo;

```

Program 9.3 Macro Implementation of the IF..ENDIF Statement

9.2.2 The HLA SWITCH/CASE Statement

HLA doesn't support a selection statement (SWITCH or CASE statement). Instead, HLA's SWITCH..CASE..DEFAULT..ENDSWITCH statement exists only as a macro in the HLA Standard Library HLL.HHF file. This section discusses HLA's macro implementation of the SWITCH statement.

The SWITCH statement is very complex so it should come as no surprise that the macro implementation is long, involved, and complex. The example appearing in this section is slightly simplified over the standard HLA version, but not by much. This discussion assumes that you're familiar with the low-level implementation of the SWITCH..CASE..DEFAULT..ENDSWITCH statement. If you are not comfortable with that implementation, or feel a little rusty, you may want to take another look at "SWITCH/CASE Statements" on page 750 before attempting to read this section. The discussion in this section is somewhat advanced and assumes a fair amount of programming skill. If you have trouble following this discussion, you may want to skip this section until you gain some more experience.

There are several different ways to implement a SWITCH statement. In this section we will assume that the `_switch.._endswitch` macro we are writing will implement the SWITCH statement using a jump table. Implementation as a sequence of *if..elseif* statements is fairly trivial and is left as an exercise. Other schemes are possible as well, this section will not consider them.

A typical SWITCH statement implementation might look like the following:

```

readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
    .
    .
    .

// switch( i )

    mov( i, eax );           // Check to see if "i" is outside the range
    cmp( eax, 5 );           // 5..7 and transfer control directly to the
    jb EndCase               // DEFAULT case if it is.
    cmp( eax, 7 );
    ja EndCase;
    jmp( JmpTbl[ eax*4 - 5*size(dword)] );

// case( 5 )
    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

// Case( 6 )
    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

```

```
// Case( 7 )
    Stmt7:
        stdout.put( "I=7" );

EndCase:
```

If you study this code carefully, with an eye to writing a macro to implement this statement, you'll discover a couple of major problems. First of all, it is exceedingly difficult to determine how many cases and the range of values those cases cover before actually processing each CASE in the SWITCH statement. Therefore, it is really difficult to emit the range check (for values outside the range 5..7) and the indirect jump before processing all the cases in the SWITCH statement. You can easily solve this problem, however, by moving the checks and the indirect jump to the bottom of the code and inserting a couple of extra JMP instructions. This produces the following implementation:

```
readonly
    JumpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
    .
    .
    .

// switch( i )

    jmp DoSwitch;                // First jump inserted into this code.

// case( 5 )
    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

// Case( 6 )
    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

// Case( 7 )
    Stmt7:
        stdout.put( "I=7" );
        jmp EndCase;           // Second jump inserted into this code.

DoSwitch:                        // Insert this label and move the range
    mov( i, eax );               // checks and indirect jump down here.
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );
    ja EndCase;
    jmp( JumpTbl[ eax*4 - 5*@size(dword)] );

// All the cases (including the default case) jump down here:

EndCase:
```

Since the range check code appears after all the cases, the macro can now process those cases and easily determine the bounds on the cases by the time it must emit the CMP instructions above that check the bounds of the SWITCH value. However, this implementation still has a problem. The entries in the *JumpTbl* table refer to labels that can only be determined by first processing all the cases in the SWITCH statement. Therefore, a macro cannot emit this table in a READONLY section that appears earlier in the source file than the SWITCH statement. Fortunately, HLA lets you embed data in the middle of the code section using the

READONLY..ENDREADONLY and STATIC..ENDSTATIC directives¹. Taking advantage of this feature allows use to rewrite the SWITCH implementation as follows:

```
// switch( i )

    jmp DoSwitch;                // First jump inserted into this code.

// case( 5 )
    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

// Case( 6 )
    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

// Case( 7 )
    Stmt7:
        stdout.put( "I=7" );
        jmp EndCase;          // Second jump inserted into this code.

DoSwitch:                        // Insert this label and move the range
    mov( i, eax );              // checks and indirect jump down here.
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );
    ja EndCase;
    jmp( JumpTbl[ eax*4 - 5*@size(dword)] );

// All the cases (including the default case) jump down here:

EndCase:

readonly
    JumpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
endreadonly;
```

HLA's macros can produce code like this when processing a SWITCH macro. So this is the type of code we will generate with a `_switch.._case.._default.._endswitch` macro.

Since we're going to need to know the minimum and maximum case values (in order to generate the appropriate operands for the CMP instructions above), the `_case` KEYWORD macro needs to compare the current case value(s) against the global minimum and maximum case values for all cases. If the current case value is less than the global minimum or greater than the global maximum, then the `_case` macro must update these global values accordingly. The `_endswitch` macro will use these global minimum and maximum values in the two CMP instructions it generates for the range checking sequence.

For each case value appearing in a `_switch` statement, the `_case` macros must save the case value and an identifying label for that case value. This is necessary so that the `_endswitch` macro can generate the jump table. What is really needed is an arbitrary list of records, each record containing a value field and a label field. Unfortunately, the HLA compile-time language does not support arbitrary lists of objects, so we will have to implement the list using a (fixed size) array of record constants. The record declaration will take the following form:

```
caseRecord:
    record
        value:uns32;
        label:uns32;
    endrecord;
```

1. HLA actually moves the data to the appropriate segment in memory, the data is not stored directly in the CODE section.

The *value* field will hold the current case value. The *label* field will hold a unique integer value for the corresponding `_case` that the macros can use to generate statement labels. The implementation of the `_switch` macro in this section will use a variant of the trick found in the section on the `_if` macro; it will convert a local macro symbol to a string and append an integer value to the end of that string to create a unique label. The integer value appended will be the value of the *label* field in the *caseRecord* list.

Processing the `_case` macro becomes fairly easy at this point. All the `_case` macro has to do is create an entry in the *caseRecord* list, bump a few counters, and emit an appropriate case label prior to the code emission. The implementation in this section uses Pascal semantics, so all but the first case in the `_switch.._endswitch` statement must first emit a jump to the statement following the `_endswitch` so the previous case's code doesn't fall into the current case.

The real work in implementing the `_switch.._endswitch` statement lies in the generation of the jump table. First of all, there is no requirement that the cases appear in ascending order in the `_switch.._endswitch` statement. However, the entries in the jump table must appear in ascending order. Second, there is no requirement that the cases in the `_switch.._endswitch` statement be consecutive. Yet the entries in the jump table must be consecutive case values². The code that emits the jump table must handle these inconsistencies.

The first task is to sort the entries in the *caseRecord* list in ascending order. This is easily accomplished by writing a little *SortCases* macro to sort all the *caseRecord* entries once the `_switch.._endswitch` macro has processed all the cases. *SortCases* doesn't have to be fancy. In fact, a bubblesort algorithm is perfect for this because:

- Bubble sort is easy to implement
- Bubble sort is efficient when sorting small lists and most SWITCH statements only have a few cases.
- Bubble sort is especially efficient on nearly sorted data and most programmers put their cases in ascending order.

After sorting the cases, only one problem remains: there may be gaps in the case values. This problem is easily handled by stepping through the *caseRecord* elements one by one and synthesizing consecutive entries whenever a gap appears in the list. Program 9.4 provides the full `_switch.._case.._default.._endswitch` macro implementation.

```

/*****
/*
/* switch.hla-
/*
/* This program demonstrates how to implement the
/* _switch.._case.._default.._endswitch statement
/* using macros.
/*
/*
*****/

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

const

    // Because this code uses an array to implement
    // the caseRecord list, we have to specify a fixed
    // number of cases. The following constant defines
    // the maximum number of possible cases in a
    // _switch statement.

```

2. Of course, if there are gaps in the case values, the jump table entries for the missing items should contain the address of the default case.

```

maxCases := 256;

type

    // The following data type hold the case value
    // and statement label information for each
    // case appearing in a _switch statement.

    caseRecord:
        record

            value:uns32;
            label:uns32;

        endrecord;

// SortCases
//
// This routine does a bubble sort on an array
// of caseRecord objects. It sorts in ascending
// order using the "value" field as the key.
//
// This is a good old fashioned bubble sort which
// turns out to be very efficient because:
//
// (1) The list of cases is usually quite small, and
// (2) The data is usually already sorted (or mostly sorted).

macro SortCases( sort_array, sort_size ):
    sort_i,
    sort_bnd,
    sort_didswap,
    sort_temp;

    ?sort_bnd := sort_size - 1;
    ?sort_didswap := true;
    #while( sort_didswap )

        ?sort_didswap := false;
        ?sort_i := 0;
        #while( sort_i < sort_bnd )

            #if
            (
                sort_array[sort_i].value >
                sort_array[sort_i+1].value
            )

                ?sort_temp := sort_array[sort_i];
                ?sort_array[sort_i] := sort_array[sort_i+1];
                ?sort_array[sort_i+1] := sort_temp;
                ?sort_didswap := true;

            #elseif
            (
                sort_array[sort_i].value =
                sort_array[sort_i+1].value
            )

```

```

        #error
        (
            "Two cases have the same value: (" +
            string( sort_array[sort_i].value ) +
            ") "
        )

    #endif
    ?sort_i := sort_i + 1;

#endwhile
?sort_bnd := sort_bnd - 1;

#endwhile;

endmacro;

// HLA Macro to implement a C SWITCH statement (using
// Pascal semantics). Note that the switch parameter
// must be a 32-bit register.

macro _switch( switch_reg ):
    switch_minval,
    switch_maxval,
    switch_otherwise,
    switch_endcase,
    switch_jmptbl,
    switch_cases,
    switch_caseIndex,
    switch_doCase,
    switch_hasotherwise;      // Just used to generate unique names.

// Verify that we have a register operand.

#if( !@isReg32( switch_reg ) )

    #error( "Switch operand must be a 32-bit register" )

#endif

// Create the switch_cases array. Allow, at most, 256 cases.

?switch_cases:caseRecord[ maxCases ];

// General initialization for processing cases.

?switch_caseIndex := 0;      // Index into switch_cases array.
?switch_minval := $FFFF_FFFF; // Minimum case value.
?switch_maxval := 0;         // Maximum case value.
?switch_hasotherwise := false; // Determines if DEFAULT section present.

```

```

// We need to process the cases to collect information like
// switch_minval prior to emitting the indirect jump. So move the
// indirect jump to the bottom of the case statement.

jmp switch_doCase;

// "case" keyword macro handles each of the cases in the
// case statement. Note that this syntax allows you to
// specify several cases in the same _case macro, e.g.,
// _case( 2, 3, 4 ). Such a situation tells this macro
// that these three values all execute the same code.

keyword _case( switch_parms[] ):
    switch_parmIndex,
    switch_parmCount,
    switch_constant;

?switch_parmCount:uns32;
?switch_parmCount := @elements( switch_parms );

#if( switch_parmCount <= 0 )

    #error( "Must have at least one case value" );
    ?switch_parms:uns32[1] := [0];

#endif

// If we have at least one case already, terminate
// the previous case by transferring control to the
// first statement after the endcase macro. Note
// that these semantics match Pascal's CASE statement,
// not C/C++'s SWITCH statement which would simply
// fall through to the next CASE.

#if( switch_caseIndex <> 0 )

    jmp switch_endcase;

#endif

// The following loop processes each case value
// supplied to the _case macro.

?switch_parmIndex:uns32;
?switch_parmIndex := 0;
#while( switch_parmIndex < switch_parmCount )

    ?switch_constant: uns32;
    ?switch_constant: uns32 :=
        uns32( @text( switch_parms[ switch_parmIndex ] ) );

    // Update minimum and maximum values based on the
    // current case value.

    #if( switch_constant < switch_minval )

        ?switch_minval := switch_constant;

```

```

#endif
#if( switch_constant > switch_maxval )

    ?switch_maxval := switch_constant;

#endif

// Emit a unique label to the source code for this case:

@text
(
    "_case"
    + @string:switch_caseIndex
    + string( switch_caseIndex )
):

// Save away the case label and the case value so we
// can build the jump table later on.

?switch_cases[ switch_caseIndex ].value := switch_constant;
?switch_cases[ switch_caseIndex ].label := switch_caseIndex;

// Bump switch_caseIndex value because we've just processed
// another case.

?switch_caseIndex := switch_caseIndex + 1;
#if( switch_caseIndex >= maxCases )

    #error( "Too many cases in statement" );

#endif

?switch_parmIndex := switch_parmIndex + 1;

#endwhile

// Handle the default keyword/macro here.

keyword _default;

// If there was not a preceding case, this is an error.
// If so, emit a jmp instruction to skip over the
// default case.

#if( switch_caseIndex < 1 )

    #error( "Must have at least one case" );

#endif

    jmp switch_endcase;

// Emit the label for this default case and set the
// switch_hasotherwise flag to true.

switch_otherwise:
?switch_hasotherwise := true;

```

```

// The endswitch terminator/macro checks to see if
// this is a reasonable switch statement and emits
// the jump table code if it is.

terminator_endswitch:
    switch_i_,
    switch_j_,
    switch_curCase_;

// If the difference between the smallest and
// largest case values is great, the jump table
// is going to be fairly large. If the difference
// between these two values is greater than 256 but
// less than 1024, warn the user that the table will
// be large. If it's greater than 1024, generate
// an error.
//
// Note: these are arbitrary limits. Feel free to
// adjust them if you like.

#if( (switch_maxval - switch_minval) > 256 )

    #if( (switch_maxval - switch_minval) > 1024 )

        // Perhaps in the future, this macro could
        // switch to generating an if..elseif..elseif...
        // chain if the range between the values is
        // too great.

        #error( "Range of cases is too great" );

    #else

        #print( "Warning: Range of cases is large" );

    #endif

#endif

// Table emission algorithm requires that the switch_cases
// array be sorted by the case values.

SortCases( switch_cases, switch_caseIndex );

// Build a string of the form:
//
//      switch_jmptbl:dword[ xx ] := [&case1, &case2, &case3...&casen];
//
// so we can output the jump table.

readonly

    switch_jmptbl:dword[ switch_maxval - switch_minval + 1 ] := [

        ?switch_i_ := 0;
        #while( switch_i_ < switch_caseIndex )

            ?switch_curCase_ := switch_cases[ switch_i_ ].value;

```

```

        // Emit the label associated with the current case:

        @text
        (
            "&"
            +   "_case"
            +   @string:switch_caseIndex
            +   string( switch_cases[ switch_i_ ].label )
            +   ","
        )

        // Emit "&switch_otherwise" table entries for any gaps present
        // in the table:

        ?switch_j_ := switch_cases[ switch_i_ + 1 ].value;
        ?switch_curCase_ := switch_curCase_ + 1;

        #while( switch_curCase_ < switch_j_ )

            &switch_otherwise,
            ?switch_curCase_ := switch_curCase_ + 1;

        #endwhile
        ?switch_i_ := switch_i_ + 1;

    #endwhile

    // Emit a dummy entry to terminate the table:

    &switch_otherwise];

endreadonly;

#if( switch_caseIndex < 1 )

    #error( "Must have at least one case" );

#endif

    // After the default case, or after the last
    // case entry, jump over the code that does
    // the conditional jump.

    jmp switch_endcase;

// Okay, here's the code that does the conditional jump.

switch_doCase:

    // If the minimum case value is zero, we don't
    // need to emit a CMP instruction for it.

    #if( switch_minval <> 0 )

        cmp( switch_reg, switch_minval );
        jb switch_otherwise;

    #endif

    cmp( switch_reg, switch_maxval );
    ja switch_otherwise;

```

```

        jmp( switch_jmptbl[ switch_reg*4 - switch_minval*4 ] );

// If there was no default case, transfer control
// to the first statement after the "endcase" clause.

#if( !switch_hasotherwise )

    switch_otherwise:

#endif

// When each of the cases complete execution,
// transfer control down here.

switch_endcase:

// The following statement deallocates the storage
// associated with the switch_cases array (this saves
// memory at compile time, it does not affect the
// execution of the resulting machine code).

?switch_cases := 0;

endmacro;

begin demoSwitch;

// A simple demonstration of the _switch.._endswitch statement:

for( mov( 0, eax ); eax < 8; inc( eax ) ) do

    _switch( eax )

        _case( 0 )

            stdout.put( "eax = 0" nl );

        _case( 1, 2 )

            stdout.put( "eax = 1 or 2" nl );

        _case( 3, 4, 5 )

            stdout.put( "eax = 3, 4, or 5" nl );

        _case( 6 )

            stdout.put( "eax = 6" nl );

        _default

            stdout.put( "eax is not in the range 0-6" nl );

    _endswitch;

```

```

        endfor;

    end demoSwitch;

```

Program 9.4 Macro Implementation of the SWITCH..ENDSWITCH Statement

9.2.3 A Modified WHILE Loop

The previous sections have shown you how to implement statements that are already available in HLA or the HLA Standard Library. While this approach lets you work with familiar statements that you should be comfortable with, it doesn't really demonstrate that you can create *new* control statements with HLA's compile-time language. In this section you will see how to create a variant of the WHILE statement that is not simply a rehash of HLA's WHILE statement. This should amply demonstrate that there are some useful control structures that HLA (and high level languages) don't provide and that you can easily use HLA compile-time language to implement specialized control structures as needed.

A common use of a WHILE loop is to search through a list and stop upon encountering some desired value or upon hitting the end of the list. A typical HLA example might take the following form:

```

while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>

endwhile;

```

The problem with this approach is that when the statement immediately following the ENDWHILE executes, that code doesn't know whether the loop terminated because it found the desired value or because it exhausted the list. The typical solution is to test to see if the loop exhausted the list and deal with that accordingly:

```

while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>

endwhile;
if( <<The list wasn't exhausted>> ) then

    << do something with the item we found >>

endif;

```

The problem with this "solution" should be obvious if you think about it a moment. We've already tested to see if the loop is empty, immediately after leaving the loop we repeat this same test. This is somewhat inefficient. A better solution would be to have something like an "else" clause in the WHILE loop that executes if you break out of the loop and doesn't execute if the loop terminates because the boolean expression evaluated false. Rather than use the keyword ELSE, let's invent a new (more readable) term: *onbreak*. The ONBREAK section of a WHILE loop executes (only once) if a BREAK or BREAKIF statement was the reason for the loop termination. With this ONBREAK clause, you could recode the previous WHILE loop a little bit more elegantly as follows:

```

while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>

```

```

onbreak

    << do something with the item we found >>

endwhile;

```

Note that if the ONBREAK clause is present, the WHILE's loop body ends at the ONBREAK keyword. The ONBREAK clause executes at most once per execution of this WHILE statement.

Implementing a `_while.._onbreak.._endwhile` statement is very easy using HLA's multi-part macros. Program 9.5 provides the complete implementation of this statement:

```

/*****
/*
/* while.hla
/*
/* This program demonstrates a variant of the
/* WHILE loop that provides a special "onbreak"
/* clause. The _onbreak clause executes if the
/* program executes a _break clause or it executes
/* a _breakif clause and the corresponding
/* boolean expression evaluates true. The _onbreak
/* section does not execute if the loop terminates
/* due to the _while boolean expression evaluating
/* false.
/*
/*
*****/

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

// _while semantics:
//
// _while( expr )
//
//    << stmts including optional _break, _breakif
//        _continue, and _continueif statements >>
//
//    _onbreak // This section is optional.
//
//    << stmts that only execute if program executes
//        a _break or _breakif (with true expression)
//        statement. >>
//
// _endwhile;

macro _while( expr ):falseLbl, breakLbl, topOfLoop, hasOnBreak;

    // hasOnBreak keeps track of whether we've seen an _onbreak
    // section.
    ?hasOnBreak:boolean:=false;

    // Here's the top of the WHILE loop.
    // Implement this as a straight-forward WHILE (test for
    // loop termination at the top of the loop).

    topOfLoop:
        jf( expr ) falseLbl;

    // Ignore the _do keyword.

```

```

keyword _do;

// _continue and _continueif (with a true expression)
// transfer control to the top of the loop where the
// _while code retests the loop termination condition.

keyword _continue;
    jmp topOfLoop;

keyword _continueif( expr1 );
    jt( expr1 ) topOfLoop;

// Unlike the _break or _breakif in a standard WHILE
// statement, we don't immediately exit the WHILE.
// Instead, this code transfers control to the optional
// _onbreak section if it is present. If it is not
// present, control transfers to the first statement
// beyond the _endwhile.

keyword _break;
    jmp breakLbl;

keyword _breakif( expr2 );
    jt( expr2 ) breakLbl;

// If we encounter an _onbreak section, this marks
// the end of the while loop body. Emit a jump that
// transfers control back to the top of the loop.
// This code also has to verify that there is only
// one _onbreak section present. Any code following
// this clause is going to execute only if the _break
// or _breakif statements execute and transfer control
// down here.

keyword _onbreak;
    #if( hasOnBreak )

        #error( "Extra _onbreak clause encountered" )

    #else

        jmp topOfLoop;
        ?hasOnBreak := true;

    breakLbl:

    #endif

terminator _endwhile;

// If we didn't have an _onbreak section, then
// this is the bottom of the _while loop body.
// Emit the jump to the top of the loop and emit
// the "breakLbl" label so the execution of a
// _break or _breakif transfers control down here.

#if( !hasOnBreak )

```

```

        jmp topOfLoop;
        breakLbl:

    #endif
    falseLbl:

endmacro;

static
    i:int32;

begin Demo_while;

    // Demonstration of standard while loop

    mov( 0, i );
    _while( i < 10 ) _do

        stdout.put( "1: i=", i, nl );
        inc( i );

    _endwhile;

    // Demonstration with BREAKIF:

    mov( 5, i );
    _while( i < 10 ) _do

        stdout.put( "2: i=", i, nl );
        _breakif( i = 7 );
        inc( i );

    _endwhile

    // Demonstration with _BREAKIF and _ONBREAK:

    mov( 0, i );
    _while( i < 10 ) _do

        stdout.put( "3: i=", i, nl );
        _breakif( i = 4 );
        inc( i );

        _onbreak

        stdout.put( "Breakif was true at i=", i, nl );

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

end Demo_while;

```

Program 9.5 The Implementation of _while.._onbreak.._endwhile

9.2.4 A Modified IF..ELSE..ENDIF Statement

The IF statement is another statement that doesn't always do exactly what you want. Like the `_while..._onbreak..._endwhile` example above, it's quite possible to redefine the IF statement so that it behaves the way we want it to. In this section you'll see how to implement a variant of the IF..ELSE..ENDIF statement that nests differently than the standard IF statement.

HLA's particular variant of the IF statement has several limitations. One of the major limitations is the inability to combine logical sub-expressions using logical conjunction (and) and logical disjunction (or). It is possible to simulate conjunction and disjunction if you carefully structure your code. Consider the following example:

// "C" code employing logical-AND operator:

```
if( expr1 && expr2 )
{
    << statements >>
}
```

// Equivalent HLA version:

```
if( expr1 ) then
    if( expr2 ) then
        << statements >>
    endif;
endif;
```

In both cases ("C" and HLA) the `<< statements >>` block executes only if both *expr1* and *expr2* evaluate true. So other than the extra typing involved, it is often very easy to simulate logical conjunction by using two IF statements in HLA.

There is one very big problem with this scheme. Consider what happens if you modify the "C" code to be the following:

// "C" code employing logical-AND operator:

```
if( expr1 && expr2 )
{
    << 'true' statements >>
}
else
{
    << 'false' statements >>
}
```

The only way to convert this to HLA (using the standard HLA high level control constructs) is by duplicating the 'false' statements. This introduces a bit of inefficiency into your code. As a result, many HLA programmers will switch to low-level control constructs or HLA's hybrid control structures (see "Hybrid Control Structures in HLA" on page 776) in order to avoid duplicating code. Unfortunately, dropping down into low-level code may make your program harder to read. It would be nice if you could efficiently handle this situation without making your code unreadable. Fortunately, you can do exactly this by creating a new version of the IF statement using HLA's multi-part macro facilities.

Before describing how to create this new type of IF statement, we must digress for a moment and explore an interesting feature of HLA's multi-part macro expansion: KEYWORD macros do not have to use unique names. Whenever you declare an HLA KEYWORD macro, HLA accepts whatever name you

choose. If that name happens to be already defined, then the KEYWORD macro name takes precedence as long as the macro is active (that is, from the point you invoke the macro name until HLA encounters the TERMINATOR macro). Therefore, the KEYWORD macro name hides the previous definition of that name until the termination of the macro. This feature applies even to the original macro name; that is, it is possible to define a KEYWORD macro with the same name as the original macro to which the KEYWORD macro belongs. This is a very useful feature because it allows you to change the definition of the macro within the scope of the opening and terminating invocations of the macro.

Although not pertinent to the IF statement we are construction, you should note that parameter and local symbols in a macro also override any previously defined symbols of the same name. So if you use that symbol between the opening macro and the terminating macro, you will get the value of the local symbol, not the global symbol. E.g.,

```
var
    i:int32;
    j:int32;
    .
    .
    .
macro abc:i;
    ?i:text := "j";
    .
    .
    .
terminator xyz;
    .
    .
    .
endmacro
    .
    .
    .
    mov( 25, i );
    mov( 10, j );
    abc
        mov( i, eax );    // Loads j's value (10), not 25 into eax.
    xyz;
```

The code above loads 10 into EAX because the "mov(i, eax);" instruction appears between the opening and terminating macros *abc..xyz*. Between those two macros the local definition of *i* takes precedence over the global definition. Since *i* is a text constant that expands to *j*, the aforementioned MOV statement is really equivalent to "mov(j, eax);" That statement, of course, loads 10 into EAX. Since this problem is difficult to see while reading your code, you should choose local symbols in multi-part macros very carefully. A good convention to adopt is to combine your local symbol name with the macro name, e.g.,

```
macro abc : i_abc;
```

You may wonder why HLA allows something so crazy to happen in your source code, in a moment you'll see why this behavior is useful (and now, with this brief message out of the way, back to our regularly scheduled discussion).

Before we digressed to discuss this interesting feature in HLA multi-part macros, we were trying to figure out how to efficiently simulate the conjunction and disjunction operators in an IF statement without resorting to low-level code. The problem in the example appearing earlier in this section is that you would have to duplicate some code in order to convert the IF..ELSE statement properly. The following code shows this problem:

```
// "C" code employing logical-AND operator:
```

```
if( expr1 && expr2 )
{
    << 'true' statements >>
```

```

}
else
{
    << 'false' statements >>
}

```

// Corresponding HLA code using the "nested-IF" algorithm:

```

if( expr1 ) then

    if( expr2 ) then

        << 'true' statements >>

    else

        << 'false' statements >>

    endif;

else

    << 'false' statements >>

endif;

```

Note that this code must duplicate the "<< 'false' statements >>" section if the logic is to exactly match the original "C" code. This means that the program will be larger and harder to read than is absolutely necessary.

One solution to this problem is to create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF.ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement the code above as follows:

```

if( expr1 ) then

    if( expr2 ) then

        << 'true' statements >>

    else

        << 'false' statements >>

    endif;

endif;

```

If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding 'false' statements) appear in this code; hence the code does not necessarily expand in size. If *expr2* evaluates true, then control falls through to the 'true' statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement's THEN block ends.

Of course, we can't actually define a new macro named "if" because you cannot redefine HLA reserved words. Nor would it be a good idea to do so even if these were legal (since it would make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high level statements.

```

/*****
/*
/* if.hla
/*
/*
/* This program demonstrates a modification of
/* the IF..ELSE..ENDIF statement using HLA's
/* multi-part macros.
/*
/*
*****/

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

// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression. Syntax:
//
// _if( expression ) _then
//
//     <<statements including nested _if clauses>>
//
// _else // this is optional
//
//     <<statements, but _if clauses are not allowed here>>
//
// _endif
//
// Note that nested _if clauses do not have a corresponding
// _endif clause. This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses
// including the first one. Of course, once the code
// encounters an _endif another _if statement may begin.

// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;

// Just ignore the _then keyword.

```

```

keyword _then;

// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro). Identical to the
// above _if implementation except this one does not
// require a matching _endif clause. The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

    // If this appears within the _else section, report
    // an error (we don't allow _if clauses nested in
    // the else section, that would create a loop).

    #if( hasElse )

        #error( "All _if clauses must appear before the _else clause" )

    #endif

// Handle the _else clause here. All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

    #else

        // Set hasElse true so we know that we've seen an _else
        // clause in this statement.

        ?hasElse := true;
        jmp ifDone;
        elseLbl:

    #endif

// _endif has two tasks. First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section. Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false conditional(s)
// in the _if clause(s) have a legal target label.

terminator _endif;

    ifDone:
    #if( !hasElse )

        elseLbl:

    #endif

endmacro;

```

```

static
    tr:boolean := true;
    f:boolean := false;

begin newIF;

    // Real quick demo of the _if statement:

    _if( tr ) _then

        _if( tr ) _then
            _if( f ) _then

                stdout.put( "error" nl );

            _else

                stdout.put( "Success" );

            _endif

        _endif

end newIF;

```

Program 9.6 Using Macros to Create a New IF Statement

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the `_if` macro as a keyword macro upon invocation of the main `_if` macro. The reason this code does this is so that any nested `_if` clauses do not require a corresponding `_endif` and don't support an `_else` clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader³.

9.3 Sample Program: A Simple Expression Compiler

This program's sample program is a bit complex. In fact, the theory behind this program is well beyond the scope of this text (since it involves compiler theory). However, this example is such a good demonstration of the capabilities of HLA's macro facilities and DSEL capabilities, it was too good not to include here. The following paragraphs will attempt to explain how this compile-time program operates. If you have difficulty understanding what's going on, don't feel too bad, this code isn't exactly the type of stuff that beginning assembly language programmers would normally develop on their own.

This program presents a (very) simple *expression compiler*. This code includes a macro, `u32expr`, that emits a sequence of instructions that compute the value of an arithmetic expression and leave that result sitting in one of the 80x86's 32-bit registers. The syntax for the `u32expr` macro invocation is the following:

```
u32expr( reg32, uns32_expression );
```

This macro emits the code that computes the following (HLL) statement:

3. I.e., I don't even want to have to think about this problem!

```
reg32 := uns32_expression;
```

For example, the macro invocation "u32expr(eax, ebx+ecx*5 - edi);" computes the value of the expression "ebx+ecx*5 - edi" and leaves the result of this expression sitting in the EAX register.

The *u32expr* macro places several restrictions on the expression. First of all, as the name implies, it only computes the result of an *uns32* expression. No other data types may appear within the expression. During computation, the macro uses the EAX and EDX registers, so expressions should not contain these registers as their values may be destroyed by the code that computes the expression (EAX or EDX may safely appear as the first operand of the expression, however). Finally, expressions may only contain the following operators:

```
<, <=, >, >=, <>, !=, =, ==
      +, -
      *, /
      (, )
```

The "<>" and "!=" operators are equivalent (not equals) and the "=" and "==" operators are also equivalent (equals). The operators above are listed in order of increasing precedence; i.e., "*" has a higher precedence than "+" (as you would expect). You can override the precedence of an operator by using parentheses in the standard manner.

It is important to remember that *u32expr* is a macro, not a function. That is, the invocation of this macro results in a sequence of 80x86 assembly language instructions that computes the desired expression. The *u32expr* invocation is not a function call. to some routine that computes the result.

To understand how this macro works, it would be a good idea to review the section on "Converting Arithmetic Expressions to Postfix Notation" on page 613. That section discusses how to convert floating point expressions to reverse polish notation; although the *u32expr* macro works with *uns32* objects rather than floating point objects, the approach it uses to translate expressions into assembly language uses this same algorithm. So if you don't remember how to translate expressions into reverse polish notation, it might be worthwhile to review that section of this text.

Converting floating point expressions to reverse polish notation is especially easy because the 80x86's FPU uses a stack architecture. Alas, the integer instructions on the 80x86 use a register architecture and efficiently translating integer expression to assembly language is a bit more difficult (see "Arithmetic Expressions" on page 577). We'll solve this problem by translating the expressions to assembly code in a somewhat less than efficient manner; we'll simulate an integer stack architecture by using the 80x86's hardware stack to hold temporary results during an integer calculation.

To push an integer constant or variable onto the 80x86 hardware stack, we need only use a PUSH or PUSHED instruction. This operation is trivial.

To add two values sitting on the top of stack together, leaving their sum on the stack, all we need do is pop those two values into registers, add the register values, and then push the result back onto the stack. We can do this operation slightly more efficiently, since addition is commutative, by using the following code:

```
// Compute X+Y where X is on NOS (next on stack) and Y is on TOS (top of stack):

pop( eax );           // Get Y's value.
add( eax, [esp] );    // Add with X's value and leave sum on TOS.
```

Subtraction is identical to addition. Although subtraction is not commutative the operands just happen to be on the stack in the proper order to efficiently compute their difference. To compute "X-Y" where X is on NOS and Y is on TOS, we can use code like the following:

```
// Compute X-y where X is on NOS and Y is on TOS:

pop( eax );
sub( eax, [esp] );
```

Multiplication of the two items on the top of stack is a little more complicated since we must use the MUL instruction (the only unsigned multiplication instruction available) and the destination operand must be the EDX:EAX register pair. Fortunately, multiplication is a commutative operation, so we can compute the product of NOS and TOS using code like the following:

```
// Compute X*Y where X is on NOS and Y is on TOS:

    pop( eax );
    mul( [esp], eax );      // Note that this wipes out the EDX register.
    mov( eax, [esp] );
```

Division is problematic because it is not a commutative operation and its operands on the stack are not in a convenient order. That is, to compute X/Y it would be really convenient if X was on TOS and Y was in the NOS position. Alas, as you'll soon see, it turns out that X is at NOS and Y is on the TOS. To resolve this issue requires slightly less efficient code than the sequences we've used above. Since the DIV instruction is so slow anyway, this will hardly matter.

```
// Compute X/Y where X is on NOS and Y is on TOS:

    mov( [esp+4], eax );    // Get X from NOS.
    xor( edx, edx );        // Zero-extend EAX into EDX:EAX
    div( [esp], edx:eax );  // Compute their quotient.
    pop( edx );             // Remove unneeded Y value from the stack.
    mov( eax, [esp] );      // Store quotient to the TOS.
```

The remaining operators are the comparison operators. These operators compare the value on NOS with the value on TOS and leave true (1) or false (0) sitting on the stack based on the result of the comparison. While it is easy to work around the non-commutative aspect of many of the comparison operators, the big challenge is converting the result to true or false. The SETcc instructions are convenient for this purpose, but they only work on byte operands. Therefore, we will have to zero extend the result of the SETcc instructions to obtain an *uns32* result we can push onto the stack. Ultimately, the code we must emit for a comparison is similar to the following:

```
// Compute X <= Y where X is on NOS and Y is on TOS.

    pop( eax );
    cmp( [esp], eax );
    setbe( al );            // This instruction changes for other operators.
    movzx( al, eax );
    mov( eax, [esp] );
```

As it turns out, the appearance of parentheses in an expression only affects the order of the instructions appearing in the sequence, it does not affect the number of type of instructions that correspond to the calculation of an expression. As you'll soon see, handling parentheses is an especially trivial operation.

With this short description of how to emit code for each type of arithmetic operator, it's time to discuss exactly how we will write a macro to automate this translation. Once again, a complete discussion of this topic is well beyond the scope of this text, however a simple introduction to compiler theory will certainly ease the understanding the *u32expr* macro.

For efficiency and reasons of convenience, most compilers are broken down into several components called *phases*. A compiler phase is collection of logically related activities that take place during compilation. There are three general compiler phases we are going to consider here: (1) *lexical analysis* (or *scanning*), (2) *parsing*, and (3) *code generation*. It is important to realize that these three activities occur concurrently during compilation; that is, they take place at the same time rather than as three separate, serial, activities. A compiler will typically run the lexical analysis phase for a short period, transfer control to the parsing phase, do a little code generation, and then, perhaps, do some more scanning and parsing and code generation (not necessarily in that order). Real compilers have additional phases, the *u32expr* macro

will only use these three phases (and if you look at the macro, you'll discover that it's difficult to separate the parsing and code generation phases).

Lexical analysis is the process of breaking down a string of characters, representing the expression to compile, into a sequence of *tokens* for use by the parser. For example, an expression of the form "MaxVal - x <= \$1c" contains five distinct tokens:

- MaxVal
- -
- x
- <=
- \$1c

Breaking any one of these tokens into smaller objects would destroy the intent of the expression (e.g., converting MaxVal to "Max" and "Val" or converting "<=" into "<" and "="). The job of the lexical analyzer is to break the string down into a sequence of constituent tokens and return this sequence of tokens to the parser (generally one token at a time, as the parser requests new tokens). Another task for the lexical analyzer is to remove any extra white space from the string of symbols (since expressions may generally contain an arbitrary amount of white space).

Fortunately, it is easy to extract the next available token in the input string by skipping all white space characters and then look at the current character. Identifiers always begin with an alphabetic character or an underscore, numeric values always begin with a decimal digit, a dollar sign ("\$"), or a percent sign ("%"). Operators always begin with the corresponding punctuation character that represents the operator. There are only two major issues here: how do we classify these tokens and how do we differentiate two or more distinct tokens that start with the same character (e.g., "<", "<=", and "<>")? Fortunately, HLA's compile-time functions provide the tools we need to do this.

Consider the declaration of the *u32expr* macro:

```
macro u32expr( reg, expr ):sexpr;
```

The *expr* parameter is a text object representing the expression to compile. The *sexpr* local symbol will contain the string equivalent of this text expression. The macro translates the text *expr* object to a string with the following statement:

```
?sexpr := @string:expr;
```

From this point forward, the macro works with the string in *sexpr*.

The *lexer* macro (compile-time function) handles the lexical analysis operation. This macro expects a single string parameter from which it extracts a single token and removes the string associated with that token from the front of the string. For example, the following macro invocation returns "2" as the function result and leaves "+3" in the parameter string (*str2Lex*):

```
?str2Lex := "2+3";
?TokenResult := lexer( str2Lex );
```

The *lexer* function actually returns a little more than the string it extracts from its parameter. The actual return value is a record constant that has the definition:

```
tokType:
  record

    lexeme:string;
    tokClass:tokEnum;

  endrecord;
```

The *lexeme* field holds that actual string (e.g., "2" in this example) that the *lexer* macro returns. The *tokClass* field holds a small numeric value (see the *tokEnum* enumerated data type) that specifies that type of the token. In this example, the call to *lexer* stores the value *intconst* into the *tokClass* field. Having a single

value (like *intconst*) when the *lexeme* could take on a large number of different forms (e.g., "2", "3", "4", ...) will help make the parser easier to write. The call to *lexer* in the previous example produces the following results:

```
str2lex : "+3"
TokenResult.lexeme: "2"
TokenResult.tokClass: intconst
```

A subsequent call to *lexer*, immediately after the call above, will process the next available character in the string and return the following values:

```
str2lex : "3"
TokenResult.lexeme: "+"
TokenResult.tokClass: plusOp
```

To see how *lexer* works, consider the first few lines of the *lexer* macro:

```
macro lexer( input ):theLexeme,boolResult;

    ?theLexeme:string;          // Holds the string we scan.
    ?boolResult:boolean;       // Used only as a dummy value.

    // Check for an identifier.

    #if( @peekCset( input, tok1stIDChar ))

        // If it began with a legal ID character, extract all
        // ID characters that follow. The extracted string
        // goes into "theLexeme" and this call also removes
        // those characters from the input string.

        ?boolResult := @oneOrMoreCset( input, tokIDChars, input, theLexeme );

        // Return a tokType constant with the identifier string and
        // the "identifier" token value:

        tokType:[ theLexeme, identifier ]

    // Check for a decimal numeric constant.

    #elseif( @peekCset( input, digits ))
        .
        .
        .
```

The real work begins with the *#IF* statement where the code uses the *@peekCset* function to see if the first character of the *input* parameter is a member of the *tok1stIDChar* set (which is the alphabetic characters plus an underscore, i.e., the set of character that may appear as the first character of an identifier). If so, the code executes the *@oneOrMoreCset* function to extract all legal identifier characters (alphanumerics plus underscore), storing the result in the *theLexeme* string variable. Note that this function call to *@oneOrMoreCset* also removes the string it matches from the front of the *input* string (see the description of *@oneOrMoreCset* for more details). This macro returns a *tokType* result by simply specifying a *tokType* constant containing *theLexeme* and the enum constant *identifier*.

If the first character of the input string is not in the *tok1stIDChar* set, then the *lexer* macro checks to see if the first character is a legal decimal digit. If so, then this macro processes that string of digits in a manner very similar to identifiers. The code handles hexadecimal and binary constants in a similar fashion. About the only thing exciting in the whole macro is the way it differentiates tokens that begin with the same sym-

bol. Once it determines that a token begins with a character common to several lexemes, it calls `@matchStr` to attempt to match the longer tokens before settling on the shorter lexeme (i.e., *lexer* attempts to match "`<=`" or "`<>`" before it decides the lexeme is just "`<`"). Other than this complication, the operation of the lexer is really quite simple.

The operation of the parser/code generation phases is a bit more complex, especially since these macros are indirectly recursive; to simplify matters we will explore the parser/code generator in a bottom-up fashion.

The parser/code generator phases consist of four separate macros: *doTerms*, *doMulOps*, *doAddOps*, and *doCmpOps*. The reason for these four separate macros is to handle the different precedences of the arithmetic operators and the parentheses. An explanation of how these four macros handle the different arithmetic precedences is beyond the scope of this text; we'll just look at how these four macros do their job.

The *doTerms* macro is responsible for handling identifiers, numeric constants, and subexpressions surrounded by parentheses. The single parameter is the current input string whose first (non-blank) character sequence is an identifier, constant, or parenthetical expression. Here is the full text for this macro:

```
macro doTerms( expr ):termToken;

    // Begin by removing any leading white space from the string:

    ?expr := @trim( expr, 0 );

    // Okay, call the lexer to extract the next token from the input:

    ?termToken:tokType := lexer( expr );

    // See if the current token is an identifier. If so, assume that
    // it's an uns32 identifier and emit the code to push its value onto
    // the stack.

    #if( termToken.tokClass = identifier )

        // If we've got an identifier, emit the code to
        // push that identifier onto the stack.

        push( @text( termToken.lexeme ) );

    // If it wasn't an identifier, see if it's a numeric constant.
    // If so, emit the code that will push this value onto the stack.

    #elseif( termToken.tokClass = intconst )

        // If we've got a constant, emit the code to push
        // that constant onto the stack.

        pushd( @text( termToken.lexeme ) );

    // If it's not an identifier or an integer constant, see if it's
    // a parenthesized subexpression. If so, invoke the doCmpOps macro
    // to do the real work of code generation for the subexpression.
    // The call to the doCmpOps macro emits all the code needed to push
    // the result of the subexpression onto the stack; note that this
    // macro doesn't need to emit any code for the parenthetical expression,
    // all the code emission is handled by doCmpOps.

    #elseif( termToken.tokClass = lparen )

        // If we've got a parenthetical expression, emit
        // the code to leave the parenthesized expression
        // sitting on the stack.
```

```

doCmpOps( expr );

// We must have a closing right parentheses after the subexpression.
// Skip any white space and check for the closing ")" here.

?expr := @trim( expr, 0 );
?termToken:tokType := lexer( expr );
#if( termToken.tokClass <> rparen )

    #error( "Expected closing parenthesis: " + termToken.lexeme )

#endif

// If we get to this point, then the lexer encountered something besides
// an identifier, a numeric constant, or a parenthetical expression.

#else

    #error( "Unexpected term: '" + termToken.lexeme + "'" )

#endif

endmacro;

```

The *doTerms* macro is responsible for leaving a single item sitting on the top of the 80x86 hardware stack. That stack item is either the value of an *uns32* identifier, the value of an *uns32* expression, or the value left on the stack via a parenthesized subexpression. The important thing to remember is that you can think of *doTerms* as a function that emits code that leaves a single item on the top of the 80x86 stack.

The *doMulOps* macro handles expressions consisting of a single term (items handled by the *doTerms* macro) optionally followed by zero or more pairs consisting of a multiplicative operator ("*" or "/") and a second term. It is especially important to remember that the *doMulOps* macro does not require the presence of a multiplicative operator; it will legally process a single term (identifier, numeric constant, or parenthetical expression). If one or more multiplicative operator and term pairs are present, the *doMulOps* macro will emit the code that will multiply the values of the two terms together and push the result onto the stack. E.g., consider the following:

$$X * 5$$

Since there is a multiplicative operator present ("*"), the *doMulOps* macro will call *doTerms* to process the two terms (pushing *X* and then *Y* onto the stack) and then the *doMulOps* macro will emit the code to multiply the two values on the stack leaving their product on the stack. The complete code for the *doMulOps* macro is the following:

```

macro doMulOps( sexpr ):opToken;

    // Process the leading term (not optional). Note that
    // this expansion leaves an item sitting on the stack.

    doTerms( sexpr );

    // Process all the MULOPS at the current precedence level.
    // (these are optional, there may be zero or more of them.)
    // Begin by removing any leading white space.

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, MulOps ))

        // Save the operator so we know what code we should
        // generate later.

```

```

?opToken := lexer( sexpr );

// Get the term following the operator.

doTerms( sexpr );

// Okay, the code for the two terms is sitting on
// the top of the stack (left operand at [esp+4] and
// the right operand at [esp]). Emit the code to
// perform the specified operation.

#if( opToken.lexeme = "*" )

    // For multiplication, compute
    // [esp+4] = [esp] * [esp+4] and
    // then pop the junk off the top of stack.

    pop( eax );
    mul( (type dword [esp]) );
    mov( eax, [esp] );

#elif( opToken.lexeme = "/" )

    // For division, compute
    // [esp+4] = [esp+4] / [esp] and
    // then pop the junk off the top of stack.

    mov( [esp+4], eax );
    xor( edx, edx );
    div( [esp], edx:eax );
    pop( edx );
    mov( eax, [esp] );

#endif
?sexpr := @trim( sexpr, 0 );

#endwhile

endmacro;

```

Note the simplicity of the code generation. This macro assumes that *doTerms* has done its job leaving two values sitting on the top of the stack. Therefore, the only code this macro has to generate is the code to pop these two values off the stack and then multiply or divide them, depending on the actual operator that is present. The code generation uses the sequences appearing earlier in this section.

The *doAddOps* and *doCmpOps* macros work in a manner nearly identical to *doMulOps*. The only difference is the operators these macros handle (and, of course, the code that they generate). See Program 9.7, below, for details concerning these macros.

Once we've got the lexer and the four parser/code generation macros written, writing the *u32expr* macro is quite easy. All that *u32expr* needs to do is call the *doCmpOps* macro to compile the expression and then pop the result off the stack and store it into the destination register appearing as the first operand. This requires little more than a single POP instruction.

About the only thing interesting in the *u32expr* macro is the presence of the RETURNS statement. This HLA statement takes the following form:

```
returns( { statements }, string_expression )
```

This statement simply compiles the sequence of statements appearing between the braces in the first operand and then it uses the second *string_expression* operand as the "returns" value for this statement. As

you may recall from the discussion of instruction composition (see “Instruction Composition in HLA” on page 538), HLA substitutes the "returns" value of a statement in place of that statement if it appears as an operand to another expression. The RETURNS statement appearing in the *u32expr* macro returns the register you specify as the first parameter as the "returns" value for the macro invocation. This lets you invoke the *u32expr* macro as an operand to many different instructions (that accept a 32-bit register as an operand). For example, the following *u32expr* macro invocations are all legal:

```
mov( u32expr( eax, i*j+k/15 - 2), m );
if( u32expr( edx, eax < (ebx-2)*ecx) ) then ... endif;
funcCall( u32expr( eax, (x*x + y*y)/z*z ), 16, 2 );
```

Well, without further ado, here's the complete code for the *u32expr* compiler and some test code that checks out the operation of this macro:

```
// u32expr.hla
//
// This program demonstrates how to write an "expression compiler"
// using the HLA compile-time language. This code defines a macro
// (u32expr) that accepts an arithmetic expression as a parameter.
// This macro compiles that expression into a sequence of HLA
// machine language instructions that will compute the result of
// that expression at run-time.
//
// The u32expr macro does have some severe limitations.
// First of all, it only support uns32 operands.
// Second, it only supports the following arithmetic
// operations:
//
// +, -, *, /, <, <=, >, >=, =, <>.
//
// The comparison operators produce zero (false) or
// one (true) depending upon the result of the (unsigned)
// comparison.
//
// The syntax for a call to u32expr is
//
// u32expr( register, expression )
//
// The macro computes the result of the expression and
// leaves this result sitting in the register specified
// as the first operand. This register overwrites the
// values in the EAX and EDX registers (though these
// two registers are fine as the destination for the
// result).
//
// This macro also returns the first (register) parameter
// as its "returns" value, so you may use u32expr anywhere
// a 32-bit register is legal, e.g.,
//
//      if( u32expr( eax, (i*3-2) < j )) then
//
//          << do something if (i*3-2) < j >>
//
//      endif;
//
// The statement above computes true or false in EAX and the
// "if" statement processes this result accordingly.
```

```

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

// Some special character classifications the lexical analyzer uses.

const

    // tok1stIDChar is the set of legal characters that
    // can begin an identifier. tokIDChars is the set
    // of characters that may follow the first character
    // of an identifier.

    tok1stIDChar := { 'a'..'z', 'A'..'Z', '_' };
    tokIDChars := { 'a'..'z', 'A'..'Z', '0'..'9', '_' };

    // digits, hexDigits, and binDigits are the sets
    // of characters that are legal in integer constants.
    // Note that these definitions don't allow underscores
    // in numbers, although it would be a simple fix to
    // allow this.

    digits := { '0'..'9' };
    hexDigits := { '0'..'9', 'a'..'f', 'A'..'F' };
    binDigits := { '0'..'1' };

    // CmpOps, PlusOps, and MulOps are the sets of
    // operator characters legal at three levels
    // of precedence that this parser supports.

    CmpOps := { '>', '<', '=', '!' };
    PlusOps := { '+', '-' };
    MulOps := { '*', '/' };

type

    // tokEnum-
    //
    // Data values the lexical analyzer returns to quickly
    // determine the classification of a lexeme. By
    // classifying the token with one of these values, the
    // parser can more quickly process the current token.
    // I.e., rather than having to compare a scanned item
    // against the two strings "+" and "-", the parser can
    // simply check to see if the current item is a "plusOp"
    // (which indicates that the lexeme is "+" or "-").
    // This speeds up the compilation of the expression since
    // only half the comparisons are needed and they are
    // simple integer comparisons rather than string comparisons.

    tokEnum:      enum
    {
        identifier,
        intconst,
        lparen,
        rparen,

```

```

        plusOp,
        mulOp,
        cmpOp
    };

// tokType-
//
// This is the "token" type returned by the lexical analyzer.
// The "lexeme" field contains the string that matches the
// current item scanned by the lexer. The "tokClass" field
// contains a generic classification for the symbol (see the
// "tokEnum" type above).

tokType:
    record

        lexeme:string;
        tokClass:tokEnum;

    endrecord;

// lexer-
//
// This is the lexical analyzer. On each call it extracts a
// lexical item from the front of the string passed to it as a
// parameter (it also removes this item from the front of the
// string). If it successfully matches a token, this macro
// returns a tokType constant as its return value.

macro lexer( input ):theLexeme,boolResult;

    ?theLexeme:string;      // Holds the string we scan.
    ?boolResult:boolean;    // Used only as a dummy value.

    // Check for an identifier.

    #if( @peekCset( input, tok1stIDChar ))

        // If it began with a legal ID character, extract all
        // ID characters that follow. The extracted string
        // goes into "theLexeme" and this call also removes
        // those characters from the input string.

        ?boolResult := @oneOrMoreCset( input, tokIDChars, input, theLexeme );

        // Return a tokType constant with the identifier string and
        // the "identifier" token value:

        tokType:[ theLexeme, identifier ]

    // Check for a decimal numeric constant.

    #elseif( @peekCset( input, digits ))

        // If the current item began with a decimal digit, extract
        // all the following digits and put them into "theLexeme".
        // Also remove these characters from the input string.

```

```

?boolResult := @oneOrMoreCset( input, digits, input, theLexeme );

// Return an integer constant as the current token.

tokType:[ theLexeme, intconst ]

// Check for a hexadecimal numeric constant.

#elseif( @peekChar( input, '$' ))

    // If we had a "$" symbol, grab it and any following
    // hexadecimal digits. Set boolResult true if there
    // is at least one hexadecimal digit. As usual, extract
    // the hex value to "theLexeme" and remove the value
    // from the input string:

    ?boolResult := @oneChar( input, '$', input ) &&
        @oneOrMoreCset( input, hexDigits, input, theLexeme );

    // Returns the hex constant string as an intconst object:

    tokType:[ '$' + theLexeme, intconst ]

// Check for a binary numeric constant.

#elseif( @peekChar( input, '%' ))

    // See the comments for hexadecimal constants. This boolean
    // constant scanner works the same way.

    ?boolResult := @oneChar( input, '%', input ) &&
        @oneOrMoreCset( input, binDigits, input, theLexeme );
    tokType:[ '%' + theLexeme, intconst ]

// Handle the "+" and "-" operators here.

#elseif( @peekCset( input, PlusOps ))

    // If it was a "+" or "-" sign, extract it from the input
    // and return it as a "plusOp" token.

    ?boolResult := @oneCset( input, PlusOps, input, theLexeme );
    tokType:[ theLexeme, plusOp ]

// Handle the "*" and "/" operators here.

#elseif( @peekCset( input, MulOps ))

    // If it was a "*" or "/" sign, extract it from the input
    // and return it as a "mulOp" token.

    ?boolResult := @oneCset( input, MulOps, input, theLexeme );
    tokType:[ theLexeme, mulOp ]

```

```

// Handle the "=" ("=="), "<>" ("!="), "<", "<=", ">", and ">="
// operators here.

#elseif( @peekCset( input, CmpOps ))

    // Note that we must check for two-character operators
    // first so we don't confuse them with the single
    // character opertors:

    #if
    (
        @matchStr( input, ">=", input, theLexeme )
        || @matchStr( input, "<=", input, theLexeme )
        || @matchStr( input, "<>", input, theLexeme )
    )

        tokType:[ theLexeme, cmpOp ]

    #elseif( @matchStr( input, "!=", input, theLexeme ))

        tokType:[ "<>", cmpOp ]

    #elseif( @matchStr( input, "==", input, theLexeme ))

        tokType:[ "=", cmpOp ]

    #elseif( @oneCset( input, {'>', '<', '='}, input, theLexeme ))

        tokType:[ theLexeme, cmpOp ]

    #else

        #error( "Illegal comparison operator: " + input )

    #endif

// Handle the parentheses down here.

#elseif( @oneChar( input, '(', input, theLexeme ))

    tokType:[ "(", lparen ]

#elseif( @oneChar( input, ')', input, theLexeme ))

    tokType:[ ")", rparen ]

// Anything else is an illegal character.

#else

    #error
    (
        "Illegal character in expression: '" +
        @substr( input, 0, 1 ) +
        "' ($" +
        string( dword( @substr( input, 0, 1 ))) +

```

```

        ")"
    )
    ?input := @substr( input, 1, @length(input) - 1 );

#endif

endmacro;

// Handle identifiers, constants, and sub-expressions within
// parentheses within this macro.
//
// terms-> identifier | intconst | '(' CmpOps ')'
//
// This compile time function does the following:
//
// (1) If it encounters an identifier, it emits the
//     following instruction to the code stream:
//
//         push( identifier );
//
// (2) If it encounters an (unsigned) integer constant, it emits
//     the following instruction to the code stream:
//
//         pushd( constant_value );
//
// (3) If it encounters an expression surrounded by parentheses,
//     then it emits whatever instruction sequence is necessary
//     to leave the value of that (unsigned integer) expression
//     sitting on the top of the stack.
//
// (4) If the current lexeme is none of the above, then this
//     macro prints an appropriate error message.
//
// The end result of the execution of this macro is the emission
// of some code that leaves a single 32-bit unsigned value sitting
// on the top of the 80x86 stack (assuming no error).

macro doTerms( expr ):termToken;

    ?expr := @trim( expr, 0 );
    ?termToken:tokType := lexer( expr );
    #if( termToken.tokClass = identifier )

        // If we've got an identifier, emit the code to
        // push that identifier onto the stack.

        push( @text( termToken.lexeme ) );

    #elseif( termToken.tokClass = intconst )

        // If we've got a constant, emit the code to push
        // that constant onto the stack.

        pushd( @text( termToken.lexeme ) );

    #elseif( termToken.tokClass = lparen )

        // If we've got a parenthetical expression, emit

```

```

// the code to leave the parenthesized expression
// sitting on the stack.

doCmpOps( expr );
?expr := @trim( expr, 0 );
?termToken:tokType := lexer( expr );
#if( termToken.tokClass <> rparen )

    #error( "Expected closing parenthesis: " + termToken.lexeme )

#endif

#else

    #error( "Unexpected term: '" + termToken.lexeme + "'" )

#endif

endmacro;

// Handle the multiplication, division, and modulo operations here.
//
// MulOps-> terms ( mulOp terms ) *
//
// The above grammar production tells us that a "MulOps" consists
// of a "terms" expansion followed by zero or more instances of a
// "mulop" followed by a "terms" expansion (like wildcard filename
// expansions, the "*" indicates zero or more copies of the things
// inside the parentheses).
//
// This code assumes that "terms" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single term (no optional mulOp/term following), then this code
// does nothing (it leaves the result on the stack that was pushed
// by the "terms" expansion). If one or more mulOp/terms pairs are
// present, then for each pair this code assumes that the two "terms"
// expansions left some value on the stack. This code will pop
// those two values off the stack and multiply or divide them and
// push the result back onto the stack (sort of like the way the
// FPU multiplies or divides values on the FPU stack).
//
// If there are three or more operands in a row, separated by
// mulops ("*" or "/") then this macro will process them in
// a left-to-right fashion, popping each pair of values off the
// stack, operating on them, pushing the result, and then processing
// the next pair. E.g.,
//
//      i * j * k
//
// yields:
//
//      push( i ); // From the "terms" macro.
//      push( j ); // From the "terms" macro.
//
//      pop( eax ); // Compute the product of i*j
//      mul( (type dword [esp]));
//      mov( eax, [esp]);
//

```

```

//      push( k ); // From the "terms" macro.
//
//      pop( eax ); // Pop K
//      mul( (type dword [esp])); // Compute K* (i*j) [i*j is value on TOS].
//      mov( eax, [esp]); // Save product on TOS.

macro doMulOps( sexpr ):opToken;

    // Process the leading term (not optional). Note that
    // this expansion leaves an item sitting on the stack.

    doTerms( sexpr );

    // Process all the MULOPs at the current precedence level.
    // (these are optional, there may be zero or more of them.)

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, MulOps ))

        // Save the operator so we know what code we should
        // generate later.

        ?opToken := lexer( sexpr );

        // Get the term following the operator.

        doTerms( sexpr );

        // Okay, the code for the two terms is sitting on
        // the top of the stack (left operand at [esp+4] and
        // the right operand at [esp]). Emit the code to
        // perform the specified operation.

        #if( opToken.lexeme = "*" )

            // For multiplication, compute
            // [esp+4] = [esp] * [esp+4] and
            // then pop the junk off the top of stack.

            pop( eax );
            mul( (type dword [esp]) );
            mov( eax, [esp] );

        #elseif( opToken.lexeme = "/" )

            // For division, compute
            // [esp+4] = [esp+4] / [esp] and
            // then pop the junk off the top of stack.

            mov( [esp+4], eax );
            xor( edx, edx );
            div( [esp], edx:eax );
            pop( edx );
            mov( eax, [esp] );

        #endif

        ?sexpr := @trim( sexpr, 0 );

    #endwhile

```

```

endmacro;

// Handle the addition, and subtraction operations here.
//
// AddOps-> MulOps ( addOp MulOps ) *
//
// The above grammar production tells us that an "AddOps" consists
// of a "MulOps" expansion followed by zero or more instances of an
// "addOp" followed by a "MulOps" expansion.
//
// This code assumes that "MulOps" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single MulOps item then this code does nothing. If one or more
// addOp/MulOps pairs are present, then for each pair this code
// assumes that the two "MulOps" expansions left some value on the
// stack. This code will pop those two values off the stack and
// add or subtract them and push the result back onto the stack.

macro doAddOps( sexpr ):opToken;

    // Process the first operand (or subexpression):

    doMulOps( sexpr );

    // Process all the ADDOPs at the current precedence level.

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, PlusOps ))

        // Save the operator so we know what code we should
        // generate later.

        ?opToken := lexer( sexpr );

        // Get the MulOp following the operator.

        doMulOps( sexpr );

        // Okay, emit the code associated with the operator.

        #if( opToken.lexeme = "+" )

            pop( eax );
            add( eax, [esp] );

        #elseif( opToken.lexeme = "-" )

            pop( eax );
            pop( edx );
            sub( eax, edx );
            push( edx );

        #endif

    #endwhile

endmacro;

```

```

// Handle the comparison operations here.
//
// CmpOps-> addOps ( cmpOp AddOps ) *
//
// The above grammar production tells us that a "CmpOps" consists
// of an "AddOps" expansion followed by zero or more instances of an
// "cmpOp" followed by an "AddOps" expansion.
//
// This code assumes that "MulOps" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single MulOps item then this code does nothing. If one or more
// addOp/MulOps pairs are present, then for each pair this code
// assumes that the two "MulOps" expansions left some value on the
// stack. This code will pop those two values off the stack and
// add or subtract them and push the result back onto the stack.

macro doCmpOps( sexpr ):opToken;

    // Process the first operand:

    doAddOps( sexpr );

    // Process all the CMPOPs at the current precedence level.

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, CmpOps ))

        // Save the operator for the code generation task later.

        ?opToken := lexer( sexpr );

        // Process the item after the comparison operator.

        doAddOps( sexpr );

        // Generate the code to compare [esp+4] against [esp]
        // and leave true/false sitting on the stack in place
        // of these two operands.

        #if( opToken.lexeme = "<" )

            pop( eax );
            cmp( [esp], eax );
            setb( al );
            movzx( al, eax );
            mov( eax, [esp] );

        #elseif( opToken.lexeme = "<=" )

            pop( eax );
            cmp( [esp], eax );
            setbe( al );
            movzx( al, eax );
            mov( eax, [esp] );

        #elseif( opToken.lexeme = ">" )

```

```

        pop( eax );
        cmp( [esp], eax );
        seta( al );
        movzx( al, eax );
        mov( eax, [esp] );

#elif( opToken.lexeme = ">=" )

        pop( eax );
        cmp( [esp], eax );
        setae( al );
        movzx( al, eax );
        mov( eax, [esp] );

#elif( opToken.lexeme = "=" )

        pop( eax );
        cmp( [esp], eax );
        sete( al );
        movzx( al, eax );
        mov( eax, [esp] );

#elif( opToken.lexeme = "<>" )

        pop( eax );
        cmp( [esp], eax );
        setne( al );
        movzx( al, eax );
        mov( eax, [esp] );

#endif

#endif

#endif

endmacro;

// General macro that does the expression compilation.
// The first parameter must be a 32-bit register where
// this macro will leave the result. The second parameter
// is the expression to compile. The expression compiler
// will destroy the value in EAX and may destroy the value
// in EDX (though EDX and EAX make fine destination registers
// for this macro).
//
// This macro generates poor machine code. It is more a
// "proof of concept" rather than something you should use
// all the time. Nevertheless, if you don't have serious
// size or time constraints on your code, this macro can be
// quite handy. Writing an optimizer is left as an exercise
// to the interested reader.

macro u32expr( reg, expr):sexpr;

    // The "returns" statement processes the first operand
    // as a normal sequence of statements and then returns
    // the second operand as the "returns" value for this
    // macro.

```

```

returns
(
    {

        ?sexpr:string := @string:expr;
        #if( !@IsReg32( reg ) )

            #error( "Expected a 32-bit register" )

        #else

            // Process the expression and leave the
            // result sitting in the specified register.

            doCmpOps( sexpr );
            pop( reg );

        #endif
    },

    // Return the specified register as the "returns"
    // value for this compilation:

    @string:reg
)

endmacro;

// The following main program provides some examples of the
// use of the above macro:

static
    x:uns32;
    v:uns32 := 5;

begin TestExpr;

    mov( 10, x );
    mov( 12, ecx );

    // Compute:
    //
    // edi := (x*3/v + %1010 == 16) + ecx;
    //
    // This is equivalent to:
    //
    // edi := (10*3/5 + %1010 == 16) + 12
    //      := ( 30/5 + %1010 == 16) + 12
    //      := ( 6 + 10 == 16) + 12
    //      := ( 16 == 16) + 12
    //      := ( 1 ) + 12
    //      := 13
    //
    // This macro invocation emits the following code:
    //
    // push(x);
    // pushd(3);

```

```

// pop(eax);
// mul( (type dword [esp]) );
// mov( eax, [esp] );
// push( v );
// mov( [esp+4], eax );
// xor edx, edx
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pushd( 10 );
// pop( eax );
// add( eax, [esp] );
// pushd( 16 );
// pop( eax );
// cmp( [esp], eax );
// sete( al );
// movzx( al, eax );
// mov( eax, [esp+0] );
// push( ecx );
// pop( eax );
// add( eax, [esp] );
// pop( edi );

u32expr( edi, (x*3/v+%1010 == 16) + ecx );
stdout.put( "Sum = ", (type uns32 edi), nl );

// Now compute:
//
//   eax := x + ecx/2
//       := 10 + 12/2
//       := 10 + 6
//       := 16
//
// This macro emits the following code:
//
//   push( x );
//   push( ecx );
//   pushd( 2 );
//   mov( [esp+4], eax );
//   xor( edx, edx );
//   div( [esp], edx:eax );
//   pop( edx );
//   mov( eax, [esp] );
//   pop( eax );
//   add( eax, [esp] );
//   pop( eax );

u32expr( eax, x+ecx/2 );
stdout.put( "x=", x, " ecx=", (type uns32 ecx), " v=", v, nl );
stdout.put( "x+ecx/2 = ", (type uns32 eax ), nl );

// Now determine if (x+ecx/2) < v
// (it is not since (x+ecx/2)=16 and v = 5.)
//
// This macro invocation emits the following code:
//

```

```

// push( x );
// push( ecx );
// pushd( 2 );
// mov( [esp+4], eax );
// xor( edx, edx );
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pop( eax );
// add( eax, [esp] );
// push( v );
// pop( eax );
// cmp( eax, [esp+0] );
// setb( al );
// movzx( al, eax );
// mov( eax, [esp+0] );
// pop( eax );

if( u32expr( eax, x+ecx/2 < v ) ) then

    stdout.put( "x+ecx/2 < v" nl );

else

    stdout.put( "x+ecx/2 >= v" nl );

endif;

end TestExpr;

```

Program 9.7 Uns32 Expression Compiler

9.4 Putting It All Together

The ability to extend the HLA language is one of the most powerful features of the HLA language. In this chapter you got to explore the use of several tools that allow you to extend the base language. Although a complete treatise on language design and implementation is beyond the scope of this chapter, further study in the area of compiler construction will help you learn new techniques for extending the HLA language. Later volumes in this text, including the volume on advanced string handling, will cover additional topics of interest to those who want to design and implement their own language constructs.

Classes and Objects

Chapter Ten

10.1 Chapter Overview

Many modern imperative high level languages now support the notion of classes and objects in their programming paradigm. C++ (an object version of C) and Delphi (an object version of Pascal) are two good examples. Of course, these high level language compilers translate their high level source code into low-level machine code, so it should be pretty obvious that some mechanism exists in machine code for implementing classes and objects.

Although it has always been possible to implement classes and objects in machine code, most assemblers provide poor support for writing object-oriented assembly language programs. Of course, HLA does not suffer from this drawback as it provides good support for writing object-oriented assembly language programs. This chapter discusses the general principles behind object-oriented programming (OOP) and how HLA supports OOP.

10.2 General Principles

Before discussing the mechanisms behind OOP, it is probably a good idea to take a step back and explore the benefits of using OOP (especially in assembly language programs). Most texts describing the benefits of OOP will mention buzz-words like “code reuse,” “abstract data types,” “improved development efficiency,” and so on. While all of these features are nice and are good attributes for a programming paradigm, a good software engineer would question the use of assembly language in an environment where “improved development efficiency” is an important goal. After all, you can probably obtain far better efficiency by using a high level language (even in a non-OOP fashion) than you can by using objects in assembly language. If the purported features of OOP don’t seem to apply to assembly language programming, why bother using OOP in assembly? This section will explore some of those reasons.

The first thing you should realize is that the use of assembly language does not negate the aforementioned OOP benefits. OOP in assembly language does promote code reuse, it provides a good method for implementing abstract data types, and it can improve development efficiency *in assembly language*. In other words, if you’re dead set on using assembly language, there are benefits to using OOP.

To understand one of the principle benefits of OOP, consider the concept of a global variable. Most programming texts strongly recommend against the use of global variables in a program (as does this text). Interprocedural communication through global variables is dangerous because it is difficult to keep track of all the possible places in a large program that modify a given global object. Worse, it is very easy when making enhancements to accidentally reuse a global object for something other than its intended purpose; this tends to introduce defects into the system.

Despite the well-understood problems with global variables, the semantics of global objects (extended lifetimes and accessibility from different procedures) are absolutely necessary in various situations. Objects solve this problem by letting the programmer decide on the lifetime of an object¹ as well as allow access to data fields from different procedures. Objects have several advantages over simple global variables insofar as objects can control access to their data fields (making it difficult for procedures to accidentally access the data) and you can also create multiple *instances* of an object allowing two separate sections of your program to use their own unique “global” object without interference from the other section.

Of course, objects have many other valuable attributes. One could write several volumes on the benefits of objects and OOP; this single chapter cannot do this subject justice. The following subsections present objects with an eye towards using them in HLA/assembly programs. However, if you are a beginning to

1. That is, the time during which the system allocates memory for an object.

OOP or wish more information about the object-oriented paradigm, you should consult other texts on this subject.

An important use for classes and objects is to create *abstract data types* (ADTs). An abstract data type is a collection of data objects and the functions (which we'll call *methods*) that operate on that data. In a pure abstract data type, the ADT's methods are the only code that has access to the data fields of the ADT; external code may only access the data using function calls to get or set data field values (these are the ADT's *accessor* methods). In real life, for efficiency reasons, most languages that support ADTs allow, at least, limited access to the data fields of an ADT by external code.

Assembly language is not a language most people associate with ADTs. Nevertheless, HLA provides several features to allow the creation of rudimentary ADTs. While some might argue that HLA's facilities are not as complete as those in a language such as C++ or Java, keep in mind that these differences exist because HLA is assembly language.

True ADTs should support *information hiding*. This means that the ADT does not allow the user of an ADT access to internal data structures and routines which manipulate those structures. In essence, information hiding restricts access to an ADT to only the accessor methods provided by the ADT. Assembly language, of course, provides very few restrictions. If you are dead set on accessing an object directly, there is very little HLA can do to prevent you from doing this. However, HLA has some facilities which will provide a small amount of information hiding capabilities. Combined with some care on your part, you will be able to enjoy many of the benefits of information hiding within your programs.

The primary facility HLA provides to support information hiding is separate compilation, linkable modules, and the `#INCLUDE`/`#INCLUDEONCE` directives. For our purposes, an abstract data type definition will consist of two sections: an *interface* section and an *implementation* section.

The interface section contains the definitions which must be visible to the application program. In general, it should not contain any specific information which would allow the application program to violate the information hiding principle, but this is often impossible given the nature of assembly language. Nevertheless, you should attempt to only reveal what is absolutely necessary within the interface section.

The implementation section contains the code, data structures, etc., to actually implement the ADT. While some of the methods and data types appearing in the implementation section may be public (by virtue of appearance within the interface section), many of the subroutines, data items, and so on will be private to the implementation code. The implementation section is where you hide all the details from the application program.

If you wish to modify the abstract data type at some point in the future, you will only have to change the interface and implementation sections. Unless you delete some previously visible object which the applications use, there will be no need to modify the applications at all.

Although you could place the interface and implementation sections directly in an application program, this would not promote information hiding or maintainability, especially if you have to include the code in several different applications. The best approach is to place the implementation section in an include file which any interested application reads using the HLA `#INCLUDE` directive and to place the implementation section in a separate module that you link with your applications.

The include file would contain `EXTERNAL` directives, any necessary macros, and other definitions you want made public. It generally would not contain 80x86 code except, perhaps, in some macros. When an application wants to make use of an ADT it would include this file.

The separate assembly file containing the implementation section would contain all the procedures, functions, data objects, etc., to actually implement the ADT. Those names which you want to be public should appear in the interface include file and have the `EXTERNAL` attribute. You should also include the interface include file in the implementation file so you do not have to maintain two sets of `EXTERNAL` directives.

One problem with using procedures for data access methods is the fact that many accessor methods are especially trivial (typically just a `MOV` instruction) and the overhead of the call and return instructions is expensive for such trivial operations. For example, suppose you have an ADT whose data object is a structure, but you do not want to make the field names visible to the application and you really do not want to allow the application to access the fields of the data structure directly (because the data structure may change in the future). The normal way to handle this is to supply a method *GetField* which returns the desired field

of the object. However, as pointed out above, this can be very slow. An alternative, for simple access methods is to use a macro to emit the code to access the desired field. Although code to directly access the data object appears in the application program (via macro expansion), it will be automatically updated if you ever change the macro in the interface section by simply assembling your application.

Although it is quite possible to create ADTs using nothing more than separate compilation and, perhaps, RECORDs, HLA does provide a better solution: the class. Read on to find out about HLA's support for classes and objects as well as how to use these to create ADTs.

10.3 Classes in HLA

HLA's classes provide a good mechanism for creating abstract data types. Fundamentally, a class is little more than a RECORD declaration that allows the definition of fields other than data fields (e.g., procedures, constants, and macros). The inclusion of other program declaration objects in the class definition dramatically expands the capabilities of a class over that of a record. For example, with a class it is now possible to easily define an ADT since classes may include data and methods that operate on that data (procedures).

The principle way to create an abstract data type in HLA is to declare a class data type. Classes in HLA always appear in the TYPE section and use the following syntax:

```
classname :   class

               << Class declaration section >>

            endcclass;
```

The class declaration section is very similar to the local declaration section for a procedure insofar as it allows CONST, VAL, VAR, and STATIC variable declaration sections. Classes also let you define macros and specify procedure, iterator, and *method* prototypes (method declarations are legal only in classes). Conspicuously absent from this list is the TYPE declaration section. You cannot declare new types within a class.

A method is a special type of procedure that appears only within a class. A little later you will see the difference between procedures and methods, for now you can treat them as being one and the same. Other than a few subtle details regarding class initialization and the use of pointers to classes, their semantics are identical². Generally, if you don't whether to use a procedure or method in a class, the safest bet is to use a method.

You do not place procedure/iterator/method code within a class. Instead you simply supply *prototypes* for these routines. A routine prototype consists of the PROCEDURE, ITERATOR, or METHOD reserved word, the routine name, any parameters, and a couple of optional attributes (RETURNS and EXTERNAL). The actual routine definition (i.e., the body of the routine and any local declarations it needs) appears outside the class.

The following example demonstrates a typical class declaration appearing in the TYPE section:

```
TYPE
  TypicalClass: class

      const
          TCconst := 5;

      val
          TCval := 6;

      var
          TCvar : uns32;           // Private field used only by TCproc.
```

2. Note, however, that the difference between procedures and methods makes all the difference in the world to the object-oriented programming paradigm. Hence the inclusion of methods in HLA's class definitions.

```

static
    TCstatic : int32;

procedure TCproc( u:uns32 ); returns( "eax" );
iterator TCiter( i:int32 ); external;
method TCmethod( c:char );

endclass;

```

As you can see, classes are very similar to records in HLA. Indeed, you can think of a record as being a class that only allows VAR declarations. HLA implements classes in a fashion quite similar to records insofar as it allocates sequential data fields in sequential memory locations. In fact, with only one minor exception, there is almost no difference between a RECORD declaration and a CLASS declaration that only has a VAR declaration section. Later you'll see exactly how HLA implements classes, but for now you can assume that HLA implements them the same as it does records and you won't be too far off the mark.

You can access the *TCvar* and *TCstatic* fields (in the class above) just like a record's fields. You access the CONST and VAL fields in a similar manner. If a variable of type *TypicalClass* has the name *obj*, you can access the fields of *obj* as follows:

```

mov ( obj.TCconst, eax );
mov( obj.TCval, ebx );
add( obj.TCvar, eax );
add( obj.TCstatic, ebx );
obj.TCproc( 20 );          // Calls the TCproc procedure in TypicalClass.
etc.

```

If an application program includes the class declaration above, it can create variables using the *TypicalClass* type and perform operations using the above methods. Unfortunately, the application program can also access the fields of the ADT data type with impunity. For example, if a program created a variable *MyClass* of type *TypicalClass*, then it could easily execute instructions like "MOV(MyClass.TCvar, eax);" even though this field might be private to the implementation section. Unfortunately, if you are going to allow an application to declare a variable of type *TypicalClass*, the field names will have to be visible. While there are some tricks we could play with HLA's class definitions to help hide the private fields, the best solution is to thoroughly comment the private fields and then exercise some restraint when accessing the fields of that class. Specifically, this means that ADTs you create using HLA's classes cannot be "pure" ADTs since HLA allows direct access to the data fields. However, with a little discipline, you can simulate a pure ADT by simply electing not to access such fields outside the class' methods, procedures, and iterators.

Prototypes appearing in a class are effectively FORWARD declarations. Like normal forward declarations, all procedures, iterators, and methods you define in a class must have an actual implementation later in the code. Alternately, you may attach the EXTERNAL keyword to the end of a procedure, iterator, or method declaration within a class to inform HLA that the actual code appears in a separate module. As a general rule, class declarations appear in header files and represent the interface section of an ADT. The procedure, iterator, and method bodies appear in the implementation section which is usually a separate source file that you compile separately and link with the modules that use the class.

The following is an example of a sample class procedure implementation:

```

procedure TypicalClass.TCproc( u:uns32 ); nodisplay;
    << Local declarations for this procedure >>
begin TCproc;

    << Code to implement whatever this procedure does >>

end TCProc;

```

There are several differences between a standard procedure declaration and a class procedure declaration. First, and most obvious, the procedure name includes the class name (e.g., *TypicalClass.TCproc*). This differentiates this class procedure definition from a regular procedure that just happens to have the name

TCproc. Note, however, that you do not have to repeat the class name before the procedure name in the BEGIN and END clauses of the procedure.

A second difference between class procedures and non-class procedures is not obvious. Some procedure attributes (EXTERNAL, RETURNS) are legal only in the prototype declaration appearing within the class while other attributes (NOFRAME, NODISPLAY, NOALIGNSTK, and ALIGN) are legal only within the procedure definition and not within the class. Fortunately, HLA provides helpful error messages if you stick the option in the wrong place, so you don't have to memorize this rule.

If a class routine's prototype does not have the EXTERNAL option, the compilation unit (that is, the PROGRAM or UNIT) containing the class declaration must also contain the routine's definition or HLA will generate an error at the end of the compilation. For small, local, classes (i.e., when you're embedding the class declaration and routine definitions in the same compilation unit) the convention is to place the class' procedure, iterator, and method definitions in the source file shortly after the class declaration. For larger systems (i.e., when separately compiling a class' routines), the convention is to place the class declaration in a header file by itself and place all the procedure, iterator, and method definitions in a separate HLA unit and compile them by themselves.

10.4 Objects

Remember, a class definition is just a type. Therefore, when you declare a class type you haven't created a variable whose fields you can manipulate. An *object* is an *instance* of a class; that is, an object is a variable whose type is some class type. You declare objects (i.e., class variables) the same way you declare other variables: in a VAR, STATIC, DATA, or STORAGE section³. A pair of sample object declarations follow:

```
var
    T1: TypicalClass;
    T2: TypicalClass;
```

For a given class object, HLA allocates storage for each variable appearing in the VAR section of the class declaration. If you have two objects, *T1* and *T2*, of type *TypicalClass* then *T1.TCvar* is unique as is *T2.TCvar*. This is the intuitive result (similar to RECORD declarations); most data fields you define in a class will appear in the VAR declaration section.

Static data objects (e.g., those you declare in the STATIC section of a class declaration) are not unique among the objects of that class; that is, HLA allocates only a single static variable that all variables of that class share. For example, consider the following (partial) class declaration and object declarations:

```
type
    sc: class

        var
            i: int32;

        static
            s: int32;
            .
            .
            .
    endclass;

var
    s1: sc;
    s2: sc;
```

In this example, *s1.i* and *s2.i* are different variables. However, *s1.s* and *s2.s* are aliases of one another. Therefore, an instruction like "mov(5, s1.s);" also stores five into *s2.s*. Generally you use static class vari-

3. Technically, you could also declare an object in a READONLY section, but HLA does not allow you to define class constants, so there is little utility in declaring class objects in the READONLY section.

ables to maintain information about the whole class while you use class VAR objects to maintain information about the specific object. Since keeping track of class information is relatively rare, you will probably declare most class data fields in a VAR section.

You can also create dynamic instances of a class and refer to those dynamic objects via pointers. In fact, this is probably the most common form of object storage and access. The following code shows how to create pointers to objects and how you can dynamically allocate storage for an object:

```
var
    pSC: pointer to sc;
    .
    .
    .
    malloc( @size( sc ) );
    mov( eax, pSC );
    .
    .
    .
    mov( pSC, ebx );
    mov( (type sc [ebx]).i, eax );
```

Note the use of type coercion to cast the pointer in EBX as type *sc*.

10.5 Inheritance

Inheritance is one of the most fundamental ideas behind object-oriented programming. The basic idea behind inheritance is that a class inherits, or copies, all the fields from some class and then possibly expands the number of fields in the new data type. For example, suppose you created a data type *point* which describes a point in the planar (two dimensional) space. The class for this point might look like the following:

```
type
    point: class
        var
            x:int32;
            y:int32;
            method distance;

    endclass;
```

Suppose you want to create a point in 3D space rather than 2D space. You can easily build such a data type as follows:

```
type
    point3D: class inherits( point )
        var
            z:int32;

    endclass;
```

The INHERITS option on the CLASS declaration tells HLA to insert the fields of *point* at the beginning of the class. In this case, *point3D* inherits the fields of *point*. HLA always places the inherited fields at the beginning of a class object. The reason for this will become clear a little later. If you have an instance of *point3D* which you call *P3*, then the following 80x86 instructions are all legal:

```
mov( P3.x, eax );
add( P3.y, eax );
mov( eax, P3.z );
```

```
P3.distance();
```

Note that the *P3.distance* method invocation in this example calls the *point.distance* method. You do not have to write a separate *distance* method for the *point3D* class unless you really want to do so (see the next section for details). Just like the *x* and *y* fields, *point3D* objects inherit *point*'s methods.

10.6 Overriding

Overriding is the process of replacing an existing method in an inherited class with one more suitable for the new class. In the *point* and *point3D* examples, the *distance* method (presumably) computes the distance from the origin to the specified point. For a point on a two-dimensional plane, you can compute the distance using the function:

$$\text{dist} = \sqrt{x^2 + y^2}$$

However, the distance for a point in 3D space is given by the equation:

$$\text{dist} = \sqrt{x^2 + y^2 + z^2}$$

Clearly, if you call the *distance* function for *point* for a *point3D* object you will get an incorrect answer. In the previous section, however, you saw that the *P3* object calls the distance function inherited from the *point* class. Therefore, this would produce an incorrect result.

In this situation the *point3D* data type must override the *distance* method with one which computes the correct value. You cannot simply redefine the *point3D* class by adding a *distance* method prototype:

```
type
  point3D:  class inherits( point )

            var
              z:int32;

            method distance;  // This doesn't work!

endclass;
```

The problem with the *distance* method declaration above is that *point3D* already has a distance method – the one that it inherits from the *point* class. HLA will complain because it doesn't like two methods by the same name in a single class.

To solve this problem, we need some mechanism by which we can override the declaration of *point.distance* and replace it with a declaration for *point3D.distance*. To do this, you use the **OVERRIDE** keyword before the method declaration:

```
type
  point3D:  class inherits( point )

            var
              z:int32;

            override method distance;  // This will work!

endclass;
```

The **OVERRIDE** prefix tells HLA to ignore the fact that *point3D* inherits a method named *distance* from the *point* class. Now, any call to the *distance* method via a *point3D* object will call the *point3D.distance* method rather than *point.distance*. Of course, once you override a method using the **OVERRIDE** prefix, you must supply the method in the implementation section of your code, e.g.,

```
method point3D.distance; nodisplay;

    << local declarations for the distance function >>
```

```
begin distance;

    << Code to implement the distance function >>

end distance;
```

10.7 Virtual Methods vs. Static Procedures

A little earlier, this chapter suggested that you could treat class methods and class procedures the same. There are, in fact, some major differences between the two (after all, why have methods if they're the same as procedures?). As it turns out, the differences between methods and procedures is crucial if you want to develop object-oriented programs. Methods provide the second feature necessary to support true polymorphism: virtual procedure calls⁴. A virtual procedure call is just a fancy name for an indirect procedure call (using a pointer associated with the object). The key benefit of virtual procedures is that the system automatically calls the right method when using pointers to generic objects.

Consider the following declarations using the *point* class from the previous sections:

```
var
    P2: point;
    p: pointer to point;
```

Given the declarations above, the following assembly statements are all legal:

```
mov( P2.x, eax );
mov( P2.y, ecx );
P2.distance();      // Calls point3D.distance.

lea( ebx, P2 );     // Store address of P2 into P.
mov( ebx, P );
P.distance();       // Calls point.distance.
```

Note that HLA lets you call a method via a pointer to an object rather than directly via an object variable. This is a crucial feature of objects in HLA and a key to implementing *virtual method calls*.

The magic behind polymorphism and inheritance is that object pointers are *generic*. In general, when your program references data indirectly through a pointer, the value of the pointer should be the address of the underlying data type associated with that pointer. For example, if you have a pointer to a 16-bit unsigned integer, you wouldn't normally use that pointer to access a 32-bit signed integer value. Similarly, if you have a pointer to some record, you would not normally cast that pointer to some other record type and access the fields of that other type⁵. With pointers to class objects, however, we can lift this restriction a bit. Pointers to objects may legally contain the address of the object's type *or the address of any object that inherits the fields of that type*. Consider the following declarations that use the *point* and *point3D* types from the previous examples:

```
var
    P2: point;
    P3: point3D;
    p: pointer to point;
    .
    .
    .
    lea( ebx, P2 );
    mov( ebx, p );
```

4. Polymorphism literally means "many-faced." In the context of object-oriented programming polymorphism means that the same method name, e.g., *distance*, can refer to one of several different methods.

5. Of course, assembly language programmers break rules like this all the time. For now, let's assume we're playing by the rules and only access the data using the data type associated with the pointer.

```

p.distance();           // Calls the point.distance method.
.
.
.
lea( ebx, P3 );
mov( ebx, p );          // Yes, this is semantically legal.
p.distance();           // Surprise, this calls point3D.distance.

```

Since *p* is a pointer to a *point* object, it might seem intuitive for *p.distance* to call the *point.distance* method. However, methods are *polymorphic*. If you've got a pointer to an object and you call a method associated with that object, the system will call the actual (overridden) method associated with the object, not the method specifically associated with the pointer's class type.

Class procedures behave differently than methods with respect to overridden procedures. When you call a class procedure indirectly through an object pointer, the system will always call the procedure associated with the underlying class associated with the pointer. So had *distance* been a procedure rather than a method in the previous examples, the "*p.distance()*;" invocation would always call *point.distance*, even if *p* is pointing at a *point3D* object. The section on Object Initialization, later in this chapter, explains why methods and procedures are different (see "Object Implementation" on page 1038).

Note that iterators are also virtual; so like methods an object iterator invocation will always call the (overridden) iterator associated with the actual object whose address the pointer contains. To differentiate the semantics of methods and iterators from procedures, we will refer to the method/iterator calling semantics as *virtual procedures* and the calling semantics of a class procedure as a *static procedure*.

10.8 Writing Class Methods, Iterators, and Procedures

For each class procedure, method, and iterator prototype appearing in a class definition, there must be a corresponding procedure, method, or iterator appearing within the program (for the sake of brevity, this section will use the term *routine* to mean procedure, method, or iterator from this point forward). If the prototype does not contain the EXTERNAL option, then the code must appear in the same compilation unit as the class declaration. If the EXTERNAL option does follow the prototype, then the code may appear in the same compilation unit or a different compilation unit (as long as you link the resulting object file with the code containing the class declaration). Like external (non-class) procedures and iterators, if you fail to provide the code the linker will complain when you attempt to create an executable file. To reduce the size of the following examples, they will all define their routines in the same source file as the class declaration.

HLA class routines must always follow the class declaration in a compilation unit. If you are compiling your routines in a separate unit, the class declarations must still precede the code with the class declaration (usually via an #INCLUDE file). If you haven't defined the class by the time you define a routine like *point.distance*, HLA doesn't know that *point* is a class and, therefore, doesn't know how to handle the routine's definition.

Consider the following declarations for a *point2D* class:

```

type
point2D: class

    const
        UnitDistance: real32 := 1.0;

    var
        x: real32;
        y: real32;

    static
        LastDistance: real32;

    method distance( fromX: real32; fromY:real32 ); returns( "st0" );
    procedure InitLastDistance;

```

```
endclass;
```

The distance function for this class should compute the distance from the object's point to (fromX,fromY). The following formula describes this computation:

$$\sqrt{(x - \text{fromX})^2 + (y - \text{fromY})^2}$$

A first pass at writing the distance method might produce the following code:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( x );           // Note: this doesn't work!
    fld( fromX );       // Compute (x-fromX)
    fsub();
    fld( st0 );         // Duplicate value on TOS.
    fmul();             // Compute square of difference.

    fld( y );           // This doesn't work either.
    fld( fromY );       // Compute (y-fromY)
    fsub();
    fld( st0 );         // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;
```

This code probably looks like it should work to someone who is familiar with an object-oriented programming language like C++ or Delphi. However, as the comments indicate, the instructions that push the *x* and *y* variables onto the FPU stack don't work – HLA doesn't automatically define the symbols associated with the data fields of a class within that class' routines.

To learn how to access the data fields of a class within that class' routines, we need to back up a moment and discover some very important implementation details concerning HLA's classes. To do this, consider the following variable declarations:

```
var
    Origin: point2D;
    PtInSpace: point2D;
```

Remember, whenever you create two objects like *Origin* and *PtInSpace*, HLA reserves storage for the *x* and *y* data fields for both of these objects. However, there is only one copy of the *point2D.distance* method in memory. Therefore, were you to call *Origin.distance* and *PtInSpace.distance*, the system would call the same routine for both method invocations. Once inside that method, one has to wonder what an instruction like “fld(*x*);” would do. How does it associate *x* with *Origin.x* or *PtInSpace.x*? Worse still, how would this code differentiate between the data field *x* and a global object *x*? In HLA, the answer is “it doesn't.” You do not specify the data field names within a class routine by simply using their names as though they were common variables.

To differentiate *Origin.x* from *PtInSpace.x* within class routines, HLA automatically passes a pointer to an object's data fields whenever you call a class routine. Therefore, you can reference the data fields indirectly off this pointer. HLA passes this object pointer in the ESI register. This is one of the few places where HLA-generated code will modify one of the 80x86 registers behind your back: **anytime you call a class routine, HLA automatically loads the ESI register with the object's address.** Obviously, you cannot count on ESI's value being preserved across class routine class nor can you pass parameters to the class routine in the ESI register. For class methods and iterators (but not procedures), HLA will also load the EDI register with the address of the class' *virtual method table* (see “Virtual Method Tables” on page 1041).

While the virtual method table address isn't as interesting as the object address, keep in mind that **HLA-generated code will overwrite any value in the EDI register when you call a method or an iterator.**

Upon entry into a class routine, ESI contains a pointer to the (non-static) data fields associated with the class. Therefore, to access fields like *x* and *y* (in our *point2D* example), you could use an address expression like the following:

```
(type point2D [esi]).x
```

Since you use ESI as the base address of the object's data fields, it's a good idea not to disturb ESI's value within the class routines (or, at least, preserve ESI's value if you need to access the objects data fields after some point where you must use ESI for some other purpose). Note that if you call an iterator or a method you do not have to preserve EDI (unless, for some reason, you need access to the virtual method table, which is unlikely).

Accessing the fields of a data object within a class' routines is such a common operation that HLA provides a shorthand notation for casting ESI as a pointer to the class object: **THIS**. Within a class in HLA, the reserved word **THIS** automatically expands to a string of the form "(type *classname* [esi])" substituting, of course, the appropriate class name for *classname*. Using the **THIS** keyword, we can (correctly) rewrite the previous distance method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;
```

```
    fld( this.x );
    fld( fromX );      // Compute (x-fromX)
    fsub();
    fld( st0 );        // Duplicate value on TOS.
    fmul();            // Compute square of difference.

    fld( this.y );
    fld( fromY );      // Compute (y-fromY)
    fsub();
    fld( st0 );        // Compute the square of the difference.
    fmul();

    fsqrt();
```

```
end distance;
```

Don't forget that calling a class routine wipes out the value in the ESI register. This isn't obvious from the syntax of the routine's invocation. It is especially easy to forget this when calling some class routine from inside some other class routine; don't forget that if you do this the internal call wipes out the value in ESI and on return from that call ESI no longer points at the original object. Always push and pop ESI (or otherwise preserve ESI's value) in this situation, e.g.,

```
.
.
.
fld( this.x );      // ESI points at current object.
.
.
.
push( esi );        // Preserve ESI across this method call.
SomeObject.SomeMethod();
pop( esi );
.
.
.
lea( ebx, this.x );      // ESI points at original object here.
```

The `THIS` keyword provides access to the class variables you declare in the `VAR` section of a class. You can also use `THIS` to call other class routines associated with the current object, e.g.,

```
this.distance( 5.0, 6.0 );
```

To access class constants and `STATIC` data fields you generally do not use the `THIS` pointer. HLA associates constant and static data fields with the whole class, not a specific object. To access these class members, just use the class name in place of the object name. For example, to access the *UnitDistance* constant in the *point2D* class you could use a statement like the following:

```
fld( point2D.UnitDistance );
```

As another example, if you wanted to update the *LastDistance* field in the *point2D* class each time you computed a distance, you could rewrite the *point2D.distance* method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );      // Compute (x-fromX)
    fsub();
    fld( st0 );        // Duplicate value on TOS.
    fmul();            // Compute square of difference.

    fld( this.y );
    fld( fromY );      // Compute (y-fromY)
    fsub();
    fld( st0 );        // Compute the square of the difference.
    fmul();

    fsqrt();

    fst( point2D.LastDistance );    // Update shared (STATIC) field.

end distance;
```

To understand why you use the class name when referring to constants and static objects but you use `THIS` to access `VAR` objects, check out the next section.

Class procedures are also static objects, so it is possible to call a class procedure by specifying the class name rather than an object name in the procedure invocation, e.g., both of the following are legal:

```
Origin.InitLastDistance();
point2D.InitLastDistance();
```

There is, however, a subtle difference between these two class procedure calls. The first call above loads `ESI` with the address of the *Origin* object prior to actually calling the *InitLastDistance* procedure. The second call, however, is a direct call to the class procedure without referencing an object; therefore, HLA doesn't know what object address to load into the `ESI` register. In this case, HLA loads `NULL` (zero) into `ESI` prior to calling the *InitLastDistance* procedure. Because you can call class procedures in this manner, it's always a good idea to check the value in `ESI` within your class procedures to verify that HLA contains an object address. Checking the value in `ESI` is a good way to determine which calling mechanism is in use. Later, this chapter will discuss constructors and object initialization; there you will see a good use for static procedures and calling those procedures directly (rather than through the use of an object).

10.9 Object Implementation

In a high level object-oriented language like C++ or Delphi, it is quite possible to master the use of objects without really understanding how the machine implements them. One of the reasons for learning assembly language programming is to fully comprehend low-level implementation details so one can make

educated decisions concerning the use of programming constructs like objects. Further, since assembly language allows you to poke around with data structures at a very low-level, knowing how HLA implements objects can help you create certain algorithms that would not be possible without a detailed knowledge of object implementation. Therefore, this section, and its corresponding subsections, explains the low-level implementation details you will need to know in order to write object-oriented HLA programs.

HLA implements objects in a manner quite similar to records. In particular, HLA allocates storage for all VAR objects in a class in a sequential fashion, just like records. Indeed, if a class consists of only VAR data fields, the memory representation of that class is nearly identical to that of a corresponding RECORD declaration. Consider the Student record declaration taken from Volume Three and the corresponding class:

```
type
  student:  record
            Name: char[65];
            Major: int16;
            SSN:  char[12];
            Midterm1: int16;
            Midterm2: int16;
            Final: int16;
            Homework: int16;
            Projects: int16;
          endrecord;

  student2: class
            Name: char[65];
            Major: int16;
            SSN:  char[12];
            Midterm1: int16;
            Midterm2: int16;
            Final: int16;
            Homework: int16;
            Projects: int16;
          endclass;
```

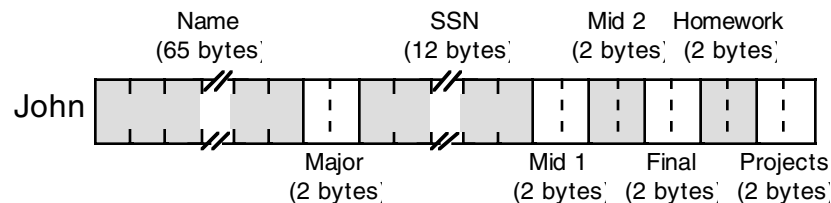


Figure 10.1 Student RECORD Implementation in Memory

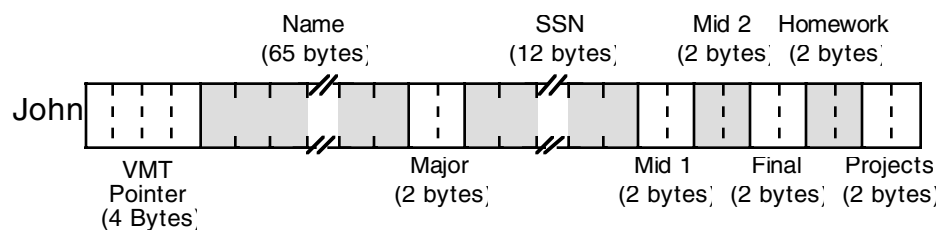


Figure 10.2 Student CLASS Implementation in Memory

If you look carefully at these two figures, you'll discover that the only difference between the class and the record implementations is the inclusion of the VMT (virtual method table) pointer field at the beginning of the class object. This field, which is always present, contains the address of the class' virtual method table which, in turn, contains the addresses of all the class' methods and iterators. The VMT field, by the way, is present even if a class doesn't contain any methods or iterators.

As pointed out in previous sections, HLA does not allocate storage for STATIC objects within the object's storage. Instead, HLA allocates a single instance of each static data field that all objects share. As an example, consider the following class and object declarations:

```
type
  tHasStatic: class

    var
      i:int32;
      j:int32;
      r:real32;

    static
      c:char[2];
      b:byte;

  endclass;

var
  hs1: tHasStatic;
  hs2: tHasStatic;
```

Figure 10.3 shows the storage allocation for these two objects in memory.

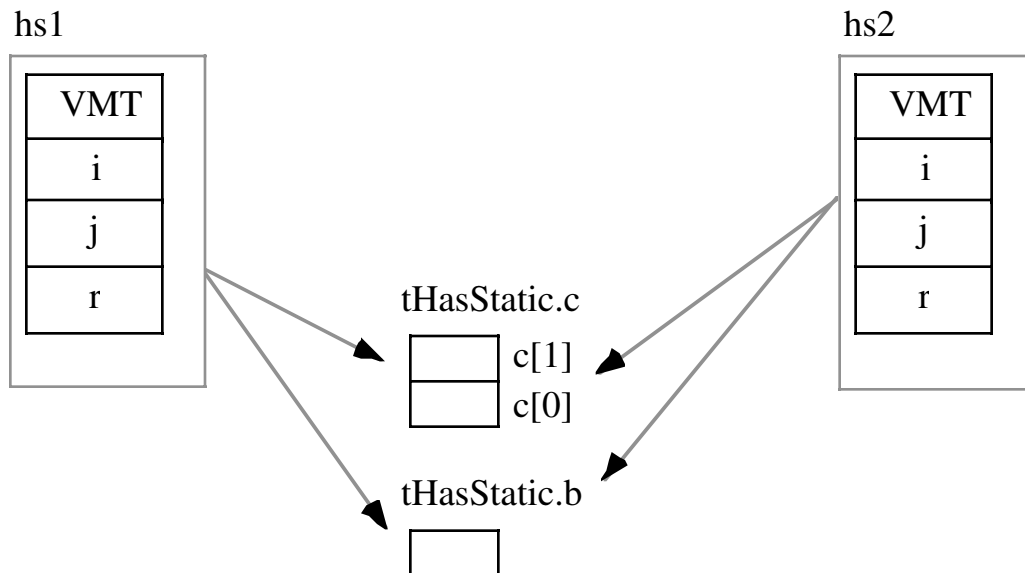


Figure 10.3 Object Allocation with Static Data Fields

Of course, CONST, VAL, and MACRO objects do not have any run-time memory requirements associated with them, so HLA does not allocate any storage for these fields. Like the STATIC data fields, you may access CONST, VAL, and MACRO fields using the class name as well as an object name. Hence, even if

tHasStatic has these types of fields, the memory organization for *tHasStatic* objects would still be the same as shown in Figure 10.3.

Other than the presence of the virtual method table pointer (VMT), the presence of methods, iterators, and procedures has no impact on the storage allocation of an object. Of course, the machine instructions associated with these routines does appear somewhere in memory. So in a sense the code for the routines is quite similar to static data fields insofar as all the objects share a single instance of the routine.

10.9.1 Virtual Method Tables

When HLA calls a class procedure, it directly calls that procedure using a CALL instruction, just like any normal non-class procedure call. Methods and iterators are another story altogether. Each object in the system carries a pointer to a virtual method table which is an array of pointers to all the methods and iterators appearing within the object's class.

SomeObject

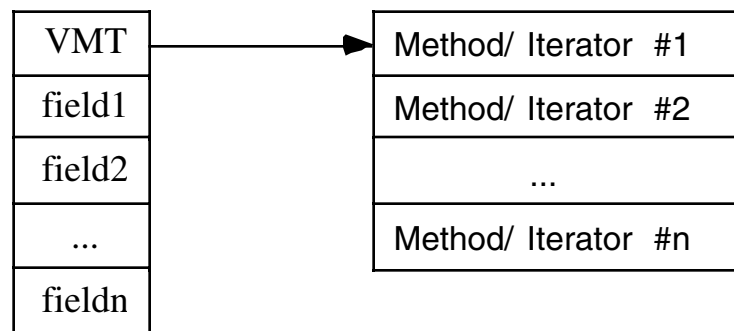


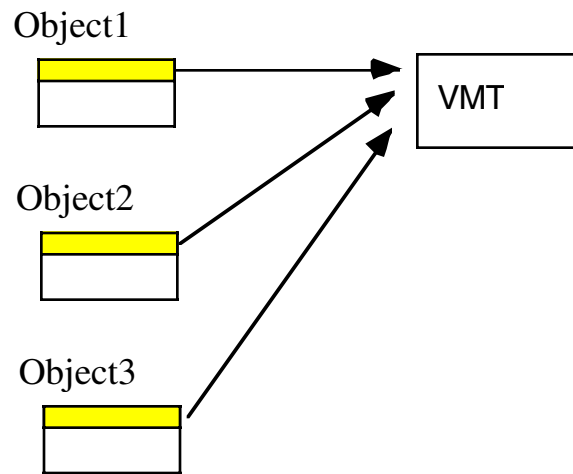
Figure 10.4 Virtual Method Table Organization

Each iterator or method you declare in a class has a corresponding entry in the virtual method table. That *dword* entry contains the address of the first instruction of that iterator or method. To call a class method or iterator is a bit more work than calling a class procedure (it requires one additional instruction plus the use of the EDI register). Here is a typical calling sequence for a method:

```

mov( ObjectAdrs, ESI );      // All class routines do this.
mov( [esi], edi );          // Get the address of the VMT into EDI
call( (type dword [edi+n])); // "n" is the offset of the method's entry
                             // in the VMT.
  
```

For a given class there is only one copy of the VMT in memory. This is a static object so all objects of a given class type share the same VMT. This is reasonable since all objects of the same class type have exactly the same methods and iterators (see Figure 10.5).



Note: Objects are all the same class type

Figure 10.5 All Objects That are the Same Class Type Share the Same VMT

Although HLA builds the VMT record structure as it encounters methods and iterators within a class, HLA does not automatically create the actual run-time virtual method table for you. You must explicitly declare this table in your program. To do this, you include a statement like the following in a STATIC or READONLY declaration section of your program, e.g.,

```
readonly
    VMT( classname );
```

Since the addresses in a virtual method table should never change during program execution, the READONLY section is probably the best choice for declaring VMTs. It should go without saying that changing the pointers in a VMT is, in general, a really bad idea. So putting VMTs in a STATIC section is usually not a good idea.

A declaration like the one above defines the variable *classname._VMT_*. In section 10.10 (see “Constructors and Object Initialization” on page 1046) you see that you’ll need this name when initializing object variables. The class declaration automatically defines the *classname._VMT_* symbol as an external static variable. The declaration above just provides the actual definition of this external symbol.

The declaration of a VMT uses a somewhat strange syntax because you aren’t actually declaring a new symbol with this declaration, you’re simply supplying the data for a symbol that you previously declared implicitly by defining a class. That is, the class declaration defines the static table variable *classname._VMT_*, all you’re doing with the VMT declaration is telling HLA to emit the actual data for the table. If, for some reason, you would like to refer to this table using a name other than *classname._VMT_*, HLA does allow you to prefix the declaration above with a variable name, e.g.,

```
readonly
    myVMT: VMT( classname );
```

In this declaration, *myVMT* is an alias of *classname._VMT_*. As a general rule, you should avoid aliases in a program because they make the program more difficult to read and understand. Therefore, it is unlikely that you would ever really need to use this type of declaration.

Like any other global static variable, there should be only one instance of a VMT for a given class in a program. The best place to put the VMT declaration is in the same source file as the class’ method, iterator, and procedure code (assuming they all appear in a single file). This way you will automatically link in the VMT whenever you link in the routines for a given class.

10.9.2 Object Representation with Inheritance

Up to this point, the discussion of the implementation of class objects has ignored the possibility of inheritance. Inheritance only affects the memory representation of an object by adding fields that are not explicitly stated in the class declaration.

Adding inherited fields from a *base class* to another class must be done carefully. Remember, an important attribute of a class that inherits fields from a base class is that you can use a pointer to the base class to access the inherited fields from that base class in another class. As an example, consider the following classes:

```
type
  tBaseClass: class
    var
      i:uns32;
      j:uns32;
      r:real32;

    method mBase;
  endclass;

  tChildClassA: class inherits( tBaseClass )
    var
      c:char;
      b:boolean;
      w:word;

    method mA;
  endclass;

  tChildClassB: class inherits( tBaseClass )
    var
      d:dword;
      c:char;
      a:byte[3];
  endclass;
```

Since both *tChildClassA* and *tChildClassB* inherit the fields of *tBaseClass*, these two child classes include the *i*, *j*, and *r* fields as well as their own specific fields. Furthermore, whenever you have a pointer variable whose base type is *tBaseClass*, it is legal to load this pointer with the address of any child class of *tBaseClass*; therefore, it is perfectly reasonable to load such a pointer with the address of a *tChildClassA* or *tChildClassB* variable, e.g.,

```
var
  B1: tBaseClass;
  CA: tChildClassA;
  CB: tChildClassB;
  ptr: pointer to tBaseClass;
  .
  .
  .
  lea( ebx, B1 );
  mov( ebx, ptr );
  << Use ptr >>
  .
  .
  .
  lea( eax, CA );
  mov( ebx, ptr );
  << Use ptr >>
```

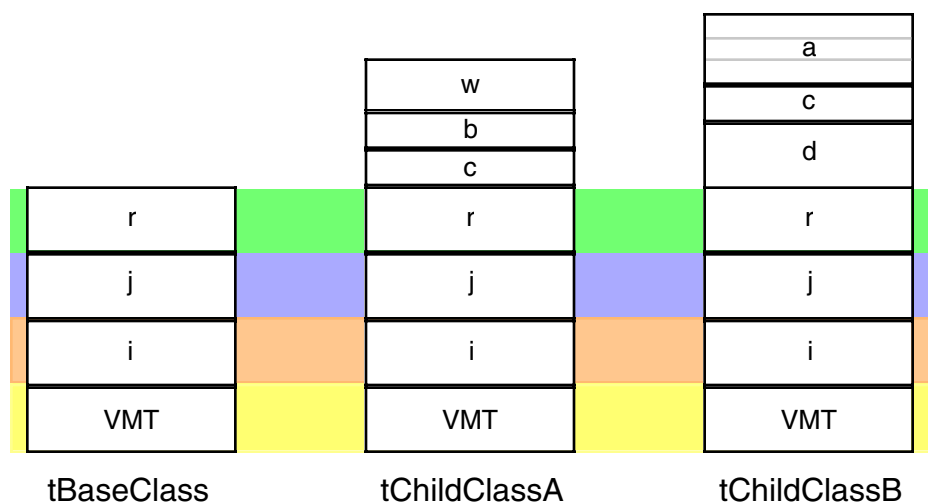
```

.
.
.
lea( eax, CB );
mov( eax, ptr );
<< Use ptr >>

```

Since *ptr* points at an object of *tBaseClass*, you may legally (from a semantic sense) access the *i*, *j*, and *r* fields of the object where *ptr* is pointing. It is not legal to access the *c*, *b*, *w*, or *d* fields of the *tChildClassA* or *tChildClassB* objects since at any one given moment the program may not know exactly what object type *ptr* references.

In order for inheritance to work properly, the *i*, *j*, and *r* fields must appear at the same offsets all child classes as they do in *tBaseClass*. This way, an instruction of the form “mov((type tBaseClass [ebx]).i, eax);” will correct access the *i* field even if EBX points at an object of type *tChildClassA* or *tChildClassB*. Figure 10.6 shows the layout of the child and base classes:



Derived (child) classes locate their inherited fields at the same offsets as those fields in the base class.

Figure 10.6 Layout of Base and Child Class Objects in Memory

Note that the new fields in the two child classes bear no relation to one another, even if they have the same name (e.g., field *c* in the two child classes does not lie at the same offset). Although the two child classes share the fields they inherit from their common base class, any new fields they add are unique and separate. Two fields in different classes share the same offset only by coincidence.

All classes (even those that aren't related to one another) place the pointer to the virtual method table at offset zero within the object. There is a single VMT associated with each class in a program; even classes that inherit fields from some base class have a VMT that is (generally) different than the base class' VMT. shows how objects of type *tBaseClass*, *tChildClassA* and *tChildClassB* point at their specific VMTs:

```

var
    B1: tBaseClass;
    CA: tChildClassA;
    CB: tChildClassB;
    CB2: tChildClassB;
    CA2: tChildClassA;

```

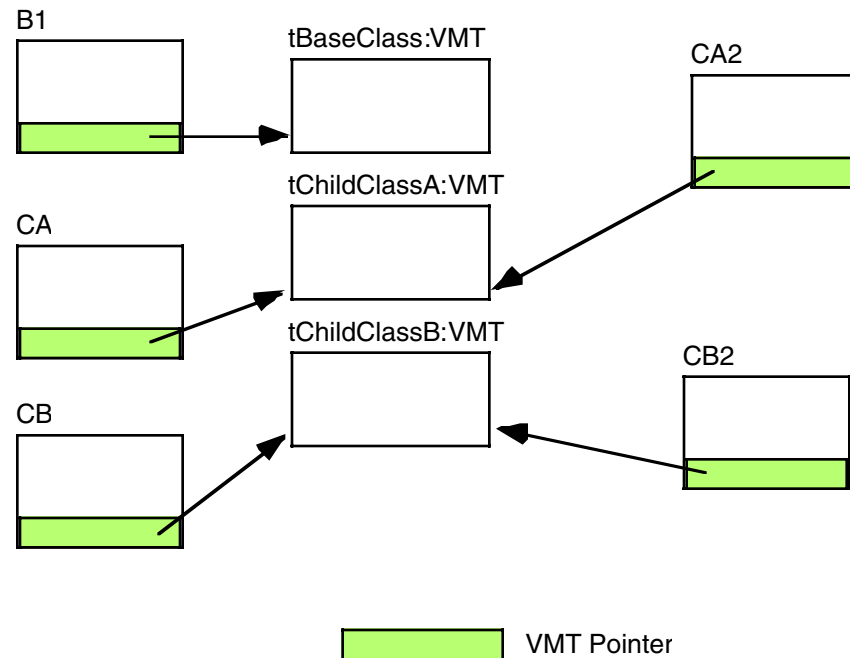


Figure 10.7 Virtual Method Table References from Objects

A virtual method table is nothing more than an array of pointers to the methods and iterators associated with a class. The address of the first method or iterator appearing in a class is at offset zero, the address of the second appears at offset four, etc. You can determine the offset value for a given iterator or method by using the @offset function. If you want to call a method or iterator directly (using 80x86 syntax rather than HLA's high level syntax), you code use code like the following:

```

var
    sc: tBaseClass;
    .
    .
    .
    lea( esi, sc );           // Get the address of the object (& VMT).
    mov( [esi], edi );       // Put address of VMT into EDI.
    call( (type dword [edi+@offset( tBaseClass.mBase )] ) );

```

Of course, if the method has any parameters, you must push them onto the stack before executing the code above. Don't forget, when making direct calls to a method, that you must load ESI with the address of the object. Any field references within the method will probably depend upon ESI containing this address. The choice of EDI to contain the VMT address is nearly arbitrary. Unless you're doing something tricky (like using EDI to obtain run-time type information), you could use any register you please here. As a general rule, you should use EDI when simulating class iterator/method calls because this is the convention that HLA employs and most programmers will expect this.

Whenever a child class inherits fields from some base class, the child class' VMT also inherits entries from the base class' VMT. For example, the VMT for class *tBaseClass* contains only a single entry – a pointer to method *tBaseClass.mBase*. The VMT for class *tChildClassA* contains two entries: a pointer to *tBaseClass.mBase* and *tChildClassA.mA*. Since *tChildClassB* doesn't define any new methods or iterators, *tChildClassB*'s VMT contains only a single entry, a pointer to the *tBaseClass.mBase* method. Note that *tChildClassB*'s VMT is identical to *tBaseClass*' VMT. Nevertheless, HLA produces two distinct VMTs. This is a critical fact that we will make use of a little later. Figure 10.8 shows the relationship between these VMTs:

Virtual Method Tables for Derived (inherited) Classes

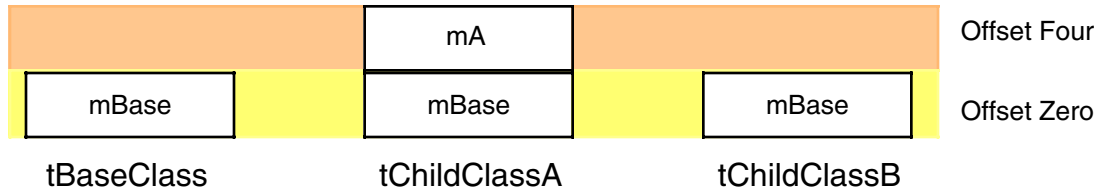


Figure 10.8 Virtual Method Tables for Inherited Classes

Although the VMT always appears at offset zero in an object (and, therefore, you can access the VMT using the address expression “[ESI]” if ESI points at an object), HLA actually inserts a symbol into the symbol table so you may refer to the VMT symbolically. The symbol *_pVMT_* (pointer to Virtual Method Table) provides this capability. So a more readable way to access the VMT pointer (as in the previous code example) is

```
lea( esi, sc );
mov( (type tBaseClass [esi])._pVMT_, edi );
call( (type dword [edi+offset( tBaseClass.mBase )] ) );
```

If you need to access the VMT directly, there are a couple ways to do this. Whenever you declare a class object, HLA automatically includes a field named *_VMT_* as part of that class. *_VMT_* is a static array of double word objects. Therefore, you may refer to the VMT using an identifier of the form *class-name._VMT_*. Generally, you shouldn't access the VMT directly, but as you'll see shortly, there are some good reasons why you need to know the address of this object in memory.

10.10 Constructors and Object Initialization

If you've tried to get a little ahead of the game and write a program that uses objects prior to this point, you've probably discovered that the program inexplicably crashes whenever you attempt to run it. We've covered a lot of material in this chapter thus far, but you are still missing one crucial piece of information – how to properly initialize objects prior to use. This section will put the final piece into the puzzle and allow you to begin writing programs that use classes.

Consider the following object declaration and code fragment:

```
var
    bc: tBaseClass;
    .
    .
    .
    bc.mBase();
```

Remember that variables you declare in the VAR section are uninitialized at run-time. Therefore, when the program containing these statements gets around to executing *bc.mBase*, it executes the three-statement sequence you've seen several times already:

```
lea( esi, sbc);
mov( [esi], edi );
call( (type dword [edi+@offset( tBaseClass.mBase )] ) );
```

The problem with this sequence is that it loads EDI with an undefined value assuming you haven't previously initialized the *bc* object. Since EDI contains a garbage value, attempting to call a subroutine at address "[EDI+@offset(tBaseClass.mBase)]" will likely crash the system. Therefore, before using an object, you must initialize the *_pVMT_* field with the address of that object's VMT. One easy way to do this is with the following statement:

```
mov( &tBaseClass._VMT_, bc._pVMT_ );
```

Always remember, **before using an object, be sure to initialize the virtual method table pointer for that field.**

Although you must initialize the virtual method table pointer for all objects you use, this may not be the only field you need to initialize in those objects. Each specific class may have its own application-specific initialization that is necessary. Although the initialization may vary by class, you need to perform the same initialization on each object of a specific class that you use. If you ever create more than a single object from a given class, it is probably a good idea to create a procedure to do this initialization for you. This is such a common operation that object-oriented programmers have given these initialization procedures a special name: *constructors*.

Some object-oriented languages (e.g., C++) use a special syntax to declare a constructor. Others (e.g., Delphi) simply use existing procedure declarations to define a constructor. One advantage to employing a special syntax is that the language knows when you define a constructor and can automatically generate code to call that constructor for you (whenever you declare an object). Languages, like Delphi, require that you explicitly call the constructor; this can be a minor inconvenience and a source of defects in your programs. HLA does not use a special syntax to declare constructors – you define constructors using standard class procedures. As such, you will need to explicitly call the constructors in your program; however, you'll see an easy method for automating this in a later section of this chapter.

Perhaps the most important fact you must remember is that **constructors must be class procedures**. You must not define constructors as methods (or iterators). The reason is quite simple: one of the tasks of the constructor is to initialize the pointer to the virtual method table and you cannot call a class method or iterator until after you've initialized the VMT pointer. Since class procedures don't use the virtual method table, you can call a class procedure prior to initializing the VMT pointer for an object.

By convention, HLA programmers use the name *Create* for the class constructor. There is no requirement that you use this name, but by doing so you will make your programs easier to read and follow by other programmers.

As you may recall, you can call a class procedure via an object reference or a class reference. E.g., if *clsProc* is a class procedure of class *tClass* and *Obj* is an object of type *tClass*, then the following two class procedure invocations are both legal:

```
tClass.clsProc();
Obj.clsProc();
```

There is a big difference between these two calls. The first one calls *clsProc* with ESI containing zero (NULL) while the second invocation loads the address of *Obj* into ESI before the call. We can use this fact to determine within a method the particular calling mechanism.

10.10.1 Dynamic Object Allocation Within the Constructor

As it turns out, most programs allocated object dynamically using *malloc* and refer to those objects indirectly using pointers. This adds one more step to the initialization process – allocating storage for the object. The constructor is the perfect place to allocate this storage. Since you probably won't need to allocate all objects dynamically, you'll need two types of constructors: one that allocates storage and then initializes the object, and another that simply initializes an object that already has storage.

Another constructor convention is to merge these two constructors into a single constructor and differentiate the type of constructor call by the value in ESI. On entry into the class' *Create* procedure, the program checks the value in ESI to see if it contains NULL (zero). If so, the constructor calls *malloc* to allocate storage for the object and returns a pointer to the object in ESI. If ESI does not contain NULL upon entry into the procedure, then the constructor assumes that ESI points at a valid object and skips over the memory allocation statements. At the very least, a constructor initializes the pointer to the VMT; therefore, the minimalist constructor will look like the following:

```
procedure tBaseClass.mBase; nodisplay;
begin mBase;

    push( EAX );    // Malloc returns its result here, so save it.
    if( ESI = 0 ) then

        malloc( @size( tBaseClass ) );
        mov( eax, esi ); // Put pointer into ESI;

    endif;

    // Initialize the pointer to the VMT:
    // (remember, "this" is shorthand for (type tBaseClass [esi])"

    mov( &tBaseClass._VMT_, this._pVMT_ );

    // Other class initialization would go here.

end mBase;
```

After you write a constructor like the one above, you choose an appropriate calling mechanism based on whether your object's storage is already allocated. For pre-allocated objects (i.e., those you've declared in VAR, STATIC, STORAGE, or DATA sections⁶ or those you've previously allocated storage for via *malloc*) you simply load the address of the object into ESI and call the constructor. For those objects you declare as a variable, this is very easy – just call the appropriate *Create* constructor:

```
var
    bc0: tBaseClass;
    bcp: pointer to tBaseClass;
    .
    .
    .
    bc0.Create(); // Initializes pre-allocated bc0 object.
    .
    .
    .
    malloc( @size( tBaseClass ) ); // Allocate storage for bcp object.
    mov( eax, bcp );
    .
    .
    .
    bcp.Create(); // Initializes pre-allocated bcp object.
```

Note that although *bcp* is a pointer to a *tBaseClass* object, the *Create* method does not automatically allocate storage for this object. The program already allocates the storage earlier. Therefore, when the program calls *bcp.Create* it loads ESI with the address contained within *bcp*; since this is not NULL, the *tBaseClass.Create* procedure does not allocate storage for a new object. By the way, the call to *bcp.Create* emits the following sequence of machine instructions:

```
mov( bcp, esi );
call tBaseClass.Create;
```

6. You generally do not declare objects in READONLY sections because you cannot initialize them.

Until now, the code examples for a class procedure call always began with an LEA instruction. This is because all the examples to this point have used object variables rather than pointers to object variables. Remember, a class procedure (method/iterator) call passes the address of the object in the ESI register. For object variables HLA emits an LEA instruction to obtain this address. For pointers to objects, however, the actual object address is the *value* of the pointer variable; therefore, to load the address of the object into ESI, HLA emits a MOV instruction that copies the value of the pointer into the ESI register.

In the example above, the program preallocates the storage for an object prior to calling the object constructor. While there are several reasons for preallocating object storage (e.g., you're creating a dynamic array of objects), you can achieve most simple object allocations like the one above by calling a standard *Create* method (i.e., one that allocates storage for an object if ESI contains NULL). The following example demonstrates this:

```
var
    bcp2: pointer to tBaseClass;
    .
    .
    .
    tBaseClass.Create();    // Calls Create with ESI=NULL.
    mov( esi, bcp2 );      // Save pointer to new class object in bcp2.
```

Remember, a call to a *tBaseClass.Create* constructor returns a pointer to the new object in the ESI register. It is the caller's responsibility to save the pointer this function returns into the appropriate pointer variable; the constructor does not automatically do this for you.

10.10.2 Constructors and Inheritance

Constructors for derived (child) classes that inherit fields from a base class represent a special case. Each class must have its own constructor but needs the ability to call the base class constructor. This section explains the reasons for this and how to do this.

A derived class inherits the *Create* procedure from its base class. However, you must override this procedure in a derived class because the derived class probably requires more storage than the base class and, therefore, you will probably need to use a different call to *malloc* to allocate storage for a dynamic object. Hence, it is very unusual for a derived class not to override the definition of the *Create* procedure.

However, overriding a base class' *Create* procedure has problems of its own. When you override the base class' *Create* procedure, you take the full responsibility of initializing the (entire) object, including all the initialization required by the base class. At the very least, this involves putting duplicate code in the overridden procedure to handle the initialization usually done by the base class constructor. In addition to make your program larger (by duplicating code already present in the base class constructor), this also violates information hiding principles since the derived class must be aware of all the fields in the base class (including those that are logically private to the base class). What we need here is the ability to call a base class' constructor from within the derived class' destructor and let that call do the lower-level initialization of the base class' fields. Fortunately, this is an easy thing to do in HLA.

Consider the following class declarations (which does things the hard way):

```
type
    tBase: class
        var
            i:uns32;
            j:int32;

        procedure Create(); returns( "esi" );
    endclass;

    tDerived: class inherits( tBase )
        var†
            r: real64;
```

```

        override procedure Create(); returns( "esi" );
    endclass;

    procedure tBase.Create; nodisplay;
    begin Create;

        if( esi = 0 ) then

            push( eax );
            mov( malloc( @size( tBase ) ), esi );
            pop( eax );

        endif;
        mov( &tBase._VMT_, this._pVMT_ );
        mov( 0, this.i );
        mov( -1, this.j );

    end Create;

    procedure tDerived.Create; nodisplay;
    begin Create;

        if( esi = 0 ) then

            push( eax );
            mov( malloc( @size( tDerived ) ), esi );
            pop( eax );

        endif;

        // Initialize the VMT pointer for this object:

        mov( &tDerived._VMT_, this._pVMT_ );

        // Initialize the "r" field of this particular object:

        fldz();
        fstp( this.r );

        // Duplicate the initialization required by tBase.Create:

        mov( 0, this.i );
        mov( -1, this.j );

    end Create;

```

Let's take a closer look at the *tDerived.Create* procedure above. Like a conventional constructor, it begins by checking ESI and allocates storage for a new object if ESI contains NULL. Note that the size of a *tDerived* object includes the size required by the inherited fields, so this properly allocates the necessary storage for all fields in a *tDerived* object.

Next, the *tDerived.Create* procedure initializes the VMT pointer field of the object. Remember, each class has its own VMT and, specifically, derived classes do not use the VMT of their base class. Therefore, this constructor must initialize the *_pVMT_* field with the address of the *tDerived* VMT.

After initializing the VMT pointer, the *tDerived* constructor initializes the value of the *r* field to 0.0 (remember, FLDZ loads zero onto the FPU stack). This concludes the *tDerived*-specific initialization.

The remaining instructions in *tDerived.Create* are the problem. These statements duplicate some of the code appearing in the *tBase.Create* procedure. The problem with code duplication becomes really apparent when you decide to modify the initial values of these fields; if you've duplicated the initialization code in

derived classes, you will need to change the initialization code in more than one *Create* procedure. More often than not, this results in defects in the derived class *Create* procedures, especially if those derived classes appear in different source files than the base class.

Another problem with burying base class initialization in derived class constructors is the violation of the information hiding principle. Some fields of the base class may be *logically private*. Although HLA does not explicitly support the concept of public and private fields in a class (as, say, C++ does), well-disciplined programmers will still partition the fields as private or public and then only use the private fields in class routines belonging to that class. Initializing these private fields in derived classes is not acceptable to such programmers. Doing so will make it very difficult to change the definition and implementation of some base class at a later date.

Fortunately, HLA provides an easy mechanism for calling the inherited constructor within a derived class' constructor. All you have to do is call the base constructor using the classname syntax, e.g., you could call *tBase.Create* directly from within *tDerived.Create*. By calling the base class constructor, your derived class constructors can initialize the base class fields without worrying about the exact implementation (or initial values) of the base class.

Unfortunately, there are two types of initialization that every (conventional) constructor does that will affect the way you call a base class constructor: all conventional constructors allocate memory for the class if ESI contains zero and all conventional constructors initialize the VMT pointer. Fortunately, it is very easy to deal with these two problems

The memory required by an object of some most base class is usually less than the memory required for an object of a class you derive from that base class (because the derived classes usually add more fields). Therefore, you cannot allow the base class constructor to allocate the storage when you call it from inside the derived class' constructor. This problem is easily solved by checking ESI within the derived class constructor and allocating any necessary storage for the object *before* calling the base class constructor.

The second problem is the initialization of the VMT pointer. When you call the base class' constructor, it will initialize the VMT pointer with the address of the base class' virtual method table. A derived class object's *_pVMT_* field, however, must point at the virtual method table for the derived class. Calling the base class constructor will always initialize the *_pVMT_* field with the wrong pointer; to properly initialize the *_pVMT_* field with the appropriate value, the derived class constructor must store the address of the derived class' virtual method table into the *_pVMT_* field after the call to the base class constructor (so that it overwrites the value written by the base class constructor).

The *tDerived.Create* constructor, rewritten to call the *tBase.Create* constructors, follows:

```
procedure tDerived.Create; nodisplay;
begin Create;

    if( esi = 0 ) then

        push( eax );
        mov( malloc( @size( tDerived ) ), esi );
        pop( eax );

    endif;

    // Call the base class constructor to do any initialization
    // needed by the base class. Note that this call must follow
    // the object allocation code above (so ESI will always contain
    // a pointer to an object at this point and tBase.Create will
    // never allocate storage).

    tBase.Create();

    // Initialize the VMT pointer for this object. This code
    // must always follow the call to the base class constructor
    // because the base class constructor also initializes this
    // field and we don't want the initial value supplied by
    // tBase.Create.
```

```

mov( &tDerived._VMT_, this._pVMT_ );

// Initialize the "r" field of this particular object:

fldz();
fstp( this.r );

end Create;

```

This solution solves all the above concerns with derived class constructors.

10.10.3 Constructor Parameters and Procedure Overloading

All the constructor examples to this point have not had any parameters. However, there is nothing special about constructors that prevent the use of parameters. Constructors are procedures therefore you can specify any number and types of parameters you choose. You can use these parameter values to initialize certain fields or control how the constructor initializes the fields. Of course, you may use constructor parameters for any purpose you'd use parameters in any other procedure. In fact, about the only issue you need concern yourself with is the use of parameters whenever you have a derived class. This section deals with those issues.

The first, and probably most important, problem with parameters in derived class constructors actually applies to all overridden procedures, iterators, and methods: the parameter list of an overridden routine must exactly match the parameter list of the corresponding routine in the base class. In fact, HLA doesn't even give you the chance to violate this rule because **OVERRIDE** routine prototypes don't allow parameter list declarations – they automatically inherit the parameter list of the base routine. Therefore, you cannot use a special parameter list in the constructor prototype for one class and a different parameter list for the constructors appearing in base or derived classes. Sometimes it would be nice if this weren't the case, but there are some sound and logical reasons why HLA does not support this⁷.

Some languages, like C++, support function overloading letting you specify several different constructors whose parameter list specifies which constructor to use. HLA does not directly support procedure overloading in this manner, but you can use macros to simulate this language feature (see “Simulating Function Overloading with Macros” on page 960). To use this trick with constructors you would create a macro with the name *Create*. The actual constructors could have names that describe their differences (e.g., *CreateDefault*, *CreateSetIJ*, etc.). The *Create* macro would parse the actual parameter list to determine which routine to call.

HLA does not support macro overloading. Therefore, you cannot override a macro in a derived class to call a constructor unique to that derived class. In certain circumstances you can create a small workaround by defining empty procedures in your base class that you intend to override in some derived class (this is similar to an abstract method, see “Abstract Methods” on page 1058). Presumably, you would never call the procedure in the base class (in fact, you would probably want to put an error message in the body of the procedure just in case you accidentally call it). By putting the empty procedure declaration in the base class, the macro that simulates function overloading can refer to that procedure and you can use that in derived classes later on.

7. Calling virtual methods and iterators would be a real problem since you don't really know which routine a pointer references. Therefore, you couldn't know the proper parameter list. While the problems with procedures aren't quite as drastic, there are some subtle problems that could creep into your code if base or derived classes allowed overridden procedures with different parameter lists.

10.11 Destructors

A destructor is a class routine that cleans up an object once a program finishes using that object. Like constructors, HLA does not provide a special syntax for creating destructors nor does HLA automatically call a destructor; unlike constructors, a destructor is usually a method rather than a procedure (since virtual destructors make a lot of sense while virtual constructors do not).

A typical destructor will close any files opened by the object, free the memory allocated during the use of the object, and, finally, free the object itself if it was created dynamically. The destructor also handles any other clean-up chores the object may require before it ceases to exist.

By convention, most HLA programmers name their destructors *Destroy*. Destructors generally do not have any parameters, so the issue of overloading the parameter list rarely arises. About the only code that most destructors have in common is the code to free the storage associated with the object. The following destructor demonstrates how to do this:

```
procedure tBase.Destroy; nodisplay;
begin Destroy;

    push( eax );    // isInHeap uses this

    // Place any other clean up code here.
    // The code to free dynamic objects should always appear last
    // in the destructor.

    /*****/

    // The following code assumes that ESI still contains the address
    // of the object.

    if( isInHeap( esi )) then

        free( esi );

    endif;
    pop( eax );

end Destroy;
```

The HLA Standard Library routine *isInHeap* returns true if its parameter is an address that *malloc* returned. Therefore, this code automatically frees the storage associated with the object if the program originally allocated storage for the object by calling *malloc*. Obviously, on return from this method call, ESI will no longer point at a legal object in memory if you allocated it dynamically. Note that this code will not affect the value in ESI nor will it modify the object if the object wasn't one you've previously allocated via a call to *malloc*.

10.12 HLA's “_initialize_” and “_finalize_” Strings

Although HLA does not automatically call constructors and destructors associated with your classes, HLA does provide a mechanism whereby you can cause these calls to happen automatically: by using the *_initialize_* and *_finalize_* compile-time string variables (i.e., VAL constants) HLA automatically declares in every procedure.

Whenever you write a procedure, iterator, or method, HLA automatically declares several local symbols in that routine. Two such symbols are *_initialize_* and *_finalize_*. HLA declares these symbols as follows:

```
val
    _initialize_: string := "";
    _finalize_: string := "";
```

HLA emits the `_initialize_` string as text at the very beginning of the routine's body, i.e., immediately after the routine's BEGIN clause⁸. Similarly, HLA emits the `_finalize_` string at the very end of the routine's body, just before the END clause. This is comparable to the following:

```
procedure SomeProc;
  << declarations >>
begin SomeProc;

  @text( _initialize_ );

  << procedure body >>

  @text( _finalize_ );

end SomeProc;
```

Since `_initialize_` and `_finalize_` initially contain the empty string, these expansions have no effect on the code that HLA generates unless you explicitly modify the value of `_initialize_` prior to the BEGIN clause or you modify `_finalize_` prior to the END clause of the procedure. So if you modify either of these string objects to contain a machine instruction, HLA will compile that instruction at the beginning or end of the procedure. The following example demonstrates how to use this technique:

```
procedure SomeProc;
  ?_initialize_ := "mov( 0, eax );";
  ?_finalize_   := "stdout.put( eax );"
begin SomeProc;

  // HLA emits "mov( 0, eax );" here in response to the _initialize_
  // string constant.

  add( 5, eax );

  // HLA emits "stdout.put( eax );" here.

end SomeProc;
```

Of course, these examples don't save you much. It would be easier to type the actual statements at the beginning and end of the procedure than assign a string containing these statements to the `_initialize_` and `_finalize_` compile-time variables. However, if we could automate the assignment of some string to these variables, so that you don't have to explicitly assign them in each procedure, then this feature might be useful. In a moment, you'll see how we can automate the assignment of values to the `_initialize_` and `_finalize_` strings. For the time being, consider the case where we load the name of a constructor into the `_initialize_` string and we load the name of a destructor in to the `_finalize_` string. By doing this, the routine will "automatically" call the constructor and destructor for that particular object.

The example above has a minor problem. If we can automate the assignment of some value to `_initialize_` or `_finalize_`, what happens if these variables already contain some value? For example, suppose we have two objects we use in a routine and the first one loads the name of its constructor into the `_initialize_` string; what happens when the second object attempts to do the same thing? The solution is simple: don't directly assign any string to the `_initialize_` or `_finalize_` compile-time variables, instead, always concatenate your strings to the end of the existing string in these variables. The following is a modification to the above example that demonstrates how to do this:

```
procedure SomeProc;
  ?_initialize_ := _initialize_ + "mov( 0, eax );";
  ?_finalize_   := _finalize_ + "stdout.put( eax );"
begin SomeProc;
```

8. If the routine automatically emits code to construct the activation record, HLA emits `_initialize_`'s text after the code that builds the activation record.

```

// HLA emits "mov( 0, eax );" here in response to the _initialize_
// string constant.

add( 5, eax );

// HLA emits "stdout.put( eax );" here.

end SomeProc;

```

When you assign values to the `_initialize_` and `_finalize_` strings, HLA almost guarantees that the `_initialize_` sequence will execute upon entry into the routine. Sadly, the same is not true for the `_finalize_` string upon exit. HLA simply emits the code for the `_finalize_` string at the end of the routine, immediately before the code that cleans up the activation record and returns. Unfortunately, “falling off the end of the routine” is not the only way that one could return from that routine. One could explicitly return from somewhere in the middle of the code by executing a RET instruction. Since HLA only emits the `_finalize_` string at the very end of the routine, returning from that routine in this manner bypassing the `_finalize_` code. Unfortunately, other than manually emitting the `_finalize_` code, there is nothing you can do about this⁹. Fortunately, this mechanism for exiting a routine is completely under your control; if you never exit a routine except by “falling off the end” then you won’t have to worry about this problem (note that you can use the EXIT control structure to transfer control to the end of a routine if you really want to return from that routine from somewhere in the middle of the code).

Another way to prematurely exit a routine which, unfortunately, you have no control over, is by raising an exception. Your routine could call some other routine (e.g., a standard library routine) that raises an exception and then transfers control immediately to whomever called your routine. Fortunately, you can easily trap and handle exceptions by putting a TRY..ENDTRY block in your procedure. Here is an example that demonstrates this:

```

procedure SomeProc;
  << declarations that modify _initialize_ and _finalize_ >>
begin SomeProc;

  << HLA emits the code for the _initialize_ string here. >>

  try    // Catch any exceptions that occur:

    << Procedure Body Goes Here >>

    anyexception

      push( eax );    // Save the exception #.
      @text( _finalize_ ); // Execute the _finalize_ code here.
      pop( eax );     // Restore the exception #.
      raise( eax );   // Reraise the exception.

    endtry;

    // HLA automatically emits the _finalize_ code here.

end SomeProc;

```

Although the code above handles some problems that exist with `_finalize_`, by no means that this handle every possible case. Always be on the look out for ways your program could inadvertently exit a routine without executing the code found in the `_finalize_` string. You should explicitly expand `_finalize_` if you encounter such a situation.

9. Note that you can manually emit the `_finalize_` code using the statement “@text(_finalize_);”.

There is one important place you can get into trouble with respect to exceptions: within the code the routine emits for the `_initialize_` string. If you modify the `_initialize_` string so that it contains a constructor call and the execution of that constructor raises an exception, this will probably force an exit from that routine without executing the corresponding `_finalize_` code. You could bury the `TRY..ENDTRY` statement directly into the `_initialize_` and `_finalize_` strings but this approach has several problems, not the least of which is the fact that one of the first constructors you call might raise an exception that transfers control to the exception handler that calls the destructors for all objects in that routine (including those objects whose constructors you have yet to call). Although no single solution that handles all problems exists, probably the best approach is to put a `TRY..ENDTRY` block around each constructor call if it is possible for that constructor to raise some exception that is possible to handle (i.e., doesn't require the immediate termination of the program).

Thus far this discussion of `_initialize_` and `_finalize_` has failed to address one important point: why use this feature to implement the “automatic” calling of constructors and destructors since it apparently involves more work than simply calling the constructors and destructors directly? Clearly there must be a way to automate the assignment of the `_initialize_` and `_finalize_` strings or this section wouldn't exist. The way to accomplish this is by using a macro to define the class type. So now it's time to take a look at another HLA feature that makes it possible to automate this activity: the `FORWARD` keyword.

You've seen how to use the `FORWARD` reserved word to create procedure and iterator prototypes (see “Forward Procedures” on page 546), it turns out that you can declare `FORWARD CONST`, `VAL`, `TYPE`, and variable declarations as well. The syntax for such declarations takes the following form:

```
ForwardSymbolName: forward( undefinedID );
```

This declaration is completely equivalent to the following:

```
?undefinedID: text := "ForwardSymbolName";
```

Especially note that this expansion does not actually define the symbol *ForwardSymbolName*. It just converts this symbol to a string and assigns this string to the specified `TEXT` object (*undefinedID* in this example).

Now you're probably wonder how something like the above is equivalent to a forward declaration. The truth is, it isn't. However, `FORWARD` declarations let you create macros that simulate type names by allowing you to defer the actual declaration of an object's type until some later point in the code. Consider the following example:

```
type
  myClass: class
    var
      i:int32;

    procedure Create; returns( "esi" );
    procedure Destroy;
  endclass;

macro _myClass: varID;
  forward( varID );
  ?_initialize_ := _initialize_ + @string:varID + ".Create(); ";
  ?_finalize_   := _finalize_ + @string:varID + ".Destroy(); ";
  varID: myClass
endmacro;
```

Note, and this is very important, that a semicolon does not follow the “`varID: myClass`” declaration at the end of this macro. You'll find out why this semicolon is missing in a little bit.

If you have the class and macro declarations above in your program, you can now declare variables of type *myClass* that automatically invoke the constructor and destructor upon entry and exit of the routine containing the variable declarations. To see how, take a look at the following procedure shell:

```
procedure HasmyClassObject;
var
```

```

    mco: _myClass;
begin HasmyClassObject;

    << do stuff with mco here >>

end HasmyClassObject;

```

Since `_myClass` is a macro, the procedure above expands to the following text during compilation:

```

procedure HasmyClassObject;
var
    mco:           // Expansion of the _myClass macro:
        forward( _0103_ ); // _0103_ symbol is and HLA supplied text symbol
                           // that expands to "mco".

    ?_initialize_ := _initialize_ + "mco" + ".Create(); ";
    ?_finalize_   := _finalize_ + "mco" + ".Destroy(); ";
    mco: myClass;

begin HasmyClassObject;

    mco.Create(); // Expansion of the _initialize_ string.

    << do stuff with mco here >>

    mco.Destroy(); // Expansion of the _finalize_ string.

end HasmyClassObject;

```

You might notice that a semicolon appears after “`mco: myClass`” declaration in the example above. This semicolon is not actually a part of the macro, instead it is the semicolon that follows the “`mco: _myClass;`” declaration in the original code.

If you want to create an array of objects, you could legally declare that array as follows:

```

var
    mcoArray: _myClass[10];

```

Because the last statement in the `_myClass` macro doesn’t end with a semicolon, the declaration above will expand to the following (almost correct) code:

```

mcoArray:           // Expansion of the _myClass macro:
    forward( _0103_ ); // _0103_ symbol is and HLA supplied text symbol
                      // that expands to "mcoArray".

    ?_initialize_ := _initialize_ + "mcoArray" + ".Create(); ";
    ?_finalize_   := _finalize_ + "mcoArray" + ".Destroy(); ";
    mcoArray: myClass[10];

```

The only problem with this expansion is that it only calls the constructor for the first object of the array. There are several ways to solve this problem; one is to append a macro name to the end of `_initialize_` and `_finalize_` rather than the constructor name. That macro would check the object’s name (*mcoArray* in this example) to determine if it is an array. If so, that macro could expand to a loop that calls the constructor for each element of the array (the implementation appears as a programming project at the end of this chapter).

Another solution to this problem is to use a macro parameter to specify the dimensions for arrays of *myClass*. This scheme is easier to implement than the one above, but it does have the drawback of requiring a different syntax for declaring object arrays (you have to use parentheses around the array dimension rather than square brackets).

The `FORWARD` directive is quite powerful and lets you achieve all kinds of tricks. However, there are a few problems of which you should be aware. First, since HLA emits the `_initialize_` and `_finalize_` code transparently, you can be easily confused if there are any errors in the code appearing within these strings. If

you start getting error messages associated with the BEGIN or END statements in a routine, you might want to take a look at the `_initialize_` and `_finalize_` strings within that routine. The best defense here is to always append very simple statements to these strings so that you reduce the likelihood of an error.

Fundamentally, HLA doesn't support automatic constructor and destructor calls. This section has presented several tricks to attempt to automate the calls to these routines. However, the automation isn't perfect and, indeed, the aforementioned problems with the `_finalize_` strings limit the applicability of this approach. The mechanism this section presents is probably fine for simple classes and simple programs. However, one piece of advice is probably worth following: if your code is complex or correctness is critical, it's probably a good idea to explicitly call the constructors and destructors manually.

10.13 Abstract Methods

An *abstract base class* is one that exists solely to supply a set of common fields to its derived classes. You never declare variables whose type is an abstract base class, you always use one of the derived classes. The purpose of an abstract base class is to provide a template for creating other classes, nothing more. As it turns out, the only difference in syntax between a standard base class and an abstract base class is the presence of at least one *abstract method* declaration. An abstract method is a special method that does not have an actual implementation in the abstract base class. Any attempt to call that method will raise an exception. If you're wondering what possible good an abstract method could be, well, keep on reading...

Suppose you want to create a set of classes to hold numeric values. One class could represent unsigned integers, another class could represent signed integers, a third could implement BCD values, and a fourth could support *real64* values. While you could create four separate classes that function independently of one another, doing so passes up an opportunity to make this set of classes more convenient to use. To understand why, consider the following possible class declarations:

```
type
  uint: class
    var
      TheValue: dword;

    method put;
    << other methods for this class >>
  endclass;

  sint: class
    var
      TheValue: dword;

    method put;
    << other methods for this class >>
  endclass;

  r64: class
    var
      TheValue: real64;

    method put;
    << other methods for this class >>
  endclass;
```

The implementation of these classes is not unreasonable. They have fields for the data, they have a *put* method (which, presumably, writes the data to the standard output device). Presumably they have other methods and procedures to implement various operations on the data. There is, however, two problems with these classes, one minor and one major, both occurring because these classes do not inherit any fields from a common base class.

The first problem, which is relatively minor, is that you have to repeat the declaration of several common fields in these classes. For example, the *put* method declaration appears in each of these classes¹⁰. This duplication of effort involves results in a harder to maintain program because it doesn't encourage you to use a common name for a common function since it's easy to use a different name in each of the classes.

A bigger problem with this approach is that it is not generic. That is, you can't create a generic pointer to a "numeric" object and perform operations like addition, subtraction, and output on that value (regardless of the underlying numeric representation).

We can easily solve these two problems by turning the previous class declarations into a set of derived classes. The following code demonstrates an easy way to do this:

```
type
  numeric: class
    procedure put;
    << Other common methods shared by all the classes >>
  endclass;

  uint: class inherits( numeric )
    var
      TheValue: dword;

    override method put;
    << other methods for this class >>
  endclass;

  sint: class inherits( numeric )
    var
      TheValue: dword;

    override method put;
    << other methods for this class >>
  endclass;

  r64: class inherits( numeric )
    var
      TheValue: real64;

    override method put;
    << other methods for this class >>
  endclass;
```

This scheme solves both the problems. First, by inheriting the *put* method from *numeric*, this code encourages the derived classes to always use the name *put* thereby making the program easier to maintain. Second, because this example uses derived classes, it's possible to create a pointer to the *numeric* type and load this pointer with the address of a *uint*, *sint*, or *r64* object. That pointer can invoke the methods found in the *numeric* class to do functions like addition, subtraction, or numeric output. Therefore, the application that uses this pointer doesn't need to know the exact data type, it only deals with numeric values in a generic fashion.

One problem with this scheme is that it's possible to declare and use variables of type *numeric*. Unfortunately, such numeric variables don't have the ability to represent any type of number (notice that the data storage for the numeric fields actually appears in the derived classes). Worse, because you've declared the *put* method in the *numeric* class, you've actually got to write some code to implement that method even though one should never really call it; the actual implementation should only occur in the derived classes. While you could write a dummy method that prints an error message (or, better yet, raises an exception), there shouldn't be any need to write "dummy" procedures like this. Fortunately, there *is* no reason to do so – if you use *abstract* methods.

10. Note, by the way, that *TheValue* is not a common class because this field has a different type in the *r64* class.

The `ABSTRACT` keyword, when it follows a method declaration, tells HLA that you are not going to provide an implementation of the method for this class. Instead, it is the responsibility of all derived class to provide a concrete implementation for the abstract method. HLA will raise an exception if you attempt to call an abstract method directly. The following is the modification to the *numeric* class to convert *put* to an abstract method:

```
type
  numeric: class
    procedure put; abstract;
    << Other common methods shared by all the classes >>
  endclass;
```

An abstract base class is a class that has at least one abstract method. Note that you don't have to make all methods abstract in an abstract base class; it is perfectly legal to declare some standard methods (and, of course, provide their implementation) within the abstract base class.

Abstract method declarations provide a mechanism by which a base class enforces the methods that the derived classes must implement. In theory, all derived classes must provide concrete implementations of all abstract methods or those derived classes are themselves abstract base classes. In practice, it's possible to bend the rules a little and use abstract methods for a slightly different purpose.

A little earlier, you read that one should never create variables whose type is an abstract base class. For if you attempt to execute an abstract method the program would immediately raise an exception to complain about this illegal method call. In practice, you actually can declare variables of an abstract base type and get away with this as long as you don't call any abstract methods. We can use this fact to provide a better form of method overloading (that is, providing several different routines with the same name but different parameter lists). Remember, the standard trick in HLA to overload a routine is to write several different routines and then use a macro to parse the parameter list and determine which actual routine to call (see "Simulating Function Overloading with Macros" on page 960). The problem with this technique is that you cannot override a macro definition in a class, so if you want to use a macro to override a routine's syntax, then that macro must appear in the base class. Unfortunately, you may not need a routine with a specific parameter list in the base class (for that matter, you may only need that particular version of the routine in a single derived class), so implementing that routine in the base class and in all the other derived classes is a waste of effort. This isn't a big problem. Just go ahead and define the abstract method in the base class and only implement it in the derived class that needs that particular method. As long as you don't call that method in the base class or in the other derived classes that don't override the method, everything will work fine.

One problem with using abstract methods to support overloading is that this trick does not apply to procedures - only methods and iterators. However, you can achieve the same effect with procedures by declaring a (non-abstract) procedure in the base class and overriding that procedure only in the class that actually uses it. You will have to provide an implementation of the procedure in the base class, but that is a minor issue (the procedure's body, by the way, should simply raise an exception to indicate that you should have never called it).

An example of routine overloading in a class appears in this chapter's sample program.

10.14 Run-time Type Information (RTTI)

When working with an object variable (as opposed to a pointer to an object), the type of that object is obvious: it's the variable's declared type. Therefore, at both compile-time and run-time the program trivially knows the type of the object. When working with pointers to objects you cannot, in the general case, determine the type of an object a pointer references. However, at run-time it is possible to determine the object's actual type. This section discusses how to detect the underlying object's type and how to use this information.

If you have a pointer to an object and that pointer's type is some base class, at run-time the pointer could point at an object of the base class or any derived type. At compile-time it is not possible to determine the exact type of an object at any instant. To see why, consider the following short example:

```
ReturnSomeObject();           // Returns a pointer to some class in ESI.
mov( esi, ptrToObject );
```

The routine *ReturnSomeObject* returns a pointer to an object in ESI. This could be the address of some base class object or a derived class object. At compile-time there is no way for the program to know what type of object this function returns. For example, *ReturnSomeObject* could ask the user what value to return so the exact type could not be determined until the program actually runs and the user makes a selection.

In a perfectly designed program, there probably is no need to know a generic object's actual type. After all, the whole purpose of object-oriented programming and inheritance is to produce general programs that work with lots of different objects without having to make substantial changes to the program. In the real world, however, programs may not have a perfect design and sometimes it's nice to know the exact object type a pointer references. Run-time type information, or RTTI, gives you the capability of determining an object's type at run-time, even if you are referencing that object using a pointer to some base class of that object.

Perhaps the most fundamental RTTI operation you need is the ability to ask if a pointer contains the address of some specific object type. Many object-oriented languages (e.g., Delphi) provide an *IS* operator that provides this functionality. *IS* is a boolean operator that returns true if its left operand (a pointer) points at an object whose type matches the left operand (which must be a type identifier). The typical syntax is generally the following:

```
ObjectPointerOrVar is ClassType
```

This operator would return true if the variable is of the specified class, it returns false otherwise. Here is a typical use of this operator (in the Delphi language)

```
if( ptrToNumeric is uint ) then begin
    .
    .
    .
end;
```

It's actually quite simple to implement this functionality in HLA. As you may recall, each class is given its own virtual method table. Whenever you create an object, you must initialize the pointer to the VMT with the address of that class' VMT. Therefore, the VMT pointer field of all objects of a given class type contain the same pointer value and this pointer value is different than the VMT pointer field of all other classes. We can use this fact to see if an object is some specific type. The following code demonstrates how to implement the Delphi statement above in HLA:

```
mov( ptrToNumeric, esi );
if( (type uint [esi])._pVMT_ = &uint._VMT_ ) then
    .
    .
    .
endif;
```

This IF statement simply compares the object's *_pVMT_* field (the pointer to the VMT) against the address of the desired class' VMT. If they are equal, then the *ptrToNumeric* variable points at an object of type *uint*.

Within the body of a class method or iterator, there is a slightly easier way to see if the object is a certain class. Remember, upon entry into a method or an iterator, the EDI register contains the address of the virtual method table. Therefore, assuming you haven't modified EDI's value, you can easily test to see if THIS (ESI) is a specific class type using an IF statement like the following:

```
if( EDI = &uint._VMT_ ) then
    .
    .
    .
endif;
```

10.15 Calling Base Class Methods

In the section on constructors you saw that it is possible to call an ancestor class' procedure within the derived class' overridden procedure. To do this, all you needed to do was to invoke the procedure using the call "classname.procedureName(parameters);". On occasion you may want to do this same operation with a class' methods as well as its procedures (that is, have an overridden method call the corresponding base class method in order to do some computation you'd rather not repeat in the derived class' method). Unfortunately, HLA does not let you directly call methods as it does procedures. You will need to use an indirect mechanism to achieve this; specifically, you will have to call the function using the address in the base class' virtual method table. This section describes how to do this.

Whenever your program calls a method it does so indirectly, using the address found in the virtual method table for the method's class. The virtual method table is nothing more than an array of 32-bit pointers with each entry containing the address of one of that class' methods. So to call a method, all you need is the index into this array (or, more properly, the offset into the array) of the address of the method you wish to call. The HLA compile-time function *@offset* comes to the rescue- it will return the offset into the virtual method table of the method whose name you supply as a parameter. Combined with the CALL instruction, you can easily call any method associated with a class. Here's an example of how you would do this:

```
type
  myCls: class
      .
      .
      .
      method m;
      .
      .
      .
  endclass;

  .
  .
  .
  call( myCls._VMT_[ @offset( myCls.m ) ] );
```

The CALL instruction above calls the method whose address appears at the specified entry in the virtual method table for *myCls*. The *@offset* function call returns the offset (i.e., index times four) of the address of *myCls.m* within the virtual method table. Hence, this code indirectly calls the *m* method by using the virtual method table entry for *m*.

There is one major drawback to calling methods using this scheme: you don't get to use the high level syntax for procedure/method calls. Instead, you must use the low-level CALL instruction. In the example above, this isn't much of an issue because the *m* procedure doesn't have any parameters. If it did have parameters, you would have to manually push those parameters onto the stack yourself (see "Passing Parameters on the Stack" on page 796). Fortunately, you'll rarely need to call ancestor class methods from a derived class, so this won't be much of an issue in real-world programs.

10.16 Sample Program

This chapter's sample program will present what is probably the epitome of object-oriented programs: a simple "drawing" program that uses objects to represent shapes to draw on the display. While limited to a demonstration program, this program does demonstrate important object-oriented concepts in assembly language.

This is an unusual drawing program insofar as it draws shapes using ASCII characters. While the shapes it draws are very rough (compared to a graphics-based drawing program), the output of this program could be quite useful for creating rudimentary diagrams to include as comments in your HLA (or other lan-

guage) programs. This sample program does not provide a “user interface” for drawing images (something you would need to effectively use this program) because the user interface represents a lot of code that won’t improve your appreciation of object-oriented programming (not to mention, this book is long enough already). Providing a mouse-based user interface to this program is left as an exercise to the interested reader.

This program consists of three source files: the class definitions in a header file, the implementation of the class’ procedures and methods in an HLA source file, and a main program that demonstrates a simple use of the class’ objects. The following listings are for these three files.

```
// Shape.hhf
// Class Definitions for the shape classes.

type

    // Generic shape class:

    shape: class

        const

            maxX: uns16 := 80;
            maxY: uns16 := 25;

        var

            x:            uns16;
            y:            uns16;
            width:        uns16;
            height:       uns16;
            fillShape:    boolean;

        procedure create; returns( "esi" ); external;

        method draw; abstract;
        method fill( f:boolean ); external;
        method moveTo( x:uns16; y:uns16 ); external;
        method resize( width: uns16; height: uns16 ); external;

    endclass;

    // Class for a rectangle shape
    //
    // +-----+
    // |       |
    // +-----+

    rect: class inherits( shape )

        override procedure create; external;
        override method draw; external;

    endclass;

    // Class for a rounded rectangle shape
    //
    // -----
    // /       \
```

```

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

roundrect: class inherits( shape )

    override procedure create; external;
    override method draw;  external;

endclass;

// Class for a diamond shape
//
//      /\
//     /\ 
//    /\ 
//   /\ 
//  /\ 

diamond: class inherits( shape )

    override procedure create; external;
    override method resize;  external;
    override method draw;    external;

endclass;

```

Program 10.1 Shapes.hhf - The Shape Class Header Files

```

// Shapes.hla-
//
// Implementation of the shape classes.

unit Shapes;
#includeonce( "stdlib.hhf" )
#includeonce( "shapes.hhf" )

// Emit the virtual method tables for the classes:

static
    vmt( shape );
    vmt( rect );
    vmt( roundrect );
    vmt( diamond );

/*****/

// Generic shape methods and procedures

// Constructor for the shape class.
//

```

```

// Note: this should really be an abstract procedure, but since
// HLA doesn't support abstract procedures we'll fake it by
// raising an exception if somebody tries to call this proc.

procedure shape.create; nodisplay; noframe;
begin create;

    // This should really be an abstract procedure,
    // but such things don't exist, so we will fake it.

    raise( ex.ExecutedAbstract );

end create;

// Generic shape.fill method.
// This is an accessor function that sets the "fill" field
// to the value of the parameter.

method shape.fill( f:boolean ); nodisplay;
begin fill;

    push( eax );
    mov( f, al );
    mov( al, this.fillShape );
    pop( eax );

end fill;

// Generic shape.moveTo method.
// Checks the coordinates passed as a parameter and
// then sets the (X,Y) coordinates of the underlying
// shape object to these values.

method shape.moveTo( x:uns16; y:uns16 ); nodisplay;
begin moveTo;

    push( eax );
    push( ebx );

    mov( x, ax );
    assert( ax < shape.maxX );
    mov( ax, this.x );

    mov( y, ax );
    assert( ax < shape.maxY );
    mov( ax, this.y );

    pop( ebx );
    pop( eax );

end moveTo;

// Generic shape.resize method.
// Sets the width and height fields of the underlying object
// to the values passed as parameters.
//
// Note: Ignores resize request if the size is less than 2x2.

```

```

method shape.resize( width:uns16; height:uns16 ); nodisplay;
begin resize;

    push( eax );
    assert( width <= shape.maxX );
    assert( height <= shape.maxY );

    if( width > 2 ) then

        if( height > 2 ) then

            mov( width, ax );
            mov( ax, this.width );

            mov( height, ax );
            mov( ax, this.height );

        endif;

    endif;
    pop( eax );

end resize;

/*****/
/*          */
/* rect's methods: */
/*          */
/*****/

// Constructor for the rectangle class:

procedure rect.create; nodisplay; noframe;
begin create;

    push( eax );

    // If called as rect.create, then allocate a new object
    // on the heap and return the pointer in ESI.

    if( esi = NULL ) then

        mov( malloc( @size( rect ) ), esi );

    endif;

    // Initialize the pointer to the VMT:

    mov( &rect._VMT_, this._pVMT_ );

    // Initialize fields to create a non-filled unit square.

    sub( eax, eax );

    mov( ax, this.x );
    mov( ax, this.y );
    inc( eax );
    mov( al, this.fillShape ); // Sets fillShape to true.

```

```

    inc( eax );
    mov( ax, this.height );
    mov( ax, this.width );

    pop( eax );
    ret();

end create;

// Here's the method to draw a text-based square on the display.

method rect.draw; nodisplay;
static
    horz: str.strvar( shape.maxX ); // Holds "+-----+..."
    spcs: str.strvar( shape.maxX ); // Holds "    " for fills.

begin draw;

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

    // Initialize the horz and spcs strings to speed up
    // drawing our rectangle.

    movzx( this.width, ebx );
    str.setstr( '-', horz, ebx );
    mov( horz, eax );
    mov( '+', (type char [eax]));
    mov( '+', (type char [eax+ebx-1]));

    // If the fillShape field contains true, then we
    // need to fill in the characters inside the rectangle.
    // If this is false, we don't want to overwrite the
    // text in the center of the rectangle. The following
    // code initializes spcs to all spaces or the empty string
    // to accomplish this.

    if( this.fillShape ) then

        sub( 2, ebx );
        str.setstr( ' ', spcs, ebx );

    else

        str.cpy( "", spcs );

    endif;

    // Okay, position the cursor and draw
    // our rectangle.

    console.gotoxy( this.y, this.x );
    stdout.puts( horz ); // Draws top horz line.

    // For each row except the top and bottom rows,
    // draw "|" characters on the left and right
    // hand sides and the fill characters (if fillShape

```

```

        // is true) inbetween them.

        mov( this.y, cx );
        mov( cx, bx );
        add( this.height, bx );
        inc( cx );
        dec( bx );
        while( cx < bx ) do

            console.gotoxy( cx, this.x );
            stdout.putc( '|' );
            stdout.puts( spcs );
            mov( this.x, dx );
            add( this.width, dx );
            dec( dx );
            console.gotoxy( cx, dx );
            stdout.putc( '|' );
            inc( cx );

        endwhile;

        // Draw the bottom horz bar:

        console.gotoxy( cx, this.x );
        stdout.puts( horz );

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

    end draw;

/*****/
/*          */
/* roundrect's methods: */
/*          */
/*****/

// This is the constructor for the roundrect class.
// See the comments in rect.create for details
// (since this is just a clone of that code with
// minor changes here and there).

procedure roundrect.create; nodisplay; noframe;
begin create;

    push( eax );
    if( esi = NULL ) then

        mov( malloc( @size( rect ) ), esi );

    endif;
    mov( &roundrect._VMT_, this._pVMT_ );

```

```

// Initialize fields to create a non-filled unit square.

sub( eax, eax );

mov( ax, this.x );
mov( ax, this.y );
inc( eax );
mov( al, this.fillShape ); // Sets fillShape to true.
inc( eax );
mov( ax, this.height );
mov( ax, this.width );

pop( eax );
ret();

```

```
end create;
```

```

// Here is the draw method for the roundrect object.
// Note: if the object is less than 5x4 in size,
// this code calls rect.draw to draw a rectangle
// since roundrects smaller than 5x4 don't look good.
//
// Typical roundrect:
//
//      -----
//      /         \
//      |           |
//      \         /
//      -----

```

```

method roundrect.draw; nodisplay;
static
    horz:      str.strvar( shape.maxX );
    spcs:      str.strvar( shape.maxX );

begin draw;

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

    if
    {
        cmp( this.width, 5 );
        jb true;
        cmp( this.height, 4 );
        jae false;
    }

    // If it's too small to draw an effective
    // roundrect, then draw it as a rectangle.

    call( rect._VMT_[ @offset( rect.draw ) ] );

else

    // Okay, it's big enough, draw it as a rounded

```

```

// rectangle object. Begin by initializing the
// horz string with a set of dashes with spaces
// at either end.

movzx( this.width, ebx );
sub( 4, ebx );
str.setstr( '-', horz, ebx );
if( this.fillShape ) then

    add( 2, ebx );
    str.setstr( ' ', spcs, ebx );

else

    str.cpy( "", spcs );

endif;

// Okay, draw the top line.

mov( this.x, ax );
add( 2, ax );
console.gotoxy( this.y, ax );
stdout.puts( horz );

// Now draw the second line and the
// as "/" and "\" with optional spaces
// inbetween (if fillShape is true).

mov( this.y, cx );
inc( cx );
console.gotoxy( cx, ax );
stdout.puts( spcs );

console.gotoxy( cx, this.x );
stdout.puts( " /" );

add( this.width, ax );
sub( 4, ax ); // Sub 4 because we added two above.
console.gotoxy( cx, ax );
stdout.puts( "\" " );

// Okay, now draw the bottom line:

mov( this.x, ax );
add( 2, ax );
mov( this.y, cx );
add( this.height, cx );
dec( cx );
console.gotoxy( cx, ax );
stdout.puts( horz );

// And draw the second from the bottom
// line as "\" and "/" with optional
// spaces inbetween (depending on fillShape)

dec( cx );
console.gotoxy( cx, this.x );
stdout.puts( spcs );

```

```

        console.gotoxy( cx, this.x );
        stdout.puts( " \" );

        mov( this.x, ax );
        add( this.width, ax );
        sub( 2, ax );           // Sub 4 because we added two above.
        console.gotoxy( cx, ax );
        stdout.puts( "/ \" );

        // Finally, draw all the lines inbetween the
        // top two and bottom two lines.

        mov( this.y, cx );
        mov( this.height, bx );
        add( cx, bx );
        add( 2, cx );
        sub( 2, bx );
        mov( this.x, ax );
        add( this.width, ax );
        dec( ax );

        while( cx < bx ) do

            console.gotoxy( cx, this.x );
            stdout.putc( '|' );
            stdout.puts( spcs );
            console.gotoxy( cx, ax );
            stdout.putc( '|' );
            inc( cx );

        endwhile;

    endif;

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

end draw;

/*****/
/*          */
/* Diamond's methods */
/*          */
/*****/

// Constructor for a diamond shape.
// See pertinent comments for the rect constructor
// for more details.

procedure diamond.create; nodisplay; noframe;
begin create;

    push( eax );
    if( esi = NULL ) then

        mov( malloc( @size( rect ) ), esi );

```

```

endif;
mov( &diamond._VMT_, this._pVMT_ );

// Initialize fields to create a 2x2 diamond.

sub( eax, eax );

mov( ax, this.x );
mov( ax, this.y );
inc( eax );
mov( al, this.fillShape ); // Sets fillShape to true.
inc( eax );               // Minimum diamond size is 2x2.
mov( ax, this.height );
mov( ax, this.width );

pop( eax );
ret();

end create;

// We have to overload the resize method for diamonds
// (unlike the other objects) because diamond shapes
// have to be symmetrical. That is, the width and
// the height have to be the same. This code enforces
// this restriction by setting both parameters to the
// minimum of the width/height parameters and then it
// calls shape.resize to do the dirty work.

method diamond.resize( width:uns16; height:uns16 ); nodisplay;
begin resize;

    // Diamonds are symmetrical shapes, so the width and
    // height must be the same. Force that here:

    push( eax );
    mov( width, ax );
    if( ax > height ) then

        mov( height, ax );

    endif;

    // Call the shape.resize method to do the actual work:

    push( eax ); // Pass the minimum value as the width.
    push( eax ); // Also pass the minimum value as the height.
    call( shape._VMT_[ @offset( shape.resize ) ] );

    pop( eax );

end resize;

// Here's the code to draw the diamond.

method diamond.draw; nodisplay;
var
    startY: uns16;
    endY: uns16;

```

```

startX: uns16;
endX:   uns16;

begin draw;

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

    if
    {
        cmp( this.width, 2 );
        jb true;
        cmp( this.height, 2 );
        jae false;
    }

    // Special cases for small diamonds.
    // Resizing prevents most of these from ever appearing.
    // However, if someone pokes around directly in the
    // width and height fields this code will save us:

        cmp( this.width, 1 );
        ja D2x1;
        cmp( this.height, 1 );
        ja D1x2;

        // At this point we must have a 1x1 diamond

        console.gotoxy( this.y, this.x );
        stdout.putc( '+' );
        jmp SmallDiamondDone;

D2x1:

    // Okay, we have a 2x1 (WxH) diamond here:

    console.gotoxy( this.y, this.x );
    stdout.puts( "<>" );
    jmp SmallDiamondDone;

D1x2:

    // We have a 1x2 (WxH) diamond here:

    mov( this.y, ax );
    console.gotoxy( ax, this.x );
    stdout.putc( '^' );
    inc( ax );
    console.gotoxy( ax, this.x );
    stdout.putc( 'V' );

SmallDiamondDone:

else

    // Okay, we're drawing a reasonable sized diamond.
    // There is still a minor problem. The best looking
    // diamonds always have a width and height that is an

```

```

// even integer. We need to do something special if
// the height or width is odd.
//
//      Odd      Odd
// Height      Width
//      .      <- That's a period
//  \ /      / \
// < >      \ /
//  \ /      '   <- That's an apostrophe
//
//      Both
//      .
//      / \
//      < >
//      \ /
//      '
//
// Step one: determine if we have an odd width.  If so,
// output the period and quote at the appropriate points.

```

```

mov( this.width, ax );
mov( this.y, cx );
test( 1, al );
if( @nz ) then

    shr( 1, ax );
    add( this.x, ax );
    console.gotoxy( cx, ax );
    stdout.putc( '.' );
    inc( cx );
    mov( cx, startY );

    add( this.height, cx );
    sub( 2, cx );
    console.gotoxy( cx, ax );
    stdout.putc( ''' );
    dec( cx );
    mov( cx, endY );

else

    mov( this.y, ax );
    mov( ax, startY );
    add( this.height, ax );
    dec( ax );
    mov( ax, endY );

endif;

```

```

// Step two: determine if we have an odd height.  If so,
// output the less than and greater than symbols at the
// appropriate spots (in the center of the diamond).

```

```

mov( this.height, ax );
mov( this.x, cx );
test( 1, al );
if( @nz ) then

    shr( 1, ax );
    add( this.y, ax );
    console.gotoxy( ax, cx );

```

```

        stdout.putc( '<' );
        inc( cx );
        mov( cx, startX );

        // Write spaces across the center if fillShape is true.

        if( this.fillShape ) then

            lea( ebx, [ecx+1] );
            mov( this.x, dx );
            add( this.width, dx );
            dec( dx );
            while( bx < dx ) do

                stdout.putc( ' ' );
                inc( bx );

            endwhile;

        endif;

        add( this.width, cx );
        sub( 2, cx );
        console.gotoxy( ax, cx );
        stdout.putc( '>' );
        dec( cx );
        mov( cx, endX );

    else

        mov( this.x, ax );
        mov( ax, startX );
        add( this.width, ax );
        dec( ax );
        mov( ax, endX );

    endif;

    // Step three: fill in the sides of the diamond
    //
    //      /\      '
    //     / \    O  / \   (or something inbetween these two).
    //      \ /    R  <  >
    //       \/\    \ /
    //          '
    // We've already drawn the points if there was an odd height
    // or width, now we've just got to fill in the sides with
    // "/" and "\" characters.
    //
    // Compute the middle two (or three) lines beginning with
    // the "/" (decY) and "\" (incY) symbols:
    //
    // decY = ( ( startY + endY - 1 ) and $FFFE )/2
    // incY = ( startY + endY )/2 + 1

    mov( startY, ax );
    add( endY, ax );
    mov( ax, bx );
    dec( ax );
    and( $FFFE, ax ); // Force value to be even.
    shr( 1, ax );

```

```

shr( 1, bx );
inc( bx );

// Fill in pairs of rows as long as we don't hit the bottom/top
// of the diamond:

while( (type int16 ax) >= (type int16 startY) ) do

    // Draw the sides on the upper half of the diamond:

    mov( startX, cx );
    mov( endX, dx );
    console.gotoxy( ax, cx );
    stdout.putc( '/' );
    if( this.fillShape ) then

        inc( cx );
        while( cx < dx ) do

            stdout.putc( ' ' );
            inc( cx );

        endwhile;

    endif;
    console.gotoxy( ax, dx );
    stdout.putc( '\\' );

    // Draw the sides on the lower half of the diamond:

    mov( startX, cx );
    mov( endX, dx );
    console.gotoxy( bx, cx );
    stdout.putc( '\\' );
    if( this.fillShape ) then

        inc( cx );
        while( cx < dx ) do

            stdout.putc( ' ' );
            inc( cx );

        endwhile;

    endif;
    console.gotoxy( bx, dx );
    stdout.putc( '/' );

    inc( bx );
    dec( ax );
    inc( startX );
    dec( endX );

endwhile;

endif;

pop( edx );

```

```

    pop( ecx );
    pop( ebx );
    pop( eax );

```

```

end draw;

```

```

end Shapes;

```

Program 10.2 Shapes.hla - The Implementation of the Shape Class

```

// This is a simple demonstration program
// that shows how to use the shape objects
// in the shape, rect, roundrect, and diamond classes.

program ASCIIDraw;
#include( "stdlib.hhf" )
#includeonce( "shapes.hhf" )

type
    pShape: pointer to shape;

// Allocate storage for various shapes:

static
    aRect1: pointer to rect;
    aRect2: pointer to rect;
    aRect3: pointer to rect;

    aRrect1: pointer to roundrect;
    aRrect2: pointer to roundrect;
    aRrect3: pointer to roundrect;

    aDiamond1: pointer to diamond;
    aDiamond2: pointer to diamond;
    aDiamond3: pointer to diamond;

    // We'll create a list of generic objects
    // in the following array in order to demonstrate
    // virtual method calls and polymorphism.

    DrawList: pShape[9];

begin ASCIIDraw;

    console.cls();

    // Initialize various rectangle, roundrect, and diamond objects.
    // This code also stores pointers to each of these objects in
    // the DrawList array.

```

```

mov( rect.create(), aRect1 ); mov( esi, DrawList[0*4] );
mov( rect.create(), aRect2 ); mov( esi, DrawList[1*4] );
mov( rect.create(), aRect3 ); mov( esi, DrawList[2*4] );

mov( roundrect.create(), aRect1 ); mov( esi, DrawList[3*4] );
mov( roundrect.create(), aRect2 ); mov( esi, DrawList[4*4] );
mov( roundrect.create(), aRect3 ); mov( esi, DrawList[5*4] );

mov( diamond.create(), aDiamond1 ); mov( esi, DrawList[6*4] );
mov( diamond.create(), aDiamond2 ); mov( esi, DrawList[7*4] );
mov( diamond.create(), aDiamond3 ); mov( esi, DrawList[8*4] );

// Size and position each of these objects:

aRect1.resize( 10, 10 );
aRect1.moveTo( 10, 10 );

aRect2.resize( 10, 10 );
aRect2.moveTo( 15, 15 );

aRect3.resize( 10, 10 );
aRect3.moveTo( 20, 20 );
aRect3.fill( false );

aRrect1.resize( 10, 10 );
aRrect1.moveTo( 40, 10 );

aRrect2.resize( 10, 10 );
aRrect2.moveTo( 45, 15 );

aRrect3.resize( 10, 10 );
aRrect3.moveTo( 50, 20 );
aRrect3.fill( false );

aDiamond1.resize( 9, 9 );
aDiamond1.moveTo( 28, 0 );

aDiamond2.resize( 9, 9 );
aDiamond2.moveTo( 28, 3 );

aDiamond3.resize( 9, 9 );
aDiamond3.moveTo( 28, 6 );
aDiamond3.fill( false );

// Note for the real fun, draw all of the objects
// on the screen using the following simple loop.

for( mov( 0, ebx ); ebx < 9; inc( ebx ) ) do

    DrawList.draw[ ebx*4 ]();

endfor;

end ASCIIIDraw;

```

Program 10.3 ShapeMain.hla - The Main Program That Demonstrates Using Shape Objects

10.17 Putting It All Together

HLA's class declarations provide a powerful tool for creating object-oriented assembly language programs. Although object-oriented programming is not as popular in assembly as in high level languages, part of the reason has been the lack of assemblers that support object-oriented programming in a reasonable fashion and an even greater lack of tutorial information on object-oriented programming in assembly language.

While this chapter cannot go into great detail about the object-oriented programming paradigm (space limitations prevent this), this chapter does explain the object-oriented facilities that HLA provides and supplies several example programs that use those facilities. From here on, it's up to you to utilize these facilities in your programs and gain experience writing object oriented assembly code.

The MMX Instruction Set

Chapter Eleven

11.1 Chapter Overview

While working on the Pentium and Pentium Pro processors, Intel was also developing an instruction set architecture extension for multimedia applications. By studying several existing multimedia applications, developing lots of multimedia related algorithms, and through simulation, Intel developed 57 instructions that would greatly accelerate the execution of multimedia applications. The end result was their multimedia extensions to the Pentium processor that Intel calls the MMX Technology Instructions.

Prior to the invention of the MMX enhancements, good quality multimedia systems required separate digital signal processors and special electronics to handle much of the multimedia workload¹. The introduction of the MMX instruction set allowed later Pentium processors to handle these multimedia tasks without these expensive digital signal processors (DSPs), thus lowering the cost of multimedia systems. So later Pentiums, Pentium II, Pentium III, and Pentium IV processors all have the MMX instruction set. Earlier Pentiums (and CPUs prior to the Pentium) and the Pentium Pro do not have these instructions available. Since the instruction set has been available for quite some time, you can probably use the MMX instructions without worrying about your software failing on many machines.

In this chapter we will discuss the MMX Technology instructions and how to use them in your assembly language programs. The use of MMX instructions, while not completely limited to assembly language, is one area where assembly language truly shines since most high level languages do not make good use of MMX instructions except in library routines. Therefore, writing fast code that uses MMX instructions is mainly the domain of the assembly language programmer. Hence, it's a good idea to learn these instructions if you're going to write much assembly code.

11.2 Determining if a CPU Supports the MMX Instruction Set

While it's almost a given that any modern CPU your software will run on will support the MMX extended instruction set, there may be times when you want to write software that will run on a machine even in the absence of MMX instructions. There are two ways to handle this problem – either provide two versions of the program, one with MMX support and one without (and let the user choose which program they wish to run), or the program can dynamically determine whether a processor supports the MMX instruction set and skip the MMX instructions if they are not available.

The first situation, providing two different programs, is the easiest solution from a software development point of view. You don't actually create two source files, of course; what you do is use conditional compilation statements (i.e., `#IF..#ELSE..#ENDIF`) to selectively compile MMX or standard instructions depending on the presence of an identifier or value of a boolean constant in your program. See “Conditional Compilation (Compile-Time Decisions)” on page 932 for more details.

Another solution is to dynamically determine the CPU type at run-time and use program logic to skip over the MMX instructions and execute equivalent standard code if the CPU doesn't support the MMX instruction set. If you're expecting the software to run on an Intel Pentium or later CPU, you can use the `CPUID` instruction to determine whether the processor supports the MMX instruction set. The `CPUID` instruction will return bit 23 as a one in the feature flags return result.

The following code illustrates how to use the `CPUID` instruction. This example does not demonstrate the entire `CPUID` sequence, but shows the portion used for detection of MMX technology.

1. A good example was the Apple Quadra 660AV and 840AV computer systems; they were built around the Motorola 68040 processor rather than a Pentium, but the 68040 was no more capable of handling multimedia applications than the Pentium. However, an on-board DSP (digital signal processor) CPU allowed the Quadas to easily handle audio applications that the 68040 could not.

```
// For a perfectly general routine, you should determine if this
// is a Pentium or later processor. We'll assume at least a Pentium
// for now, since most Win32 OS expect a Pentium or better processor.

mov( 1, eax );           // Request for CPUID feature flags.
CPUID();                 // Get the feature flags into EDX.
test( $80_0000, edx );   // Is bit 23 set?
jnz HasMMX;
```

This code assumes at least the presence of a Pentium Processor. If your code needs to run on a 486 or 386 processor, you will have to detect that the system is using one of these processors. There is tons of code on the net that detects different processors, but most of it will not run under Windows since that code typically uses protected (non-user-mode) instructions. There is also a Windows API call (`Get_Machine_Info`) that will return this status. We'll not go into the details here because 99% of the users out there that are running modern versions of Windows have a CPU that supports the MMX instruction set or, at least, the CPUID instruction.

11.3 The MMX Programming Environment

The MMX architecture extends the Pentium architecture by adding the following:

- Eight MMX registers (MM0..MM7).
- Four MMX data types (packed bytes, packed words, packed double words, and quad word).
- 57 MMX Instructions.

11.3.1 The MMX Registers

The MMX architecture adds eight 64-bit registers to the Pentium. The MMX instructions refer to these registers as MM0, MM1, MM2, MM3, MM4, MM5, MM6, and MM7. These are strictly data registers, you cannot use them to hold addresses nor are they suitable for calculations involving addresses.

Although MM0..MM7 appear as separate registers in the Intel Architecture, the Pentium processors alias these registers with the FPU's registers (ST0..ST7). Each of the eight MMX 64-bit registers is physically equivalent to the L.O. 64-bits of each of the FPU's registers (see Figure 11.1). The MMX registers overlay the FPU registers in much the same way that the 16-bit general purpose registers overlay the 32-bit general purpose registers.

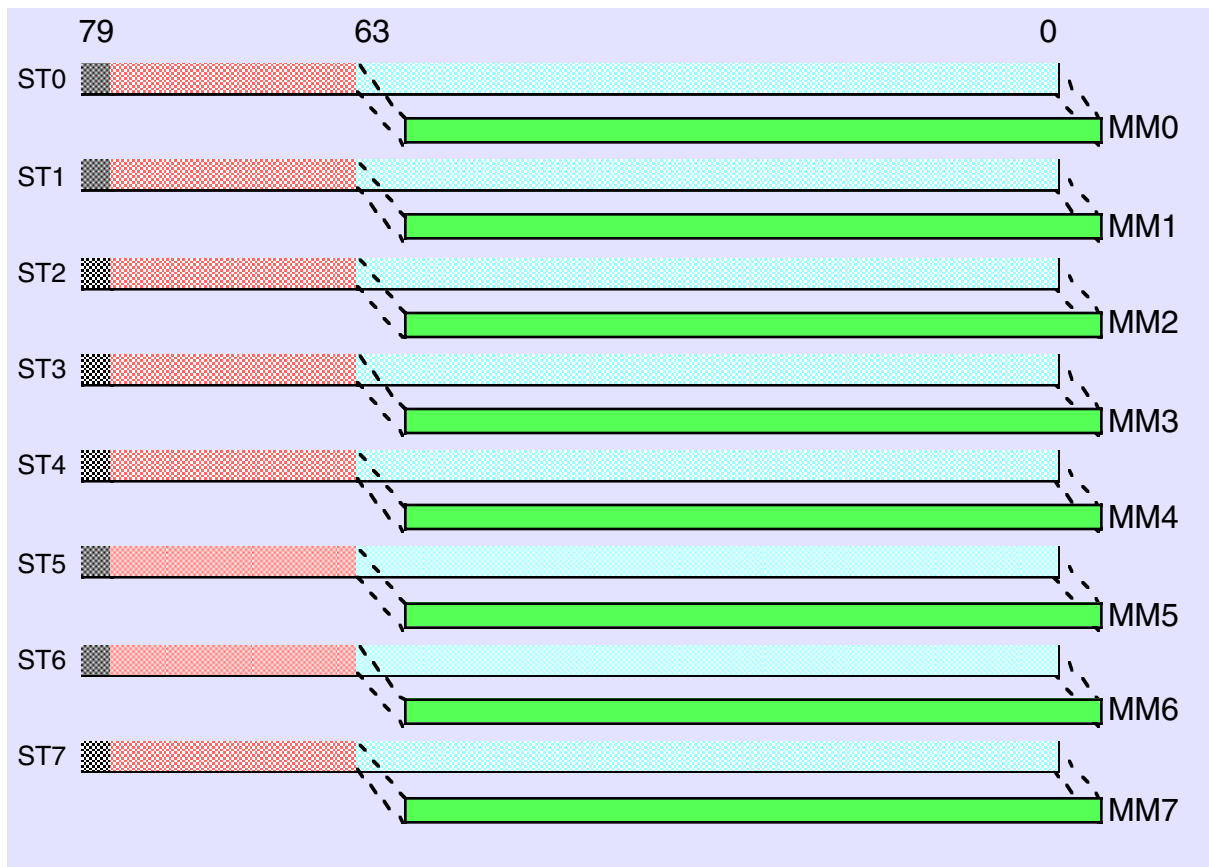


Figure 11.1 MMX and FPU Register Aliasing

Because the MMX registers overlay the FPU registers, you cannot mix FPU and MMX instructions in the same computation sequence. You can begin executing an MMX instruction sequence at any time; however, once you execute an MMX instruction you cannot execute another FPU instruction until you execute a special MMX instruction, EMMS (Exit MMX Machine State). This instruction resets the FPU so you may begin a new sequence of FPU calculations. The CPU does not save the FPU state across the execution of the MMX instructions; executing EMMS clears all the FPU registers. Because saving FPU state is very expensive, and the EMMS instruction is quite slow, it's not a good idea to frequently switch between MMX and FPU calculations. Instead, you should attempt to execute the MMX and FPU instructions at different times during your program's execution.

You're probably wondering why Intel chose to alias the MMX registers with the FPU registers. Intel, in their literature, brags constantly about what a great idea this was. You see, by aliasing the MMX registers with the FPU registers, Microsoft and other multitasking OS vendors did not have to write special code to save the MMX state when the CPU switched from one process to another. The fact that the OS automatically saved the FPU state means that the CPU would automatically save the MMX state as well. This meant that the new Pentium chips with MMX technology that Intel created were automatically compatible with Windows 95, Windows NT, and Linux without any changes to the operating system code.

Of course, those operating systems have long since been upgraded and Microsoft (and Linux developers) could have easily provided a "service pack" to handle the new registers (had Intel chosen not to alias the FPU and MMX registers). So while aliasing MMX with the FPU provided a very short-lived and temporary benefit, in retrospect Intel made a big mistake with this decision. They've obviously realized their mistake, because as they've introduced new "streaming" instructions (the floating point equivalent of the MMX instruction set) they've added new registers (XMM0..XMM7) without using this trick. It's too bad they

don't fix the problem in their current CPUs (there is no technical reason why they can't create separate MMX and FPU registers at this point). Oh well, you'll just have to live with the fact that you can't execute interleaved FPU and MMX instructions.

11.3.2 The MMX Data Types

The MMX instruction set supports four different data types: an eight-byte array, a four-word array, a two dword array, and a quadword object. Each MMX registers processes one of these four data types (see Figure 11.2).

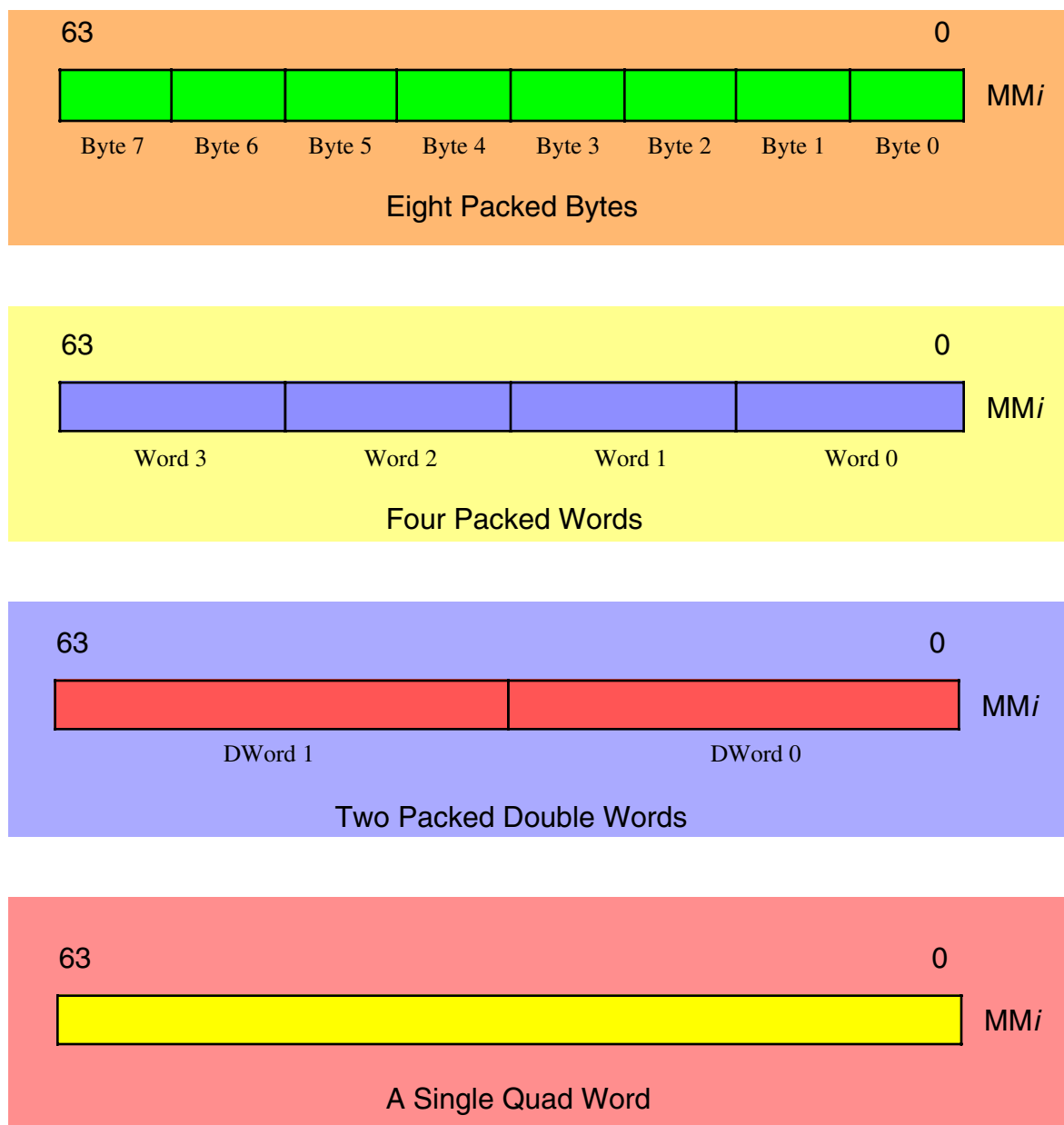


Figure 11.2 The MMX Data Types

Despite the presence of 64-bit registers, the MMX instruction set does not extend the 32-bit Pentium processor to 64-bits. Instead, after careful study Intel added only those 64-bit instructions that were useful for multimedia operations. For example, you cannot add or subtract two 64-bit integers with the MMX instruction set. In fact, only the logical and shift operations directly manipulate 64 bits.

The MMX instruction set was not designed to provide general 64-bit capabilities to the Pentium. Instead, the MMX instruction set provides the Pentium with the capability of performing multiple eight-, sixteen-, or thirty-two bit operations simultaneously. In other words, the MMX instructions are generally SIMD (Single Instruction Multiple Data) instructions (see “Parallel Processing” on page 258 for an explanation of SIMD). For example, a single MMX instruction can add eight separate pairs of byte values together. This is not the same as adding two 64-bit values since the overflow from the individual bytes does not carry over into the higher order bytes. This can accelerate a program that needs to add a long string of bytes together since a single MMX instruction can do the work of eight regular Pentium instructions. This is how the MMX instruction set speeds up multimedia applications – by processing multiple data objects in parallel with a single instruction. Given the data types the MMX instruction set supports, you can process up to eight byte objects in parallel, four word objects in parallel, or two double words in parallel.

11.4 The Purpose of the MMX Instruction Set

The Single Instruction Multiple Data model the MMX architecture supports may not look all that impressive when viewed with a SISD (Single Instruction, Single Data) bias. Once you’ve mastered the basic integer instructions on the 80x86, it’s difficult to see the application of the MMX’s SIMD instruction set. However, the MMX instructions directly address the needs of modern media, communications, and graphics applications, which often use sophisticated algorithms that perform the same operations on a large number of small data types (bytes, words, and double words).

For example, most programs use a stream of bytes or words to represent audio and video data. The MMX instructions can operate on eight bytes or four words with a single instruction, thus accelerating the program by almost a factor of four or eight.

One drawback to the MMX instruction set is that it is not general purpose. Intel’s research that led to the development of these new instructions specifically targeted audio, video, graphics, and another multimedia applications. Although some of the instructions are applicable in many general programs, you’ll find that many of the instructions have very little application outside their limited domain. Although, with a lot of deep thought, you can probably dream up some novel uses of many of these instructions that have nothing whatsoever at all to do with multimedia, you shouldn’t get too frustrated if you cannot figure out why you would want to use a particular instruction; that instruction probably has a specific purpose and if you’re not trying to code a solution for that problem, you may not be able to use the instruction. If you’re questioning why Intel would put such limited instructions in their instruction set, just keep in mind that although you can use the instruction(s) for lots of different purposes, they are invaluable for the few purposes they are uniquely suited.

11.5 Saturation Arithmetic and Wraparound Mode

The MMX instruction set supports saturating arithmetic (see “Sign Extension, Zero Extension, Contraction, and Saturation” on page 63). When manipulating standard integer values and an overflow occurs, the integer instructions maintain the correct L.O. bits of the value in the integer while truncating any overflow². This form of arithmetic is known as *wraparound* mode since the L.O. bits wrap back around to zero. For example, if you add the two eight-bit values \$02 and \$FF you wind up with a carry and the result \$01. The actual sum is \$101, but the operation truncates the ninth bit and the L.O. byte wraps around to \$01.

In saturation mode, results of an operation that overflow or underflow are clipped (saturated) to some maximum or minimum value depending on the size of the object and whether it is signed or unsigned. The

2. For some instructions the overflow may appear in another register or the carry flag, but in the destination register the high order bits are lost.

result of an operation that exceeds the range of a data-type saturates to the maximum value of the range. A result that is less than the range of a data type saturates to the minimum value of the range.

Table 1:

Data Type	Decimal		Hexadecimal	
	Lower Limit	Upper Limit	Lower Limit	Upper Limit
Signed Byte	-128	+127	\$80	\$7f
Unsigned Byte	0	255	0	\$ff
Signed Word	-32768	+32767	\$8000	\$7fff
Unsigned Word	0	65535	0	\$ffff

For example, when the result exceeds the data range limit for signed bytes, it is saturated to \$7f; if a value is less than the data range limit, it is saturated to \$80 for signed bytes. If a value exceeds the range for unsigned bytes, it is saturated to \$ff or \$00.

This saturation effect is very useful for audio and video data. For example, if you are amplifying an audio signal by multiplying the words in the 44.1 kHz audio stream by 1.5, clipping the value at +32767, while introducing distortion, sounds far better than allowing the waveform to wrap around to -32768. Similarly, if you are mixing colors in a 24-bit graphic or video image, saturating to white produces much more meaningful results than wrap-around.

Since Intel created the MMX architecture to support audio, graphics, and video, it should come as no surprise that the MMX instruction set supports saturating arithmetic. For those applications that require saturating arithmetic, having the CPU automatically handle this process (rather than having to explicitly check after each calculation) is another way the MMX architecture speeds up multimedia applications.

11.6 MMX Instruction Operands

Most MMX instructions operate on two operands, a source and a destination operand. A few instructions have three operands with the third operand being a small immediate (constant) value. In this section we'll take a look at the common MMX instruction operands.

The destination operand is almost always an MMX register. In fact, the only exceptions are those instructions that store an MMX register into memory. The MMX instructions always leave the result of MMX calculations in an MMX register.

The source operand can be an MMX register or a memory location. The memory location is usually a quad word entity, but certain instructions operate on double word objects. Note that, in this context, "quad word" and "double word" mean eight or four consecutive bytes in memory; they do not necessarily imply that the MMX instruction is operating on a qword or dword object. For example, if you add eight bytes together using the PADDB (packed add bytes) instruction, PADDB references a qword object in memory, but actually adds together eight separate bytes.

For most MMX instructions, the generic HLA syntax is one of the following:

```
mmxInstr( source, dest );
```

The specific forms are

```
mmxInstr( mmi, mmi );    // i=0..7
mmxInstr( mem, mmi );    // i=0..7
```

MMX instructions access memory using the same addressing modes as the standard integer instructions. Therefore, any legal 80x86 addressing mode is usable in an MMX instruction. For those instructions that reference a 64-bit memory location, HLA requires that you specify an anonymous memory object (e.g., “[ebx]” or “[ebp+esi*8+6]”) or a *qword* variable.

A few instructions require a small immediate value (or constant). For example, the shift instructions let you specify a shift count as an immediate value in the range 0..63. Another instruction uses the immediate value to specify a set of four different count values in the range 0..3 (i.e., four two-bit count values). These instructions generally take the following form:

```
mmxInstr( imm8, source, dest );
```

Note that, in general, MMX instructions do not allow you to specify immediate constants as operands except for a few special cases (such as shift counts). In particular, the source operand to an MMX instruction has to be a register or a quad word variable, it cannot be a 64-bit constant. To achieve the same effect as specifying a constant as the source operand, you must initialize a quad word variable in the READONLY (or STATIC) section of your program and specify this variable as the source operand. Unfortunately, HLA does not support 64-bit constants, so initializing the value is going to be a bit of a problem. There are two solutions to this problem: break the constant into smaller pieces (bytes, words, or double words) and emit the constant in pieces that HLA can process; or you can write your own numeric conversion routine(s) using the HLA compile-time language to allow the emission of a 64-bit constant. We'll explore both of those approaches here.

The first approach is the one you will most commonly use. Very few MMX instructions actually operate on 64-bit data operands; instead, they typically operate on a (small) array of bytes, words, or double words. Since HLA provides good support for byte, word, and double word constant expressions, specifying a 64-bit MMX memory operand as a short array of objects is probably the best way to create this data. Since the MMX instructions that fetch a source value from memory expect a 64-bit operand, you must declare such objects as *qword* variables, e.g.,

```
static
    mmxVar:qword;
```

The big problem with this declaration is that the *qword* type does not allow an initializer (since HLA cannot handle 64-bit constant expressions). Since this declaration occurs in the STATIC segment, HLA will initialize *mmxVar* with zero; probably not the value you're interested in supplying here.

There are two ways to solve this problem. The first way is to put the MMX variable declarations in the DATA segment rather than the STATIC segment. As you may recall (see “The Data and Static Sections” on page 159) HLA does not actually allocate storage for variables you declare in the DATA section. Instead, you must explicitly reserve space using the BYTE, WORD, DWORD, etc., directives. The data declarations that immediately follow the variable definition provide the initial data for that variable. Here's an example of such a declaration:

```
data
    mmxDVar: qword;
    dword $1234_5678, $90ab_cdef;
```

Note that the DWORD directive above stores the double word constants in successive memory locations. Therefore, \$1234_5678 will appear in the L.O. double word of the 64-bit value and \$90ab_cdef will appear in the H.O. double word of the 64-bit value. Always keep in mind that the L.O. objects come first in the list following the DWORD (or BYTE, or WORD, or ???) directive; this appears to be opposite of the way you're used to reading values.

One problem with using DATA segment declarations is that they are read/write objects. Further, objects you declare in the DATA segment aren't necessarily contiguous with objects you declare in the STATIC segment. Therefore, putting objects in the DATA segment isn't always possible. The second solution is to use the NOSTORAGE attribute on a variable declaration (see “The NOSTORAGE Attribute” on page 163). For example, if you wanted to declare a read-only 64-bit object, you could use the following declaration:

```
readonly
    mmxConst: qword: nostorage;
```

```
dword $FFFF_0000, $FFFF_0000;
```

The examples up to this point have all used a pair of `DWORD` directives to provide the initialization constant. However, you can use any data declaration directive, or even a combination of directives, as long as you allocate at least eight bytes (64-bits) for each *qword* constant. The following data declaration, for example, initializes eight eight-bit constants for an MMX operand; this would be perfect for a `PADDB` instruction or some other instruction that operates on eight bytes in parallel:

```
static
  eightBytes: qword: nostorage;
    byte 0, 1, 2, 3, 4, 5, 6, 7;
```

Although most MMX instructions operate on small arrays of bytes, words, or double words, a few actually do operate on 64-bit quantities. For such memory operands you would probably prefer to specify a 64-bit constant rather than break it up into its constituent double word values. This way, you don't have to remember to put the L.O. double word first and perform other mental adjustments.

Although HLA does not support 64-bit constants in the compile time language, HLA is flexible enough to allow you to extend the language to handle such declarations. Program 11.1 demonstrates how to write a macro to accept a 64-bit hexadecimal constant. This macro will automatically emit two `DWORD` declarations containing the L.O. and H.O. components of the 64-bit value you specify as the *qword16* (quadword constant, base 16) macro parameter. You would typically use the *qword16* macro as follows:

```
static
  HOO nes: qword: nostorage;
    qword16( $FFFF_FFFF_0000_0000 );
```

The *qword16* macro would emit the following:

```
dword 0;
dword $FFFF_FFFF;
```

Without further ado, here's the macro (and a sample test program):

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

// The following macro accepts a 64-bit hexadecimal constant
// and emits two dword objects in place of the constant.

macro qword16( theHexVal ):hs, len, dwval, mplier, curch, didLO;

  // Remove whitespace around the macro parameter (shouldn't
  // be any, but just in case something weird is going on) and
  // convert all lower case characters to upper case.

  ?hs := @uppercase( @trim( @string:theHexVal, 0 ), 0 );

  // If there is a leading "$" symbol, strip it from the string.

  #if( @substr( hs, 0, 1 ) = "$" )

    ?hs := @substr( hs, 1, 256 );

  #endif

  // Process each character in the string from the L.O. digit
  // through to the H.O. digit. Add the digit, multiplied by
  // some successive power of 16, to the current sum we're
  // accumulating in dwval. When we cross a dword boundary,
```

```

// emit the L.O. dword and start over.

?len := @length( hs );      // Number of characters to process.
?dwval:dword := 0;          // Accumulate value here.
?mplier:dword := 1;         // Power of 16 to multiply by.
?didLO:boolean := false;    // Checks for overflow.
#while( len > 0 )            // Repeat for each char in string.

    // For each character in the string, verify that it is
    // a legal hexadecimal character and merge it in with the
    // current accumulated value if it is. Print an error message
    // if we come across an illegal character.

    ?len := len - 1;         // Next available char.
    ?curch := char( @substr( hs, len, 1 )); // Get the character.
    #if( curch in {'0'..'9'} ) // See if valid decimal digit.

        // Accumulate result if decimal digit.

        ?dwval := dwval +
            (uns8( curch ) - uns8( '0' )) * mplier;

    #elseif( curch in {'A'..'F'} ) // See if valid hex digit.

        // Accumulate result if a hexadecimal digit.

        ?dwval := dwval +
            (uns8( curch ) - uns8( 'A' ) + 10) * mplier;

    // Ignore underscore characters and report an error for anything
    // else we find in the string.

    #elseif( curch <> '_' )

        #error( "Illegal character in 64-bit hexadecimal constant" )
        #print( "Character = '", curch, "' Rest of string: '", hs, "'" )

    #endif

    // If it's not an underscore character, adjust the multiplier value.
    // If we cross a dword boundary, emit the L.O. value as a dword
    // and reset everything for the H.O. dword.

    #if( curch <> '_' )

        // If the current value fits in 32 bits, process this
        // as though it were a dword object.

        #if( mplier < $1000_0000 )

            ?mplier := mplier * 16;

        #elseif( len > 0 )

            // Down here we've just processed the last hex
            // digit that will fit into 32 bits. So emit the
            // L.O. dword and reset the mplier and dwval constants.

            ?mplier := 1;
            dword dwval;

```

```

        ?dwval := 0;

        // If we've been this way before, we've got an
        // overflow.

        #if( didLO )

            #error( "64-bit overflow in constant" );

        #endif

        ?didLO := true;

    #endif

#endif

#endwhile

// Emit the H.O. dword here.

dword dwval;

// If the constant only consumed 32 bits, we've got to emit a zero
// for the H.O. dword at this point.

#if( !didLO )

    dword 0;

#endif

endmacro;

static
    x:qword:nostorage;
    qword16( $1234_5678_90ab_cdef );
    qword16( 100 );

begin qwordConstType;

    stdout.put( "64-bit value of x = $" );
    stdout.putqw( x );
    stdout.newln();

end qwordConstType;

```

Program 11.1 qword16 Macro to Process 64-bit Hexadecimal Constants

Although it's a little bit more difficult, you could also write a *qword10* macro that lets you specify decimal constants as the macro operand rather than hexadecimal constants. The implementation of *qword10* is left as a programming exercise at the end of this volume.

11.7 MMX Technology Instructions

The following subsections describe each of the MMX instructions in detail. The organization is as follows:

- Data Transfer Instructions,
- Conversion Instructions,
- Packed Arithmetic Instructions,
- Comparisons,
- Logical Instructions,
- Shift and Rotate Instructions,
- the EMMS Instruction.

These sections describe *what* these instructions do, not *how* you would use them. Later sections will provide examples of how you can use several of these instructions.

11.7.1 MMX Data Transfer Instructions

```
movd( reg32, mmi );
movd( mem32, mmi );
movd( mmi, reg32 );
movd( mmi, mem32 );

movq( mem64, mmi );
movq( mmi, mem64 );
movq( mmi, mmi );
```

The MOVD (move double word) instruction copies data between a 32-bit integer register or double word memory location and an MMX register. If the destination is an MMX register, this instruction zero-extends the value while moving it. If the destination is a 32-bit register or memory location, this instruction copies the L.O. 32-bits of the MMX register to the destination.

The MOVQ (move quadword) instruction copies data between two MMX registers or between an MMX register and memory. If either the source or destination operand is a memory object, it must be a *qword* variable or HLA will complain.

11.7.2 MMX Conversion Instructions

```
packssdw( mem64, mmi );
packssdw( mmi, mmi );

packsswb( mem64, mmi );
packsswb( mmi, mmi );

packusdw( mem64, mmi );
packusdw( mmi, mmi );

packuswb( mem64, mmi );
packuswb( mmi, mmi );

punpckhbw( mem64, mmi );
punpckhbw( mmi, mmi );

punpckhdq( mem64, mmi );
```

```

punpckhdq( mmi, mmi );

punpckhwd( mem64, mmi );
punpckhwd( mmi, mmi );

punpcklbw( mem64, mmi );
punpcklbw( mmi, mmi );

punpckldq( mem64, mmi );
punpckldq( mmi, mmi );

punpcklwd( mem64, mmi );
punpcklwd( mmi, mmi );

```

The `PACKSSxx` instructions pack and saturate signed values. They convert a sequence of larger values to a sequence of smaller values via saturation. Those instructions with the *dw* suffix pack four double words into four words; those with the *wb* suffix saturate and pack eight signed words into eight signed bytes.

The `PACKSSDW` instruction takes the two double words in the source operand and the two double words in the destination operand and converts these to four signed words via saturation. The instruction packs these four words together and stores the result in the destination MMX register. See Figure 11.3 for details.

The `PACKSSWB` instruction takes the four words from the source operand and the four signed words from the destination operand and converts, via signed saturation, these values to eight signed bytes. This instruction leaves the eight bytes in the destination MMX register. See Figure 11.4 for details.

One application for these pack instructions is to convert UNICODE to ASCII (ANSI). You can convert UNICODE (16-bit) character to ANSI (8-bit) character if the H.O. eight bits of each UNICODE character is zero. The `PACKUSWB` instruction will take eight UNICODE characters and pack them into a string that is eight bytes long with a single instruction. If the H.O. byte of any UNICODE character contains a non-zero value, then the `PACKUSWB` instruction will store `$FF` in the respective byte; therefore, you can use `$FF` as a conversion error indication.

Another use for the `PACKSSWB` instruction is to translate a 16-bit audio stream to an eight-bit stream. Assuming you've scaled your sixteen-bit values to produce a sequence of values in the range -128..+127, you can use the `PACKSSWB` instruction to convert that sequence of 16-bit values into a packed sequence of eight bit values.

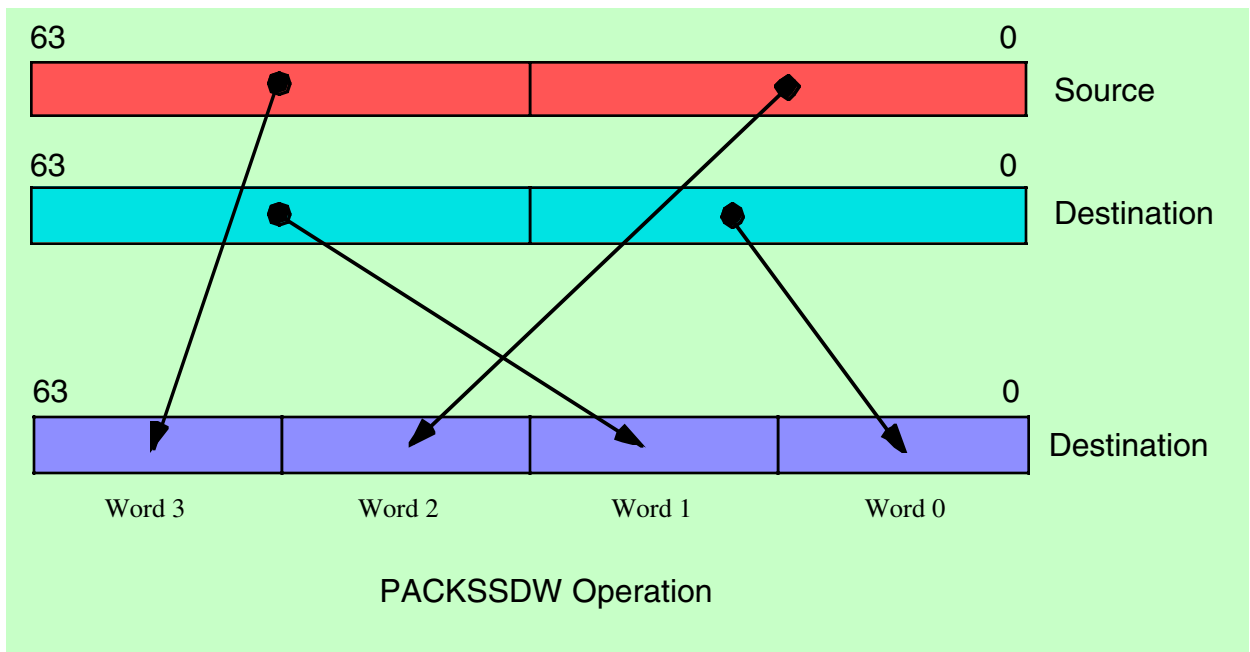


Figure 11.3 **PACKSSDW Instruction**

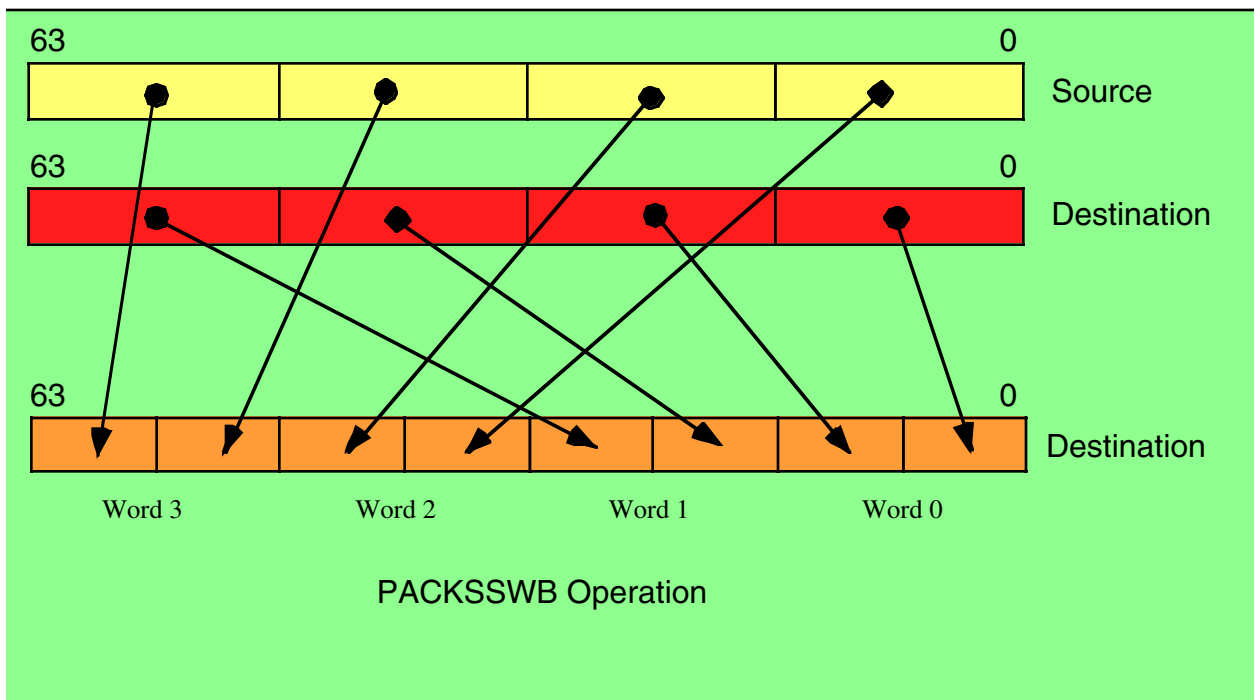


Figure 11.4 **PACKSSWB Instruction**

The unpack instructions (PUNPCKxxx) provide the converse operation to the pack instructions. The unpack instructions take a sequence of smaller, packed, values and translate them into larger values. There is one problem with this conversion, however. Unlike the pack instructions, where it took two 64-bit operands to generate a single 64-bit result, the unpack operations will produce a 64-bit result from a single 32-bit result. Therefore, these instructions cannot operate directly on full 64-bit source operands. To overcome this limitation, there are two sets of unpack instructions: one set unpacks the data from the L.O. double word of a 64-bit object, the other set of instructions unpacks the H.O. double word of a 64-bit object. By executing one instruction from each set you can unpack a 64-bit object into a 128-bit object.

The PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ instructions merge (unpack) the L.O. double words of their source and destination operands and store the 64-bit result into their destination operand.

The PUNPCKLBW instruction unpacks and interleaves the low-order four bytes of the source (first) and destination (second) operands. It places the L.O. four bytes of the destination operand at the even byte positions in the destination and it places the L.O. four bytes of the source operand in the odd byte positions of the destination operand.(see Figure 11.5).

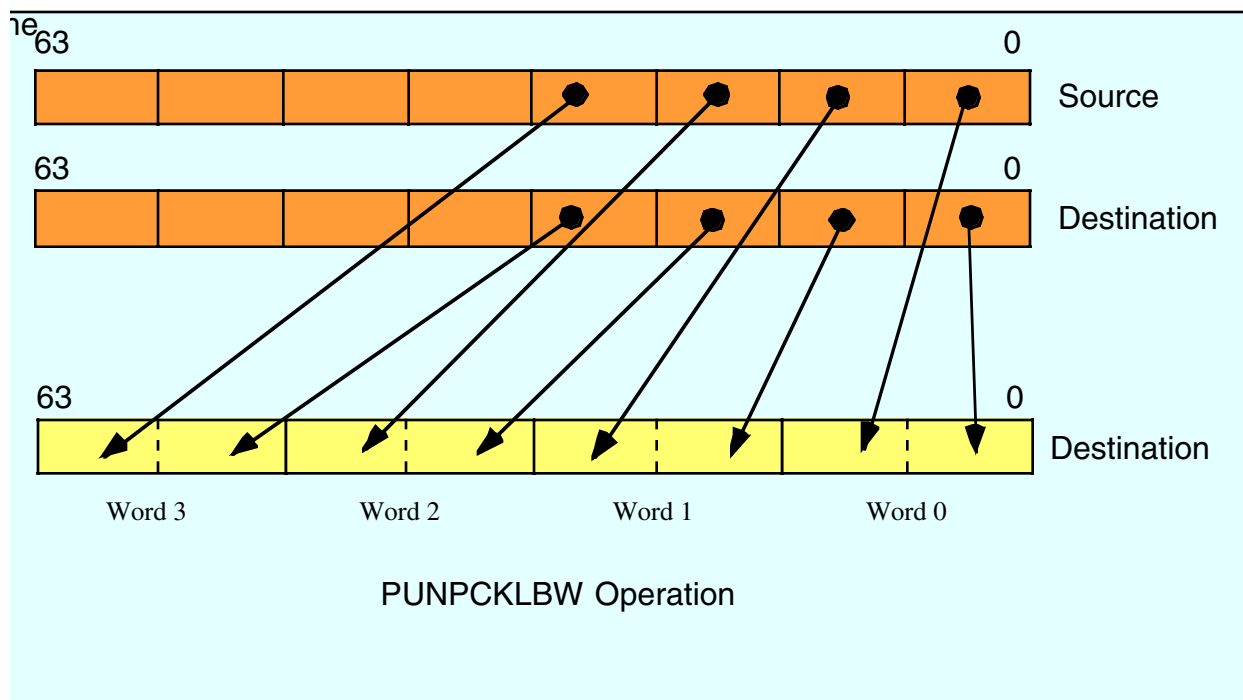


Figure 11.5 UNPCKLBW Instruction

The PUNPCKLWD instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand (see Figure 11.6).

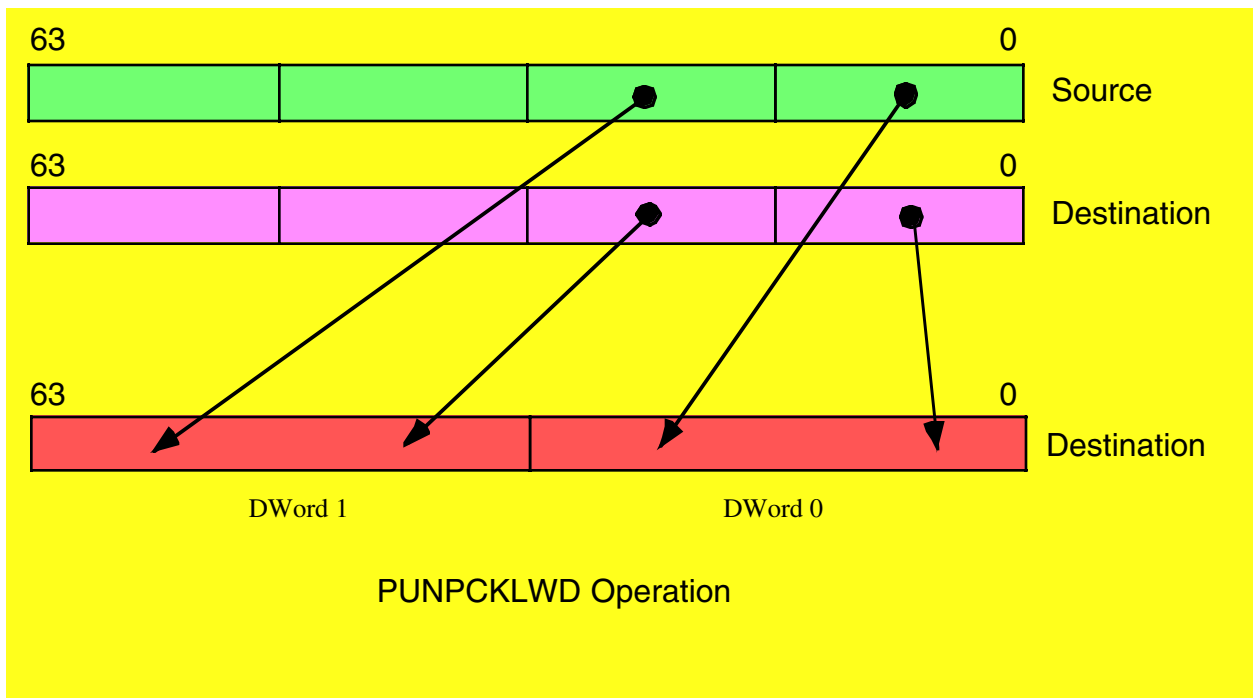


Figure 11.6 The PUNPCKLWD Instruction

The PUNPCKDQ instruction copies the L.O. dword of the source operand to the L.O. dword of the destination operand and it copies the (original) L.O. dword of the destination operand to the L.O. dword of the destination (i.e., it doesn't change the L.O. dword of the destination, see Figure 11.7).

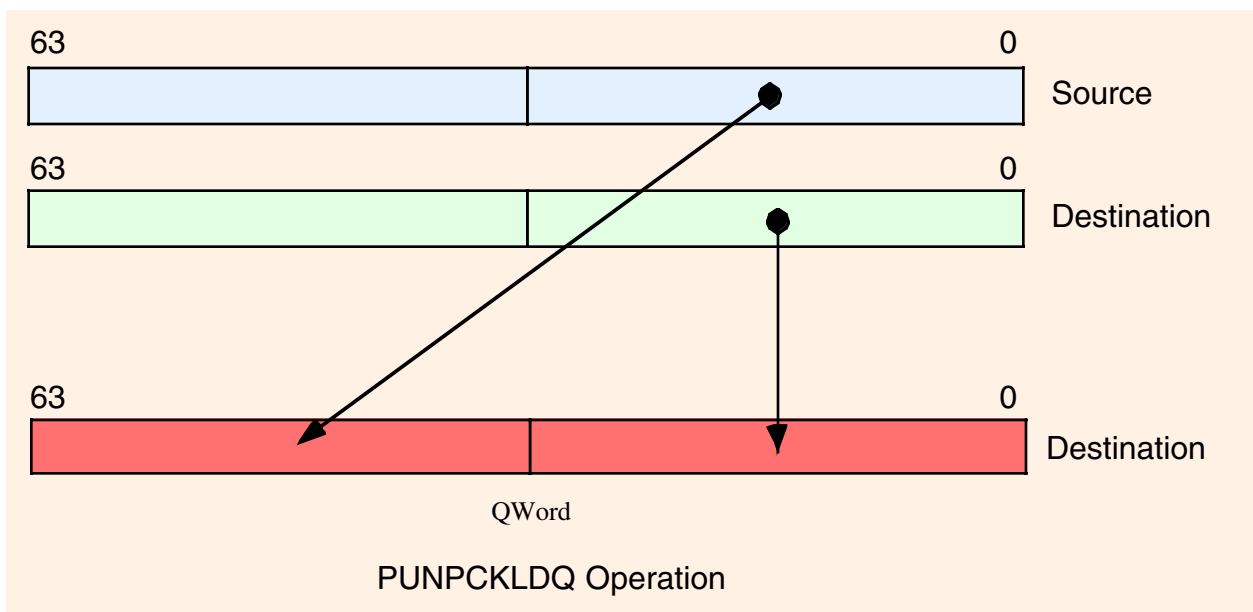


Figure 11.7 PUNPCKLDQ Instruction

The PUNPCKHBW instruction is quite similar to the PUNPCKLBW instruction. The difference is that it unpacks and interleaves the high-order four bytes of the source (first) and destination (second) operands. It places the H.O. four bytes of the destination operand at the even byte positions in the destination and it places the H.O. four bytes of the source operand in the odd byte positions of the destination operand (see Figure 11.8).

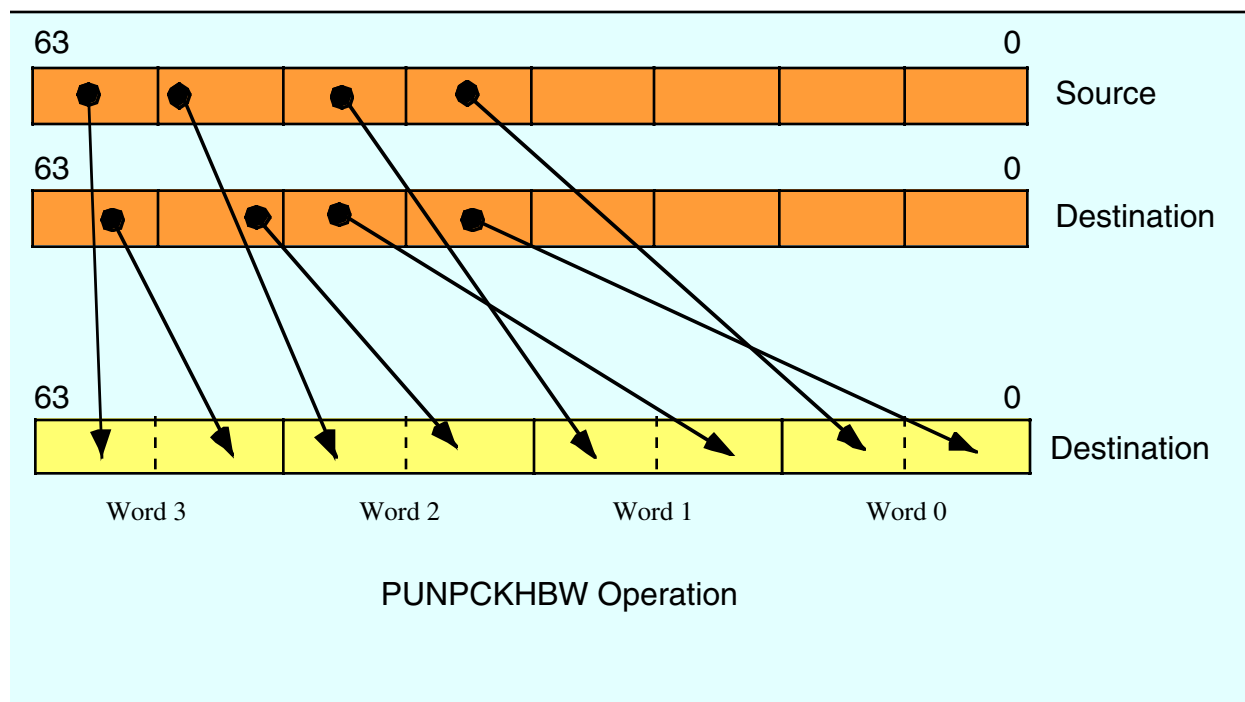


Figure 11.8 PUNPCKHBW Instruction

The PUNPCKHWD instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand (see Figure 11.9)

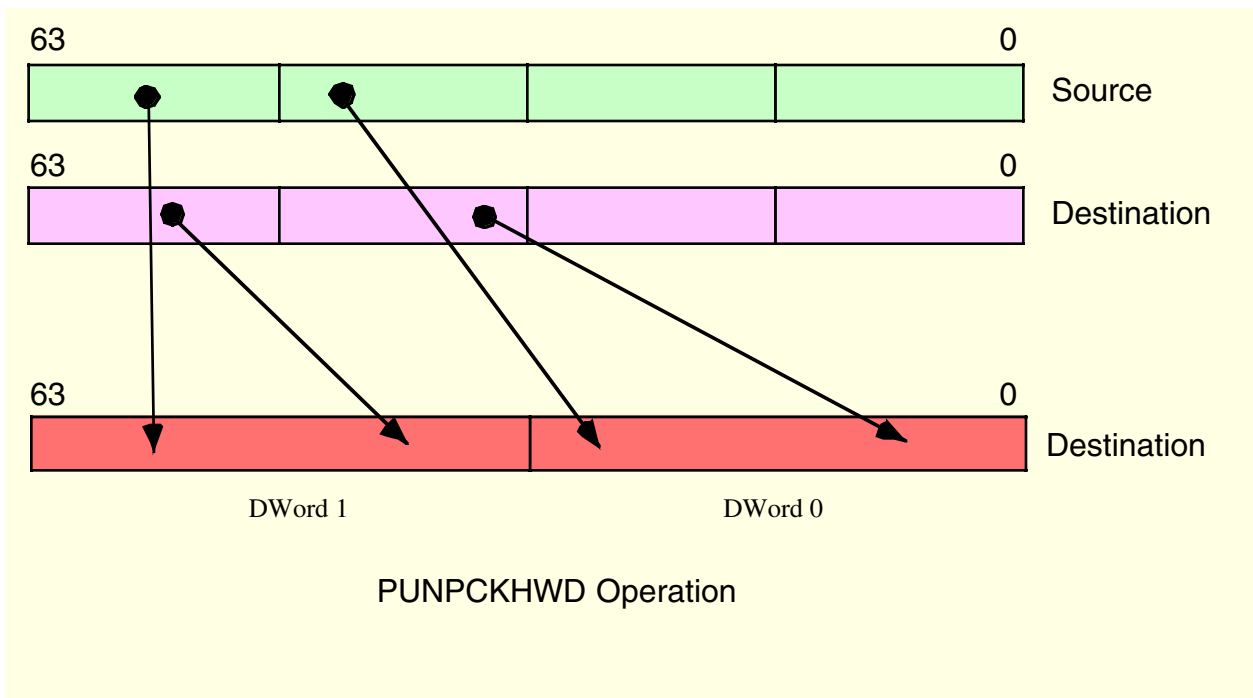


Figure 11.9 PUNPCKHWD Instruction

The PUNPCKHDQ instruction copies the H.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) H.O. dword of the destination operand to the L.O. dword of the destination (see Figure 11.10).

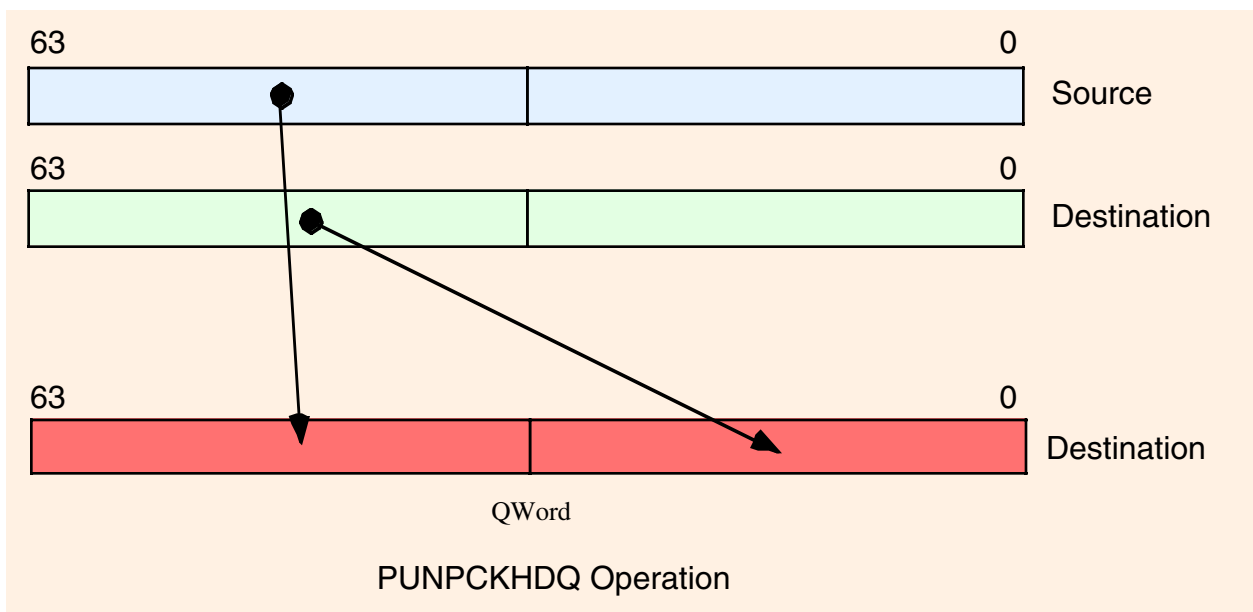


Figure 11.10 PUNPCKDQ Instruction

Since the unpack instructions provide the converse operation of the pack instructions, it should come as no surprise that you can use these instructions to perform the inverse algorithms of the examples given earlier for the pack instructions. For example, if you have a string of eight-bit ANSI characters, you can convert them to their UNICODE equivalents by setting one MMX register (the source) to all zeros. You can convert each four characters of the ANSI string to UNICODE by loading those four characters into the L.O. double word of an MMX register and executing the PUNPCKLBW instruction. This will interleave each of the characters with a zero byte, thus converting them from ANSI to UNICODE.

Of course, the unpack instructions are quite valuable any time you need to interleave data. For example, if you have three separate images containing the blue, red, and green components of a 24-bit image, it is possible to merge these three bytes together using the PUNPCKLBW instruction³.

11.7.3 MMX Packed Arithmetic Instructions

```

paddb( mem64, mmi );
paddb( mmi, mmi );

paddw( mem64, mmi );
paddw( mmi, mmi );

padd( mem64, mmi );
padd( mmi, mmi );

paddsb( mem64, mmi );
paddsb( mmi, mmi );

paddsw( mem64, mmi );
paddsw( mmi, mmi );

paddusb( mem64, mmi );
paddusb( mmi, mmi );

paddusw( mem64, mmi );
paddusw( mmi, mmi );

psubb( mem64, mmi );
psubb( mmi, mmi );

psubw( mem64, mmi );
psubw( mmi, mmi );

psubd( mem64, mmi );
psubd( mmi, mmi );

psubsb( mem64, mmi );
psubsb( mmi, mmi );

psubsw( mem64, mmi );
psubsw( mmi, mmi );

psubusb( mem64, mmi );
psubusb( mmi, mmi );

psubusw( mem64, mmi );
psubusw( mmi, mmi );

```

3. Typically you would merge in a fourth byte of zero and then store the resulting double word every three bytes in memory to overwrite the zeros.

```

pmulhuw( mem64, mmi );
pmulhuw( mmi, mmi );

pmulhw( mem64, mmi );
pmulhw( mmi, mmi );

pmullw( mem64, mmi );
pmullw( mmi, mmi );

pmaddwd( mem64, mmi );
pmaddwd( mmi, mmi );

```

The packed arithmetic instructions operate on a set of bytes, words, or double words within a 64-bit block. For example, the PADDW instruction computes four 16-bit sums of two operand simultaneously. None of these instructions affect the CPU's FLAGS register. Therefore, there is no indication of overflow, underflow, zero result, negative result, etc. If you need to test a result after a packed arithmetic computation, you will need to use one of the packed compare instructions (see "MMX Comparison Instructions" on page 1101).

The PADDB, PADDW, and PADDD instructions add the individual bytes, words, or double words in the two 64-bit operands using a wrap-around (i.e., non-saturating) addition. Any carry out of a sum is lost; it is your responsibility to ensure that overflow never occurs. As for the integer instructions, these packed add instructions add the values in the source operand to the destination operand, leaving the sum in the destination operand. These instructions produce correct results for signed or unsigned operands (assuming overflow/underflow does not occur).

The PADDSB and PADDSW instructions add the eight eight-bit or four 16-bit operands in the source and destination locations together using signed saturation arithmetic. The PADDUSB and PADDUSW instructions add their eight eight-bit or four 16-bit operands together using unsigned saturation arithmetic. Notice that you must use different instructions for signed and unsigned value since saturation arithmetic is different depending upon whether you are manipulating signed or unsigned operands. Also note that the instruction set does not support the saturated addition of double word values.

The PSUBB, PSUBW, and PSUBD instructions work just like their addition counterparts, except of course, they compute the wrap-around difference rather than the sum. These instructions compute $\text{dest} = \text{dest} - \text{src}$. Likewise, the PSUBSB, PSUBSW, PSUBUSB, and PSUBUSW instruction compute the difference of the destination and source operands using saturation arithmetic.

While addition and subtraction can produce a one-bit carry or borrow, multiplication of two n-bit operands can produce as large as a $2 \times n$ bit result. Since overflow is far more likely in multiplication than in addition or subtraction, the MMX packed multiply instructions work a little differently than their addition and subtraction counterparts. To successfully multiply two packed values requires two instructions - one to compute the L.O. component of the result and one to produce the H.O. component of the result. The PMULLW, PMULHW, and PMULHUW instructions handle this task.

The PMULLW instruction multiplies the four words of the source operand by the four words of the destination operand and stores the four L.O. words of the four double word results into the destination operand. This instruction ignores the H.O. words of the results. Used by itself, this instruction computes the wrap-around product of an unsigned or signed set of operands; this is also the L.O. words of the four products.

The PMULHW and PMULHUW instructions complete the calculation. After computing the L.O. words of the four products with the PMULLW instruction, you use either the PMULHW or PMULHUW instruction to compute the H.O. words of the products. These two instruction multiply the four words in the source by the four words in the destination and then store the H.O. words of the results in the destination MMX register. The difference between the two is that you use PMULHW for signed operands and PMULHUW for unsigned operands. If you compute the full product by using a PMULLW and a PMULHW (or PMULHUW) instruction pair, then there is no overflow possible, hence you don't have to worry about wrap-around or saturation arithmetic.

The PMADDWD instruction multiplies the four words in the source operand by the four words in the destination operand to produce four double word products. Then it adds the two L.O. double words together

and stores the result in the L.O. double word of the destination MMX register; it also adds together the two H.O. double words and stores their sum in the H.O. word of the destination MMX register.

11.7.4 MMX Logic Instructions

```
pand( mem64, mmi );
pand( mmi, mmi );

pandn( mem64, mmi );
pandn( mmi, mmi );

por( mem64, mmi );
por( mmi, mmi );

pxor( mem64, mmi );
pxor( mmi, mmi );
```

The packed logic instructions are some examples of MMX instructions that actually operate on 64-bit values. There are no packed byte, packed word, or packed double word versions of these instructions. Of course, there is no need for special byte, word, or double word versions of these instructions since they would all be equivalent to the 64-bit logic instruction. Hence, if you want to logically AND eight bytes together in parallel, you use the PAND instruction; likewise, if you want to logically AND four words or two double words together, you just use the PAND instruction.

The PAND, POR, and PXOR instructions do the same thing as their 32-bit integer instruction counterparts (AND, OR, XOR) except, of course, they operate on two 64-bit MMX operands. Hence, no further discussion of these instructions is really necessary here. The PANDN (AND NOT) instruction is a new logic instruction, so it bears a little bit of a discussion. The PANDN instruction computes the following result:

```
dest := dest and (not source);
```

As you may recall from the chapter on Introduction to Digital Design, this is the inhibition function. If the destination operand is B and the source operand is A, this function computes $B = BA'$. (see “Boolean Functions and Truth Tables” on page 197 for details of the inhibition function). If you’re wondering why Intel chose to include such a weird function in the MMX instruction set, well, this instruction has one very useful property: it forces bits to zero in the destination operand everywhere there is a one bit in the source operand. This is an extremely useful function for merging to 64-bit quantities together. The following code sequence demonstrates this:

```
readonly
AlternateNibbles: qword: nostorage;
    qword16( $F0F0_F0F0_F0F0_F0F0 ); // Note: needs qword16 macro!
    .
    .
    .
// Create a 64-bit value in MM0 containing the Odd nibbles from MM1 and
// the even nibbles from MM0:

pandn( AlternateNibbles, mm0 ); // Clear the odd numbered nibbles.
pand( AlternateNibbles, mm1 ); // Clear the even numbered nibbles.
por( mm1, mm0 ); // Merge the two.
```

The PANDN operation is also useful for compute the set difference of two character sets. You could implement the *cs.difference* function using only six MMX instructions:

```
// Compute csdest := csdest - cssrc;

movq( (type qword csdest), mm0 );
pandn( (type qword cssrc), mm0 );
movq( mm0, (type qword csdest) );
```

```
movq( (type qword cdest[8]), mm0 );
pandn( (type qword cssrc[8]), mm0 );
movq( mm0, (type qword cdest[8]) );
```

Of course, if you want to improve the performance of the HLA Standard Library character set functions, you can use the MMX logic instructions throughout that module. Examples of such code appear later in this chapter.

11.7.5 MMX Comparison Instructions

```
pcmpeqb( mem64, mmi );
pcmpeqb( mmi, mmi );

pcmpeqw( mem64, mmi );
pcmpeqw( mmi, mmi );

pcmpeqd( mem64, mmi );
pcmpeqd( mmi, mmi );

pcmpgtb( mem64, mmi );
pcmpgtb( mmi, mmi );

pcmpgtw( mem64, mmi );
pcmpgtw( mmi, mmi );

pcmpgtd( mem64, mmi );
pcmpgtd( mmi, mmi );
```

The packed comparison instructions compare the destination (second) operand to the source (first) operand and to test for equality or greater than. These instructions compare eight pairs of bytes (PCMPEQB, PCMPGTB), four pairs of words (PCMPEQW, PCMPGTW), or two pairs of double words (PCMPEQD, PCMPGTD).

The first big difference to notice about these packed comparison instructions is that they compare the second operand to the first operand. This is exactly opposite of the standard CMP instruction (that compares the first operand to the second operand). The reason for this will become clear in a moment; however, you do have to keep in mind when using these instructions that the operands are opposite what you would normally expect. If this ordering bothers you, you can create macros to reverse the operands; we will explore this possibility a little later in this section.

The second big difference between the packed comparisons and the standard integer comparison is that these instructions test for a specific condition (equality or greater than) rather than doing a generic comparison. This is because these instructions, like the other MMX instructions, do not affect any condition code bits in the FLAGS register. This may seem contradictory, after all the whole purpose of the CMP instruction is to set the condition code bits. However, keep in mind that these instructions simultaneously compare two, four, or eight operands; that implies that you would need two, four, or eight sets of condition code bits to hold the results of the comparisons. Since the FLAGS register maintains only one set of condition code bits, it is not possible to reflect the comparison status in the FLAGS. This is why the packed comparison instructions test a specific condition - so they can return true or false to indicate the result of their comparison.

Okay, so where do these instructions return their true or false values? In the destination operand, of course. This is the third big difference between the packed comparisons and the standard integer CMP instruction – the packed comparisons modify their destination operand. Specifically, the PCMPEQB and PCMPGTB instruction compare each pair of bytes in the two operands and write false (\$00) or true (\$FF) to the corresponding byte in the destination operand, depending on the result of the comparison. For example, the instruction “pcmpgtb(MM1, MM0);” compares the L.O. byte of MM0 (A) with the L.O. byte of MM1 (B) and writes \$00 to the L.O. byte of MM0 if A is not greater than B. It writes \$FF to the L.O. byte of MM0 if A is greater than B (see Figure 11.11).

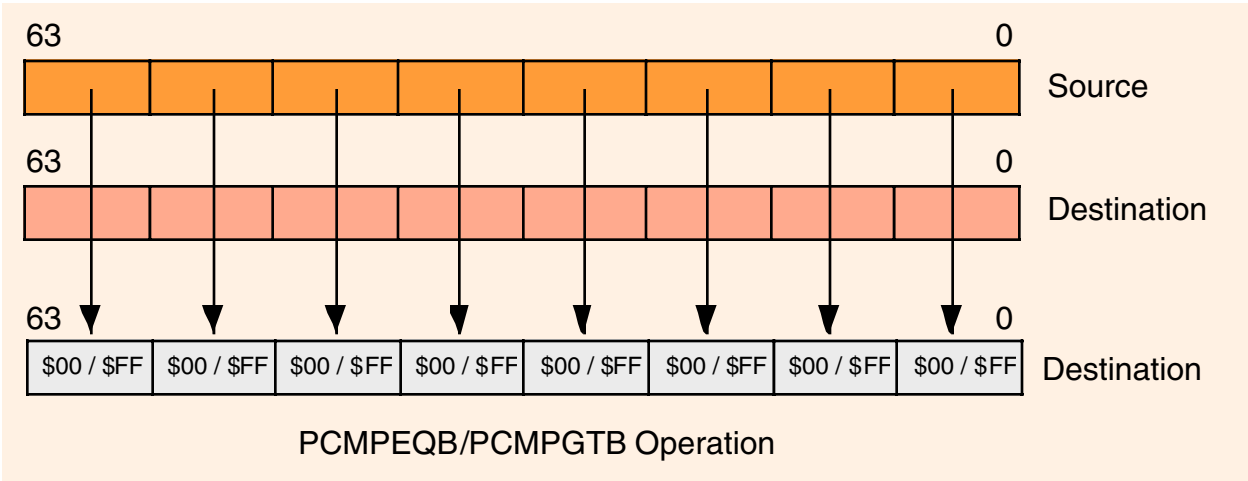


Figure 11.11 PCMPEQB and PCMPGTB Instructions

The PCMPEQW, PCMPGTW, PCMPEQD, and PCMPGTD instructions work in an analogous fashion except, of course, they compare words and double words rather than bytes (see Figure 11.12 and Figure 11.13).

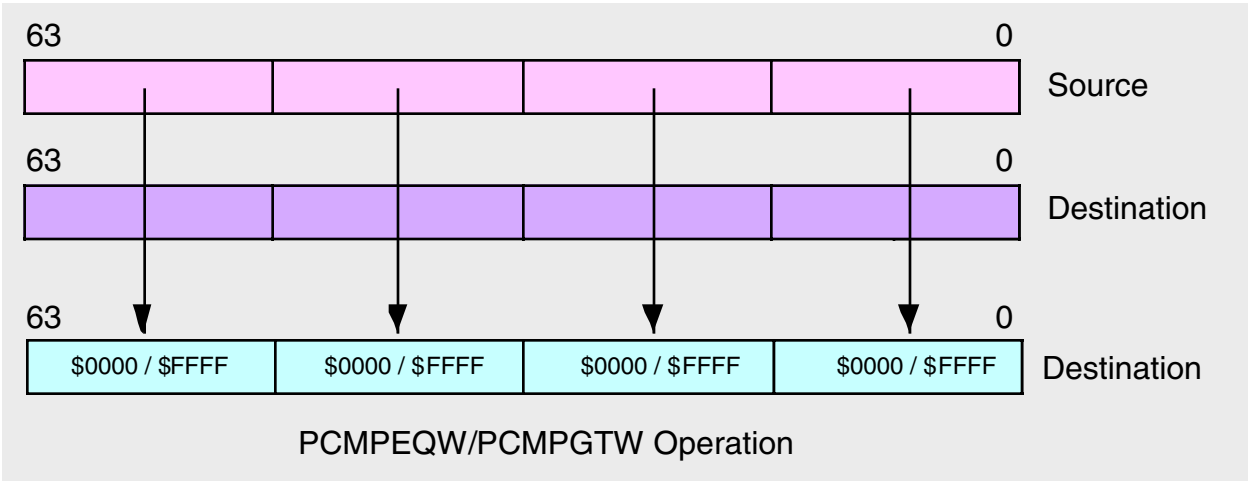


Figure 11.12 PCMPEQW and PCMPGTW Instructions

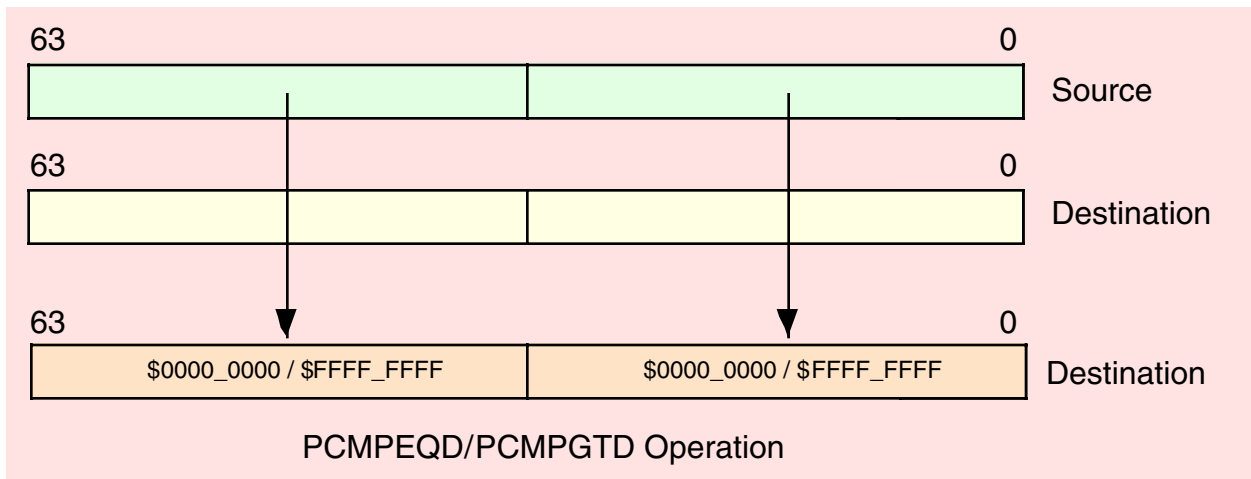


Figure 11.13 PCMPEQD and PCMPGTD Instructions

You've probably already noticed that there isn't a set of PCMPPLTx instructions. Intel chose not to provide these instructions because you can simulate them with the PCMPGTD instructions by reversing the operands. That is, $A > B$ implies $B < A$. Therefore, if you want to do a concurrent comparison of multiple operands for less than, you can use the PCMPGTD instructions to do this by simply reversing the operands. The only time this isn't directly possible is if your source operand is a memory operand; since the destination operand of the packed comparison instructions has to be an MMX register, you would have to move the memory operand into an MMX register before comparing them.

In addition to the lack of a packed less than comparison, you're also missing the not equals, less than or equal, and greater than or equal comparisons. You can easily synthesize these comparisons by executing a PXOR or POR instruction after the packed comparison.

To simulate a PCMPNEx instruction, all you've got to do is invert all the bits in the destination operand after executing a PCMPEQx instruction, e.g.,

```
pcmpeqb( mm1, mm0 );
pxor( AllOnes, mm0 ); // Assumption: AllOnes is a qword variable
                        // containing $FFFF_FFFF_FFFF_FFFF.
```

Of course, you can save the PXOR instruction by testing for zeros in the destination operand rather than ones (that is, use your program's logic to invert the result rather than actually computing the inverse).

To simulate the PCMPGEx and PCMPLEx instructions, you must do two comparisons, one for equality and one for greater than or less than, and then logically OR the results. Here's an example that computes $MM0 \leq MM1$:

```
movq( mm1, mm2 ); // Need a copy of destination operand.
pcmpgtb( mm0, mm1 ); // Remember: A < B is equal to B > A, so we're
pcmpeqb( mm0, mm2 ); // MM0 < MM1 and MM0 = MM1 here.
por( mm2, mm1 ); // Leaves boolean results in MM1.
```

If it really bothers you to have to reverse the operands, you can create macros to create your own PCMPPLTx instructions. The following example demonstrates how to create the PCMPPLTB macro:

```
macro pcmpltb( mmOp1, mmOp2 );

    pcmpgtb( mmOp2, mmOp1 );

endmacro;
```

Of course, you must keep in mind that there are two very big differences between this PCMPLTB “instruction” and a true PCMPLTB instruction. First, this form leaves the result in the first operand, not the second operand, hence the semantics of this “instruction” are different than the other packed comparisons. Second, the first operand has to be an MMX register while the second operand can be an MMX register or a quad word variable; again, just the opposite of the other packed instructions. The fact that this instruction’s operands behave differently than the PCMPGTB instruction may create some problems. So you will have to carefully consider whether you really want to use this scheme to create a PCMPLTB “instruction” for use in your programs. If you decide to do this, it would help tremendously if you always commented each invocation of the macro to point out that the first operand is the destination operand, e.g.,

```
pcmpltb( mm0, mm1 ); // Computes mm0 := mm1<mm0!
```

If the fact that the packed comparison instruction’s operands are reversed bothers you, you can also use macros to swap those operands. The following example demonstrates how to write such macros for the PEQB (PCMPEQB), PGTB (PCMPGTB), and PLTB (packed less than, byte) instructions.

```
macro peqb( leftOp, rightOp );

    pcmpeqb( rightOp, leftOp );

endmacro;

macro pgtb( leftOp, rightOp );

    cmpgtb( rightOp, leftOp );

endmacro;

macro pltb( leftOp, rightOp );

    cmpgtb( leftOp, rightOp );

endmacro;
```

Note that these macros don’t solve the PLTB problem of having the wrong operand as the destination. However, these macros do compare the first operand to the second operand, just like the standard CMP instruction.

Of course, once you obtain a boolean result in an MMX register, you’ll probably want to test the results at one point or another. Unfortunately, the MMX instructions only provide a couple of ways to move comparison information in and out of the MMX processor – you can store an MMX register value into memory or you can copy 32-bits of an MMX register to a general-purpose integer register. Since the comparison instructions produce a 64-bit result, writing the destination of a comparison to memory is the easiest way to gain access to the comparison results in your program. Typically, you’d use an instruction sequence like the following:

```
pcmpeqb( mm1, mm0 ); // Compare 8 bytes in mm1 to mm0.
movq( mm0, qwordVar ); // Write comparison results to memory.
if((type boolean qwordVar )) then

    << do this if byte #0 contained true ($FF, which is non-zero). >>

endif;
if((type boolean qwordVar[1])) then

    << do this if byte #1 contained true. >>

endif;
etc.
```

11.7.6 MMX Shift Instructions

```

psllw( mem, mmi );
psllw( mmi, mmi );
psllw( imm8, mmi );

pslld( mem, mmi );
pslld( mmi, mmi );
pslld( imm8, mmi );

psllq( mem, mmi );
psllq( mmi, mmi );
psllq( imm8, mmi );

pslrw( mem, mmi );
pslrw( mmi, mmi );
pslrw( imm8, mmi );

psrld( mem, mmi );
psrld( mmi, mmi );
psrld( imm8, mmi );

pslrq( mem, mmi );
pslrq( mmi, mmi );
pslrq( imm8, mmi );

psarw( mem, mmi );
psarw( mmi, mmi );
psarw( imm8, mmi );

psard( mem, mmi );
psard( mmi, mmi );
psard( imm8, mmi );

```

The MMX shift, like the arithmetic instructions, allow you to simultaneously shift several different values in parallel. The PSLRx instructions perform a packed shift left logical operation, the PSRLRx instructions do a packed logical shift right operation, and the PSARx instruction do a packed arithmetic shift right operation. These instructions operate on word, double word, and quad word operands. Note that Intel does not provide a version of these instructions that operate on bytes.

The first operand to these instructions specifies a shift count. This should be an unsigned integer value in the range 0..15 for word shifts, 0..31 for double word operands, and 0..63 for quadword operands. If the shift count is outside these ranges, then these instructions set their destination operands to all zeros. If the count (first) operand is not an immediate constant, then it must be an MMX register or a *qword* memory location.

The PSLW instruction simultaneously shifts the four words in the destination MMX register to the left one bit position. The instruction shifts zero into the L.O. bit of each word and the bit shifted out of the H.O. bit of each word is lost. There is no carry from one word to the other (since that would imply a larger shift operation). This instruction, like all the other MMX instructions, does not affect the FLAGS register (including the carry flag).

The PSLD instruction simultaneously shifts the two double words in the destination MMX register to the left one bit position. Like the PSLW instruction, this instruction shifts zeros into the L.O. bits and any bits shifted out of the H.O. positions are lost.

The PSLQ is one of the few MMX instructions that operates on 64-bit quantities. This instruction shifts the entire 64-bit destination register to the left the number of bits specified by the count operand. In addition to allowing you to manipulate 64-bit integer quantities, this instruction is especially useful for moving data around in MMX registers so you can pack or unpack data as needed.

Although there is no PSLLB instruction to shift bits, you can simulate this instruction using a PSLLW and a PANDN instruction. After shifting the word values to the left the specified number of bits, all you've got to do is clear the L.O. n bits of each byte, where n is the shift count. For example, to shift the bytes in MM0 to the left three positions you could use the following two instructions:

```
static
ThreeBitsZero: byte: nostorage;
    byte $F8, $F8, $F8, $F8, $F8, $F8, $F8, $F8;
    .
    .
    .
    psllw( 3, mm0 );
    pandn( ThreeBitsZero, mm0 );
```

The PSLRW, PSLRD, and PSLRQ instructions work just like their left shift counterparts except that these instructions shift their operands to the right rather than to the left. They shift zeros into the vacated H.O. positions of the destination values and bits they shift out of the L.O. bits are lost. As with the shift left instructions, there is no PSLRB instruction but you can easily simulate this with a PSLRW and a PANDN instruction.

The PSARW and PSARD instructions do an arithmetic shift right operation on the words or double words in the destination MMX register. Note that there isn't a PSARQ instruction. While shifting data to the right, these instructions replicate the H.O. bit of each word, double word, or quad word rather than shifting in zeros. As for the logical shift right instructions, bits that these instructions shift out of the L.O. bits are lost forever.

The PSLLQ and PSLRQ instructions provide a convenient way to shift a quad word to the left or right. However, the MMX shift instructions are not generally useful for extended precision shifts since all data shifted out of the operands is lost. If you need to do an extended precision shift other than 64 bits, you should stick with the SHLD and SHRD instructions. The MMX shift instructions are mainly useful for shifting several values in parallel or (PSLLQ and PSLRQ) repositioning data in an MMX register.

11.8 The EMMS Instruction

```
emms();
```

The EMMS (Empty MMX Machine State) instruction restores the FPU status on the CPU so that it can begin processing FPU instructions again after an MMX instruction sequence. You should always execute the EMMS instruction once you complete some MMX sequence. Failure to do so may cause any floating point instructions to fail.

When an MMX instruction executes, the floating point tag word is marked valid (00s). Subsequent floating-point instructions that will be executed may produce unexpected results because the floating-point stack seems to contain valid data. The EMMS instruction marks the floating point tag word as empty. This must occur before the execution of any following floating point instructions.

Of course, you don't have to execute the EMMS instruction immediately after an MMX sequence if you're going to execute some additional MMX instructions prior to executing any FPU instructions, but you must take care to execute this instruction if

- You call any library routines or Windows APIs (that might possibly use the FPU).
- You switch tasks in a cooperative fashion (for example, see the chapter on Coroutines in the Volume on Advanced Procedures).
- You execute any FPU instructions.

If the EMMS instruction is not used when trying to execute a floating-point instruction, the following may occur:

- Depending on the exception mask bits of the floating-point control word, a floating point exception event may be generated.

- A “soft exception” may occur. In this case floating-point code continues to execute, but generates incorrect results.

The EMMS instruction is rather slow, so you don’t want to unnecessarily execute it, but it is critical that you execute it at the appropriate times. Of course, better safe than sorry; if you’re not sure you’re going to execute more MMX instructions before any FPU instructions, then go ahead and execute the EMMS instruction to clear the state.

11.9 The MMX Programming Paradigm

In general, you don’t learn scalar (non-MMX) 80x86 assembly language programming and then use that same mindset when writing programs using the MMX instruction set. While it is possible to directly use various MMX instructions the same way you would the general purpose integer instructions, one phrase comes to mind when working with MMX: think parallel. This text has spent many hundreds of pages up to this point attempting to get you to think in assembly language; to think that this small section can teach you how to design optimal MMX sequence would be ludicrous. Nonetheless, a few simple examples are useful to help start you thinking about how to use the MMX instructions to your benefit in your programs. This section will begin by presenting some fairly obvious uses for the MMX instruction set, and then it will attempt to present some examples that exploit the inherent parallelism of the MMX instructions.

Since the MMX registers are 64-bits wide, you can double the speed of certain data movement operations by using MMX registers rather than the 32-bit general purpose registers. For example, consider the following code from the HLA Standard Library that copies one character set object to another:

```
procedure cs.cpy( src:cset; var dest:cset ); nodisplay;
begin cpy;

    push( eax );
    push( ebx );
    mov( dest, ebx );
    mov( (type dword src), eax );
    mov( eax, [ebx] );
    mov( (type dword src[4]), eax );
    mov( eax, [ebx+4] );
    mov( (type dword src[8]), eax );
    mov( eax, [ebx+8] );
    mov( (type dword src[12]), eax );
    mov( eax, [ebx+12] );
    pop( ebx );
    pop( eax );

end cpy;
```

Program 11.2 HLA Standard Library cs.cpy Routine

This is a relatively simple code sequence. Indeed, a fair amount of the execution time is spent copying the parameters (20 bytes) onto the stack, calling the routine, and returning from the routine. This entire sequence can be reduced to the following four MMX instructions:

```
movq( (type qword src), mm0 );
movq( (type qword src[8]), mm1 );
movq( mm0, (type qword dest) );
movq( mm1, (type qword dest[8]) );
```

Of course, this sequence assumes two things: (1) it's okay to wipe out the values in MM0 and MM1, and (2) you'll execute the EMMS instruction a little later on after the execution of some other MMX instructions. If either, or both, of these assumptions is incorrect, the performance of this sequence won't be quite as good (though probably still better than the *cs.cpy* routine). However, if these two assumptions do hold, then it's relatively easy to implement the *cs.cpy* routine as an in-line function (i.e., a macro) and have it run much faster. If you really need this operation to occur inside a procedure and you need to preserve the MMX registers, and you don't know if any MMX instructions will execute shortly thereafter (i.e., you'll need to execute EMMS), then it's doubtful that using the MMX instructions will help here. However, in those cases when you can put the code in-line, using the MMX instructions will be faster.

Warning: don't get too carried away with the MMX MOVQ instruction. Several programmers have gone to great extremes to use this instruction as part of a high performance MOVSD replacement. However, except in very special cases on very well designed systems, the limit factor for a block move is the speed of memory. Since Intel has optimized the operation of the MOVSD instruction, you're best off using the MOVSD instructions when moving blocks of memory around.

Earlier, this chapter used the *cs.difference* function as an example when discussing the PANDN instruction. Here's the HLA Standard Library implementation of this function:

```

procedure cs.difference( src:cset; var dest:cset ); nodisplay;
begin difference;

    push( eax );
    push( ebx );
    mov( dest, ebx );
    mov( (type dword src), eax );
    not( eax );
    and( eax, [ebx] );
    mov( (type dword src[4]), eax );
    not( eax );
    and( eax, [ebx+4] );
    mov( (type dword src[8]), eax );
    not( eax );
    and( eax, [ebx+8] );
    mov( (type dword src[12]), eax );
    not( eax );
    and( eax, [ebx+12] );
    pop( ebx );
    pop( eax );

end difference;

```

Program 11.3 HLA Standard Library cs.difference Routine

Once again, the high-level nature of HLA is hiding the fact that calling this function is somewhat expensive. A typical call to *cs.difference* emits five or more instructions just to push the parameters (it takes four 32-bit PUSH instructions to pass the *src* character set because it is a value parameter). If you're willing to wipe out the values in MM0 and MM1, and you don't need to execute an EMMS instruction right away, it's possible to compute the set difference with only six instructions – that's about the same number of instructions (and often fewer) than are needed to call this routine, much less do the actual work. Here are those six instructions:

```

movq( dest, mm0 );
movq( dest[8], mm1 );
pandn( src, mm0 );
pandn( src[8], mm1 );

```

```
movq( mm0, dest );
movq( mm1, dest[8] );
```

These six instructions replace 12 of the instructions in the body of the function. The sequence is sufficiently short that it's reasonable to code it in-line rather than in a function. However, were you to bury this code in the `cs.difference` routine, and you needed to preserve MM0 and MM1⁴, and you needed to execute EMMS afterwards, this would cost more than it's worth. As an in-line macro, however, it is going to be significantly faster since it avoids passing parameters and the call/return sequence.

If you want to compute the intersection of two character sets, the instruction sequence is identical to the above except you substitute PAND for PANDN. Similarly, if you want to compute the union of two character sets, use the code sequence above substituting POR for PANDN. Again, both approaches pay off handsomely if you insert the code in-line rather than burying it in a procedure and you don't need to preserve MMX registers or execute EMMS afterwards.

We can continue with this exercise of working our way through the HLA Standard Library character set (and other) routines substituting MMX instructions in place of standard integer instructions. As long as we don't need to preserve the MMX machine state (i.e., registers) and we don't have to execute EMMS, most of the character set operations will be short enough to code in-line. Unfortunately, we're not buying that much over code the standard implementations of these functions in-line from a performance point of view (though the code would be quite a bit shorter). The problem here is that we're not "thinking in MMX." We're still thinking in scalar (non-parallel mode) and the fact that the MMX instruction set requires a lot of set-up (well, "tear-down" actually) negates many of the advantages of using MMX instructions in our programs.

The MMX instructions truly shine when you compute multiple results in parallel. The problem with the character set examples above is that we're not even processing a whole data object with a single instruction; we're actually only processing a half of a character set with a sequence of three MMX instructions (i.e., it requires six instructions to compute the intersection, union, or difference of two character sets). At best, we can only expect the code to run about twice as fast since we're processing 64 bits at a time instead of 32 bits. Executing EMMS (and, God help us, having to preserve MMX registers) negates much of what we might gain by using the MMX instructions. Again, we're only going to see a speed improvement if we process multiple objects with a single MMX instruction. We're not going to do that manipulating large objects like character sets.

One data type that will let us easily manipulate up to eight objects at one time is a character string. We can speed up many character string operations by operating on eight characters in the string at one time. Consider the HLA Standard Library *str.uppercase* procedure. This function steps through each character of a string, tests to see if it's a lower case character, and if so, converts the lower case character to upper case. A good question to ask is "can we process eight characters at a time using the MMX instructions?" The answer turns out to be yes and the MMX implementation of this function provides an interesting perspective on writing MMX code.

At first glance it might seem impractical to use the MMX instructions to test for lower case characters and convert them to upper case. Consider the typical scalar approach that tests and converts a single character at a time:

```
<< Get character to convert into the AL register >>

    cmp( al, 'a' );
    jb noConversion;
    cmp( al, 'z' );
    ja noConversion;
    sub( $20, al );    // Could also use AND($5f, al); here.
noConversion:
```

This code first checks the value in AL to see if it's actually a lower case character (that's the CMP and Jcc instructions in the code above). If the character is outside the range 'a'..'z' then this code skips over the conversion (the SUB instruction); however, if the code is in the specified range, then the sequence above drops through to the SUB instruction and converts the lower case character to upper case by subtracting \$20 from

4. Actually, the code could be rewritten easily enough to use only one MMX register.

the lower case character's ASCII code (since lower case characters always have bit #5 set, subtracting \$20 always clears this bit).

Any attempt to convert this code directly to an MMX sequence is going to fail. Comparing and branching around the conversion instruction only works if you're converting one value at a time. When operating on eight character simultaneously, any mixture of the eight characters may or may not require conversion from lower case to upper case. Hence, we need to be able to perform some calculation that is benign if the character is not lower case (i.e., doesn't affect the character's value) while converting the character to upper case if it was lower case to begin with. Worse, we have to do this with pure computation since flow of control isn't going to be particularly effective here (if we test each individual result in our MMX register we won't really save anything over the scalar approach). To save you some suspense, yes, such a calculation does exist.

Consider the following algorithm that converts lower case characters to upper case:

```
<< Get character to test into AL >>
cmp( al, 'a' );
setae( bl );      // bl := al >= 'a'
cmp( al, 'z' );
setbe( bh );      // bh := al <= 'z'
and( bh, bl );    // bl := (al >= 'a') && (al <= 'z' );
dec( bl );        // bl := $FF/$00 if false/true.
not( bl );        // bl := $FF/$00 if true/false.
and( $20, bl );   // bl := $20/$00 if true/false.
sub( bl, al );    // subtract $20 if al was lowercase.
```

This code sequence is fairly straight-forward up until the DEC instruction above. It computes true/false in BL depending on whether AL is in the range 'a'..'z'. At the point of the DEC instruction, BL contains one if AL is a lower case character, it contains zero if AL's value is not lower case. After the DEC instruction, BL contains \$FF for false (AL is not lower case) and \$00 for true (AL is lowercase). The code is going to use this as a mask a little later, but it really needs true to be \$FF and false \$00, hence the NOT instruction that follows. The (second) AND instruction above converts true to \$20 and false to \$00 and the final SUB instruction subtracts \$20 if AL contained lower case, it subtracts \$00 from AL if AL did not contain a lower case character (subtracting \$20 from a lower case character will convert it to upper case).

Whew! This sequence probably isn't very efficient when compared to the simpler code given previously. Certainly there are more instructions in this version (nearly twice as many). Whether this code without any branches runs faster or slower than the earlier code with two branches is a good question. The important thing to note here, though, is that we converted the lower case characters to upper case (leaving other characters unchanged) using only a calculation; no program flow logic is necessary. This means that the code sequence above is a good candidate for conversion to MMX. Even if the code sequence above is slower than the previous algorithm when converting one character at a time to upper case, it's positively going to scream when it converts eight characters at a shot (since you'll only need to execute the sequence one-eighth as many times).

The following is the code sequence that will convert the eight characters starting at location [EDI] in memory to upper case:

```
data
A:qword;
    byte $60, $60, $60, $60, $60, $60, $60, $60; // Note: $60 = 'a'-1.
Z:qword;
    byte $7B, $7B, $7B, $7B, $7B, $7B, $7B, $7B; // Note: $7B = 'z' + 1.
ConvFactor:qword;
    byte $20, $20, $20, $20, $20, $20, $20, $20; // Magic value for lc->UC.
    .
    .
    .
    movq( ConvFactor, mm4 ); // Eight copies of conversion value.
    movq( A, mm2 );         // Put eight "a" characters in mm2.
    movq( Z, mm3 );         // Put eight "z" characters in mm3.
    movq( [edi], mm0 );     // Get next eight characters of our string.
```

```

movq( mm0, mm1 );    // We need two copies.
pcmpgtb( mm2, mm1 ); // Generate 1's in MM1 everywhere chars >= 'a'
pcmpgtb( mm0, mm3 ); // Generate 1's in MM3 everywhere chars <= 'z'
pand( mm3, mm1 );    // Generate 1's in MM1 when 'a'<=chars<='z'
pand( mm4, mm1 );    // Generates $20 in each spot we have a l.c. char
psubb( mm1, mm0 );   // Convert l.c. chars to U.C. by adding $20.
movq( mm0, [edi]);

```

Note that this code compares the characters that [EDI] points at to 'a'-1 and 'z'+1 because we only have a greater than comparison rather than a greater or equal comparison (this saves a few extra instructions). Other than setting up the MMX registers and taking advantage of the fact that the PCMPGTB instructions automatically produce \$FF for true and \$00 for false, this is a faithful reproduction of the previous algorithm except it operates on eight bytes simultaneously. So if we put this code in a loop and execute it once for each eight characters in the string, there will be one-eighth the iterations of a similar loop using the scalar instructions.

Of course, there is one problem with this code. Not all strings have lengths that are an even multiple of eight bytes. Therefore, we've got to put some special case code into our algorithm to handle strings that are less than eight characters long and handle strings whose length is not an even multiple of eight characters. In the following program, the `mmxupper` function simply borrows the scalar code from the HLA Standard Library's `str.upper` procedure to handle the leftover characters. The following example program provides both an MMX and a scalar solution with a main program that compares the running time of both. If you're wondering, the MMX version is about three times faster (on a Pentium III) for strings around 35 characters long, containing mostly lower case (mostly lower case favors the scalar algorithm since fewer branches are taken with lower case characters; longer strings favor the MMX algorithm since it spends more time in the MMX code compared to the scalar code at the end).

```

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

// The following code was stolen from the
// HLA Standard Library's str.upper function.
// It is not optimized, but then none of this
// code is optimized other than to use the MMX
// instruction set (later).

procedure strupper( dest: string ); nodisplay;
begin strupper;

    push( edi );
    push( eax );

    mov( dest, edi );
    if( edi = 0 ) then

        raise( ex.AttemptToDerefNULL );

    endif;

    // Until we encounter a zero byte, convert any lower
    // case characters to upper case.

    forever

        mov( [edi], al );
        breakif( al = 0 );    // Quit when we find a zero byte.

        // If a lower case character, convert it to upper case

```

```

        // and store the result back into the destination string.

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

        and( $5f, al );    // Magic lc->UC translation.
        mov( al, [edi] );  // Save result.

    endif;

    // Move on to the next character.

    inc( edi );

endfor;

pop( edi );
pop( eax );

end strupper;

```

```

procedure mmxupper( dest: string ); nodisplay;
const
    zCh:char := char( uns8( 'z' ) + 1 );
    aCh:char := char( uns8( 'a' ) - 1 );

data

    // Create eight copies of the A-1 and Z+1 characters
    // so we can compare eight characters at once:

    A:qword;
    byte aCh, aCh, aCh, aCh, aCh, aCh, aCh, aCh;

    Z:qword;
    byte zCh, zCh, zCh, zCh, zCh, zCh, zCh, zCh;

    // Conversion factor: UC := LC - $20.

    ConvFactor: qword;
    byte $20, $20, $20, $20, $20, $20, $20, $20;

begin mmxupper;

    push( edi );
    push( eax );

    mov( dest, edi );
    if( edi = 0 ) then

        raise( ex.AttemptToDerefNULL );

    endif;

```

```

// Some invariant operations (things that don't
// change on each iteration of the loop):

movq( A, mm2 );
movq( ConvFactor, mm4 );

// Get the string length from the length field:

mov( (type str.strRec [edi]).length, eax );

// Process the string in blocks of eight characters:

while( (type int32 eax) >= 8 ) do

    movq( [edi], mm0 );    // Get next eight characters of our string.
    movq( mm0, mm1 );     // We need two copies.
    movq( Z, mm3 );       // Need to refresh on each loop.
    pcmptgb( mm2, mm1 );  // Generate 1's in MM1 everywhere chars >= 'a'
    pcmptgb( mm0, mm3 );  // Generate 1's in MM3 everywhere chars <= 'z'
    pand( mm3, mm1 );     // Generate 1's in MM1 when 'a'<=chars<='z'
    pand( mm4, mm1 );     // Generates $20 in each spot we have a l.c. char
    psubb( mm1, mm0 );    // Convert l.c. chars to U.C. by adding $20.
    movq( mm0, (type qword [edi]));

    // Move on to the next eight characters in the string.

    sub( 8, eax );
    add( 8, edi );

endwhile;

// If we're processing less than eight characters, do it the old-fashioned
// way (one character at a time). This also handles the last 1..7 chars
// if the number of characters is not an even multiple of eight. This
// code was swiped directly from the HLA str.upper function (above).

if( eax != 0 ) then

    forever

        mov( [edi], al );
        breakif( al = 0 );    // Quit when we find a zero byte.

        // If a lower case character, convert it to upper case
        // and store the result back into the destination string.

        if
        {
            cmp( al, 'a' );
            jnb false;
            cmp( al, 'z' );
            ja false;
        }

        and( $5f, al );    // Magic lc->UC translation.
        mov( al, [edi] );  // Save result.

    endif;

    // Move on to the next character.

```

```

        inc( edi );

    endfor;

endif;
emms(); // Clean up MMX state.

pop( edi );
pop( eax );

end mmxupper;

static
    MyStr: string := "Hello There, MMX Uppercase Routine!";
    destStr:string;
    mmxCycles:qword;
    strCycles:qword;

begin UpperCase;

    // Charge up the cache (prefetch the code and data
    // to avoid cache misses later).

    mov( str.a_cpy( MyStr ), destStr );
    mmxupper( destStr );
    strupper( destStr );

    // Okay, time the execution of the MMX version:

    mov( str.a_cpy( MyStr ), destStr );

    rdtsc();
    mov( eax, (type dword mmxCycles));
    mov( edx, (type dword mmxCycles[4]));
    mmxupper( destStr );
    rdtsc();
    sub( (type dword mmxCycles), eax );
    sbb( (type dword mmxCycles[4]), edx );
    mov( eax, (type dword mmxCycles));
    mov( edx, (type dword mmxCycles[4]));

    stdout.put( "Dest String = '", destStr, "'", nl );

    // Okay, time the execution of the HLA version:

    mov( str.a_cpy( MyStr ), destStr );

    rdtsc();
    mov( eax, (type dword strCycles));
    mov( edx, (type dword strCycles[4]));
    strupper( destStr );
    rdtsc();
    sub( (type dword strCycles), eax );
    sbb( (type dword strCycles[4]), edx );
    mov( eax, (type dword strCycles));

```

```

mov( edx, (type dword strCycles[4]));

stdout.put( "Dest String(2) = '", destStr, "'", nl );

stdout.put( "MMX cycles:" );
stdout.puti64( mmxCycles );
stdout.put( nl "HLA cycles: " );
stdout.puti64( strCycles );
stdout.newln();

end UpperCase;

```

Program 11.4 MMX Implementation of the HLA Standard Library str.upper Procedure

Other string functions, like a case insensitive string comparison, can greatly benefit from the use of parallel computation via the MMX instruction set. Implementation of other string functions is left as an exercise to the reader; interested readers should consider converting string functions that involve calculations and tests on each individual characters in a string as candidates for optimization via MMX.

11.10 Putting It All Together

Intel's MMX enhancements to the basic Pentium instruction set allow the acceleration of certain algorithms. Unfortunately, the MMX instruction set isn't generally applicable to a wide range of problems. The MMX instructions, with their SIMD orientation, are generally useful for manipulating a large amount of data organized as byte, word, or double word arrays where the MMX instructions can calculate several values in parallel. Learning to effectively use the MMX instruction set requires a paradigm shift on the part of the programmer. You don't apply the same rules for scalar 80x86 instructions to the MMX instructions. However, if you take the time to master parallel programming techniques with the MMX instructions, then you will be able to accelerate many of your applications.

Mixed Language Programming

Chapter Twelve

12.1 Chapter Overview

Most assembly language code doesn't appear in a stand-alone assembly language program. Instead, most assembly code is actually part of a library package that programs written in a high level language wind up calling. Although HLA makes it really easy to write standalone assembly applications, at one point or another you'll probably want to call an HLA procedure from some code written in another language or you may want to call code written in another language from HLA. This chapter discusses the mechanisms for doing this in three languages: MASM (low-level assembly), C/C++, and Delphi. The mechanisms for other languages are usually similar to one of these three, so the material in this chapter will still apply even if you're using some other high level language.

12.2 Mixing HLA and MASM Code in the Same Program

It may seem kind of weird to mix MASM and HLA code in the same program. After all, they're both assembly languages and almost anything you can do with MASM can be done in HLA. So why bother trying to mix the two in the same program? Well, there are two reasons:

- You've already got a lot of code written in MASM and you don't want to convert it to HLA's syntax.
- There are a few things MASM does that HLA cannot, and you happen to need to do one of those things.

In this section, we'll discuss two ways to merge MASM and HLA code in the same program: via in-line assembly code and through linking OBJ files.

12.2.1 In-Line (MASM) Assembly Code in Your HLA Programs

As you're probably aware, the HLA compiler doesn't actually produce machine code directly from your HLA source files. Instead, it first compiles the code to a MASM-compatible assembly language source file and then it calls MASM to assemble this code to object code. If you're interested in seeing the MASM output HLA produces, just edit the *filename.ASM* file that HLA creates after compiling your *filename.HLA* source file. The output assembly file isn't amazingly readable, but it is fairly easy to correlate the assembly output with the HLA source file.

HLA provides two mechanisms that let you inject raw MASM code directly into the output file it produces: the `#ASM..#ENDASM` sequence and the `#EMIT` statement. The `#ASM..#ENDASM` sequence copies all text between these two clauses directly to the MASM output file, e.g.,

```
#asm

    mov eax, 0          ;MASM syntax for MOV( 0, EAX );
    add eax, ebx        ; "      "      " ADD( ebx, eax );

#endasm
```

The `#ASM..#ENDASM` sequence is how you inject in-line (MASM) assembly code into your HLA programs. For the most part there is very little need to use this feature, but in a few instances it is valuable. For example, if you're writing structured exception handling code under Windows, you'll need to access the double word at address `FS:[0]` (offset zero in the segment pointed at by the 80x86's FS segment register). Unfortunately, HLA does not support segmentation and the use of segment registers. However, you can drop into MASM for a statement or two in order to access this value:

```
#asm
    mov ebx, fs:[0]      ; Loads process pointer into EBX
#endasm
```

At the end of this instruction sequence, EBX will contain the pointer to the process information structure that Windows maintains.

HLA blindly copies all text between the #ASM and #ENDASM clauses directly to the MASM output file. HLA does not check the syntax of this code or otherwise verify its correctness. If you introduce an error within this section of your program, MASM will report the error when HLA assembles your code by calling MASM.

The #EMIT statement also writes text directly to the assembly output file. However, this statement does not simply copy the text from your source file to the output file; instead, this statement copies the value of a string (constant) expression to the output file. The syntax for this statement is as follows:

```
#emit( string_expression );
```

This statement evaluates the expression and verifies that it's a string expression. Then it copies the string data to the output file. Like the #ASM/#ENDASM statement, the #EMIT statement does not check the syntax of the MASM statement it writes to the assembly file. If there is a syntax error, MASM will catch it later on when HLA assembles the output file.

When HLA compiles your programs into assembly language, it does not use the same symbols in the assembly language output file that you use in the HLA source files. There are several technical reasons for this, but the bottom line is this: you cannot easily reference your HLA identifiers in your in-line assembly code. The only exception to this rule are external identifiers. HLA external identifiers use the same name in the MASM file as in the HLA source file. Therefore, you can refer to external objects within your in-line assembly sequences or in the strings you output via #EMIT.

One advantage of the #EMIT statement is that it lets you construct MASM statements under (compile-time) program control. You can write an HLA compile-time program that generates a sequence of strings and emits them to the MASM file via the #EMIT statement. The compile-time program has access to the HLA symbol table; this means that you can extract the identifiers that HLA emits to the assembly file and use these directly, even if they aren't external objects.

The @StaticName compile-time function returns the name that HLA uses to refer to most static objects in your program. The following program demonstrates a simple use of this compile-time function to obtain the MASM name of an HLA procedure:

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

    procedure myProc;
    begin myProc;

        stdout.put( "Inside MyProc" nl );

    end myProc;

begin emitDemo;

    ?stmt:string := "call " + @StaticName( myProc );
    #emit( stmt );

end emitDemo;
```

Program 12.1 Using the @StaticName Function

This example creates a string value (stmt) that contains something like “call ?741_myProc” and emits this MASM instruction directly to the source file (“?741_myProc” is typical of the type of name mangling that HLA does to static names it writes to the output file). If you compile and run this program, it should display “Inside MyProc” and then quit. If you look at the MASM file that HLA emits, you will see that it has given the *myProc* procedure the same name it appends to the CALL instruction¹.

The @StaticName function is only valid for static symbols. This includes STATIC, DATA, READ-ONLY, and STORAGE variables, procedures, and iterators. It does not include VAR objects, constants, macros, or methods.

You can access VAR variables by using the [EBP+offset] addressing mode, specifying the offset of the desired local variable. You can use the @offset compile-time function to obtain the offset of a VAR object or a parameter. The following program demonstrates how to do this:

```

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

var
    i:int32;

begin offsetDemo;

    mov( -255, i );
    ?stmt := "mov eax, [ebp+(\" + string( @offset( i )) + \")]";
    #print( "Emitting '", stmt, "'" );
    #emit( stmt );
    stdout.put( "eax = ", (type int32 eax), nl );

end offsetDemo;
```

Program 12.2 Using the @Offset Compile-Time Function

This example emits the statement “mov eax, [ebp+(-8)]” to the assembly language source file. It turns out that -8 is the offset of the *i* variable in the offsetDemo program’s activation record.

Of course, the examples of #EMIT up to this point have been somewhat ridiculous since you can achieve the same results by using HLA statements. One very useful purpose for the #emit statement, however, is to create some instructions that HLA does not support. For example, as of this writing HLA does not support the LES instruction because you can’t really use it under Win32. However, if you found a need for this instruction, you could easily write a macro to emit this instruction and appropriate operands to the MASM source file. Using the #EMIT statement gives you the ability to reference HLA objects, something you cannot do with the #ASM..#ENDASM sequence.

12.2.2 Linking MASM-Assembled Modules with HLA Modules

Although you can do some interesting things with HLA’s in-line assembly statements, you’ll probably never use them. Further, future versions of HLA may not even support these statements, so you should avoid them as much as possible even if you see a need for them. Of course, HLA does most of the stuff you’d want to do with the #ASM/#ENDASM and #EMIT statements anyway, so there is very little reason to use them at

1. HLA may assign a different name than “?741_myProc” when you compile the program. The exact symbol HLA chooses varies from version to version of the assembler (it depends on the number of symbols defined prior to the definition of *myProc*). In this example, there were 741 static symbols defined in the HLA Standard Library before the definition of *myProc*.

all. If you're going to combine MASM (or other assembler) code and HLA code together in a program, most of the time this will occur because you've got a module or library routine written in some other assembly language and you would like to take advantage of that code in your HLA programs. Rather than convert the other assembler's code to HLA, the easy solution is to simply compile that other code to an OBJ file and link it with your HLA programs.

Once you've compiled or assembled a source file to an OBJ file, the routines in that module are callable from almost any machine code that can handle the routines' calling sequences. If you have an OBJ file that contains a SQRT function, for example, it doesn't matter whether you compiled that function with HLA, MASM, TASM, NASM, Gas, or even a high level language; if it's object code and it exports the proper symbols, you can call it from your HLA program.

Compiling a module in MASM and linking that with your HLA program is little different than linking other HLA modules with your main HLA program. In the MASM source file you will have to export some symbols (using the PUBLIC directive) and in your HLA program you've got to tell HLA that those symbols appear in a separate module (using the EXTERNAL option).

Since the two modules are written in assembly language, there is very little language imposed structure on the calling sequence and parameter passing mechanisms. If you're calling a function written in MASM from your HLA program, then all you've got to do is to make sure that your HLA program passes parameters in the same locations where the MASM function is expecting them.

About the only issue you've got to deal with is the case of identifiers in the two programs. By default, MASM is case insensitive. HLA, on the other hand, enforces case neutrality (which, essentially, means that it is case sensitive). Fortunately, there is a MASM command line option ("/Cp") that tells MASM to preserve case in all public symbols. It's a real good idea to use this option when assembling modules you're going to link with HLA so that MASM doesn't mess with the case of your identifiers during assembly.

Of course, once MASM starts processing symbols in a case sensitive manner, it's possible to create two separate identifiers that are the same except for alphabetic case. HLA enforces case neutrality so it won't let you (directly) create two different identifiers that differ only in case. In general, this is such a bad programming practice that one would hope you never encounter it (and God forbid you actually do this yourself). However, if you inherit some MASM code written by a C hacker, it's quite possible the code uses this technique. The way around this problem is to use two separate identifiers in your HLA program and use the extended form of the EXTERNAL directive to provide the external names. For example, suppose that in MASM you have the following declarations:

```

        public  AVariable
        public  avariable
        .
        .
        .
        .data
AVariable dword    ?
avariable byte     ?

```

If you assemble this code with the "/Cp" or "/Cx" (total case sensitivity) command line options, MASM will emit these two external symbols for use by other modules. Of course, were you to attempt to define variables by these two names in an HLA program, HLA would complain about a duplicate symbol definition. However, you can connect two different HLA variables to these two identifiers using code like the following:

```

static
  AVariable: dword; external( "AVariable" );
  AnotherVar: byte; external( "avariable" );

```

HLA does not check the strings you supply as parameters to the EXTERNAL clause. Therefore, you can supply two names that are the same except for case and HLA will not complain. Note that when HLA calls MASM to assemble its output file, HLA specifies the "/Cp" option that tells MASM to preserve case in public and global symbols.

The following program demonstrates how to call a MASM subroutine from an HLA main program:

```

// To compile this module and the attendant MASM file, use the following
// command line:
//
//      hla masmdemo1.hla masmupper.asm
//
// Sorry about no make file for this code, but these two files are in
// the HLA Vol4/Ch12 subdirectory that has it's own makefile for building
// all the source files in the directory and I wanted to avoid confusion.

program MasmDemo1;
#include( "stdlib.hhf" )

    // The following external declaration defines a function that
    // is written in MASM to convert the character in AL from
    // lower case to upper case.

    procedure masmUpperCase( c:char in al ); external( "masmUpperCase" );

static
    s: string := "Hello World!";

begin MasmDemo1;

    stdout.put( "String converted to uppercase: " );
    mov( s, edi );
    while( mov( [edi], al ) <> #0 ) do

        masmUpperCase( al );
        stdout.putc( al );
        inc( edi );

    endwhile;
    stdout.put( "" nl );

end MasmDemo1;

```

Program 12.3 Main HLA Program to Link with a MASM Program

```

; MASM source file to accompany the MasmDemo1.HLA source
; file. This code compiles to an object module that
; gets linked with an HLA main program. The function
; below converts the character in AL to upper case if it
; is a lower case character.

    .586
    .model flat, pascal

    .code
    public masmUpperCase
    masmUpperCase proc near32

```

```

        .if al >= 'a' && al <= 'z'
        and al, 5fh
        .endif
        ret
masmUpperCase    endp
end

```

Program 12.4 Calling a MASM Procedure from an HLA Program: MASM Module

It is also possible to call an HLA procedure from a MASM program (this should be obvious since HLA compiles its source code to a MASM source file and that MASM source file can call HLA procedures such as those found in the HLA Standard Library). There are a few restrictions when calling HLA code from some other language. First of all, you can't easily use HLA's exception handling facilities in the modules you call from other languages (including MASM). The HLA main program initializes the exception handling system; this initialization is probably not done by your MASM programs. Further, the HLA main program exports a couple of important symbols needed by the exception handling subsystem; again, it's unlikely your main MASM program provides these public symbols. In the Volume on Advanced Procedures this text will discuss how to deal with Window's Structured Exception Handling subsystem and HLA's layer on top of this. However, that topic is a little too advanced for this chapter. Until you get to the point you can write code in MASM to properly set up the HLA exception handling system, you should not execute any code that uses the TRY..ENDTRY, RAISE, or any other exception handling statements.

Warning; a large percentage of the HLA Standard Library routines include exception handling statements or call other routines that use exception handling statements. Unless you've set up the HLA exception handling subsystem properly, you should not call any HLA Standard Library routines from non-HLA programs.

Other than the issue of exception handling, calling HLA procedures from MASM code is really easy. All you've got to do is put an EXTERNAL prototype in the HLA code to make the symbol you wish to access public and then include an EXTERN (or EXTERNDEF) statement in the MASM source file to provide the linkage. Then just compile the two source files and link them together.

About the only issue you need concern yourself with when calling HLA procedures from MASM is the parameter passing mechanism. Of course, if you pass all your parameters in registers (the best place), then communication between the two languages is trivial. Just load the registers with the appropriate parameters in your MASM code and call the HLA procedure. Inside the HLA procedure, the parameter values will be sitting in the appropriate registers (sort of the converse of what happened in Program 12.4).

If you decide to pass parameters on the stack, note that HLA uses the PASCAL language calling model. Therefore, you push parameters on the stack in the order they appear in a parameter list (from left to right) and it is the called procedure's responsibility to remove the parameters from the stack. Note that you can specify the PASCAL calling convention for use with MASM's INVOKE statement using the ".model" directive, e.g.,

```

.586
.model flat, pascal
.
.
.

```

Of course, if you manually push the parameters on the stack yourself, then the specific language model doesn't really matter.

This section is not going to attempt to go into gory details about MASM syntax. There is an appendix in this text that contrasts the HLA language with MASM; you should be able to get a rough idea of MASM syntax from that appendix if you're completely unfamiliar with MASM. Another alternative is to read a copy of the DOS/16-bit edition of this text that uses the MASM assembler. That text describes MASM syntax in much greater detail, albeit from a 16-bit perspective. Finally, this section isn't going to go into any further detail because, quite frankly, the need to call MASM code from HLA (or vice versa) just isn't that

great. After all, most of the stuff you can do with MASM can be done directly in HLA so there really is little need to spend much more time on this subject. Better to move on to more important questions, like how do you call HLA routines from C or Pascal...

12.3 Programming in Delphi and HLA

Delphi is a marvelous language for writing Win32 GUI-based applications. Its support for Rapid Application Design (RAD) and visual programming is superior to almost every other Win32 programming approach available. However, being Pascal-based, there are some things that just cannot be done in Delphi and many things that cannot be done as efficiently in Delphi as in assembly language. Fortunately, Delphi lets you call assembly language procedures and functions so you can overcome Delphi's limitations.

Delphi provides two ways to use assembly language in the Pascal code: via a built-in assembler (BASM) or by linking in separately compiled assembly language modules. The built-in "Borland Assembler" (BASM) is a very weak Intel-syntax assembler. It is suitable for injecting a few instructions into your Pascal source code or perhaps writing a very short assembly language function or procedure. It is not suitable for serious assembly language programming. If you know Intel syntax and you only need to execute a few machine instructions, then BASM is perfect. However, since this is a text on assembly language programming, the assumption here is that you want to write some serious assembly code to link with your Pascal/Delphi code. To do that, you will need to write the assembly code and compile it with a different assembler (e.g., HLA) and link the code into your Delphi application. That is the approach this section will concentrate on. For more information about BASM, check out the Delphi documentation.

Before we get started discussing how to write HLA modules for your Delphi programs, you must understand two very important facts:

HLA's exception handling facilities are not directly compatible with Delphi's. This means that you cannot use the TRY..ENDTRY and RAISE statements in the HLA code you intend to link to a Delphi program. This also means that you cannot call library functions that contain such statements. Since the HLA Standard Library modules use exception handling statements all over the place, this effectively prevents you from calling HLA Standard Library routines from the code you intend to link with Delphi².

Although you can write console applications with Delphi, 99% of Delphi applications are Windows/GUI applications. You cannot call console-related functions (e.g., stdin.xxxx or stdout.xxxx) from a GUI application. Even if HLA's console and standard input/output routines didn't use exception handling, you wouldn't be able to call them from a standard Delphi application.

Given the rich set of language features that Delphi supports, it should come as no surprise that the interface between Delphi's Object Pascal language and assembly language is somewhat complex. Fortunately there are two facts that reduce this problem. First, HLA uses many of the same calling conventions as Pascal; so much of the complexity is hidden from sight by HLA. Second, the other complex stuff you won't use very often, so you may not have to bother with it.

Note: the following sections assume you are already familiar with Delphi programming. They make no attempt to explain Delphi syntax or features other than as needed to explain the Delphi assembly language interface. If you're not familiar with Delphi, you will probably want to skip this section.

12.3.1 Linking HLA Modules With Delphi Programs

The basic unit of interface between a Delphi program and assembly code is the procedure or function. That is, to combine code between the two languages you will write procedures in HLA (that correspond to

2. Note that the HLA Standard Library source code is available; feel free to modify the routines you want to use and remove any exception handling statements contained therein.

procedures or functions in Delphi) and call these procedures from the Delphi program. Of course, there are a few mechanical details you've got to worry about, this section will cover those.

To begin with, when writing HLA code to link with a Delphi program you've got to place your HLA code in an HLA UNIT. An HLA PROGRAM module contains start up code and other information that Windows uses to determine where to begin program execution when it loads an EXE file from disk. However, the Delphi program also supplies this information and specifying two starting addresses confuses the linker, therefore, you must place all your HLA code in a UNIT rather than a PROGRAM module.

Within the HLA UNIT you must create EXTERNAL procedure prototypes for each procedure you wish to call from Delphi. If you prefer, you can put these prototype declarations in a header file and #INCLUDE them in the HLA code, but since you'll probably only reference these declarations from this single file, it's okay to put the EXTERNAL prototype declarations directly in the HLA UNIT module. These EXTERNAL prototype declarations tell HLA that the associated functions will be public so that Delphi can access their names during the link process. Here's a typical example:

```
unit LinkWithDelphi;

    procedure prototype; external;

    procedure prototype;
    begin prototype;

        << Code to implement prototype's functionality >>

    end prototype;

end LinkWithDelphi;
```

After creating the module above, you'd compile it using HLA's "-s" (compile to assembly only) command line option. This will produce an ASM file. Were this just about any other language, you'd then assemble the ASM file with MASM. Unfortunately, Delphi doesn't like OBJ files that MASM produces. For all but the most trivial of assembly modules, Delphi will reject the MASM's output. Borland Delphi expects external assembly modules to be written with Borland's assembler, TASM32.EXE (the 32-bit Turbo Assembler). Unfortunately, HLA's output is intended for assembly by MASM and claims by Borland to the contrary, TASM doesn't always assemble the files that HLA produces. To solve this problem, there is a little program called "m2t.exe" (MASM to TASM) that translates HLA's output files so that you can assemble them with TASM³. The m2t.exe program uses the following command line syntax:

```
m2t filename.asm
```

The program writes the TASM-compatible output to the *filename.tas* (turbo assembler) output file.

Once you create the .TAS file, you can assemble it to an OBJ file using the TASM32.EXE program. For example, to assemble the filename.tas file in the current example you'd use the following command:

```
tasm32 filename.tas
```

Here are all the commands to compile and assemble the module given earlier:

```
hla -s LinkWithDelphi.hla
m2t LinkWithDelphi.asm
tasm32 LinkWithDelphi.tas
```

You should not attempt to use the HLA "-c" command line option to compile with HLA and assemble with MASM all in one step. As noted above, you must assemble the output with TASM32.EXE and the HLA "-c" option assembles the code using MASM.

3. For those who don't have a copy of TASM and want to use MASM to link code with Delphi, there is a section on how to accomplish this a little later in this chapter; for this most part, however, this chapter's discussion of Delphi will assume that you're using the Turbo Assembler, V4.0 or later.

Of course, if you don't like typing three commands to compile and assemble your HLA code, you can always create a make file or a batch file that will let you do both operations with a single command. See the chapter on Managing Large Programs for more details (see "Make Files" on page 557).

Once you've created the HLA code, the next step is to tell Delphi that it needs to call the HLA/assembly code. There are two steps needed to achieve this: You've got to inform Delphi that a procedure (or function) is written in assembly language (rather than Pascal) and you've got to tell Delphi to link in the OBJ file you've created when compiling the Delphi code.

The second step above, telling Delphi to include the HLA object module, is the easiest task to achieve. All you've got to do is insert a compiler directive of the form "{ \$L *objectFileName.obj* }" to the Delphi program before declaring and calling your object module. A good place to put this is after the **implementation** reserved word in the module that calls your assembly procedure. The code examples a little later in this section will demonstrate this.

The next step is to tell Delphi that you're supplying an external procedure or function. This is done using the Delphi EXTERNAL directive on a procedure or function prototype. For example, a typical external declaration for the *prototype* procedure appearing earlier is

```
procedure prototype; external; // This may look like HLA code, but it's
                             // really Delphi code!
```

As you can see here, Delphi's syntax for declaring external procedures is nearly identical to HLA's (in fact, in this particular example the syntax is identical). This is not an accident, much of HLA's syntax was borrowed directly from Pascal.

The next step is to call the assembly procedure from the Delphi code. This is easily accomplished using standard Pascal procedure calling syntax. The following two listings provide a complete, working, example of an HLA procedure that a Delphi program can call. This program doesn't accomplish very much other than to demonstrate how to link in an assembly procedure. The Delphi program contains a form with a single button on it. Pushing the button calls the HLA procedure, whose body is empty and therefore returns immediately to the Delphi code without any visible indication that it was ever called. Nevertheless, this code does provide all the syntactical elements necessary to create and call an assembly language routine from a Delphi program.

```
unit LinkWithDelphi;

    procedure CalledFromDelphi; external; // Just to make this symbol public

    procedure CalledFromDelphi;
    begin CalledFromDelphi;
    end CalledFromDelphi;

end LinkWithDelphi;
```

Program 12.5 CalledFromDelphi.HLA Module Containing the Assembly Code

```
unit DelphiEx1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls;
```

```

type
  TDelphiEx1Form = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  DelphiEx1Form: TDelphiEx1Form;

implementation

{$R *.DFM}

// Add the following statement in order to link the HLA file with this\
// Delphi code:

{$L CalledFromDelphi.obj }

// Add the following external declaration so you can call
// the assembly procedure from this Delphi program:

procedure CalledFromDelphi; external;

// The Delphi program will call the following procedure whenever
// you push the button on the form.  This procedure, in turn,
// calls the "CalledFromDelphi" procedure we wrote in assembly language.

procedure TDelphiEx1Form.Button1Click(Sender: TObject);
begin
    CalledFromDelphi();
end;

end.

```

Program 12.6 DelphiEx1– Delphi Source Code that Calls an Assembly Procedure

The full Delphi and HLA source code for the programs appearing in Program 12.5 and Program 12.6 accompanies the HLA software distribution in the appropriate subdirectory for this chapter in the Example code module. If you've got a copy of Delphi 5 or later, you might want to load this module and try compiling it. To compile the HLA code for this example, you would use the following commands from the command prompt:

```

hla -s CalledFromDelphi.hla
m2t CalledFromDelphi.asm
tasm32 CalledFromDelphi.tas

```

After producing the CalledFromDelphi object module with the two commands above, you'd enter the Delphi Integrated Development Environment and tell it to compile the DelphiEx1 code (i.e., you'd load the DelphiEx1Project file into Delphi and compile the code). This process automatically links in the HLA code and when you run the program you can call the assembly code by simply pressing the single button on the Delphi form.

12.3.2 Register Preservation

Delphi code expects all procedures to preserve the EBX, ESI, EDI, and EBP registers. Routines written in assembly language may freely modify the contents of EAX, ECX, and EDX without preserving their values. The HLA code will have to modify the ESP register to remove the activation record (and, possibly, some parameters). Of course, HLA procedures (unless you specify the NOFRAME option) automatically preserve and set up EBP for you, so you don't have to worry about preserving this register's value; of course, you will not usually manipulate EBP's value since it points at your procedure's parameters and local variables.

Although you can modify EAX, ECX, and EDX to your heart's content and not have to worry about preserving their values, don't get the idea that these registers are available for your procedure's exclusive use. In particular, Delphi may pass parameters into a procedure within these registers and you may need to return function results in some of these registers. Details on the further use of these registers appears in later sections of this chapter.

Whenever Delphi calls a procedure, that procedure can assume that the direction flag is clear. On return, all procedures must ensure that the direction flag is still clear. So if you manipulate the direction flag in your assembly code (or call a routine that might set the direction flag), be sure to clear the direction flag before returning to the Delphi code.

If you use any MMX instructions within your assembly code, but sure to execute the EMMS instruction before returning. Delphi code assumes that it can manipulate the floating point stack without running into problems. Note that many versions of TASM do not support the MMX instruction set. So use this with care.

Although the Delphi documentation doesn't explicitly state this, experiments with Delphi code seem to suggest that you don't have to preserve the FPU (or MMX) registers across a procedure call other than to ensure that you're in FPU mode (versus MMX mode) upon return to Delphi.

12.3.3 Function Results

Delphi generally expects functions to return their results in a register. For ordinal return results, a function should return a byte value in AL, a word value in AX, or a double word value in EAX. Functions return pointer values in EAX. Functions return real values in ST0 on the FPU stack. The code example in this section demonstrates each of these parameter return locations.

For other return types (e.g., arrays, sets, records, etc.), Delphi generally passes an extra VAR parameter containing the address of the location where the function should store the return result. We will not consider such return results in this text, see the Delphi documentation for more details.

The following Delphi/HLA program demonstrates how to return different types of scalar (ordinal and real) parameters to a Delphi program from an assembly language function. The HLA functions return boolean (one byte) results, word results, double word results, a pointer (PChar) result, and a floating point result when you press an appropriate button on the form. See the DelphiEx2 example code in the HLA/Art of Assembly examples code for the full project. Note that the following code doesn't really do anything useful other than demonstrate how to return Function results in EAX and ST0.

```

unit DelphiEx2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TDelphiEx2Form = class(TForm)
    BoolBtn: TButton;
  end;

```

```

    BooleanLabel: TLabel;
    WordBtn: TButton;
    WordLabel: TLabel;
    DWordBtn: TButton;
    DWordLabel: TLabel;
    PtrBtn: TButton;
    PCharLabel: TLabel;
    FltBtn: TButton;
    RealLabel: TLabel;
    procedure BoolBtnClick(Sender: TObject);
    procedure WordBtnClick(Sender: TObject);
    procedure DWordBtnClick(Sender: TObject);
    procedure PtrBtnClick(Sender: TObject);
    procedure FltBtnClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    DelphiEx2Form: TDelphiEx2Form;

implementation

{$R *.DFM}

// Here's the directive that tells Delphi to link in our
// HLA code.

{$L ReturnBoolean.obj }
{$L ReturnWord.obj }
{$L ReturnDWord.obj }
{$L ReturnPtr.obj }
{$L ReturnReal.obj }

// Here are the external function declarations:

function ReturnBoolean:boolean; external;
function ReturnWord:smallint; external;
function ReturnDWord:integer; external;
function ReturnPtr:pchar; external;
function ReturnReal:real; external;

// Demonstration of calling an assembly language
// procedure that returns a byte (boolean) result.

procedure TDelphiEx2Form.BoolBtnClick(Sender: TObject);
var
    b:boolean;

begin
    // Call the assembly code and return its result:

    b := ReturnBoolean;

    // Display "true" or "false" depending on the return result.

    if( b ) then

```

```

        booleanLabel.caption := 'Boolean result = true '
    else
        BooleanLabel.caption := 'Boolean result = false';
end;

// Demonstrate calling an assembly language function that
// returns a word result.

procedure TDelphiEx2Form.WordBtnClick(Sender: TObject);
var
    si:smallint;    // Return result here.
    strVal:string;  // Used to display return result.
begin
    si := ReturnWord();    // Get result from assembly code.
    str( si, strVal );    // Convert result to a string.
    WordLabel.caption := 'Word Result = ' + strVal;

end;

// Demonstration of a call to an assembly language routine
// that returns a 32-bit result in EAX:

procedure TDelphiEx2Form.DWordBtnClick(Sender: TObject);
var
    i:integer;      // Return result goes here.
    strVal:string;  // Used to display return result.
begin
    i := ReturnDWord(); // Get result from assembly code.
    str( i, strVal );   // Convert that value to a string.
    DWordLabel.caption := 'Double Word Result = ' + strVal;

end;

// Demonstration of a routine that returns a pointer
// as the function result. This demo is kind of lame
// because we can't initialize anything inside the
// assembly module, but it does demonstrate the mechanism
// even if this example isn't very practical.

procedure TDelphiEx2Form.PtrBtnClick(Sender: TObject);
var
    p:pchar;    // Put returned pointer here.
begin
    // Get the pointer (to a zero byte) from the assembly code.

    p := ReturnPtr();

    // Display the empty string that ReturnPtr returns.

    PCharLabel.caption := 'PChar Result = "' + p + '"';

end;

```

```
// Quick demonstration of a function that returns a
// floating point value as a function result.

procedure TDelphiEx2Form.FltBtnClick(Sender: TObject);
var
    r:real;
    strVal:string;
begin
    // Call the assembly code that returns a real result.

    r := ReturnReal();      // Always returns 1.0

    // Convert and display the result.

    str( r:13:10, strVal );
    RealLabel.caption := 'Real Result = ' + strVal;

end;
end.
```

Program 12.7 DelphiEx2: Pascal Code for Assembly Return Results Example

```
// ReturnBooleanUnit-
//
// Provides the ReturnBoolean function for the DelphiEx2 program.

// The following MASM code must appear at this point because Delphi

#code( "_TEXT", "para", "CODE" )
#static( "_DATA", "para", "DATA" )
#readonly( "_DATA", "para", "DATA" )
#storage( "_DATA", "para", "DATA" )
#const( "_INIDATA_", "para", "INITDATA" )

unit ReturnBooleanUnit;

// Tell HLA that ReturnBoolean is a public symbol:

procedure ReturnBoolean; external;

// Demonstration of a function that returns a byte value in AL.
// This function simply returns a boolean result that alternates
// between true and false on each call.

procedure ReturnBoolean; nodisplay; noalignstk; noframe;
static b:boolean:=false;
begin ReturnBoolean;

    xor( 1, b );    // Invert boolean status
    and( 1, b );    // Force to zero (false) or one (true).
    mov( b, al );   // Function return result comes back in AL.
    ret();
```

```
end ReturnBoolean;
end ReturnBooleanUnit;
```

Program 12.8 ReturnBoolean: Demonstrates Returning a Byte Value in AL

```
// ReturnWordUnit-
//
// Provides the ReturnWord function for the DelphiEx2 program.

#code( "_TEXT", "para", "CODE" )
#static( "_DATA", "para", "DATA" )
#readonly( "_DATA", "para", "DATA" )
#storage( "_DATA", "para", "DATA" )
#const( "_INIDATA_", "para", "INITDATA" )

unit ReturnWordUnit;

procedure ReturnWord; external;

procedure ReturnWord; nodisplay; noalignstk; noframe;
static w:int16 := 1234;
begin ReturnWord;

    // Increment the static value by one on each
    // call and return the new result as the function
    // return value.

    inc( w );
    mov( w, ax );
    ret();

end ReturnWord;
end ReturnWordUnit;
```

Program 12.9 ReturnWord: Demonstrates Returning a Word Value in AX

```
// ReturnDWordUnit-
//
// Provides the ReturnDWord function for the DelphiEx2 program.

#code( "_TEXT", "para", "CODE" )
#static( "_DATA", "para", "DATA" )
#readonly( "_DATA", "para", "DATA" )
#storage( "_DATA", "para", "DATA" )
#const( "_INIDATA_", "para", "INITDATA" )

unit ReturnDWordUnit;

procedure ReturnDWord; external;

// Same code as ReturnWord except this one returns a 32-bit value
// in EAX rather than a 16-bit value in AX.
```

```

procedure ReturnDWord; nodisplay; noalignstk; noframe;
static
    d:int32 := -7;
begin ReturnDWord;

    inc( d );
    mov( d, eax );
    ret();

end ReturnDWord;
end ReturnDWordUnit;

```

Program 12.10 ReturnDWord: Demonstrates Returning a DWord Value in EAX

```

// ReturnPtrUnit-
//
// Provides the ReturnPtr function for the DelphiEx2 program.

#code( "_TEXT", "para", "CODE" )
#static( "_DATA", "para", "DATA" )
#readonly( "_DATA", "para", "DATA" )
#storage( "_DATA", "para", "DATA" )
#const( "_INIDATA_", "para", "INITDATA" )

unit ReturnPtrUnit;

procedure ReturnPtr; external;

// This function, which is lame, returns a pointer to a zero
// byte in memory (i.e., an empty pchar string). Although
// not particularly useful, this code does demonstrate how
// to return a pointer in EAX.

procedure ReturnPtr; nodisplay; noalignstk; noframe;
static
    stringData:byte:nostorage;
    byte "Pchar object", 0;

begin ReturnPtr;

    lea( eax, stringData );
    ret();

end ReturnPtr;
end ReturnPtrUnit;

```

Program 12.11 ReturnPtr: Demonstrates Returning a 32-bit Address in EAX

```

// ReturnRealUnit-
//
// Provides the ReturnReal function for the DelphiEx2 program.

```

```

#code( "_TEXT", "para", "CODE" )
#static( "_DATA", "para", "DATA" )
#readonly( "_DATA", "para", "DATA" )
#storage( "_DATA", "para", "DATA" )
#const( "_INIDATA_", "para", "INITDATA" )

unit ReturnRealUnit;

procedure ReturnReal; external;
procedure ReturnReal; nodisplay; noalignstk; noframe;
static
    realData: real80 := 1.234567890;

begin ReturnReal;

    fld( realData );
    ret();

end ReturnReal;
end ReturnRealUnit;

```

Program 12.12 ReturnReal: Demonstrates Returning a Real Value in ST0

The second thing to note is the `#code`, `#static`, etc., directives at the beginning of each file to change the segment name declarations. You'll learn the reason for these segment renaming directives a little later in this chapter.

12.3.4 Calling Conventions

Delphi supports five different calling mechanisms for procedures and functions: **register**, **pascal**, **cdecl**, **stdcall**, and **safecall**. The **register** and **pascal** calling methods are very similar except that the **pascal** parameter passing scheme always passes all parameters on the stack while the **register** calling mechanism passes the first three parameters in CPU registers. We'll return to these two mechanisms shortly since they are the primary mechanisms we'll use. The **cdecl** calling convention uses the C/C++ programming language calling convention. We'll study this scheme more in the section on interfacing C/C++ with HLA. There is no need to use this scheme when calling HLA procedures from Delphi. If you must use this scheme, then see the section on the C/C++ languages for details. The **stdcall** convention is used to call Windows API functions. Again, there really is no need to use this calling convention, so we will ignore it here. See the Delphi documentation for more details. **Safecall** is another specialized calling convention that we will not use. See, we've already reduced the complexity from five mechanisms to two! Seriously, though, when calling assembly language routines from Delphi code that you're writing, you only need to use the **pascal** and **register** conventions.

The calling convention options specify how Delphi passes parameters between procedures and functions as well as who is responsible for cleaning up the parameters when a function or procedure returns to its caller. The **pascal** calling convention passes all parameters on the stack and makes it the procedure or function's responsibility to remove those parameters from the stack. The pascal calling convention mandates that the caller push parameters in the order the compiler encounters them in the parameter list (i.e., left to right). This is exactly the calling convention that HLA uses (assuming you don't use the "IN register" parameter option). Here's an example of a Delphi external procedure declaration that uses the **pascal** calling convention:

```

procedure UsesPascal( parm1:integer; parm2:integer; parm3:integer );

```

The following program provides a quick example of a Delphi program that calls an HLA procedure (function) using the **pascal** calling convention.

```

unit DelphiEx3;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    callUsesPascalBtn: TButton;
    UsesPascalLabel: TLabel;
    procedure callUsesPascalBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
{$L usespascal.obj}

function UsesPascal
(
  parm1:integer;
  parm2:integer;
  parm3:integer
):integer; pascal; external;

procedure TForm1.callUsesPascalBtnClick(Sender: TObject);
var
  i:      integer;
  strVal: string;
begin
  i := UsesPascal( 5, 6, 7 );
  str( i, strVal );
  UsesPascalLabel.caption := 'Uses Pascal = ' + strVal;

end;
end.

```

Program 12.13 DelphiEx3 – Sample Program that Demonstrates the **pascal Calling Convention**

```

// UsesPascalUnit-
//

```

```
// Provides the UsesPascal function for the DelphiEx3 program.

unit UsesPascalUnit;

// Tell HLA that UsesPascal is a public symbol:

procedure UsesPascal( parm1:int32; parm2:int32; parm3:int32 ); external;

// Demonstration of a function that uses the PASCAL calling convention.
// This function simply computes parm1+parm2-parm3 and returns the
// result in EAX. Note that this function does not have the "NOFRAME"
// option as was the case in the previous Delphi examples. Since this
// code is expecting parameters on the stack, we have to build the
// activation record (stack frame).

procedure UsesPascal( parm1:int32; parm2:int32; parm3:int32 );
    nodisplay; noalignstk;

begin UsesPascal;

    mov( parm1, eax );
    add( parm2, eax );
    sub( parm3, eax );

end UsesPascal;

end UsesPascalUnit;
```

Program 12.14 UsesPascal – HLA Function the Previous Delphi Code Will Call

To compile the HLA code, you would use the following two commands in a command window:

```
hla -s UsesPascal.hla
m2t UsesPascal.asm
tasm32 UsesPascal.tas
```

Once you produce the OBJ file with the above two commands, you can get into Delphi and compile the Pascal code.

The **register** calling convention also processes parameters from left to right and requires the procedure/function to clean up the parameters upon return; the difference is that procedures and functions that use the **register** calling convention will pass their first three (ordinal) parameters in the EAX, EDX, and ECX registers (in that order) rather than on the stack. You can use HLA's "*IN register*" syntax to specify that you want the first three parameters passed in this registers, e.g.,

```
procedure UsesRegisters
(
    parm1:int32 in EAX;
    parm2:int32 in EDX;
    parm3:int32 in ECX
);
```

If your procedure had four or more parameters, you would not specify registers as their locations. Instead, you'd access those parameters on the stack. Since most procedures have three or fewer parameters, the **register** calling convention will typically pass all of a procedure's parameters in a register.

Although you can use the **register** keyword just like **pascal** to force the use of the **register** calling convention, the register calling convention is the default mechanism in Delphi. Therefore, a Delphi declaration like the following will automatically use the **register** calling convention:

```

procedure UsesRegisters
(
    parm1:integer;
    parm2:integer;
    parm3:integer
); external;

```

The following program is a modification of the previous program in this section that uses the **register** calling convention rather than the **pascal** calling convention.

```

unit DelphiEx4;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls;

type
    TForm1 = class(TForm)
        callUsesRegisterBtn: TButton;
        UsesRegisterLabel: TLabel;
        procedure callUsesRegisterBtnClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}
{$L usesregister.obj}

function UsesRegister
(
    parm1:integer;
    parm2:integer;
    parm3:integer;
    parm4:integer
):integer; external;

procedure TForm1.callUsesRegisterBtnClick(Sender: TObject);
var
    i:      integer;
    strVal: string;
begin
    i := UsesRegister( 5, 6, 7, 3 );
    str( i, strVal );
    UsesRegisterLabel.caption := 'Uses Register = ' + strVal;

end;
end.

```

Program 12.15 DelphiEx4 – Using the `register` Calling Convention

```
// UsesRegisterUnit-
//
// Provides the UsesRegister function for the DelphiEx4 program.

unit UsesRegisterUnit;

// Tell HLA that UsesRegister is a public symbol:

procedure UsesRegister
(
    parm1:int32 in eax;
    parm2:int32 in edx;
    parm3:int32 in ecx;
    parm4:int32
); external;

// Demonstration of a function that uses the REGISTER calling convention.
// This function simply computes (parm1+parm2-parm3)*parm4 and returns the
// result in EAX. Note that this function does not have the
// "NOFRAME" option because it needs to build the activation record
// (stack frame) in order to access the fourth parameter. Furthermore, this
// code must clean up the fourth parameter upon return (another chore handled
// automatically by HLA if the "NOFRAME" option is not present).

procedure UsesRegister
(
    parm1:int32 in eax;
    parm2:int32 in edx;
    parm3:int32 in ecx;
    parm4:int32
); nodisplay; noalignstk;

begin UsesRegister;

    mov( parm1, eax );
    add( parm2, eax );
    sub( parm3, eax );
    intmul( parm4, eax );

end UsesRegister;

end UsesRegisterUnit;
```

Program 12.16 HLA Code to support the DelphiEx4 Program

To compile the HLA code, you would use the following two commands in a command window:

```
hla -s UsesRegister.hla
m2t UsesRegister.asm
tasm32 UsesRegister.tas
```

Once you produce the OBJ file with the above two commands, you can get into Delphi and compile the Pascal code.

12.3.5 Pass by Value, Reference, CONST, and OUT in Delphi

A Delphi program can pass parameters to a procedure or function using one of four different mechanisms: pass by value, pass by reference, CONST parameters, and OUT parameters. The examples up to this point in this chapter have all used Delphi's (and HLA's) default pass by value mechanism. In this section we'll look at the other parameter passing mechanisms.

HLA and Delphi also share a (mostly) common syntax for pass by reference parameters. The following two lines provide an external declaration in Delphi and the corresponding external (public) declaration in HLA for a pass by reference parameter using the **pascal** calling convention:

```
procedure HasRefParm( var refparm: integer ); pascal; external; // Delphi
procedure HasRefParm( var refparm: int32 ); external;           // HLA
```

Like HLA, Delphi will pass the 32-bit address of whatever actual parameter you specify when calling the *HasRefParm* procedure. Don't forget, inside the HLA code, that you must dereference this pointer to access the actual parameter data. See the chapter on Intermediate Procedures for more details (see "Pass by Reference" on page 791).

The CONST and OUT parameter passing mechanisms are virtually identical to pass by reference. Like pass by reference these two schemes pass a 32-bit address of their actual parameter. The difference is that the called procedure is not supposed to write to CONST objects since they're, presumably, constant. Conversely, the called procedure is supposed to write to an OUT parameter (and not assume that it contains any initial value of consequence) since the whole purpose of an OUT parameter is to return data from a procedure or function. Other than the fact that the Delphi compiler will check procedures and functions (written in Delphi) for compliance with this rules, there is no difference between CONST, OUT, and reference parameters. Delphi passes all such parameters by reference to the procedure or function. Note that in HLA you would declare all CONST and OUT parameters as pass by reference parameters. HLA does not enforce the readonly attribute of the CONST object nor does it check for an attempt to access an uninitialized OUT parameter; those checks are the responsibility of the assembly language programmer.

As you learned in the previous section, by default Delphi uses the **register** calling convention. If you pass one of the first three parameters by reference to a procedure or function, Delphi will pass the address of that parameter in the EAX, EDX, or ECX register. This is very convenient as you can immediately apply the register indirect addressing mode without first loading the parameter into a 32-bit register.

Like HLA, Delphi lets you pass untyped parameters by reference (or by CONST or OUT). The syntax to achieve this in Delphi is the following:

```
procedure UntypedRefParm( var parm1; const parm2; out parm3 ); external;
```

Note that you do not supply a type specification to these parameters. Delphi will compute the 32-bit address of these objects and pass them on to the *UntypedRefParm* procedure without any further type checking. In HLA, you can use the VAR keyword as the data type to specify that you want an untyped reference parameter. Here's the corresponding prototype for the *UntypedRefParm* procedure in HLA:

```
procedure UntypedRefParm( var parm1:var; var parm2:var; var parm3:var );
external;
```

As noted above, you use the VAR keyword (pass by reference) when passing CONST and OUT parameters. Inside the HLA procedure it's your responsibility to use these pointers in a manner that is reasonable given the expectations of the Delphi code.

12.3.6 Scalar Data Type Correspondence Between Delphi and HLA

When passing parameters between Delphi and HLA procedures and functions, it's very important that the calling code and the called code agree on the basic data types for the parameters. In this section we will draw a correspondence between the Delphi (v5.0) scalar data types and the HLA (v1.x) data types⁴.

Assembly language supports any possible data format, so HLA's data type capabilities will always be a superset of Delphi's. Therefore, there may be some objects you can create in HLA that have no counterpart in Delphi, but the reverse is not true. Since the assembly functions and procedures you write are generally manipulating data that Delphi provides, you don't have to worry too much about not being able to process some data passed to an HLA procedure by Delphi⁵.

Delphi 5.0 provides a wide range of different integer data types. The following table lists the Delphi types and the HLA equivalents:

Table 1: Delphi and HLA Integer Types

Delphi	HLA Equivalent	Range	
		Minimum	Maximum
integer	int32 ^a	-2147483648	2147483647
cardinal	uns32 ^b	0	4294967295
shortint	int8	-128	127
smallint	int16	-32768	32767
longint	int32	-2147483648	2147483647
int64	qword	-2 ⁶³	(2 ⁶³ -1)
byte	uns8	0	255
word	uns16	0	65535
longword	uns32	0	4294967295
subrange types	Depends on range	minimum range value	maximum range value

a. Int32 is the implementation of integer in Delphi 5. This may change in later releases.

b. Uns32 is the implementation of cardinal in Delphi 5. This may change in later releases.

In addition to the integer values, Delphi supports several non-integer ordinal types. The following table provides their HLA equivalents:

4. Scalar data types are the ordinal, pointer, and real types. It does not include strings or other composite data types.

5. Delphi string objects are an exception. For reasons that have nothing to do with data representation, you should not manipulate string parameters passed in from Delphi to an HLA routine. This section will explain the problems more fully a little later.

Table 2: Non-integer Ordinal Types in Delphi and HLA

Delphi	HLA	Range	
		Minimum	Maximum
char	char	#0	#255
widechar	word	chr(0)	chr(65535)
boolean	boolean	false (0)	true(1)
bytebool	byte	0(false)	255 (non-zero is true)
wordbool	word	0 (false)	65535 (non-zero is true)
longbool	dword	0 (false)	4294967295 (non-zero is true)
enumerated types	enum, byte, or word	0	Depends on number of items in the enumeration list. Usually the upper limit is 256 symbols

Like the integer types, Delphi supports a wide range of real numeric formats. The following table presents these types and their HLA equivalents.

Table 3: Real Types in Delphi and HLA

Delphi	HLA	Range	
		Minimum	Maximum
real	real64	5.0 E-324	1.7 E+308
single	real32	1.5 E-45	3.4 E+38
double	real64	5.0 E-324	1.7 E+308
extended	real80	3.6 E-4951	1.1 E+4932
comp	real80	$-2^{63}+1$	$2^{63}-1$
currency	real80	-922337203685477.5808	922337203685477.5807
real48 ^a	byte[6]	2.9 E-39	1.7 E+38

a. real48 is an obsolete type that depends upon a software floating point library. You should never use this type in assembly code. If you do, you are responsible for writing the necessary floating point sub-routines to manipulate the data.

The last scalar type of interest is the pointer type. Both HLA and Delphi use a 32-bit address to represent pointers, so these data types are completely equivalent in both languages.

12.3.7 Passing String Data Between Delphi and HLA Code

Delphi supports a couple of different string formats. The native string format is actually very similar to HLA's string format. A string object is a pointer that points at a zero terminated sequence of characters. In the four bytes preceding the first character of the string, Delphi stores the current dynamic length of the string (just like HLA). In the four bytes before the length, Delphi stores a reference count (unlike HLA, which stores a maximum length value in this location). Delphi uses the reference count to keep track of how many different pointers contain the address of this particular string object. Delphi will automatically free the storage associated with a string object when the reference count drops to zero (this is known as garbage collection).

The Delphi string format is just close enough to HLA's to tempt you to use some HLA string functions in the HLA Standard Library. This will fail for two reasons: (1) many of the HLA Standard Library string functions check the maximum length field, so they will not work properly when they access Delphi's reference count field; (2) HLA Standard Library string functions have a habit of raising string overflow (and other) exceptions if they detect a problem (such as exceeding the maximum string length value). Remember, the HLA exception handling facility is not directly compatible with Delphi's, so you should never call any HLA code that might raise an exception.

Of course, you can always grab the source code to some HLA Standard Library string function and strip out the code that raises exceptions and checks the maximum length field (this is usually the same code that raises exceptions). However, you could still run into problems if you attempt to manipulate some Delphi string. In general, it's okay to read the data from a string parameter that Delphi passes to your assembly code, but you should never change the value of such a string. To understand the problem, consider the following HLA code sequence:

```
static
  s:string := "Hello World";
  sref:string;
  scopy:string;
  .
  .
  .
  str.a_cpy( s, scopy ); // scopy has its own copy of "Hello World"

  mov( s, eax );        // After this sequence, s and sref point at
  mov( eax, sref );      // the same character string in memory.
```

After the code sequence above, any change you would make to the *scopy* string would affect only *scopy* because it has its own copy of the "Hello World" string. On the other hand, if you make any changes to the characters that *s* points at, you'll also be changing the string that *sref* points at because *sref* contains the same pointer value as *s*; in other words, *s* and *sref* are aliases of the same data. Although this aliasing process can lead to the creation of some killer defects in your code, there is a big advantage to using copy by reference rather than copy by value: copy by reference is much quicker since it only involves copying a single four-byte pointer. If you rarely change a string variable after you assign one string to that variable, copy by reference can be very efficient.

Of course, what happens if you use copy by reference to copy *s* to *sref* and then you want to modify the string that *sref* points at without changing the string that *s* points at? One way to do this is to make a copy of the string at the time you want to change *sref* and then modify the copy. This is known as *copy on write semantics*. In the average program, copy on write tends to produce faster running programs because the typical program tends to assign one string to another without modification more often than it assigns a string value and then modifies it later. Of course, the real problem is "how do you know whether multiple string variables are pointing at the same string in memory?" After all, if only one string variable is pointing at the string data, you don't have to make a copy of the data, you can manipulate the string data directly. The *reference counter field* that Delphi attaches to the string data solves this problem. Each time a Delphi program assigns one string variable to another, the Delphi code simply copies a pointer and then increments the reference counter. Similarly, if you assign a string address to some Delphi string variable and that variable was previously pointing at some other string data, Delphi decrements the reference counter field of that previous

string value. When the reference count hits zero, Delphi automatically deallocates storage for the string (this is the garbage collection operation).

Note that Delphi strings don't need a maximum length field because Delphi dynamically allocates (standard) strings whenever you create a new string. Hence, string overflow doesn't occur and there is no need to check for string overflow (and, therefore, no need for the maximum length field). For literal string constants (which the compiler allocates statically, not dynamically on the heap), Delphi uses a reference count field of -1 so that the compiler will not attempt to deallocate the static object.

It wouldn't be that hard to take the HLA Standard Library strings module and modify it to use Delphi's dynamically allocated string format. There is, however, one problem with this approach: Borland has not published the internal string format for Delphi strings (the information appearing above is the result of sleuthing through memory with a debugger). They have probably withheld this information because they want the ability to change the internal representation of their string data type without breaking existing Delphi programs. So if you poke around in memory and modify Delphi string data (or allocate or deallocate these strings on your own), don't be surprised if your program malfunctions when a later version of Delphi appears. By the way, the discussion above is current as of Delphi v5.0. Who knows if this explanation will still apply in Kylix (the Linux version of Delphi), Delphi 6.0, or any later versions.

Like HLA strings, a Delphi string is a pointer that happens to contain the address of the first character of a zero terminated string in memory. As long as you don't modify this pointer, you don't modify any of the characters in that string, and you don't attempt to access any bytes before the first character of the string or after the zero terminating byte, you can safely access the string data in your HLA programs. Just remember that you cannot use any Standard Library routines that check the maximum string length or raise any exceptions. If you need the length of a Delphi string that you pass as a parameter to an HLA procedure, it would be wise to use the Delphi *Length* function to compute the length and pass this value as an additional parameter to your procedure. This will keep your code working should Borland ever decide to change their internal string representation.

Delphi also supports a *ShortString* data type. This data type provides backwards compatibility with older versions of Borland's Turbo Pascal (Borland Object Pascal) product. *ShortString* objects are traditional length-prefixed strings (see "Character Strings" on page 401). A short string variable is a sequence of one to 256 bytes where the first byte contains the current dynamic string length (a value in the range 0..255) and the following *n* bytes hold the actual characters in the string (*n* being the value found in the first byte of the string data). If you need to manipulate the value of a string variable within an assembly language module, you should pass that parameter as a *ShortString* variable (assuming, of course, that you don't need to handle strings longer than 256 characters). For efficiency reasons, you should always pass *ShortString* variables by reference (or CONST or OUT) rather than by value. If you pass a short string by value, Delphi must copy all the characters allocated for that string (even if the current length is shorter) into the procedure's activation record. This can be very slow. If you pass a *ShortString* by reference, then Delphi will only need to pass a pointer to the string's data; this is very efficient.

Note that *ShortString* objects do not have a zero terminating byte following the string data. Therefore, your assembly code should use the length prefix byte to determine the end of the string, it should not search for a zero byte in the string.

If you need the maximum length of a *ShortString* object, you can use the Delphi *high* function to obtain this information and pass it to your HLA code as another parameter. Note that the *high* function is an compiler intrinsic much like HLA's @size function. Delphi simply replaces this "function" with the equivalent constant at compile-time; this isn't a true function you can call. This maximum size information is not available at run-time (unless you've used the Delphi *high* function) and you cannot compute this information within your HLA code.

12.3.8 Passing Record Data Between HLA and Delphi

Records in HLA are (mostly) compatible with Delphi records. Syntactically their declarations are very similar and if you've specified the correct Delphi compiler options you can easily translate a Delphi record to an HLA record. In this section we'll explore how to do this and learn about the incompatibilities that exist between HLA records and Delphi records.

For the most part, translating Delphi records to HLA is a no brainer. The two record declarations are so similar syntactically that conversion is trivial. The only time you really run into a problem in the conversion process is when you encounter case variant records in Delphi; fortunately, these don't occur very often and when they do, HLA's anonymous unions within a record come to the rescue.

Consider the following Pascal record type declaration:

```
type
  recType =
    record

      day: byte;
      month:byte;
      year:integer;
      dayOfWeek:byte;

    end;
```

The translation to an HLA record is, for the most part, very straight-forward. Just translate the field types accordingly and use the HLA record syntax (see "Records" on page 465) and you're in business. The translation is the following:

```
type
  recType:
    record

      day: byte;
      month: byte;
      year:int32;
      dayOfWeek:byte;

    endrecord;
```

There is one minor problem with this example: data alignment. By default Delphi aligns each field of a record on the size of that object and pads the entire record so its size is an even multiple of the largest (scalar) object in the record. This means that the Delphi declaration above is really equivalent to the following HLA declaration:

```
type
  recType:
    record

      day: byte;
      month: byte;
      padding:byte[2];      // Align year on a four-byte boundary.
      year:int32;
      dayOfWeek:byte;
      morePadding: byte[3]; // Make record an even multiple of four bytes.

    endrecord;
```

Of course, a better solution is to use HLA's ALIGN directive to automatically align the fields in the record:

```
type
  recType:
    record

      day: byte;
      month: byte;
      align( 4 );      // Align year on a four-byte boundary.
      year:int32;
      dayOfWeek:byte;
```

```

    align(4);           // Make record an even multiple of four bytes.

endrecord;

```

Alignment of the fields is good insofar as access to the fields is faster if they are aligned appropriately. However, aligning records in this fashion does consume extra space (five bytes in the examples above) and that can be expensive if you have a large array of records whose fields need padding for alignment.

The alignment parameters for an HLA record should be the following:

Table 4: Alignment of Record Fields

Data Type	Alignment
Ordinal Types	Size of the type: 1, 2, or 4 bytes.
Real Types	2 for real48 and extended, 4 bytes for other real types
ShortString	1
Arrays	Same as the element size
Records	Same as the largest alignment of all the fields.
Sets	1 or two if the set has fewer than 8 or 16 elements, 4 otherwise
All other types	4

Another possibility is to tell Delphi not to align the fields in the record. There are two ways to do this: use the **packed** reserved word or use the {\$A-} compiler directive.

The packed keyword tells Delphi not to add padding to a specific record. For example, you could declare the original Delphi record as follows:

```

type
  recType =
    packed record

      day: byte;
      month: byte;
      year: integer;
      dayOfWeek: byte;

    end;

```

With the **packed** reserved word present, Delphi does not add any padding to the fields in the record. The corresponding HLA code would be the original record declaration above, e.g.,

```

type
  recType:
    record

      day: byte;
      month: byte;
      year: int32;
      dayOfWeek: byte;

    endrecord;

```

The nice thing about the **packed** keyword is that it lets you explicitly state whether you want data alignment/padding in a record. On the other hand, if you've got a lot of records and you don't want field alignment on any of them, you'll probably want to use the "{A-}" (turn data alignment off) option rather than add the **packed** reserved word to each record definition. Note that you can turn data alignment back on with the "{A+}" directive if you want a sequence of records to be packed and the rest of them to be aligned.

While it's far easier (and syntactically safer) to use packed records when passing record data between assembly language and Delphi, you will have to determine on a case-by-case basis whether you're willing to give up the performance gain in exchange for using less memory (and a simpler interface). It is certainly the case that packed records are easier to maintain in HLA than aligned records (since you don't have to carefully place ALIGN directives throughout the record in the HLA code). Furthermore, on new processors most mis-aligned data accesses aren't particularly expensive. However, if performance really matters you will have to measure the performance of your program and determine the cost of using packed records.

Case variant records in Delphi let you add mutually exclusive fields to a record with an optional tag field. Here are two examples:

```
type
  r1=
    record

      stdField: integer;
      case choice:boolean of
        true:( i:integer );
        false:( r:real );
      end;

  r2=
    record
      s2:real;
      case boolean of // Notice no tag object here.
        true:( s:string );
        false:( c:char );
      end;
```

HLA does not support the case variant syntax, but it does support anonymous unions in a record that let you achieve the same semantics. The two examples above, converted to HLA (assuming "{A-}") are

```
type
  r1:
    record

      stdField: int32;
      choice: boolean; // Notice that the tag field is just another field
      union

        i:int32;
        r:real64;

      endunion;

    endrecord;

  r2:
    record

      s2:real64;
      union

        s: string;
        c: char;
```

```

        endunion;

    endrecord;

```

Again, you should insert appropriate `ALIGN` directives if you're not creating a packed record. Note that you shouldn't place any `ALIGN` directives inside the anonymous union section; instead, place a single `ALIGN` directive before the `UNION` reserved word that specifies the size of the largest (scalar) object in the union as given by the table "Alignment of Record Fields" on page 1144.

In general, if the size of a record exceeds about 16 bytes, you should pass the record by reference rather than by value.

12.3.9 Passing Set Data Between Delphi and HLA

Sets in Delphi can have between 1 and 256 elements. Delphi implements sets using an array of bits, exactly as HLA implements character sets (see "Character Sets" on page 423). Delphi reserves one to 32 bytes for each set; the size of the set (in bytes) is $(\text{Number_of_elements} + 7) \text{ div } 8$. Like HLA's character sets, Delphi uses a set bit to indicate that a particular object is a member of the set and a zero bit indicates absence from the set. You can use the bit test (and set/complement/reset) instructions and all the other bit manipulation operations to manipulate character sets. Furthermore, the MMX instructions might provide a little added performance boost to your set operations (see "The MMX Instruction Set" on page 1081). For more details on the possibilities, consult the Delphi documentation and the chapters on character sets and the MMX instructions in this text.

Generally, sets are sufficiently short (maximum of 32 bytes) that passing the by value isn't totally horrible. However, you will get slightly better performance if you pass larger sets by reference. Note that HLA often passes character sets by value (16 bytes per set) to various Standard Library routines, so don't be totally afraid of passing sets by value.

12.3.10 Passing Array Data Between HLA and Delphi

Passing array data between some procedures written in Delphi and HLA is little different than passing array data between two HLA procedures. Generally, if the arrays are large, you'll want to pass the arrays by reference rather than value. Other than that, you should declare an appropriate array type in HLA to match the type you're passing in from Delphi and have at it. The following code fragments provide a simple example:

```

type
    PascalArray = array[0..127, 0..3] of integer;

procedure PassedArray( var ary: PascalArray ); external;

```

Corresponding HLA code:

```

type
    PascalArray: int32[ 128, 4];

procedure PassedArray( var ary: PascalArray ); external;

```

As the above examples demonstrate, Delphi's array declarations specify the starting and ending indices while HLA's array bounds specify the number of elements for each dimension. Other than this difference, however, you can see that the two declarations are very similar.

Delphi uses row-major ordering for arrays. So if you're accessing elements of a Delphi multi-dimensional array in HLA code, be sure to use the row-major order computation (see "Row Major Ordering" on page 451).

12.3.11 Delphi Limitations When Linking with MASM Code

Delphi places a couple of restrictions on OBJ files that it links with the Pascal code. Some of these restrictions appears to be defects in the implementation of the linker, but only Borland can say for sure if these are defects or they are design deficiencies.

The first restriction, which is probably a design deficiency going back to Delphi's early roots as the Borland Turbo Pascal compiler, is that Delphi only allows certain segments in the code you link with Delphi. The code segment should be named `"_TEXT"`, the data segment (STATIC and DATA) should be named `"_DATA"`, and the STORAGE segment should be named `"_BSS"`. Delphi does not provide support for a read-only segment⁶. Although these are the same segment names that HLA uses by default, Delphi requires dword alignment while HLA's default is para alignment. To live within these restrictions we must not use the default HLA segment alignments. We have to use the `#code`, `#static`, `#readonly`, `#storage`, and `#const` directives to change HLA's segment names to names that are more to Delphi's liking. The following statements achieve this:

```
#code( "_TEXT", "dword", "CODE" )           // Changes alignment to dword.
#static( "_DATA", "dword", "DATA" )         // Changes alignment to dword.
#readonly( "_DATA", "dword", "DATA" )       // HLA default is READONLY
#storage( "_BSS", "dword", "BSS" )          // Changes alignment to dword.
#const( "_INITDATA_", "dword", "INITDATA" ) // HLA default is CONST.
```

Here are the limitations on OBJ found in Borland documents:

- A: No more than 10 segments (HLA automatically defines five, so you get to define five more).
- No more than 255 external symbols (generally not a problem).
- No more than 50 local names in LNames records (not an issue).
- LEData and LIData records must be in offset order (not an issue).
- No THREAD subrecords are supported in FIXU32 records (not an issue).
- Only 32-bit offsets can be fixed up (not an issue).
- Only segment and self-relative fixups (not an issue).
- Target of a fixup must be a segment, a group, or an EXTDEF (this is a problem with string variables).

In addition to this list, a document created by Morten Elling suggests the following two problems:

- B: Delphi ignore PUBDEF records that contain more than one public name string (a big problem).
- C: The fixup subrecord of a FIXU32 record cannot contain a target displacement (another big problem).

The first real defect is that Delphi doesn't seem to recognize more than one external procedure name per OBJ file when that OBJ is compiled with MASM (problem B above). It seems to work fine if you compile an assembly file with TASM, but HLA does not produce TASM compatible output, so this is not an option. For some reason, the linker ignores all public procedure names except for one. This happens because MASM emits all the public names within in single "PUBDEF" record (which is perfectly legitimate) but Delphi allows only one symbol per "PUBDEF" record. The only solution to this problem is to put each HLA procedure in its own segment or in its own source file. There is a sneaky way to put multiple code segments in and HLA assembly output file, but this trick requires MASM knowledge so we will not use that trick here; instead, you should put each procedure in a separate source file⁷. This is rather cumbersome, but it does provide a workable solution.

The second defect is the real killer. Delphi does not allow assembly code to access more than one static variable per OBJ file if that object file was assembled with MASM (again, TASM seems to work okay). This is largely due to problem "C" above. Unfortunately, splitting up your data across multiple source files is not a solution to this problem. Often you will need tables of pointers (e.g., for a SWITCH/CASE statement) or

6. Actually, segment names must end with these strings. E.g., Delphi will accept a segment with the name `"mycode_TEXT"` and it will append the data to the end of the `_TEXT` segment.

7. The trick, by the way, is to use the `#asm.#endasm` sequence to insert MASM segment and end segment directives around each procedure. Keep in mind the 10 segment limitation of Delphi, however.

even a single pointer to a second object. If you attempt to link an OBJ file containing two references to different data objects in the same OBJ file, Delphi complains about a bad object file format.

There are a couple of workarounds. First, this problem doesn't apply to automatic variables. You can create as many variables as you please in the VAR section. Another way to skirt this problem is to create a single record variable in your static section and specify all your static objects as fields of this record. Then you can access the fields of that record as though they were separate variables. Another alternative is to declare your static variables in the INTERFACE section as structured constants (a structured constant is Delphi's version of an initialized static variable; despite the name, you can write to such an object).

The bottom line, however, is that you cannot initialize pointers in your assembly module with the address of some other object in the assembly module. Delphi rejects, out of hand, any object module assembled with MASM that contains such data. Because HLA string objects are pointers, you cannot create a static declaration like the following:

```
static
  s: string := "Hello";
```

The problem is that HLA emits code to initialize *s* with the address of the first character of the string. This is a pointer initialization and Delphi will reject the object file containing the declaration above. Note that the following code is legal since HLA doesn't attempt to initialize any pointers:

```
static
  s: byte: nostorage;
    byte "Hello", 0;
```

Unfortunately, there doesn't seem to be any way around the problem with initialized pointers in assembly code causing Delphi to fail during the link phase. Hopefully Borland will fix this problem before too long given that there are a lot of MASM users out there.

If you decide to go ahead and use MASM to assemble the HLA output, you cannot use the HLA "-c" option (compile and assemble to OBJ file, without linking). The problem is that HLA will tell MASM to produce a COFF (common object file format) file and Delphi wants OMF (object module format) files. To compile an HLA program to an OMF format file, you'd use the following commands:

```
hla -s filename.hla      -- produces filename.asm
ml -c -Cp filename.asm  -- produces filename.obj
```

After this sequence, you can link the OBJ file with your Delphi program, assuming you've handled the restrictions this section discusses.

12.3.12 Referencing Delphi Objects from HLA Code

Symbols you declare in the INTERFACE section of a Delphi program are public. Therefore, you can access these objects from HLA code if you declare those objects as external in the HLA program. The following sample program demonstrates this fact by declaring a structured constant (*y*) and a function (*callme*) that the HLA code uses when you press the button on a form. The HLA code calls the *callme* function (which returns the value 10) and then the HLA code stores the function return result into the *y* structured constant (which is really just a static variable).

```
unit DelphiEx5;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
```

```

TDataTable = class(TForm)
    GetDataBtn: TButton;
    DataLabel: TLabel;
    procedure GetDataBtnClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

// Here's a static variable that we will export to
// the HLA source code (in Delphi, structured constants
// are initialized static variables).

const
    y:integer = 12345;

var
    DataTable: TDataTable;

    // Here's the function we will export to the HLA code:

    function callme:integer;

implementation

{$R *.DFM}
{$L TableData.obj }

function TableData:integer; external;

// This function will simply return 10 as the function
// result (remember, function results come back in EAX).

function callme;
begin

    callme := 10;

end;

procedure TDataTable.GetDataBtnClick(Sender: TObject);
var
    strVal: string;
    yVal:  string;
begin

    // Display the value that TableData returns.
    // Also display the value of y, which TableValue modifies

    str( TableData(), strVal );
    str( y, yVal );
    DataLabel.caption := 'Data = ' + strVal + ' y=' + yVal;

end;

end.

```

Program 12.17 DelphiEx5 – Static Data and Delphi Public Symbols Demonstration

```

#code( "_TEXT", "para", "CODE" )
#static( "_DATA", "para", "DATA" )
#readonly( "_DATA", "para", "DATA" )
#storage( "_DATA", "para", "DATA" )
#const( "_INITDATA_", "para", "INITDATA" )

unit TableDataUnit;

type
  dataseg:
    record
      index: uns32;
      TheTable: int32;
    endrecord;

  static
    y: int32; external;
    d: dataseg: nostorage;

    dword -1; // index initial value;
    dword -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6; // TheTable values.

  procedure callme; external;

  procedure TableData; external;
  procedure TableData; nodisplay; noalignstk; noframe;
  begin TableData;

    callme();
    mov( eax, y );
    inc( d.index );
    mov( d.index, eax );
    if( eax > 11 ) then

      xor( eax, eax );
      mov( eax, d.index );

    endif;
    mov( d.TheTable[ eax*4 ], eax );
    ret();

  end TableData;
end TableDataUnit;

```

Program 12.18 HLA Code for DelphiEx5 Example

12.4 Programming in C/C++ and HLA

Without question, the most popular language used to develop Win32 applications is, uh, Visual Basic. We're not going to worry about interfacing Visual Basic to assembly in this text for two reasons: (1) Visual Basic programmers will get better control and performance from their code if they learn Delphi, and (2) Visual Basic's interface to assembly is very similar to Pascal's (Delphi's) so teaching the interface to Visual Basic would repeat a lot of the material from the previous section. Coming in second as the Win32 development language of choice is C/C++. The C/C++ interface to assembly language is a bit different than Pascal/Delphi. That's why this section appears in this text.

Unlike Delphi, that has only a single vendor, there are many different C/C++ compilers available on the market. Each vendor (Microsoft, Borland, Watcom, GNU, etc.) has their own ideas about how C/C++ should interface to external code. Many vendors have their own extensions to the C/C++ language to aid in the interface to assembly and other languages. For example, Borland provides a special keyword to let Borland C++ (and C++ Builder) programmers call Pascal code (or, conversely, allow Pascal code to call the C/C++ code). Microsoft, who stopped making Pascal compilers years ago, no longer supports this option. This is unfortunate since HLA uses the Pascal calling conventions. Fortunately, HLA is assembly language, so it's easy enough to adjust to the demands of the different C/C++ systems.

Before we get started discussing how to write HLA modules for your C/C++ programs, you must understand two very important facts:

HLA's exception handling facilities are not directly compatible with C/C++'s exception handling facilities. This means that you cannot use the TRY.ENDTRY and RAISE statements in the HLA code you intend to link to a C/C++ program. This also means that you cannot call library functions that contain such statements. Since the HLA Standard Library modules use exception handling statements all over the place, this effectively prevents you from calling HLA Standard Library routines from the code you intend to link with Delphi⁸.

Although you can write console applications with C/C++, a good percentage of C/C++ (and nearly all C++ Builder) applications are Windows/GUI applications. You cannot call console-related functions (e.g., `stdin.xxxx` or `stdout.xxxx`) from a GUI application. Even if HLA's console and standard input/output routines didn't use exception handling, you wouldn't be able to call them from a standard C/C++ application. Even if you are writing a console application in C/C++, you still shouldn't call the `stdin.xxxx` or `stdout.xxx` routines because they use the RAISE statement.

Given the rich set of language features that C/C++ supports, it should come as no surprise that the interface between the C/C++ language and assembly language is somewhat complex. Fortunately there are two facts that reduce this problem. First, it is relatively easy to simulate C/C++'s calling conventions in HLA. Second, the other complex stuff you won't use very often, so you may not have to bother with it.

Note: the following sections assume you are already familiar with C/C++ programming. They make no attempt to explain C/C++ syntax or features other than as needed to explain the C/C++ assembly language interface. If you're not familiar with C/C++, you will probably want to skip this section.

Also note: although this text uses the generic term "C/C++" when describing the interface between HLA and various C/C++ compilers, the truth is that you're really interfacing HLA with the C language. There is a fairly standardized interface between C and assembly language that most vendors follow. No such standard exists for the C++ language and every vendor, if they even support an interface between C++ and assembly, uses a different scheme. In this text we will stick to interfacing HLA with the C language. Fortunately, all popular C++ compilers support the C interface to assembly, so this isn't much of a problem.

8. Note that the HLA Standard Library source code is available; feel free to modify the routines you want to use and remove any exception handling statements contained therein.

The examples in this text will use the Borland C++ compiler and Microsoft's Visual C++ compiler. There may be some minor adjustments you need to make if you're using some other C/C++ compiler; please see the vendor's documentation for more details. This text will note differences between Borland's and Microsoft's offerings, as necessary.

12.4.1 Linking HLA Modules With C/C++ Programs

One big advantage of C/C++ over Delphi is that (most) C/C++ compiler vendors' products emit standard OBJ files. Well, almost standard. You wouldn't, for example, want to attempt to link the output of Microsoft's Visual C++ with TLINK (Borland's Turbo Linker) nor would you want to link the output of Borland C++ with Microsoft's linker. Nevertheless, working with OBJ files and a true linker is much nicer than having to deal with Delphi's built-in linker. As nice as the Delphi system is, interfacing with assembly language is much easier in C/C++ than in Delphi.

Note: the HLA Standard Library was created using Microsoft tools. This means that you will probably not be able to link this library module using the Borland TLINK program. Of course, you probably shouldn't be linking Standard Library code with C/C++ code anyway, so this shouldn't matter. However, if you really want to link some module from the HLA Standard Library with Borland C/C++, you should recompile the module and use the OBJ file directly rather than attempt to link the HLALIB.LIB file.

The Visual C++ compiler works with COFF object files. The Borland C++ compiler works with OMF object files. Both forms of object files use the OBJ extension, so you can't really tell by looking at a directory listing which form you've got. Fortunately, if you need to create a single OBJ file that will work with both, the Visual C++ compiler will also accept OMF files and convert them to a COFF file during the link phase. Of course, most of the time you will not be using both compilers, so you can pick whichever OBJ file format you're comfortable with and use that.

By default, HLA tells MASM to produce a COFF file when assembling the HLA output. This means that if you compile and HLA program using a command line like the following, you will not be able to directly link the code with Borland C++ code:

```
hla -c filename.hla      // The "-c" option tells HLA to compile and assemble.
```

If you want to create an OMF file rather than a COFF file, you can do so by using the following two commands:

```
hla -s filename.hla      // The "-s" option tells HLA to compile to an ASM file
ml -c -Cp filename.asm    // The "-c" option tells MASM to produce an OBJ file
```

The execution of the above two commands produces an OMF object file that both VC++ and BCC (Borland C++) will accept (though VC++ prefers COFF, it accepts OMF).

Both BCC and VC++ look at the extension of the source file names you provide on the command line to determine whether they are compiling a C or a C++ program. There are some minor syntactical differences between the external declarations for a C and a C++ program. This text assumes that you are compiling C++ programs that have a ".cpp" extension. The difference between a C and a C++ compilation occurs in the external declarations for the functions you intend to write in assembly language. For example, in a C source file you would simply write:

```
extern char* RetHW( void );
```

However, in a C++ environment, you would need the following external declaration:

```
extern "C"
{
    extern char* RetHW( void );
};
```

The 'extern "C"' clause tells the compiler to use standard C linkage even though the compiler is processing a C++ source file (C++ linkage is different than C and definitely far more complex; this text will not con-

sider pure C++ linkage since it varies so much from vendor to vendor). If you're going to compile C source files with VC++ or BCC (i.e., files with a ".c" suffix), simply drop the 'extern "C"' and the curly braces from around the external declarations.

The following sample program demonstrates this external linkage mechanism by writing a short HLA program that returns the address of a string ("Hello World") in the EAX register (like Delphi, C/C++ expects functions to return their results in EAX). The main C/C++ program then prints this string to the console device.

```
#include <stdlib.h>
#include "ratc.h"

extern "C"
{
    extern char* ReturnHW( void );
};

int main()
_begin( main )

    printf( "%s\n", ReturnHW() );
    _return 0;

_end( main )
```

Program 12.19 Cex1 - A Simple Example of a Call to an Assembly Function from C++

```
unit ReturnHWUnit;

    procedure ReturnHW; external( "_ReturnHW" );
    procedure ReturnHW; nodisplay; noframe; noalignstk;
    begin ReturnHW;

        lea( eax, "Hello World" );
        ret();

    end ReturnHW;

end ReturnHWUnit;
```

Program 12.20 RetHW.hla - Assembly Code that Cex1 Calls

There are several new things in both the C/C++ and HLA code that might confuse you at first glance, so let's discuss these things real quick here.

The first strange thing you will notice in the C++ code is the #include "ratc.h" statement. RatC is a C/C++ macro library that adds several new features to the C++ language. RatC adds several interesting features and capabilities to the C/C++ language, but a primary purpose of RatC is to help make C/C++ programs a little more readable. Of course, if you've never seen RatC before, you'll probably argue that it's not as readable as pure C/C++, but even someone who has never seen RatC before can figure out 80% of RatC within a minutes. In the example above, the _begin and _end clauses clearly map to the "{" and "}" symbols (notice how the use of _begin and _end make it clear what function or statement associates with the braces;

unlike the guesswork you've got in standard C). The `_return` statement is clearly equivalent to the C return statement. As you'll quickly see, all of the standard C control structures are improved slightly in RatC. You'll have no trouble recognizing them since they use the standard control structure names with an underscore prefix. This text promotes the creation of readable programs, hence the use of RatC in the examples appearing in this chapter⁹. You can find out more about RatC on Webster at <http://webster.cs.ucr.edu>.

The C/C++ program isn't the only source file to introduce something new. If you look at the HLA code you'll notice that the LEA instruction appears to be illegal. It takes the following form:

```
lea( eax, "Hello World" );
```

The LEA instruction is supposed to have a memory and a register operand. This example has a register and a constant; what is the address of a constant, anyway? Well, this is a syntactical extension that HLA provides to 80x86 assembly language. If you supply a constant instead of a memory operand to LEA, HLA will create a static (readonly) object initialized with that constant and the LEA instruction will return the address of that object. In this example, HLA will emit the string to the STRINGS (#const) segment and then load EAX with the address of the first character of that string. Since HLA strings always have a zero terminating byte, EAX will contain the address of a zero-terminated string which is exactly what C++ wants. If you look back at the original C++ code, you will see that *RetHW* returns a *char** object and the main C++ program displays this result on the console device.

If you haven't figured it out yet, this is a round-about version of the venerable "Hello World" program.

Microsoft VC++ users can compile this program from the command line by using the following commands¹⁰:

```
hla -c RetHW.hla           // Compiles and assembles RetHW.hla to RetHW.obj
cl Cex1.cpp RetHW.obj      // Compiles C++ code and links it with RetHW.obj
```

If you're a Borland C++ user, you'd use the following command sequence:

```
hla -s RetHW.hla           // Compile HLA file to an ASM file.
ml -c RetHW.asm            // Assemble ASM file to produce an OMF object file.
bcc32i Cex1.cpp RetHW.obj  // Compile and link C++ and assembly code.
                           // Could also use the BCC32 compiler.
```

12.4.2 Register Preservation

Unlike Delphi, a single language with a single vendor, there is no single list of registers that you can freely use as scratchpad values within an assembly language function. The list changes by vendor and even changes between versions from the same vendor. However, you can safely assume that EAX is available for scratchpad use since C functions return their result in the EAX register. You should probably preserve everything else.

12.4.3 Function Results

C/C++ compiler universally seem to return ordinal and pointer function results in AL, AX, or EAX depending on the operand's size. The compilers probably return floating point results on the top of the FPU stack as well. Other than that, check your C/C++ vendor's documentation for more details.

9. If RatC really annoys you, just keep in mind that you've only got to look at a few RatC programs in this chapter. Then you can go back to the old-fashioned C code and hack to your heart's content!

10. This text assumes you've executed the VCVARS32.BAT file that sets up the system to allow the use of VC++ from the command line.

12.4.4 Calling Conventions

The standard C/C++ calling convention is probably the biggest area of contention between the C/C++ and HLA languages. Although it's certainly possible to call HLA procedures from C/C++ and vice versa, the interface between the languages is nowhere near as convenient to use as HLA and Pascal/Delphi (and other languages). The problem is that C/C++ passes its parameters backwards compared to the rest of the world. Fortunately, with a little effort, it's easy enough to overcome this problem.

VC++ and BCC both support multiple calling conventions. You can greatly simplify your life if you're willing to use some vendor-specific extensions to the C++ language (and why not, it's not like your code that's calling an assembly language function is going to be portable anyway). BCC even supports the Pascal calling convention that HLA uses, making it trivial to write HLA functions for BCC programs¹¹. However, before we get into the details of these other calling conventions, it's probably a wise idea to first discuss the standard C/C++ calling convention.

Both VC++ and BCC *decorate* the function name when you declare an external function. For external "C" functions, the decoration consists of an underscore. If you look back at Program 12.20 you'll notice that the external name the HLA program actually uses is "_RetHW" rather than simply "RetHW". The HLA program itself, of course, uses the symbol "RetHW" to refer to the function, but the external name (as specified by the optional parameter to the EXTERNAL option) is "_RetHW". In the C/C++ program (Program 12.19) there is no explicit indication of this decoration; you simply have to read the compiler documentation to discover that the compiler automatically prepends this character to the function name¹². Fortunately, HLA's EXTERNAL option syntax allows us to *undecorate* the name, so we can refer to the function using the same name as the C/C++ program.

Name decoration is a trivial matter, easily fixed by HLA. To bad the other problems aren't as easy to solve. The next big problem is the fact that C/C++ pushes parameters on the stack in the opposite direction of just about every other (non-C base) language on the planet; specifically, C/C++ pushes actual parameters on the stack from right to left instead of the more common left to right. This means that you cannot declare a C/C++ function with two or more parameters and use a simple translation of the C/C++ external declaration as your HLA procedure declaration, i.e., the following are not equivalent:

```
external void CToHLA( int p, unsigned q, double r );
procedure CToHLA( p:int32; q:uns32; r:real64 ); external( "_CToHLA" );
```

Were you to call *CToHLA* from the C/C++ program, the compiler would push the *r* parameter first, the *q* parameter second, and the *p* parameter third - exactly the opposite order that the HLA code expects. As a result, the HLA code would use the L.O. double word of *r* as *p*'s value, the H.O. double word of *r* as *q*'s value, and the combination of *p* and *q*'s values as the value for *r*. Obviously, you'd most likely get an incorrect result from this calculation. Fortunately, there's an easy solution to this problem: just reverse the parameters in the HLA procedure declaration:

```
procedure CToHLA( r:real64; q:uns32; p:int32 ); external( "_CToHLA" );
```

Now when the C/C++ code calls this procedure, it push the parameters on the stack and the HLA code will retrieve them in the proper order.

Reversing the parameters in the HLA declaration works great if the C/C++ program is the only one that calls this function. Suppose, however, that the function is actually written in C/C++ and the HLA module is calling the function. Again, the solution is relatively simple, you just reverse the parameters in the HLA external declaration and everything works fine. Unfortunately, it's a real headache to remember to reverse the parameters when making the call from HLA, especially if you call the function from C/C++ as well. Some function calls will have the parameters in one order while other function calls will have the parameters in the reverse order. This is very confusing and can easily lead to the injection of defects in your code. Fortunately, there is a little trick we can play in HLA to allow us to swap the reverse the parameters when we

11. Microsoft used to support the Pascal calling convention, but when they stopped supporting their QuickPascal language, they dropped support for this option.

12. Most compilers provide an option to turn this off if you don't want this to occur. We will assume that this option is active in this text since that's the standard for external C names.

call the function: create a macro and invoke the macro to call the function. Consider the following HLA code:

```
procedure _CToHLA( r:real64; q:uns32; p:int32 ); external;
macro CToHLA( a, b, c );

    _CToHLA( c, b, a );

endmacro;

.
.
.
CToHLA( pValue, qValue, rValue );
```

Notice two things here: we use the decorated name in the actual external procedure declaration and we wrote a macro that uses the undecorated name and calls the function after reversing the parameters. This scheme automatically pushes the parameters in the correct order while preserving the C/C++ calling sequence for this function.

Well, reversing the parameters solved on big problem that exists between C/C++ and HLA, but there is still another big problem – HLA procedures automatically clean up after themselves by removing all parameters pass to a procedure prior to returning to the caller. C/C++, on the other hand, requires the caller, not the procedure, to clean up the parameters. This has two important ramifications: (1) if you call a C/C++ function (or one that uses the C/C++ calling sequence), then your code has to remove any parameters it pushed upon return from that function; (2) your HLA code cannot automatically remove parameter data from the stack if C/C++ code calls it.

Removing parameters from the stack when a C/C++ function returns to your code is very easy, just execute an “add(constant, esp);” instruction where *constant* is the number of parameter bytes you’ve pushed on the stack. For example, the *CToHLA* function has 16 bytes of parameters (two *int32* objects and one *real64* object) so the calling sequence (in HLA) would look something like the following:

```
CToHLA( pVal, qVal, rVal ); // Assume this is the macro version.
add( 16, esp );             // Remove parameters from the stack.
```

Dealing with cleaning up after a call is easy enough. However, if you’re writing the function that must leave it up to the caller to remove the parameters from the stack, then you’ve got a tiny problem – by default, HLA procedures always clean up after themselves. If you want to leave the parameters on the stack for the caller to remove, then you must write the standard entry and exit sequences for the procedure that build and destroy the activation record (see “The Standard Entry Sequence” on page 787 and “The Standard Exit Sequence” on page 788). This means you’ve got to use the *NOFRAME* (and *NODISPLAY*) options on your procedures that C/C++ will call. Here’s a sample implementation of the *CToHLA* procedure that builds and destroys the activation record:

```
procedure _CToHLA( rValue:real64; q:uns32; p:int32 ); nodisplay; noframe;
begin _CToHLA;

    push( ebp );           // Standard Entry Sequence
    mov( esp, ebp );
    // sub( _vars_, esp ); // Needed if you have local variables.
    .
    .                       // Code to implement the function’s body.
    .
    mov( ebp, esp );       // Restore the stack pointer.
    pop( ebp );            // Restore link to previous activation record.
    ret();                 // Note that we don’t remove any parameters.

end _CToHLA;
```

If you’re willing to use some vendor extensions to the C/C++ programming language, then you can make the interface to HLA much simpler. For example, if you’re using Borland’s C++ product, it has an

option you can apply to function declarations to tell the compiler to use the Pascal calling convention. Since HLA uses the Pascal calling convention, specifying this option in your BCC programs will make the interface to HLA trivial. In Borland C++ you can specify the Pascal calling convention for an external function using the following syntax:

```
extern type _pascal funcname( parameters )
```

Example:

```
extern void _pascal CToHLA( int p, unsigned q, double r );
```

The Pascal calling convention does not decorate the name, so the HLA name would not have a leading underscore. The Pascal calling convention uses case insensitive names; BCC achieves this by converting the name to all uppercase. Therefore, you'd probably want to use an HLA declaration like the following:

```
procedure CToHLA( p:int32; q:uns32; r:real64 ); external( "CTOHLA" );
```

Procedures using the Pascal calling convention push their parameters from left to right and leave it up to the procedure to clean up the stack upon return; exactly what HLA does by default. When using the Pascal calling convention, you could write the *CToHLA* function as follows:

```
procedure CToHLA( rValue:real64; q:uns32; p:int32 ); external( "CTOHLA" );

procedure CToHLA( rValue:real64; q:uns32; p:int32 ); nodisplay; noalignstk;
begin CToHLA;
    .
    .          // Code to implement the function's body.
    .
end CToHLA;
```

Note that you don't have to supply the standard entry and exit sequences. HLA provides those automatically.

Of course, Microsoft isn't about to support the Pascal calling sequence since they don't have a Pascal compiler. So this option isn't available to VC++ users.

Both Borland and Microsoft support the so-call *StdCall* calling convention. This is the calling convention that Windows uses, so nearly every language that operates under Windows provides this calling convention. The StdCall calling convention is a combination of the C and Pascal calling conventions. Like C, the functions need to have their parameters pushed on the stack in a right to left order; like Pascal, it is the caller's responsibility to clean up the parameters when the function returns; like C, the function name is case sensitive; like Pascal, the function name is not decorated (i.e., the external name is the same as the function declaration). The syntax for a StdCall function is the same in both VC++ and BCC, it is the following:

```
extern void _stdcall CToHLA( int p, unsigned q, double r );
```

You can use the macro trick in HLA to reverse the parameters if you need to call this function from HLA. Because the name is undecorated, you'd probably use a prototype and macro like the following:

```
procedure _CToHLA( r:real64; q:uns32; p:int32 ); external( "CTOHLA" );
macro CToHLA( a, b, c );

    _CToHLA( c, b, a );

endmacro;

.
.
.
procedure _CToHLA( r:real64; q:uns32; p:int32 ); nodisplay; nostkalign;
begin _CToHLA;
    .
    . // Function body
    .
end _CToHLA;
```

```

.
.
.
CToHLA( pValue, qValue, rValue ); // Demo of a call to CToHLA.

```

Notice how the external declaration maps a decorated version of the name to the undecorated external name. The program does this so that the macro can use the undecorated name in order to reverse the parameters.

12.4.5 Pass by Value and Reference in C/C++

A C/C++ program can pass parameters to a procedure or function using one of two different mechanisms: pass by value and pass by reference. Since pass by reference parameters use pointers, this parameter passing mechanism is completely compatible between HLA and C/C++. The following two lines provide an external declaration in C++ and the corresponding external (public) declaration in HLA for a pass by reference parameter using the calling convention:

```

extern void HasRefParm( int& refparm );           // C++
procedure HasRefParm( var refparm: int32 ); external; // HLA

```

Like HLA, C++ will pass the 32-bit address of whatever actual parameter you specify when calling the *HasRefParm* procedure. Don't forget, inside the HLA code, that you must dereference this pointer to access the actual parameter data. See the chapter on Intermediate Procedures for more details (see "Pass by Reference" on page 791).

Like HLA, C++ lets you pass untyped parameters by reference. The syntax to achieve this in C++ is the following:

```
extern void UntypedRefParm( void* parm1 );
```

Actually, this is not a reference parameter, but a value parameter with an untyped pointer. Whenever you call this function you must explicitly take the address of the actual parameter using the "&" operator.

In HLA, you can use the VAR keyword as the data type to specify that you want an untyped reference parameter. Here's the corresponding prototype for the *UntypedRefParm* procedure in HLA:

```

procedure UntypedRefParm( var parm1:var );
external;

```

12.4.6 Scalar Data Type Correspondence Between Delphi and HLA

When passing parameters between C/C++ and HLA procedures and functions, it's very important that the calling code and the called code agree on the basic data types for the parameters. In this section we will draw a correspondence between the C/C++ scalar data types and the HLA (v1.x) data types.

Assembly language supports any possible data format, so HLA's data type capabilities will always be a superset of C/C++'s. Therefore, there may be some objects you can create in HLA that have no counterpart in C/C++, but the reverse is not true. Since the assembly functions and procedures you write are generally manipulating data that C/C++ provides, you don't have to worry too much about not being able to process some data passed to an HLA procedure by C/C++.

C/C++ provides a wide range of different integer data types. Unfortunately, the exact representation of these types is implementation specific. The following table lists the C/C++ types as currently implemented by Borland C++ and Microsoft VC++. This table may very well change as 64-bit compilers become available.

Table 5: Delphi and HLA Integer Types

C/C++	HLA Equivalent	Range	
		Minimum	Maximum
int	int32	-2147483648	2147483647
unsigned	uns32	0	4294967295
signed char	int8	-128	127
short	int16	-32768	32767
long	int32	-2147483648	2147483647
unsigned char	uns8	0	255
unsigned short	uns16	0	65535

In addition to the integer values, Delphi supports several non-integer ordinal types. The following table provides their HLA equivalents:

Table 6: Non-integer Ordinal Types in C/C++ and HLA

C/C++	HLA	Range	
		Minimum	Maximum
wchar, TCHAR	word	0	65535
BOOL	boolean	false (0)	true (not zero)

Like the integer types, C/C++ supports a wide range of real numeric formats. The following table presents these types and their HLA equivalents.

Table 7: Real Types in C/C++ and HLA

C/C++	HLA	Range	
		Minimum	Maximum
double	real64	5.0 E-324	1.7 E+308
float	real32	1.5 E-45	3.4 E+38
long double	real80	3.6 E-4951	1.1 E+4932

The last scalar type of interest is the pointer type. Both HLA and C/C++ use a 32-bit address to represent pointers, so these data types are completely equivalent in both languages.

12.4.7 Passing String Data Between C/C++ and HLA Code

C/C++ uses zero terminated strings. Algorithms that manipulate zero-terminated strings are not as efficient as functions that work on length-prefixed strings; on the plus side, however, zero-terminated strings are very easy to work with. HLA's strings are downwards compatible with C/C++ strings since HLA places a zero byte at the end of each HLA string. Since you'll probably not be calling HLA Standard Library string routines, the fact that C/C++ strings are not upwards compatible with HLA strings generally won't be a problem. If you do decide to modify some of the HLA string functions so that they don't raise exceptions, you can always translate the `str.cStrToStr` function that translates zero-terminated C/C++ strings to HLA strings.

A C/C++ string variable is typically a `char*` object or an array of characters. In either case, C/C++ will pass the address of the first character of the string to an external procedure whenever you pass a string as a parameter. Within the procedure, you can treat the parameter as an indirect reference and dereference to pointer to access characters within the string.

12.4.8 Passing Record/Structure Data Between HLA and C/C++

Records in HLA are (mostly) compatible with C/C++ structs. You can easily translate a C/C++ struct to an HLA record. In this section we'll explore how to do this and learn about the incompatibilities that exist between HLA records and C/C++ structures.

For the most part, translating C/C++ records to HLA is a no brainer. Just grab the "guts" of a structure declaration and translate the declarations to HLA syntax within a `RECORD..ENDRECORD` block and you're done.

Consider the following C/C++ structure type declaration:

```
typedef struct
{
    unsigned char day;
    unsigned char month;
    int year;
    unsigned char dayOfWeek;
} dateType;
```

The translation to an HLA record is, for the most part, very straight-forward. Just translate the field types accordingly and use the HLA record syntax (see "Records" on page 465) and you're in business. The translation is the following:

```
type
    recType:
        record

            day: byte;
            month: byte;
            year: int32;
            dayOfWeek: byte;

        endrecord;
```

There is one minor problem with this example: data alignment. Depending on your compiler and whatever defaults it uses, C/C++ might not pack the data in the structure as compactly as possible. Some C/C++ compilers will attempt to align the fields on double word or other boundaries. With double word alignment of objects larger than a byte, the previous C/C++ **typedef** statement is probably better modelled by

```
type
    recType:
        record
```

```

    day: byte;
    month: byte;
    padding: byte[2];      // Align year on a four-byte boundary.
    year: int32;
    dayOfWeek: byte;
    morePadding: byte[3]; // Make record an even multiple of four bytes.

endrecord;

```

Of course, a better solution is to use HLA's ALIGN directive to automatically align the fields in the record:

```

type
  recType:
    record

      day: byte;
      month: byte;
      align( 4 );      // Align year on a four-byte boundary.
      year: int32;
      dayOfWeek: byte;
      align(4);        // Make record an even multiple of four bytes.

    endrecord;

```

Alignment of the fields is good insofar as access to the fields is faster if they are aligned appropriately. However, aligning records in this fashion does consume extra space (five bytes in the examples above) and that can be expensive if you have a large array of records whose fields need padding for alignment.

You will need to check your compiler vendor's documentation to determine whether it packs or pads structures by default. Most compilers give you several options for packing or padding the fields on various boundaries. Padded structures might be a bit faster while packed structures (i.e., no padding) are going to be more compact. You'll have to decide which is more important to you and then adjust your HLA code accordingly.

Note that by default, C/C++ passes structures by value. A C/C++ program must explicitly take the address of a structure object and pass that address in order to simulate pass by reference. In general, if the size of a structure exceeds about 16 bytes, you should pass the structure by reference rather than by value.

12.4.9 Passing Array Data Between HLA and C/C++

Passing array data between some procedures written in C/C++ and HLA is little different than passing array data between two HLA procedures. First of all, C/C++ can only pass arrays by reference, never by value. Therefore, you must always use pass by reference inside the HLA code. The following code fragments provide a simple example:

```

    int CArray[128][4];

extern void PassedArray( int array[128][4] );

```

Corresponding HLA code:

```

type
  CArray: int32[ 128, 4];

procedure PassedArray( var ary: CArray ); external;

```

As the above examples demonstrate, C/C++'s array declarations are similar to HLA's insofar as you specify the bounds of each dimension in the array.. Other than this difference, however, you can see that the two declarations are very similar.

C/C++ uses row-major ordering for arrays. So if you're accessing elements of a C/C++ multi-dimensional array in HLA code, be sure to use the row-major order computation (see "Row Major Ordering" on page 451).

12.5 Putting It All Together

Most real-world assembly code that is written consists of small modules that programmers link to programs written in other languages. Most languages provide some scheme for interfacing that language with assembly (HLA) code. Unfortunately, the number of interface mechanisms is sufficiently close to the number of language implementations to make a complete exposition of this subject impossible. In general, you will have to refer to the documentation for your particular compiler in order to learn sufficient details to successfully interface assembly with that language.

Fortunately, nagging details aside, most high level languages do share some common traits with respect to assembly language interface. Parameter passing conventions, stack clean up, register preservation, and several other important topics often apply from one language to the next. Therefore, once you learn how to interface a couple of languages to assembly, you'll quickly be able to figure out how to interface to others (given the documentation for the new language).

This chapter discusses the interface between the Delphi and C/C++ languages and assembly language. Although there are more popular languages out there (e.g., Visual Basic), Delphi and C/C++ introduce most of the concepts you'll need to know in order to interface a high level language with assembly language. Beyond that point, all you need is the documentation for your specific compiler and you'll be interfacing assembly with that language in no time.

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 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 NOALIGN-STK option in the procedure?
- 47) Which instruction in 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, EDI, and ECX 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 185 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 1201). 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 player 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 820) 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 1199 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 818) 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<sup>1</sup> 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  $\text{gcd}(A, B)$  is equal to the non-zero value (or zero if they are both zero).

If A and B are not zero, then  $\text{gcd}(A, B)$  is equal to  $\text{gcd}(B, A \bmod B)$  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 1199 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 874) “cheats” on decimal output by converting the output value to a signed 64-bit quantity and then calling *std-out.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 7.8 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 7.9 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 MOVSB Instruction" on page 908). 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 13.17 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 13.17 currently provides.
- 54) Modify Program 13.17 to include an \_if..elseif..else..endif statement with the tracing facilities.
- 55) Modify Program 13.17 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 13.17 currently provides.
- 56) Modify Program 13.17 to include an \_switch..case..default..endswitch statement with tracing facilities. 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:

```

/\
--

 /\
/\

 /\
/\
/\

 /\
/\
/\
/\

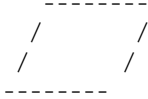
```

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

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).

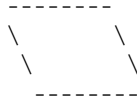
Modify the main program to draw several triangles in addition to the other shapes (demonstrate the correct operation of filled and unfilled triangles in your output).

- 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:



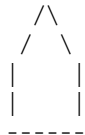
Don't forget to initialize the *slantRight* field (to true) inside the constructor.

- 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);

 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 690 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(JumpTbl[esi*4]); // Jump to the specified case handler.

 case123: // Handles 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 always 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
 // variables 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;

```

```

set(input < 0, a);
set(input >= -10, b);
set(input <= 10, c);
set(input = 0, 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 776). 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 787).

### 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.END-SWITCH” on page 723) 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 723 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. Your 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?

```
; EX15_1.asm
;
; This program demonstrates the proper use of the 80x86 string instructions.

 .386
 option segment:use16

 include stdlib.a
 includelib stdlib.lib

dseg segment para public 'data'

Buffer1 byte 2048 dup (0)
Buffer2 byte 2048 dup (0)

dseg ends

cseg segment para public 'code'
 assume cs:cseg, ds:dseg

Main proc
 mov ax, dseg
 mov ds, ax
 mov es, ax
 meminit

; Demo of the movsb, movsw, and movsd instructions

 print
 byte "The following code moves a block of 2,048 bytes "
 byte "around 100,000 times.", cr, lf
 byte "The first phase does this using the movsb "
 byte "instruction; the second", cr, lf
 byte "phase does this using the movsw instruction; "
 byte "the third phase does", cr, lf
 byte "this using the movsd instruction.", cr, lf, lf, lf
 byte "Press any key to begin phase one:", 0

 getc
 putcr

 mov edx, 100000
```

```

movsbLp:leasi, Buffer1
 lea di, Buffer2
 cld
 mov cx, 2048
rep movsb
 dec edx
 jnz movsbLp

 print
 bytecr,lf
 byte"Phase one complete",cr,lf,lf
 byte"Press any key to begin phase two:",0

 getc
 putcr

 mov edx, 100000

movswLp:leasi, Buffer1
 lea di, Buffer2
 cld
 mov cx, 1024
rep movsw
 dec edx
 jnz movswLp

 print
 bytecr,lf
 byte"Phase two complete",cr,lf,lf
 byte"Press any key to begin phase three:",0

 getc
 putcr

 mov edx, 100000

movsdLp:leasi, Buffer1
 lea di, Buffer2
 cld
 mov cx, 512
rep movsd
 dec edx
 jnz movsdLp

Quit: ExitPgm ;DOS macro to quit program.
Main endp

cseg ends

sseg segmentpara stack 'stack'
stk db 1024 dup ("stack ")
sseg ends

zzzzzzsegmentpara public 'zzzzzz'
LastBytesdb16 dup (?)
zzzzzzsegends
 end Main

```

### 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).

```
; EX15_2.asm
;
; This program compares the performance of the MOVS instruction against
; a manual block move operation. It also compares MOVS against a LODS/STOS
; loop.

.386
option segment:use16

include stdlib.a
includelibstdlib.lib

dseg segmentpara public 'data'

Buffer1byte2048 dup (0)
Buffer2byte2048 dup (0)

dseg ends

cseg segmentpara public 'code'
assume cs:cseg, ds:dseg

Main proc
mov ax, dseg
mov ds, ax
mov es, ax
meminit

; MOVS version done here:

print
byte"The following code moves a block of 2,048 bytes "
byte"around 100,000 times.",cr,lf
byte"The first phase does this using the movsb "
byte"instruction; the second",cr,lf
byte"phase does this using the lods/stos instructions; "
byte"the third phase does",cr,lf
byte"this using a loop with MOV "
byte"instructions.",cr,lf,lf,lf
byte"Press any key to begin phase one:",0

getc
putcr

mov edx, 100000

movsbLp:leasi, Buffer1
lea di, Buffer2
```

```

 cld
 mov cx, 2048
rep movsb
 dec edx
 jnz movsbLp

 print
 bytecr,lf
 byte"Phase one complete",cr,lf,lf
 byte"Press any key to begin phase two:",0

 getc
 putcr

 mov edx, 100000

LodsStosLp:leasi, Buffer1
 lea di, Buffer2
 cld
 mov cx, 2048
lodsstoslp2:lodsb
 stosb
 loopLodsStosLp2
 dec edx
 jnz LodsStosLp

 print
 bytecr,lf
 byte"Phase two complete",cr,lf,lf
 byte"Press any key to begin phase three:",0

 getc
 putcr

 mov edx, 100000

MovLp: lea si, Buffer1
 lea di, Buffer2
 cld
 mov cx, 2048
MovLp2:mov al, ds:[si]
 mov es:[di], al
 inc si
 inc di
 loopMovLp2
 dec edx
 jnz MovLp

Quit: ExitPgm ;DOS macro to quit program.
Main endp

cseg ends

sseg segmentpara stack 'stack'
stk db 1024 dup ("stack ")
sseg ends

zzzzzzsegmentpara public 'zzzzzz'
LastBytesdb16 dup (?)
zzzzzzsegends

```

end Main

### 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.

```
; EX15_3.asm
;
; This program compares the performance of the MOVS instruction against
; a manual block move operation. It also compares MOVS against a LODS/STOS
; loop. This version does so in such a way as to wipe out the on-chip CPU
; cache.

 .386
 option segment:use16

 include stdlib.a
 includelibstdlib.lib

dseg segmentpara public 'data'

Buffer1byte16384 dup (0)
Buffer2byte16384 dup (0)

dseg ends

cseg segmentpara public 'code'
 assume cs:cseg, ds:dseg

Main proc
 mov ax, dseg
 mov ds, ax
 mov es, ax
 meminit

; MOVSB version done here:

 print
 byte"The following code moves a block of 16,384 bytes "
 byte"around 12,500 times.",cr,lf
 byte"The first phase does this using the movsb "
 byte"instruction; the second",cr,lf
 byte"phase does this using the lods/stos instructions; "
 byte"the third phase does",cr,lf
 byte"this using a loop with MOV instructions."
 byte"Press any key to begin phase one:",0

 getc
 putcr

 mov edx, 12500

movsbLp:leasi, Buffer1
 lea di, Buffer2
```

```

 cld
 mov cx, 16384
rep movsb
 dec edx
 jnz movsbLp

 print
 bytecr,lf
 byte"Phase one complete",cr,lf,lf
 byte"Press any key to begin phase two:",0

 getc
 putcr

 mov edx, 12500

LodsStosLp:leasi, Buffer1
 lea di, Buffer2
 cld
 mov cx, 16384
lodsstoslp2:lodsb
 stosb
 loopLodsStosLp2
 dec edx
 jnz LodsStosLp

 print
 bytecr,lf
 byte"Phase two complete",cr,lf,lf
 byte"Press any key to begin phase three:",0

 getc
 putcr

 mov edx, 12500

MovLp: lea si, Buffer1
 lea di, Buffer2
 cld
 mov cx, 16384
MovLp2:mov al, ds:[si]
 mov es:[di], al
 inc si
 inc di
 loopMovLp2
 dec edx
 jnz MovLp

Quit: ExitPgm ;DOS macro to quit program.
Main endp
cseg ends

sseg segmentpara stack 'stack'
stk db 1024 dup ("stack ")
sseg ends

zzzzzzsegmentpara public 'zzzzzz'
LastBytesdb16 dup (?)
zzzzzzsegends
 end Main

```

### 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.

```
; EX15_4.asm
;
; This program compares the performance of length prefixed strings versus
; zero terminated strings using some simple examples.
;
; Note: these routines all assume that the strings are in the data segment
; and both ds and es already point into the data segment.

.386
option segment:use16

include stdlib.a
includelibstdlib.lib

dseg segmentpara public 'data'

LStr1 byte17,"This is a string."
LResultbyte256 dup (?)

ZStr1 byte"This is a string",0
ZResultbyte256 dup (?)

dseg ends

cseg segmentpara public 'code'
assume cs:cseg, ds:dseg

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

LStrCpyproc
 pushsi
 pushdi
 pushcx

 cld

 mov cl, [si];Get length of string.
 mov ch, 0
 inc cx ;Include length byte.
 rep movsb

 pop cx
 pop di
 pop si
 ret
LStrCpyendp
```

```

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

LStrCatproc
 pushsi
 pushdi
 pushcx

 cld

; Compute the final length of the concatenated string

 mov cl, [di] ;Get orig length.
 mov ch, [si] ;Get 2nd Length.
 add [di], ch ;Compute new length.

; Move SI to the first byte beyond the end of the first string.

 mov ch, 0 ;Zero extend orig len.
 add di, cx ;Skip past str.
 inc di ;Skip past length byte.

; Concatenate the second string (SI) to the end of the first string (DI)

 rep movsb ;Copy 2nd to end of orig.

 pop cx
 pop di
 pop si
 ret
LStrCatendp

; LStrCmp-String comparison using two length prefixed strings.
; SI points at the first string, DI points at the
; string to compare it against.

LStrCmpproc
 pushsi
 pushdi
 pushcx

 cld

; 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 cl, [si];Get the minimum of the two lengths
 mov ch, [di]
 cmp cl, ch
 jb HasMin
 mov cl, ch
HasMin: mov ch, 0

 repecmpsb ;Compare the two strings.
 je CmpLen
 pop cx
 pop di
 pop si
 ret

```

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

```
CmpLen: pop cx
 pop di
 pop si
```

```
 mov cl, [si]
 cmp cl, [di]
 ret
```

```
LStrCmpendp
```

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

```
ZStrCpyproc
 pushsi
 pushdi
 pushax
```

```
ZSCLp: mov al, [si]
 inc si
 mov [di], al
 inc di
 cmp al, 0
 jne ZSCLp
```

```
 pop ax
 pop di
 pop si
 ret
```

```
ZStrCpyendp
```

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

```
ZStrCatproc
 pushsi
 pushdi
 pushcx
 pushax
```

```
 cld
```

```
; Find the end of the destination string:
```

```
 mov cx, 0FFFFh
 mov al, 0 ;Look for zero byte.
 repnscasb
```

```
; Copy the source string to the end of the destination string:
```

```
ZcatLp: mov al, [si]
 inc si
 mov [di], al
 inc di
 cmp al, 0
 jne ZcatLp
```

```

 pop ax
 pop cx
 pop di
 pop si
 ret
ZStrCatendp

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

ZStrCmpproc
 pushcx
 pushsi
 pushdi

; 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.

ZCmpLp:mov al, [si]
 inc si
 cmp al, [di]
 jne ZCmpDone
 inc di
 cmp al, 0
 jne ZCmpLp

ZCmpDone:popdi
 pop si
 pop cx
 ret
ZStrCmpendp

Main proc
 mov ax, dseg
 mov ds, ax
 mov es, ax
 meminit

 print
 byte"The following code does 1,000,000 string "
 byte"operations using",cr,lf
 byte"length prefixed strings. Measure the amount "
 byte"of time this code",cr,lf
 byte"takes to run.",cr,lf,lf
 byte"Press any key to begin:",0

 getc
 putcr

 mov edx, 1000000
LStrCpyLp:leasi, LStr1
 lea di, LResult
 callLStrCpy
 callLStrCat
 callLStrCat
 callLStrCat

```

```

 callLStrCpy
 callLStrCmp
 callLStrCat
 callLStrCmp

 dec edx
 jne LStrCpyLp

 print
 byte"The following code does 1,000,000 string "
 byte"operations using",cr,lf
 byte"zero terminated strings. Measure the amount "
 byte"of time this code",cr,lf
 byte"takes to run.",cr,lf,lf
 byte"Press any key to begin:",0

 getc
 putcr

 mov edx, 1000000
ZStrCpyLp:leasi, ZStr1
 lea di, ZResult
 callZStrCpy
 callZStrCat
 callZStrCat
 callZStrCat
 callZStrCpy
 callZStrCmp
 callZStrCat
 callZStrCmp

 dec edx
 jne ZStrCpyLp

Quit: ExitPgm ;DOS macro to quit program.
Main endp

cseg ends

sseg segmentpara stack 'stack'
stk db 1024 dup ("stack ")
sseg ends

zzzzzzsegmentpara public 'zzzzzz'
LastBytesdb16 dup (?)
zzzzzzsegends
 end Main

```

---

### 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 13.11 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 13.12 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"<sup>2</sup> header file contains a macro, *assert*, that is quite useful for debugging purposes. This macro is defined as follows:

```
macro assert(expr):
```

---

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

```

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 demonstrates how to detect a common error (array bounds violation) using 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.

---



---

```

// 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 13.13 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 774 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 13.14 #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 6.7, 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 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 947.). 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 13.15 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]);

end i32Max;
```

```

 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 13.16 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 973.) 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 973). 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 13.17 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 13.18 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 13.19 Code for `_initialize_/_finalize_` 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.

---

```
// Laboratory Exercise 11.3:
//
// Using RTTI in a program.

program LabExercise_11_3;
#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; // a method that is specific to this class.

endclass;

anotherDerivedClass: class inherits(baseClass)

 override procedure Create;
 override procedure aProc;
 override method aMethod;
 method adcNewMethod; // a method that is specific to this class.

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 LabExercise_11_3;

 // 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 (leaving a pointer to the
 // randomly specified object in pbc).

 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 LabExercise_11_3;
```

---

### Program 13.20 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.