Volume Three: Basic Assembly Language

Volume Three:

Basic Assembly Language

This Volume provides a basic introduction to assembly language programming. By the end of this volume you should be able to write meaningful programs using HLA. This Volume plus Volume Four present all the basic skills a typical assembly language programmer needs to write real-world applications in assembly language.

Chapters One through Seven provide information about important data types and data structures found in typical assembly language programs. For courses that have a limited amount of time available, Chapters One, Four, and Five from this set are the most important, closely followed by Chapters Two and Seven. Chapters Three and Six are optional though students should read these on their own.

Chapters Eight and Ten are also essential. Chapters Nine and Eleven are important and the course should cover them if time permits. Chapter Twelve discusses an optimization that is becoming less and less important as CPU speeds vastly outstrip memory access times. Those interested in programming embedded systems should read this chapter, other instructors may elect to skip this material.

Constants, Variables, and Data Types Chapter One

Volume One discussed the basic format for data in memory. Volume Two covered how a computer system physically organizes that data. This chapter finishes this discussion by connecting the concept of *data representation* to its actual physical representation. As the title implies, this chapter concerns itself with three main topics: constants, variables and data structures. This chapter does not assume that you've had a formal course in data structures, though such experience would be useful.

1.1 Chapter Overview

This chapter discusses how to declare and use constants, scalar variables, integers, reals, data types, pointers, arrays, and structures. You must master these subjects before going on to the next chapter. Declaring and accessing arrays, in particular, seems to present a multitude of problems to beginning assembly language programmers. However, the rest of this text depends on your understanding of these data structures and their memory representation. Do not try to skim over this material with the expectation that you will pick it up as you need it later. You will need it right away and trying to learn this material along with later material will only confuse you more.

1.2 Some Additional Instructions: INTMUL, BOUND, INTO

This chapter introduces arrays and other concepts that will require the expansion of your 80x86 instruction set knowledge. In particular, you will need to learn how to multiply two values; hence the first instruction we will look at is the intmul (integer multiply) instruction. Another common task when accessing arrays is to check to see if an array index is within bounds. The 80x86 bound instruction provides a convenient way to check a register's value to see if it is within some range. Finally, the into (interrupt on overflow) instruction provides a quick check for signed arithmetic overflow. Although into isn't really necessary for array (or other data type access), its function is very similar to bound, hence the presentation at this point.

The intmul instruction takes one of the following forms:

```
// The following compute destreg = destreg * constant

intmul( constant, destreg16 );
intmul( constant, destreg32 );

// The following compute dest = src * constant

intmul( constant, srcreg16, destreg16 );
intmul( constant, srcmem16, destreg16 );

intmul( constant, srcreg32, destreg32 );
intmul( constant, srcmem32, destreg32 );

// The following compute dest = dest * src

intmul( srcreg16, destreg16 );
intmul( srcmem16, destreg16 );
intmul( srcreg32, destreg32 );
intmul( srcmem32, destreg32 );
```

Note that the syntax of the intmul instruction is different than the add and sub instructions. In particular, note that the destination operand must be a register (add and sub both allow a memory operand as a destination). Also note that intmul allows three operands when the first operand is a constant. Another important difference is that the intmul instruction only allows 16-bit and 32-bit operands; it does not allow eight-bit operands.

intmul computes the product of its specified operands and stores the result into the destination register. If an overflow occurs (which is always a signed overflow, since intmul only multiplies signed integer values), then this instruction sets both the carry and overflow flags. intmul leaves the other condition code flags undefined (so, for example, you cannot check the sign flag or the zero flag after intmul and expect them to tell you anything about the intmul operation).

The bound instruction checks a 16-bit or 32-bit register to see if it is between one of two values. If the value is outside this range, the program raises an exception and aborts. This instruction is particularly useful for checking to see if an array index is within a given range. The bound instruction takes one of the following forms:

bound(reg_{16}, *LBconstant, UBconstant*);
bound(reg_{32}, *LBconstant, UBconstant*);

bound(reg_{16}, $Mem_{16}[2]$);[1]
bound(reg_{32}, $Mem_{32}[2]$);[2]

The bound instruction compares its register operand against an unsigned lower bound value and an unsigned upper bound value to ensure that the register is in the range:

```
lower_bound <= register <= upper_bound
```

The form of the bound instruction with three operands compares the register against the second and third parameters (the lower bound and upper bound, respectively)[3]. The bound instruction with two operands checks the register against one of the following ranges:

$Mem_{16}[0] <= register_{16} <= Mem_{16}[2]$
$Mem_{32}[0] <= register_{32} <= Mem_{32}[4]$

If the specified register is not within the given range, then the 80x86 raises an exception. You can trap this exception using the HLA try..endtry exception handling statement. The excepts.hhf header file defines an exception, *ex.BoundInstr*, specifically for this purpose. The following code fragment demonstrates how to use the bound instruction to check some user input:

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

static
    InputValue:int32;
    GoodInput:boolean;

begin BoundDemo;

    // Repeat until the user enters a good value:

    repeat

        // Assume the user enters a bad value.

        mov( false, GoodInput );
```

1. The "[2]" suggests that this variable must be an array of two consecutive word values in memory.
2. Likewise, this memory operand must be two consecutive dwords in memory.
3. This form isn't a true 80x86 instruction. HLA converts this form of the bound instruction to the two operand form by creating two readonly memory variables initialized with the specified constant.

```
        // Catch bad numeric input via the try..endtry statement.

        try

            stdout.put( "Enter an integer between 1 and 10: " );
            stdin.flushInput();
            stdin.geti32();

            mov( eax, InputValue );

            // Use the BOUND instruction to verify that the
            // value is in the range 1..10.

            bound( eax, 1, 10 );

            // If we get to this point, the value was in the
            // range 1..10, so set the boolean "GoodInput"
            // flag to true so we can exit the loop.

            mov( true, GoodInput );


            // Handle inputs that are not legal integers.

          exception( ex.ConversionError )

            stdout.put( "Illegal numeric format, reenter", nl );


            // Handle integer inputs that don't fit into an int32.

          exception( ex.ValueOutOfRange )

            stdout.put( "Value is *way* too big, reenter", nl );


            // Handle values outside the range 1..10 (BOUND instruction)

          /*
          exception( ex.BoundInstr )

            stdout.put
            (
                "Value was ",
                InputValue,
                ", it must be between 1 and 10, reenter",
                nl
            );
          */

        endtry;

    until( GoodInput );
    stdout.put( "The value you entered, ", InputValue, " is valid.", nl );

end BoundDemo;
```

Program 1.1 Demonstration of the BOUND Instruction

The into instruction, like bound, also generates an exception under certain conditions. Specifically, into generates an exception if the overflow flag is set. Normally, you would use into immediately after a signed arithmetic operation (e.g., intmul) to see if an overflow occurs. If the overflow flag is not set, the system ignores the into instruction; however, if the overflow flag is set, then the into instruction raises the HLA *ex.IntoInstr* exception. The following code sample demonstrates the use of the into instruction:

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

static
    LOperand:int8;
    ResultOp:int8;

begin INTOdemo;

    // The following try..endtry checks for bad numeric
    // input and handles the integer overflow check:

    try

        // Get the first of two operands:

        stdout.put( “Enter a small integer value (-128..+127):” );
        stdin.geti8();
        mov( al, LOperand );

        // Get the second operand:

        stdout.put( “Enter a second small integer value (-128..+127):” );
        stdin.geti8();

        // Produce their sum and check for overflow:

        add( LOperand, al );
        into();

        // Display the sum:

        stdout.put( “The eight-bit sum is “, (type int8 al), nl );


        // Handle bad input here:

      exception( ex.ConversionError )

        stdout.put( “You entered illegal characters in the number”, nl );


        // Handle values that don’t fit in a byte here:

      exception( ex.ValueOutOfRange )

        stdout.put( “The value must be in the range -128..+127”, nl );


        // Handle integer overflow here:

      /*
      exception( ex.IntoInstr )
```

```
            stdout.put
            (
                “The sum of the two values is outside the range -128..+127”,
                nl
            );
          */

        endtry;

end INTOdemo;
```

Program 1.2 Demonstration of the INTO Instruction

1.3 The QWORD and TBYTE Data Types

HLA lets you declare eight-byte and ten-byte variables using the *qword*, and *tbyte* data types, respectively. Since HLA does not allow the use of 64-bit or 80-bit non-floating point constants, you may not associate an initializer with these two data types. However, if you wish to reserve storage for a 64-bit or 80-bit variable, you may use these two data types to do so.

The *qword* type lets you declare *quadword* (eight byte) variables. Generally, *qword* variables will hold 64-bit integer or unsigned integer values, although HLA and the 80x86 certainly don’t enforce this. The HLA Standard Library contains several routines to let you input and display 64-bit signed and unsigned integer values. The chapter on advanced arithmetic will discuss how to calculate 64-bit results on the 80x86 if you need integers of this size.

The *tbyte* directive allocates ten bytes of storage. There are two data types indigenous to the 80x87 (math coprocessor) family that use a ten byte data type: ten byte BCD values and extended precision (80 bit) floating point values. Since you would normally use the *real80* data type for floating point values, about the only purpose of tbyte in HLA is to reserve storage for a 10-byte BCD value (or other data type that needs 80 bits). Once again, the chapter on advanced arithmetic may provide some insight into the use of this data type. However, except for very advanced applications, you could probably ignore this data type and not suffer.

1.4 HLA Constant and Value Declarations

HLA’s CONST and VAL sections let you declare symbolic constants. The CONST section lets you declare identifiers whose value is constant throughout compilation and run-time; the VAL section lets you declare symbolic constants whose value can change at compile time, but whose values are constant at run-time (that is, the same name can have a different value at several points in the source code, but the value of a VAL symbol at a given point in the program cannot change while the program is running).

The CONST section appears in the same declaration section of your program that contains the STATIC, READONLY, STORAGE, and VAR, sections. It begins with the CONST reserved word and has a syntax that is nearly identical to the READONLY section, that is, the CONST section contains a list of identifiers followed by a type and a constant expression. The following example will give you an idea of what the CONST section looks like:

```
const
    pi:             real32 := 3.14159;
    MaxIndex:       uns32  := 15;
    Delimiter:      char   := ‘/’;
    BitMask:        byte   := $F0;
    DebugActive:   boolean:= true;
```

Once you declare these constants in this manner, you may use the symbolic identifiers anywhere the corresponding literal constant is legal. These constants are known as *manifest constants*. A manifest constant is a symbolic representation of a constant that allows you to substitute the literal value for the symbol anywhere in the program. Contrast this with READONLY variables; a READONLY variable is certainly a constant value since you cannot change such a variable at run time. However, there is a memory location associated with READONLY variables and the operating system, not the HLA compiler, enforces the read-only attribute at run-time. Although it will certainly crash your program when it runs, it is perfectly legal to write an instruction like "MOV(EAX, ReadOnlyVar);" On the other hand, it is no more legal to write "MOV(EAX, MaxIndex);" (using the declaration above) than it is to write "MOV(EAX, 15);" In fact, both of these statements are equivalent since the compiler substitutes "15" for *MaxIndex* whenever it encounters this manifest constant.

If there is absolutely no ambiguity about a constant's type, then you may declare a constant by specifying only the name and the constant's value, omitting the type specification. In the example earlier, the *pi*, *Delimiter*, *MaxIndex*, and *DebugActive* constants could use the following declarations:

```
const
    pi            := 3.14159;       // Default type is real80.
    MaxIndex      := 15;            // Default type is uns32.
    Delimiter:    := '/';           // Default type is char.
    DebugActive:  := true;          // Default type is boolean.
```

Symbol constants that have an integer literal constant are always given the type *uns32* if the constant is zero or positive, or *int32* if the value is negative. This is why *MaxIndex* was okay in this CONST declaration but *BitMask* was not. Had we included the statement "BitMask := $F0;" in this latter CONST section, the declaration would have been legal but *BitMask* would be of type *uns32* rather than *byte*.

Constant declarations are great for defining "magic" numbers that might possibly change during program modification. The following provides an example of using constants to parameterize "magic" values in the program.

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

const
    MemToAllocate   := 4_000_000;
    NumDWords       := MemToAllocate div 4;
    MisalignBy      := 62;

    MainRepetitions := 1000;
    DataRepetitions := 999_900;

    CacheLineSize   := 16;

begin ConstDemo;

    //console.cls();
    stdout.put
    (
        "Memory Alignment Exercise",nl,
        nl,
        "Using a watch (preferably a stopwatch), time the execution of", nl
        "the following code to determine how many seconds it takes to", nl
        "execute.", nl
        nl
        "Press Enter to begin timing the code:"
    );
```

```
// Allocate enough dynamic memory to ensure that it does not
// all fit inside the cache.  Note: the machine had better have
// at least four megabytes free or virtual memory will kick in
// and invalidate the timing.

malloc( MemToAllocate );

// Zero out the memory (this loop really exists just to
// ensure that all memory is mapped in by the OS).

mov( NumDWords, ecx );
repeat

    dec( ecx );
    mov( 0, (type dword [eax+ecx*4]));

until( !ecx );  // Repeat until ECX = 0.


// Okay, wait for the user to press the Enter key.

stdin.readLn();

// Note: as processors get faster and faster, you may
// want to increase the size of the following constant.
// Execution time for this loop should be approximately
// 10-30 seconds.

mov( MainRepetitions, edx );
add( MisalignBy, eax );     // Force misalignment of data.

repeat

    mov( DataRepetitions, ecx );
    align( CacheLineSize );
    repeat

        sub( 4, ecx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );

    until( !ecx );
    dec( edx );

until( !edx ); // Repeat until EAX is zero.

stdout.put( stdio.bell, “Stop timing and record time spent”, nl, nl );


// Okay, time the aligned access.

stdout.put
(
    “Press Enter again to begin timing access to aligned variable:”
);
stdin.readLn();
```

```
    // Note: if you change the constant above, be sure to change
    // this one, too!

    mov( MainRepetitions, edx );
    sub( MisalignBy, eax );     // Realign the data.
    repeat

        mov( DataRepetitions, ecx );
        align( CacheLineSize );
        repeat

            sub( 4, ecx );
            mov( [eax+ecx*4], ebx );
            mov( [eax+ecx*4], ebx );
            mov( [eax+ecx*4], ebx );
            mov( [eax+ecx*4], ebx );

        until( !ecx );
        dec( edx );

    until( !edx ); // Repeat until EAX is zero.

    stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );
    free( eax );


end ConstDemo;
```

Program 1.3 Data Alignment Program Rewritten Using CONST Definitions

1.4.1 Constant Types

Manifest constants can be any of the HLA primitive types plus a few of the composite types this chapter discusses. Volumes One and Two discussed most of the primitive types; these primitive types include the following:

- Boolean constants (true or false)
- Uns8 constants (0..255)
- Uns16 constants (0..65535)
- Uns32 constants (0..4,294,967,295)
- Int8 constants (-128..+127)
- Int16 constants (-32768..+32767)
- Int32 constants (-2,147,483,648..+2,147,483,647)
- Char constants (any ASCII character with a character code in the range 0..255)
- Byte constants (any eight-bit value including integers, booleans, and characters)
- Word constants (any 16-bit value)
- DWord constants (any 32-bit value)
- Real32 constants (floating point values)
- Real64 constants (floating point values)
- Real80 constants (floating point values)

In addition to the constant types appearing above, the CONST section supports six additional constant types:

- String constants
- Text constants
- Enumerated constant values

- Array constants
- Record/Union constants
- Character set constants

These data types are the subject of this Volume and the discussion of most of them appears in later chapters. However, the string and text constants are sufficiently important to warrant an early discussion of these constant types.

1.4.2 String and Character Literal Constants

HLA, like most programming languages, draws a distinction between a sequence of characters, a *string*, and a single character. This distinction is present both in the type declarations and in the syntax for literal character and string constants. Until now, this text has not drawn a fine distinction between character and string literal constants; now it is time to do so.

String literal constants consist of a sequence of zero or more characters surrounded by the ASCII quote characters. The following are all examples of legal literal string constants:

```
“This is a string”       // String with 16 characters.
““                       // Zero length string.
“a”                      // String with a single character.
“123”                    // String of length three.
```

A string of length one is not the same thing as a character constant. HLA uses two completely different internal representations for character and string values. Hence, “a” is not a character value, it is a string value that just happens to contain a single character.

Character literal constants take a couple forms, but the most common consist of a single character surrounded by ASCII apostrophe characters:

```
‘2’                      // Character constant equivalent to ASCII code $32.
‘a’                      // Character constant for lower case ‘A’.
```

As noted above, “a” and ‘a’ are not equivalent.

Those who are familiar with C/C++/Java probably recognize these literal constant forms, since they are similar to the character and string constants in C/C++/Java. In fact, this text has made a tacit assumption to this point that you are somewhat familiar with C/C++ insofar as examples appearing up to this point use character and string constants without an explicit definition of them[4].

Another similarity between C/C++ strings and HLA's is the automatic concatenation of adjacent literal string constants within your program. For example, HLA concatenates the two string constants

```
“First part of string, “    “second part of string”
```

to form the single string constant

```
“First part of string, second part of string”
```

Beyond these few similarities, however, HLA strings and C/C++ strings are different. For example, C/C++ strings let you specify special character values using the escape character sequence consisting of a backslash character followed by one or more special characters; HLA does not use this escape character mechanism. HLA does provide, however, several other ways to achieve this same goal.

Since HLA does not allow escape character sequences in literal string and character constants, the first question you might ask is “How does one embed quote characters in string constants and apostrophe characters in character constants?” To solve this problem, HLA uses the same technique as Pascal and many other

4. Apologies are due to those of you who do not know C/C++/Java or a language that shares these string and constant definitions.

languages: you insert two quotes in a string constant to represent a single quote or you place two apostrophes in a character constant to represent a single apostrophe character, e.g.,

```
        “He wrote a ““Hello World”” program as an example.”
```

The above is equivalent to:

```
        He wrote a “Hello World” program as an example.

        ‘’’’
```

The above is equivalent to a single apostrophe character.

HLA provides a couple of other features that eliminate the need for escape characters. In addition to concatenating two adjacent string constants to form a longer string constant, HLA will also concatenate any combination of adjacent character and string constants to form a single string constant:

```
        ‘1’  ‘2’  ‘3’            // Equivalent to “123”
        “He wrote a “  ‘”’ “Hello World”  ‘”’ “ program as an example.”
```

Note that the two “He wrote...” strings in the above examples are identical to HLA.

HLA provides a second way to specify character constants that handles all the other C/C++ escape character sequences: the ASCII code literal character constant. This literal character constant form uses the syntax:

```
                                #integer_constant
```

This form creates a character constant whose value is the ASCII code specified by *integer_constant*. The numeric constant can be a decimal, hexadecimal, or binary value, e.g.,

```
        #13    #$d    #%1101        // All three are the same character, a
                                    //   carriage return.
```

Since you may concatenate character literals with strings, and the *#constant* form is a character literal, the following are all legal strings:

```
    “Hello World” #13 #10           // #13 #10 is the Windows newline sequence
                                    //  (carriage return followed by line feed).

    “Error: Bad Value” #7           // #7 is the bell character.
    “He wrote a “ #$22 “Hello World” #$22 “ program as an example.”
```

Since $22 is the ASCII code for the quote character, this last example is yet a third form of the “He wrote...” string literal.

1.4.3 String and Text Constants in the CONST Section

String and text constants in the CONST section use the following declaration syntax:

```
const
    AStringConst:     string := “123”;
    ATextConst:       text   := “123”;
```

Other than the data type of these two constants, their declarations are identical. However, their behavior in an HLA program is quite different.

Whenever HLA encounters a symbolic string constant within your program, it substitutes the string literal constant in place of the string name. So a statement like “stdout.put(AStringConst);” prints the string “123” (without quotes, of course) to the display. No real surprise here.

Whenever HLA encounters a symbolic text constant within your program, it substitutes the text of that string (rather than the string literal constant) for the identifier. That is, HLA substitutes the characters between the delimiting quotes in place of the symbolic text constant. Therefore, the following statement is perfectly legal given the declarations above:

```
        mov( ATextConst, al );       // equivalent to mov( 123, al );
```

Note that substituting *AStringConst* for *ATextConst* in this example is illegal:

```
        mov( AStringConst, al );     // equivalent to mov( "123", al );
```

This latter example is illegal because you cannot move a string literal constant into the AL register.

Whenever HLA encounters a symbolic text constant in your program, it immediately substitutes the value of the text constant's string for that text constant and continues the compilation as though you had written the text constant's value rather than the symbolic identifier in your program. This can save some typing and help make your programs a little more readable if you often enter some sequence of text in your program. For example, consider the *nl* (newline) text constant declaration found in the HLA stdio.hhf library header file:

```
const
    nl: text := "#$d #$a";  // Windows version.  Linux is just a line feed.
```

Whenever HLA encounters the symbol *nl*, it immediately substitutes the value of the string "#$d #$a" for the *nl* identifier. When HLA sees the #$d (carriage return) character constant followed by the #$a (line feed) character constants, it concatenates the two to form the string containing the Windows newline sequence (a carriage return followed by a line feed). Consider the following two statements:

```
        stdout.put( "Hello World", nl );
        stdout.put( "Hello World"  nl );
```

(Notice that the second statement above does not separate the string literal and the nl symbol with a comma.) In the first example, HLA emits code that prints the string "Hello World" and then emits some additional code that prints a newline sequence. In the second example, HLA expands the *nl* symbol as follows:

```
        stdout.put( "Hello World" #$d #$a );
```

Now HLA sees a string literal constant ("Hello World") followed by two character constants. It concatenates the three of them together to form a single string and then prints this string with a single call. Therefore, leaving off the comma between the string literal and the *nl* symbol produces slightly more efficient code. Keep in mind that this only works with string literal constants. You cannot concatenate string variables, or a string variable with a string literal, by using this technique.

Linux users should note that the Linux end of line sequence is just a single linefeed character. Therefore, the declaration for *nl* is slightly different in Linux.

In the constant section, if you specify only a constant identifier and a string constant (i.e., you do not supply a type), HLA defaults to type *string*. If you want to declare a *text* constant you must explicitly supply the type.

```
const
    AStrConst := "String Constant";
    ATextConst: text := "mov( 0, eax );";
```

1.4.4 Constant Expressions

Thus far, this chapter has given the impression that a symbolic constant definition consists of an identifier, an optional type, and a literal constant. Actually, HLA constant declarations can be a lot more sophisticated than this because HLA allows the assignment of a constant expression, not just a literal constant, to a symbolic constant. The generic constant declaration takes one of the following two forms:

```
        Identifier : typeName := constant_expression ;
        Identifier := constant_expression ;
```

Constant expressions take the familiar form you're used to in high level languages like C/C++ and Pascal. They may contain literal constant values, previously declared symbolic constants, and various arithmetic operators. The following lists some of the operations possible in a constant expression:

```
Arithmetic Operators

    -       (unary negation)  Negates the expression immediately following the "-".
    *       Multiplies the integer or real values around the asterisk.
    div     Divides the left integer operand by the right integer operand
            producing an integer (truncated) result.
    mod     Divides the left integer operand by the right integer operand
            producing an integer remainder.
    /       Divides the left numeric operand by the second numeric operand
            producing a floating point result.
    +       Adds the left and right numeric operands.
    -       Subtracts the right numeric operand from the left numeric operand.

Comparison Operators

    =, ==   Compares left operand with right operand. Returns TRUE if equal.
    <>, !=  Compares left operand with right operand. Returns TRUE if not equal.
    <       Returns true if left operand is less than right operand.
    <=      Returns true if left operand is <= right operand.
    >       Returns true if left operand is greater than right operand.
    >=      Returns true if left operand is >= right operand.

Logical Operators[5]:

    &       For boolean operands, returns the logical AND of the two operands.
    |       For boolean operands, returns the logical OR of the two operands.
    ^       For boolean operands, returns the logical exclusive-OR.
    !       Returns the logical NOT of the single operand following "!".

Bitwise Logical Operators:

    &       For integer numeric operands, returns bitwise AND of the operands.
    |       For integer numeric operands, returns bitwise OR of the operands.
    ^       For integer numeric operands, returns bitwise XOR of the operands.
    !       For an integer numeric operand, returns bitwise NOT of the operand.

String Operators:

    '+'     Returns the concatenation of the left and right string operands.
```

The constant expression operators follow standard precedence rules; you may use the parentheses to override the precedence if necessary. See the HLA reference in the appendix for the exact precedence relationships between the operators. In general, if the precedence isn't obvious, use parentheses to exactly state the order of evaluation. HLA actually provides a few more operators than these, though the ones above are the ones you will most commonly use. Please see the HLA documentation for a complete list of constant expression operators.

If an identifier appears in a constant expression, that identifier must be a constant identifier that you have previously defined in your program. You may not use variable identifiers in a constant expression; their values are not defined at compile-time when HLA evaluates the constant expression. Also, don't confuse compile-time and run-time operations:

```
// Constant expression, computed while HLA is compiling your program:
```

5. Note to C/C++ and Java users. HLA's constant expressions use complete boolean evaluation rather than short-circuit boolean evaluation. Hence, HLA constant expressions do not behave identically to C/C++/Java expressions.

```
const
       x      := 5;
       y      := 6;
       Sum    := x + y;


// Run-time calculation, computed while your program is running, long after
// HLA has compiled it:

    mov( x, al );
    add( y, al );
```

HLA directly interprets the value of a constant expression during compilation. It does not emit any machine instructions to compute "x+y" in the constant expression above. Instead, it directly computes the sum of these two constant values. From that point forward in the program, HLA associates the value 11 with the constant *Sum* just as if the program had contained the statement "Sum := 11;" rather than "Sum := x+y;" On the other hand, HLA does not precompute the value 11 in AL for the MOV and ADD instructions above[6], it faithfully emits the object code for these two instructions and the 80x86 computes their sum when the program is run (sometime after the compilation is complete).

In general, constant expressions don't get very sophisticated. Usually, you're adding, subtracting, or multiplying two integer values. For example, the following CONST section defines a set of constants that have consecutive values:

```
const
    TapeDAT    :=  1;
    Tape8mm    :=  TapeDAT + 1;
    TapeQIC80  :=  Tape8mm + 1;
    TapeTravan :=  TapeQIC80 + 1;
    TapeDLT    :=  TapeTravan + 1;
```

The constants above have the following values: TapeDAT = 1, Tape8mm = 2, TapeQIC80 = 3, TapeTravan = 4, and TapeDLT = 5.

1.4.5 Multiple CONST Sections and Their Order in an HLA Program

Although CONST sections must appear in the declaration section of an HLA program (e.g., between the "PROGRAM *pgmname*;" header and the corresponding "BEGIN *pgmname*;" statement), they do not have to appear before or after any other items in the declaration section. In fact, like the variable declaration sections, you can place multiple CONST sections in the declaration section. The only restriction on HLA constant declarations is that you must declare any constant symbol before you use it in your program.

Some C/C++ programmers, for example, are more comfortable writing their constant declarations as follows (since this is closer to C/C++'s syntax for declaring constants):

```
const  TapeDAT    :=  1;
const  Tape8mm    :=  TapeDAT + 1;
const  TapeQIC80  :=  Tape8mm + 1;
const  TapeTravan :=  TapeQIC80 + 1;
const  TapeDLT    :=  TapeTravan + 1;
```

The placement of the CONST section in a program seems to be a personal issue among programmers. Other than the requirements of defining all constants before you use them, you may feel free to insert the constant declaration section anywhere in the declaration section. Some programmers prefer to put all their

6. Technically, if HLA had an optimizer it could replace these two instructions with a single "MOV(11, al);" instruction. HLA v1.x, however, does not do this.

CONST declarations at the beginning of their declaration section, some programmers prefer to spread them throughout declaration section, defining the constants just before they need them for some other purpose. Putting all your constants at the beginning of an HLA declaration section is probably the wisest choice right now. Later in this text you'll see reasons why you might want to define your constants later in a declaration section.

1.4.6 The HLA VAL Section

You cannot change the value of a constant you define in the CONST section. While this seems perfectly reasonable (constants after all, are supposed to be, well, constant), there are different ways we can define the term *constant* and CONST objects only follow the rules of one specific definition. HLA's VAL section lets you define constant objects that follow slightly different rules. This section will discuss the VAL section and the difference between VAL constants and CONST constants.

The concept of "*const-ness*" can exist at two different times: while HLA is compiling your program and later when your program executes (and HLA is no longer running). All reasonable definitions of a constant require that a value not change while the program is running. Whether or not the value of a "constant" can change during compilation is a separate issue. The difference between HLA CONST objects and HLA VAL objects is whether the value of the constant can change during compilation.

Once you define a constant in the CONST section, the value of that constant is immutable from that point forward *both at run-time and while HLA is compiling your program*. Therefore, an instruction like "mov(SymbolicCONST, EAX);" always moves the same value into EAX, regardless of where this instruction appears in the HLA main program. Once you define the symbol *SymbolicCONST* in the CONST section, this symbol has the same value from that point forward.

The HLA VAL section lets you declare symbolic constants, just like the CONST section. However, HLA VAL constants can change their value throughout the source code in your program. The following HLA declarations are perfectly legal:

```
val    InitialValue  := 0;
const  SomeVal       := InitialValue + 1;     // = 1
const  AnotherVal    := InitialValue + 2;     // = 2

val    InitialValue  := 100;
const  ALargerVal    := InitialValue;         // = 100
const  LargeValTwo   := InitialValue*2;       // = 200
```

All of the symbols appearing in the CONST sections use the symbolic value *InitialValue* as part of the definition. Note, however, that *InitialValue* has different values at different points in this code sequence; at the beginning of the code sequence *InitialValue* has the value zero, while later it has the value 100.

Remember, at run-time a VAL object is not a variable; it is still a manifest constant and HLA will substitute the current value of a VAL identifier for that identifier[7]. Statements like "MOV(25, InitialValue);" are no more legal than "MOV(25, 0);" or "MOV(25, 100);"

1.4.7 Modifying VAL Objects at Arbitrary Points in Your Programs

If you declare all your VAL objects in the declaration section, it would seem that you would not be able to change the value of a VAL object between the BEGIN and END statements of your program. After all, the VAL section must appear in the declaration section of the program and the declaration section ends before the BEGIN statement. Later, you will learn that most VAL object modifications occur between the BEGIN and END statements; hence, HLA must provide someway to change the value of a VAL object outside the declaration section. The mechanism to do this is the "?" operator.

7. In this context, *current* means the value last assigned to a VAL object looking backward in the source code.

Not only does HLA allow you to change the value of a VAL object outside the declaration section, it allows you to change the value of a VAL object almost *anywhere* in the program. Anywhere a space is allowed inside an HLA program, you can insert a statement of the form:

```
? ValIdentifier := constant_expression ;
```

This means that you could write a short program like the following:

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

val
    NotSoConstant := 0;

begin VALdemo;

    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

    ?NotSoConstant := 10;
    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

    ?NotSoConstant := 20;
    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

    ?NotSoConstant := 30;
    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

end VALdemo;
```

Program 1.4 Demonstration of VAL Redefinition Using “?” Operator

You probably won’t have much use for VAL objects at this time. However, later on you’ll see (in the chapter on the HLA compile-time language) how useful VAL objects can be to you.

1.5 The HLA TYPE Section

Let’s say that you simply do not like the names that HLA uses for declaring byte, word, double word, real, and other variables. Let’s say that you prefer Pascal’s naming convention or, perhaps, C’s naming convention. You want to use terms like *integer*, *float*, *double*, or whatever. If this were Pascal you could redefine the names in the **type** section of the program. With C you could use a **#define** or a **typedef** statement to accomplish the task. Well, HLA, like Pascal, has it’s own TYPE statement that also lets you create aliases of these names. The following example demonstrates how to set up some C/C++/Pascal compatible names in your HLA programs:

```
type
   integer:   int32;
```

```
    float:     real32;
    double:    real64;
    colors:    byte;
```

Now you can declare your variables with more meaningful statements like:

```
static
    i:             integer;
    x:             float;
    HouseColor:    colors;
```

If you are an Ada, C/C++, or FORTRAN programmer (or any other language, for that matter), you can pick type names you're more comfortable with. Of course, this doesn't change how the 80x86 or HLA reacts to these variables one iota, but it does let you create programs that are easier to read and understand since the type names are more indicative of the actual underlying types. One warning for C/C++ programmers: don't get too excited and go off and define an *int* data type. Unfortunately, INT is an 80x86 machine instruction (interrupt) and therefore, this is a reserved word in HLA.

The TYPE section is useful for much more than creating type isomorphism (that is, giving a new name to an existing type). The following sections will demonstrate many of the possible things you can do in the TYPE section.

1.6 ENUM and HLA Enumerated Data Types

In a previous section discussing constants and constant expressions, you saw the following example:

```
const  TapeDAT    :=  1;
const  Tape8mm    :=  TapeDAT + 1;
const  TapeQIC80  :=  Tape8mm + 1;
const  TapeTravan :=  TapeQIC80 + 1;
const  TapeDLT    :=  TapeTravan + 1;
```

This example demonstrates how to use constant expressions to develop a set of constants that contain unique, consecutive, values. There are, however, a couple of problems with this approach. First, it involves a lot of typing (and extra reading when reviewing this program). Second, it's very easy make a mistake when creating long lists of unique constants and reuse or skip some values. The HLA ENUM type provides a better way to create a list of constants with unique values.

ENUM is an HLA type declaration that lets you associate a list of names with a new type. HLA associates a unique value with each name (that is, it *enumerates* the list). The ENUM keyword typically appears in the TYPE section and you use it as follows:

```
type
    enumTypeID:    enum { comma_separated_list_of_names };
```

The symbol *enumTypeID* becomes a new type whose values are specified by the specified list of names. As a concrete example, consider the data type *TapeDrives* and a corresponding variable declaration of type *TypeDrives*:

```
type
    TapeDrives: enum{ TapeDAT, Tape8mm, TapeQIC80, TapeTravan, TapeDLT};

static
    BackupUnit:    TapeDrives := TapeDAT;

    .
    .
    .

    mov( BackupUnit, al );
```

```
if( al = Tape8mm ) then

    ...

endif;

// etc.
```

By default, HLA reserves one byte of storage for enumerated data types. So the *BackupUnit* variable will consume one byte of memory and you would typically use an eight-bit register to access it[8]. As for the constants, HLA associates consecutive *uns8* constant values starting at zero with each of the enumerated identifiers. In the TapeDrives example, the tape drive identifiers would have the values *TapeDAT*=0, *Tape8mm*=1, *TapeQIC80*=2, *TapeTravan*=3, and *TapeDLT*=4. You may use these constants exactly as though you had defined them with these values in a CONST section.

1.7 Pointer Data Types

Some people refer to pointers as scalar data types, others refer to them as composite data types. This text will treat them as scalar data types even though they exhibit some tendencies of both scalar and composite data types.

Of course, the place to start is with the question "What is a pointer?" Now you've probably experienced pointers first hand in the Pascal, C, or Ada programming languages and you're probably getting worried right now. Almost everyone has a real bad experience when they first encounter pointers in a high level language. Well, fear not! Pointers are actually *easier* to deal with in assembly language. Besides, most of the problems you had with pointers probably had nothing to do with pointers, but rather with the linked list and tree data structures you were trying to implement with them. Pointers, on the other hand, have lots of uses in assembly language that have nothing to do with linked lists, trees, and other scary data structures. Indeed, simple data structures like arrays and records often involve the use of pointers. So if you've got some deep-rooted fear about pointers, well forget everything you know about them. You're going to learn how *great* pointers really are.

Probably the best place to start is with the definition of a pointer. Just exactly what is a pointer, anyway? Unfortunately, high level languages like Pascal tend to hide the simplicity of pointers behind a wall of abstraction. This added complexity (which exists for good reason, by the way) tends to frighten programmers because *they don't understand what's going on*.

Now if you're afraid of pointers, well, let's just ignore them for the time being and work with an array. Consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

Even if you don't know Pascal, the concept here is pretty easy to understand. M is an array with 1024 integers in it, indexed from *M[0]* to *M[1023]*. Each one of these array elements can hold an integer value that is independent of all the others. In other words, this array gives you 1024 different integer variables each of which you refer to by number (the array index) rather than by name.

If you encountered a program that had the statement "M[0]:=100;" you probably wouldn't have to think at all about what is happening with this statement. It is storing the value 100 into the first element of the array *M*. Now consider the following two statements:

```
i := 0; (* Assume "i" is an integer variable *)
M [i] := 100;
```

You should agree, without too much hesitation, that these two statements perform the same exact operation as "M[0]:=100;". Indeed, you're probably willing to agree that you can use any integer expression in the

8. HLA provides a mechanism by which you can specify that enumerated data types consume two or four bytes of memory. See the HLA documentation for more details.

range 0…1023 as an index into this array. The following statements *still* perform the same operation as our single assignment to index zero:

```
i := 5;    (* assume all variables are integers*)
j := 10;
k := 50;
m [i*j-k] := 100;
```

"Okay, so what's the point?" you're probably thinking. "Anything that produces an integer in the range 0…1023 is legal. So what?" Okay, how about the following:

```
M [1] := 0;
M [ M [1] ] := 100;
```

Whoa! Now that takes a few moments to digest. However, if you take it slowly, it makes sense and you'll discover that these two instructions perform the exact same operation you've been doing all along. The first statement stores zero into array element *M[1]*. The second statement fetches the value of *M[1]*, which is an integer so you can use it as an array index into *M*, and uses that value (zero) to control where it stores the value 100.

If you're willing to accept the above as reasonable, perhaps bizarre, but usable nonetheless, then you'll have no problems with pointers. *Because m[1] is a pointer!* Well, not really, but if you were to change "M" to "memory" and treat this array as all of memory, this is the exact definition of a pointer.

1.7.1 Using Pointers in Assembly Language

A pointer is simply a memory location whose value is the address (or index, if you prefer) of some other memory location. Pointers are very easy to declare and use in an assembly language program. You don't even have to worry about array indices or anything like that.

An HLA pointer is a 32 bit value that may contain the address of some other variable. If you have a dword variable *p* that contains $1000_0000, then *p* "points" at memory location $1000_0000. To access the dword that *p* points at, you could use code like the following:

```
mov( p, ebx );        // Load EBX with the value of pointer p.
mov( [ebx], eax );    // Fetch the data that p points at.
```

By loading the value of *p* into EBX this code loads the value $1000_0000 into EBX (assuming *p* contains $1000_0000 and, therefore, points at memory location $1000_0000). The second instruction above loads the EAX register with the word starting at the location whose offset appears in EBX. Since EBX now contains $1000_0000, this will load EAX from locations $1000_0000 through $1000_0003.

Why not just load EAX directly from location $1000_0000 using an instruction like "MOV(mem, EAX);" (assuming *mem* is at address $1000_0000)? Well, there are lots of reasons. But the primary reason is that this single instruction always loads EAX from location *mem*. You cannot change the location from which it loads EAX. The former instructions, however, always load EAX from the location where *p* is pointing. This is very easy to change under program control. In fact, the simple instruction "MOV(&mem2, p);" will cause those same two instructions above to load EAX from *mem2* the next time they execute. Consider the following instructions:

```
mov( &i, p );         // Assume all variables are STATIC variables.
    .
    .
    .
if( some_expression ) then


    mov( &j, p );     // Assume the code above skips this instruction and
    .                 // you get to the next instruction by jumping
    .                 // to this point from somewhere else.
    .
```

```
        endif;
        mov( p, ebx );       // Assume both of the above code paths wind up
        mov( [ebx], eax );   // down here.
```

This short example demonstrates two execution paths through the program. The first path loads the variable *p* with the address of the variable *i*. The second path through the code loads *p* with the address of the variable *j*. Both execution paths converge on the last two MOV instructions that load EAX with *i* or *j* depending upon which execution path was taken. In many respects, this is like a *parameter* to a procedure in a high level language like Pascal. Executing the same instructions accesses different variables depending on whose address (*i* or *j*) winds up in *p*.

1.7.2 Declaring Pointers in HLA

Since pointers are 32 bits long, you could simply use the dword directive to allocate storage for your pointers. However, there is a much better way to do this: HLA provides the POINTER TO phrase specifically for declaring pointer variables. Consider the following example:

```
static
    b:              byte;
    d:              dword;
    pByteVar:       pointer to byte := &b;
    pDWordVar:      pointer to dword := &d;
```

This example demonstrates that it is possible to initialize as well as declare pointer variables in HLA. Note that you may only take addresses of static variables (STATIC, READONLY, and STORAGE objects) with the address-of operator, so you can only initialize pointer variables with the addresses of static objects.

You can also define your own pointer types in the TYPE section of an HLA program. For example, if you often use pointers to characters, you'll probably want to use a TYPE declaration like the one in the following example:

```
type
    ptrChar:   pointer to char;

static
    cString: ptrChar;
```

1.7.3 Pointer Constants and Pointer Constant Expressions

HLA allows two literal pointer constant forms: the address-of operator followed by the name of a static variable or the constant zero. In addition to these two literal pointer constants, HLA also supports simple pointer constant expressions.

The constant zero represents the NULL or NIL pointer, that is, an illegal address that does not exist[9]. Programs typically initialize pointers with NULL to indicate that a pointer has explicitly *not* been initialized. The HLA Standard Library predefines both the "NULL" and "nil" constants in the memory.hhf header file[10].

In addition to simple address literals and the value zero, HLA allows very simple constant expressions wherever a pointer constant is legal. Pointer constant expressions take one of the two following forms:

```
        &StaticVarName + PureConstantExpression
        &StaticVarName - PureConstantExpression
```

9. Actually, address zero does exist, but if you try to access it under Windows or Linux you will get a general protection fault.
10. NULL is for C/C++ programmers and nil is familiar to Pascal/Delphi programmers.

The *PureConstantExpression* term is a numeric constant expression that does not involve any pointer constants. This type of expression produces a memory address that is the specified number of bytes before or after ("-" or "+", respectively) the *StaticVarName* variable in memory.

Since you can create pointer constant expressions, it should come as no surprise to discover that HLA lets you define manifest pointer constants in the CONST section. The following program demonstrates how you can do this.

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

static
    b:  byte := 0;
        byte    1, 2, 3, 4, 5, 6, 7;

const
    pb:= &b + 1;

begin PtrConstDemo;

    mov( pb, ebx );
    mov( [ebx], al );
    stdout.put( "Value at address pb = $", al, nl );

end PtrConstDemo;
```

Program 1.5 Pointer Constant Expressions in an HLA Program

Upon execution, this program prints the value of the byte just beyond *b* in memory (which contains the value $01).

1.7.4 Pointer Variables and Dynamic Memory Allocation

Pointer variables are the perfect place to store the return result from the HLA Standard Library *malloc* function. The *malloc* function returns the address of the storage it allocates in the EAX register; therefore, you can store the address directly into a pointer variable with a single MOV instruction immediately after a call to *malloc*:

```
type
    bytePtr:   pointer to byte;

var
    bPtr: bytePtr;
        .
        .
        .
    malloc( 1024 );          // Allocate a block of 1,024 bytes.
    mov( eax, bPtr );        // Store address of block in bPtr.
        .
        .
        .
    free( bPtr );     // Free the allocated block when done using it.
```

```
.
.
.
```

In addition to *malloc* and *free*, the HLA Standard Library provides a *realloc* procedure. The *realloc* routine takes two parameters, a pointer to a block of storage that *malloc* (or *realloc*) previously created, and a new size. If the new size is less than the old size, realloc releases the storage at the end of the allocated block back to the system. If the new size is larger than the current block, then *realloc* will allocate a new block and move the old data to the start of the new block, then free the old block.

Typically, you would use *realloc* to correct a bad guess about a memory size you'd made earlier. For example, suppose you want to read a set of values from the user but you won't know how many memory locations you'll need to hold the values until after the user has entered the last value. You could make a wild guess and then allocate some storage using *malloc* based on your estimate. If, during the input, you discover that your estimate was too low, simply call *realloc* with a larger value. Repeat this as often as required until all the input is read. Once input is complete, you can make a call to *realloc* to release any unused storage at the end of the memory block.

The *realloc* procedure uses the following calling sequence:

```
realloc( ExistingPointer, NewSize );
```

Realloc returns a pointer to the newly allocated block in the EAX register.

One danger exists when using *realloc*. If you've made multiple copies of pointers into a block of storage on the heap and then call *realloc* to resize that block, all the existing pointers are now invalid. Effectively *realloc* frees the existing storage and then allocates a new block. That new block may not be in the same memory location at the old block, so any existing pointers (into the block) that you have will be invalid after the *realloc* call.

1.7.5 Common Pointer Problems

There are five common problems programmers encounter when using pointers. Some of these errors will cause your programs to immediately stop with a diagnostic message; other problems are more subtle, yielding incorrect results without otherwise reporting an error or simply affecting the performance of your program without displaying an error. These five problems are

- Using an uninitialized pointer
- Using a pointer that contains an illegal value (e.g., NULL)
- Continuing to use malloc'd storage after that storage has been free'd
- Failing to free storage once the program is done using it
- Accessing indirect data using the wrong data type.

The first problem above is using a pointer variable before you have assigned a valid memory address to the pointer. Beginning programmers often don't realize that declaring a pointer variable only reserves storage for the pointer itself, it does not reserve storage for the data that the pointer references. The following short program demonstrates this problem:

```
// Program to demonstrate use of
// an uninitialized pointer.  Note
// that this program should terminate
// with a Memory Access Violation exception.

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

static
```

```
    // Note: by default, varibles in the
    // static section are initialized with
    // zero (NULL) hence the following
    // is actually initialized with NULL,
    // but that will still cause our program
    // to fail because we haven’t initialized
    // the pointer with a valid memory address.

    Uninitialized: pointer to byte;

begin UninitPtrDemo;

    mov( Uninitialized, ebx );
    mov( [ebx], al );
    stdout.put( “Value at address Uninitialized: = $”, al, nl );

end UninitPtrDemo;
```

Program 1.6 Uninitialized Pointer Demonstration

Although variables you declare in the STATIC section are, technically, initialized; static initialization still doesn’t initialize the pointer in this program with a valid address.

Of course, there is no such thing as a truly uninitialized variable on the 80x86. What you really have are variables that you’ve explicitly given an initial value and variables that just happen to inherit whatever bit pattern was in memory when storage for the variable was allocated. Much of the time, these garbage bit patterns laying around in memory don’t correspond to a valid memory address. Attempting to *dereference* such a pointer (that is, access the data in memory at which it points) raises a Memory Access Violation exception.

Sometimes, however, those random bits in memory just happen to correspond to a valid memory location you can access. In this situation, the CPU will access the specified memory location without aborting the program. Although to a naive programmer this situation may seem preferable to aborting the program, in reality this is far worse because your defective program continues to run with a defect without alerting you to the problem. If you store data through an uninitialized pointer, you may very well overwrite the values of other important variables in memory. This defect can produce some very difficult to locate problems in your program.

The second problem programmers have with pointers is storing invalid address values into a pointer. The first problem, above, is actually a special case of this second problem (with garbage bits in memory supplying the invalid address rather than you producing via a miscalculation). The effects are the same; if you attempt to dereference a pointer containing an invalid address you will either get a Memory Access Violation exception or you will access an unexpected memory location.

The third problem listed above is also known as the dangling pointer problem. To understand this problem, consider the following code fragment:

```
malloc( 256 );    // Allocate some storage.
mov( eax, ptr );  // Save address away in a pointer variable.
    .
    .             // Code that use the pointer variable “ptr”.
    .
free( ptr );      // Free the storage associated with “ptr”.
    .
    .             // Code that does not change the value in “ptr”.
    .
mov( ptr, ebx );
mov( al, [ebx] );
```

In this example you will note that the program allocates 256 bytes of storage and saves the address of that storage away in the *ptr* variable. Then the code uses this block of 256 bytes for a while and frees the storage, returning it to the system for other uses. Note that calling *free* does not change the value of *ptr* in any way; *ptr* still points at the block of memory allocated by *malloc* earlier. Indeed, *free* does not change any data in this block, so upon return from *free*, *ptr* still points at the data stored into the block by this code. However, note that the call to *free* tells the system that this 256-byte block of memory is no longer needed by the program and the system can use this region of memory for other purposes. The *free* function cannot enforce that fact that you will never access this data again, you are simply promising that you won't. Of course, the code fragment above breaks this promise; as you can see in the last two instructions above the program fetches the value in *ptr* and accesses the data it points at in memory.

The biggest problem with dangling pointers is that you can get away with using them a good part of the time. As long as the system doesn't reuse the storage you've free'd, using a dangling pointer produces no ill effects in your program. However, with each new call to *malloc*, the system may decide to reuse the memory released by that previous call to *free*. When this happens, any attempt to dereference the dangling pointer may produce some unintended consequences. The problems range from reading data that has been overwritten (by the new, legal, use of the data storage), to overwriting the new data, to (the worst case) overwriting system heap management pointers (doing so will probably cause your program to crash). The solution is clear: *never use a pointer value once you free the storage associated with that pointer.*

Of all the problems, the fourth (failing to free allocated storage) will probably have the least impact on the proper operation of your program. The following code fragment demonstrates this problem:

```
malloc( 256 );
mov( eax, ptr );
    .          // Code that uses the data where ptr is pointing.
    .          // This code does not free up the storage
    .          // associated with ptr.
malloc( 512 );
mov( eax, ptr );

// At this point, there is no way to reference the original
// block of 256 bytes pointed at by ptr.
```

In this example the program allocates 256 bytes of storage and references this storage using the *ptr* variable. At some later time, the program allocates another block of bytes and overwrites the value in *ptr* with the address of this new block. Note that the former value in *ptr* is lost. Since this address no longer exists in the program, there is no way to call *free* to return the storage for later use. As a result, this memory is no longer available to your program. While making 256 bytes of memory inaccessible to your program may not seem like a big deal, imagine now that this code is in a loop that repeats over and over again. With each execution of the loop the program loses another 256 bytes of memory. After a sufficient number of loop iterations, the program will exhaust the memory available on the heap. This problem is often called a *memory leak* because the effect is the same as though the memory bits were leaking out of your computer (yielding less and less available storage) during program execution[11].

Memory leaks are far less damaging than using dangling pointers. Indeed, there are only two problems with memory leaks: the danger of running out of heap space (which, ultimately, may cause the program to abort, though this is rare) and performance problems due to virtual memory page swapping. Nevertheless, you should get in the habit of always free all storage once you are done using it. Note that when your program quits, the operating system reclaims all storage including the data lost via memory leaks. Therefore, memory lost via a leak is only lost to your program, not the whole system.

The last problem with pointers is the lack of type-safe access. HLA cannot and does not enforce pointer type checking. For example, consider the following program:

11. Note that the storage isn't lost from you computer; once your program quits it returns all memory (including unfree'd storage) to the O/S. The next time the program runs it will start with a clean slate.

```
// Program to demonstrate use of
// lack of type checking in pointer
// accesses.

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

static
    ptr:    pointer to char;
    cnt:    uns32;

begin BadTypePtrDemo;

    // Allocate sufficient characters
    // to hold a line of text input
    // by the user:

    malloc( 256 );
    mov( eax, ptr );


    // Okay, read the text a character
    // at a time by the user:

    stdout.put( “Enter a line of text: “ );
    stdin.flushInput();
    mov( 0, cnt );
    mov( ptr, ebx );
    repeat

        stdin.getc();       // Read a character from the user.
        mov( al, [ebx] );   // Store the character away.
        inc( cnt );         // Bump up count of characters.
        inc( ebx );         // Point at next position in memory.

    until( stdin.eoln());


    // Okay, we’ve read a line of text from the user,
    // now display the data:

    mov( ptr, ebx );
    for( mov( cnt, ecx ); ecx > 0; dec( ecx )) do

        mov( [ebx], eax );
        stdout.put( “Current value is $”, eax, nl );
        inc( ebx );

    endfor;
    free( ptr );


end BadTypePtrDemo;
```

Program 1.7 Type-Unsafe Pointer Access Example

This program reads in data from the user as character values and then displays the data as double word hexadecimal values. While a powerful feature of assembly language is that it lets you ignore data types at will and automatically coerce the data without any effort, this power is a two-edged sword. If you make a mistake and access indirect data using the wrong data type, HLA and the 80x86 may not catch the mistake and your program may produce inaccurate results. Therefore, you need to take care when using pointers and indirection in your programs that you use the data consistently with respect to data type.

1.8 Putting It All Together

This chapter contains an eclectic combination of subjects. It begins with a discussion of the INTMUL, BOUND, and INTO instructions that will prove useful throughout this text. Then this chapter discusses how to declare constants and data types, including enumerated data types. This chapter also introduces constant expressions and pointers. The following chapters in this text will make extensive use of these concepts.

Introduction to Character Strings Chapter Two

2.1 Chapter Overview

This chapter discusses how to declare and use character strings in your programs. While not a complete treatment of this subject (additional material appears later in this text), this chapter will provide sufficient information to allow basic string manipulation within your HLA programs.

2.2 Composite Data Types

Composite data types are those that are built up from other (generally scalar) data types. This chapter will cover one of the more important composite data types – the character string. A string is a good example of a composite data type – it is a data structure built up from a sequence of individual characters and some other data.

2.3 Character Strings

After integer values, character strings are probably the most popular data type that modern programs use. The 80x86 does support a handful of string instructions, but these instructions are really intended for block memory operations, not a specific implementation of a character string. Therefore, this section will concentrate mainly on the HLA definition of character strings and also discuss the string handling routines available in the HLA Standard Library.

In general, a *character string* is a sequence of ASCII characters that possesses two main attributes: a *length* and the *character data*. Different languages use different data structures to represent strings. To better understand the reasoning behind HLA strings, it is probably instructive to look at two different string representations popularized by various high level languages.

Without question, *zero-terminated strings* are probably the most common string representation in use today because this is the native string format for C/C++ and programs written in C/C++. A zero terminated string consists of a sequence of zero or more ASCII characters ending with a byte containing zero. For example, in C/C++, the string "abc" requires four characters: the three characters 'a', 'b', and 'c' followed by a byte containing zero. As you'll soon see, HLA character strings are upwards compatible with zero terminated strings, but in the meantime you should note that it is very easy to create zero terminated strings in HLA. The easiest place to do this is in the STATIC section using code like the following:

```
static
    zeroTerminatedString: char; @nostorage;
            byte “This is the zero terminated string”, 0;
```

Remember, when using the @NOSTORAGE option, no space is actually reserved for a variable declaration, so the *zeroTerminatedString* variable's address in memory corresponds to the first character in the following BYTE directive. Whenever a character string appears in the BYTE directive as it does here, HLA emits each character in the string to successive memory locations. The zero value at the end of the string properly terminates this string.

Zero terminated strings have two principle attributes: they are very simple to implement and the strings can be any length. On the other hand, zero terminated string haves a few drawbacks. First, though not usually important, zero terminated strings cannot contain the NUL character (whose ASCII code is zero). Generally, this isn't a problem, but it does create havoc once in a great while. The second problem with zero terminated strings is that many operations on them are somewhat inefficient. For example, to compute the length of a zero terminated string you must scan the entire string looking for that zero byte (counting each character as you encounter it). The following program fragment demonstrates how to compute the length of the string above:

```
        mov( &zeroTerminatedString, ebx );
        mov( 0, eax );
        while( (type byte [ebx]) <> 0 ) do

            inc( ebx );
            inc( eax );

        endwhile;

        // String length is now in EAX.
```

As you can see from this code, the time it takes to compute the length of the string is proportional to the length of the string; as the string gets longer it will take longer to compute its length.

A second string format, *length-prefixed strings*, overcomes some of the problems with zero terminated strings. Length-prefixed strings are common in languages like Pascal; they generally consist of a length byte followed by zero or more character values. The first byte specifies the length of the string, the remaining bytes (up to the specified length) are the character data itself. In a length-prefixed scheme, the string "abc" would consist of the four bytes $03 (the string length) followed by 'a', 'b', and 'c'. You can create length prefixed strings in HLA using code like the following:

```
data
    lengthPrefixedString:char;
            byte 3, "abc";
```

Counting the characters ahead of time and inserting them into the byte statement, as was done here, may seem like a major pain. Fortunately, there are ways to have HLA automatically compute the string length for you.

Length-prefixed strings solve the two major problems associated with zero-terminated strings. It is possible to include the NUL character in length-prefixed strings and those operations on zero terminated strings that are relatively inefficient (e.g., string length) are more efficient when using length prefixed strings. However, length prefixed strings suffer from their own drawbacks. The principal drawback to length-prefixed strings, as described, is that they are limited to a maximum of 255 characters in length (assuming a one-byte length prefix).

HLA uses an expanded scheme for strings that is upwards compatible with both zero-terminated and length-prefixed strings. HLA strings enjoy the advantages of both zero-terminated and length-prefixed strings without the disadvantages. In fact, the only drawback to HLA strings over these other formats is that HLA strings consume a few additional bytes (the overhead for an HLA string is nine bytes compared to one byte for zero-terminated or length-prefixed strings; the overhead being the number of bytes needed above and beyond the actual characters in the string).

An HLA string value consists of four components. The first element is a double word value that specifies the maximum number of characters that the string can hold. The second element is a double word value specifying the current length of the string. The third component is the sequence of characters in the string. The final component is a zero terminating byte. You could create an HLA-compatible string in the STATIC section using the following code[1]:

```
static
        dword 11;
        dword 11;
    TheString: char; @nostorage;
        byte "Hello there";
        byte 0;
```

1. Actually, there are some restrictions on the placement of HLA strings in memory. This text will not cover those issues. See the HLA documentation for more details.

Note that the address associated with the HLA string is the address of the first character, not the maximum or current length values.

"So what is the difference between the current and maximum string lengths?" you're probably wondering. Well, in a fixed string like the above they are usually the same. However, when you allocate storage for a string variable at run-time, you will normally specify the maximum number of characters that can go into the string. When you store actual string data into the string, the number of characters you store must be less than or equal to this maximum value. The HLA Standard Library string routines will raise an exception if you attempt to exceed this maximum length (something the C/C++ and Pascal formats can't do).

The terminating zero byte at the end of the HLA string lets you treat an HLA string as a zero-terminated string if it is more efficient or more convenient to do so. For example, most calls to Windows and Linux require zero-terminated strings for their string parameters. Placing a zero at the end of an HLA string ensures compatibility with Windows, Linux, and other library modules that use zero-terminated strings.

2.4 HLA Strings

As noted in the previous section, HLA strings consist of four components: a maximum length, a current string length, character data, and a zero terminating byte. However, HLA never requires you to create string data by manually emitting these components yourself. HLA is smart enough to automatically construct this data for you whenever it sees a string literal constant. So if you use a string constant like the following, understand that somewhere HLA is creating the four-component string in memory for you:

```
stdout.put( "This gets converted to a four-component string by HLA" );
```

HLA doesn't actually work directly with the string data described in the previous section. Instead, when HLA sees a string object it always works with a *pointer* to that object rather than the object directly. Without question, this is the most important fact to know about HLA strings, and is the biggest source of problems beginning HLA programmers have with strings in HLA: *strings are pointers!* A string variable consumes exactly four bytes, the same as a pointer (because it *is* a pointer!). Having said all that, let's take a look at a simple string variable declaration in HLA:

```
static
        StrVariable:   string;
```

Since a string variable is a pointer, you must initialize it before you can use it. There are three general ways you may initialize a string variable with a legal string address: using static initializers, using the *stralloc* routine, or calling some other HLA Standard Library that initializes a string or returns a pointer to a string.

In one of the static declaration sections that allow initialized variables (STATIC, and READONLY) you can initialize a string variable using the standard initialization syntax, e.g.,

```
static
    InitializedString: string := "This is my string";
```

Note that this does not initialize the string variable with the string data. Instead, HLA creates the string data structure (see the previous section) in a special, hidden, memory segment and initializes the *InitializedString* variable with the address of the first character in this string (the "T" in "This"). *Remember, strings are pointers!* The HLA compiler places the actual string data in a read-only memory segment. Therefore, you cannot modify the characters of this string literal at run-time. However, since the string variable (a pointer, remember) is in the static section, you can change the string variable so that it points at different string data.

Since string variables are pointers, you can load the value of a string variable into a 32-bit register. The pointer itself points at the first character position of the string. You can find the current string length in the double word four bytes prior to this address, you can find the maximum string length in the double word eight bytes prior to this address. The following program demonstrates one way to access this data[2].

```
// Program to demonstrate accessing Length and Maxlength fields of a string.

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

static
    theString:string := “String of length 19”;

begin StrDemo;

    mov( theString, ebx );  // Get pointer to the string.

    mov( [ebx-4], eax );    // Get current length
    mov( [ebx-8], ecx );    // Get maximum length

    stdout.put
    (
        “theString = ‘”, theString, “‘”, nl,
        “length( theString )= “, (type uns32 eax ), nl,
        “maxLength( theString )= “, (type uns32 ecx ), nl
    );

end StrDemo;
```

Program 2.1 Accessing the Length and Maximum Length Fields of a String

When accessing the various fields of a string variable it is not wise to access them using fixed numeric offsets as done in this example. In the future, the definition of an HLA string may change slightly. In particular, the offsets to the maximum length and length fields are subject to change. A safer way to access string data is to coerce your string pointer using the *str.strRec* data type. The *str.strRec* data type is a record data type (see “Records, Unions, and Name Spaces” on page 483) that defines symbolic names for the offsets of the length and maximum length fields in the string data type. Were the offsets to the length and maximum length fields to change in a future version of HLA, then the definitions in *str.strRec* would also change, so if you use *str.strRec* then recompiling your program would automatically make any necessary changes to your program.

To use the *str.strRec* data type properly, you must first load the string pointer into a 32-bit register, e.g., “MOV(SomeString, EBX);” Once the pointer to the string data is in a register, you can coerce that register to the *str.strRec* data type using the HLA construct “(type str.strRec [EBX])”. Finally, to access the length or maximum length fields, you would use either “(type str.strRec [EBX]).length” or “(type str.strRec [EBX]).MaxStrLen” (respectively). Although there is a little more typing involved (versus using simple offsets like “-4” or “-8”), these forms are far more descriptive and much safer than straight numeric offsets. The following program corrects the previous example by using the *str.strRec* data type.

```
// Program to demonstrate accessing Length and Maxlength fields of a string.

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

static
```

2. Note that this scheme is not recommended. If you need to extract the length information from a string, use the routines provided in the HLA string library for this purpose.

```
    theString:string := "String of length 19";

begin LenMaxlenDemo;

    mov( theString, ebx );  // Get pointer to the string.

    mov( (type str.strRec [ebx]).length, eax );     // Get current length
    mov( (type str.strRec [ebx]).MaxStrLen, ecx );  // Get maximum length

    stdout.put
    (
        "theString = '", theString, "'", nl,
        "length( theString )= ", (type uns32 eax ), nl,
        "maxLength( theString )= ", (type uns32 ecx ), nl
    );

end LenMaxlenDemo;
```

Program 2.2 Correct Way to Access Length and MaxStrLen Fields of a String

A second way to manipulate strings in HLA is to allocate storage on the heap to hold string data. Because strings can't directly use pointers returned by *malloc* (since strings need to access eight bytes prior to the pointer address), you shouldn't use *malloc* to allocate storage for string data. Fortunately, the HLA Standard Library memory module provides a memory allocation routine specifically designed to allocate storage for strings: *stralloc*. Like *malloc*, *stralloc* expects a single dword parameter. This value specifies the (maximum) number of characters needed in the string. The *stralloc* routine will allocate the specified number of bytes of memory, plus between nine and thirteen additional bytes to hold the extra string information[3].

The *stralloc* routine will allocate storage for a string, initialize the maximum length to the value passed as the *stralloc* parameter, initialize the current length to zero, and store a zero (terminating byte) in the first character position of the string. After all this, *stralloc* returns the address of the zero terminating byte (that is, the address of the first character element) in the EAX register.

Once you've allocated storage for a string, you can call various string manipulation routines in the HLA Standard Library to operate on the string. The next section will discuss the HLA string routines in detail; this section will introduce a couple of string related routines for the sake of example. The first such routine is the "stdin.gets(strvar)". This routine reads a string from the user and stores the string data into the string storage pointed at by the string parameter (*strvar* in this case). If the user attempts to enter more characters than you've allocated for the string, then stdin.gets raises the *ex.StringOverflow* exception. The following program demonstrates the use of *stralloc*.

```
// Program to demonstrate stralloc and stdin.gets.

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

static
    theString:string;

begin strallocDemo;

    stralloc( 16 );         // Allocate storage for the string and store
```

3. *Stralloc* may allocate more than nine bytes for the overhead data because the memory allocated to an HLA string must always be double word aligned and the total length of the data structure must be an even multiple of four.

```
    mov( eax, theString );  //  the pointer into the string variable.

    // Prompt the user and read the string from the user:

    stdout.put( “Enter a line of text (16 chars, max): “ );
    stdin.flushInput();
    stdin.gets( theString );

    // Echo the string back to the user:

    stdout.put( “The string you entered was: “, theString, nl );

end strallocDemo;
```

Program 2.3 Reading a String from the User

If you look closely, you see a slight defect in the program above. It allocates storage for the string by calling *stralloc* but it never frees the storage allocated. Even though the program immediately exits after the last use of the string variable, and the operating system will deallocate the storage anyway, it's always a good idea to explicitly free up any storage you allocate. Doing so keeps you in the habit of freeing allocated storage (so you don't forget to do it when it's important) and, also, programs have a way of growing such that an innocent defect that doesn't affect anything in today's program becomes a show-stopping defect in tomorrow's version.

To free storage allocated via *stralloc*, you must call the corresponding *strfree* routine, passing the string pointer as the single parameter. The following program is a correction of the previous program with this minor defect corrected:

```
// Program to demonstrate stralloc, strfree, and stdin.gets.

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

static
    theString:string;

begin strfreeDemo;

    stralloc( 16 );         // Allocate storage for the string and store
    mov( eax, theString );  //  the pointer into the string variable.

    // Prompt the user and read the string from the user:

    stdout.put( “Enter a line of text (16 chars, max): “ );
    stdin.flushInput();
    stdin.gets( theString );

    // Echo the string back to the user:

    stdout.put( “The string you entered was: “, theString, nl );

    // Free up the storage allocated by stralloc:

    strfree( theString );

end strfreeDemo;
```

Program 2.4 Corrected Program that Reads a String from the User

When looking at this corrected program, please take note that the *stdin.gets* routine expects you to pass it a string parameter that points at an allocated string object. Without question, one of the most common mistakes beginning HLA programmers make is to call *stdin.gets* and pass it a string variable that has not been initialized. This may be getting old now, but keep in mind that *strings are pointers!* Like pointers, if you do not initialize a string with a valid address, your program will probably crash when you attempt to manipulate that string object. The call to *stralloc* plus moving the returned result into *theString* is how the programs above initialize the string pointer. If you are going to use string variables in your programs, you must ensure that you allocate storage for the string data prior to writing data to the string object.

Allocating storage for a string option is such a common operation that many HLA Standard Library routines will automatically do the allocation to save you the effort. Generally, such routines have an "a_" prefix as part of their name. For example, the *stdin.a_gets* combines a call to *stralloc* and *stdin.gets* into the same routine. This routine, which doesn't have any parameters, reads a line of text from the user, allocates a string object to hold the input data, and then returns a pointer to the string in the EAX register. The following program is an adaptation of the previous two programs that uses *stdin.a_gets*:

```
// Program to demonstrate  strfree and stdin.a_gets.

program strfreeDemo2;
#include( “stdlib.hhf” );

static
    theString:string;

begin strfreeDemo2;


    // Prompt the user and read the string from the user:

    stdout.put( “Enter a line of text: “ );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, theString );

    // Echo the string back to the user:

    stdout.put( “The string you entered was: “, theString, nl );

    // Free up the storage allocated by stralloc:

    strfree( theString );

end strfreeDemo2;
```

Program 2.5 Reading a String from the User with stdin.a_gets

Note that, as before, you must still free up the storage *stdin.a_gets* allocates by calling the *strfree* routine. One big difference between this routine and the previous two is the fact that HLA will automatically allocate exactly enough space for the string read from the user. In the previous programs, the call to *stralloc* only allocates 16 bytes. If the user types more than this then the program raises an exception and quits. If the user types less than 16 characters, then some space at the end of the string is wasted. The *stdin.a_gets* routine, on the other hand, always allocates the minimum necessary space for the string read from the user. Since it allocates the storage, there is little chance of overflow[4].

2.5 Accessing the Characters Within a String

Extracting individual characters from a string is a very common and easy task. In fact, it is so easy that HLA doesn't provide any specific procedure or language syntax to accomplish this - it's easy enough just to use machine instructions to accomplish this. Once you have a pointer to the string data, a simple indexed addressing mode will do the rest of the work for you.

Of course, the most important thing to keep in mind is that *strings are pointers*. Therefore, you cannot apply an indexed addressing mode directly to a string variable an expect to extract characters from the string. I.e, if *s* is a string variable, then "MOV(s[ebx], al);" does not fetch the character at position EBX in string *s* and place it in the AL register. Remember, *s* is just a pointer variable, an addressing mode like *s[ebx]* will simply fetch the byte at offset EBX in memory starting at the address of *s* (see Figure 2.1).

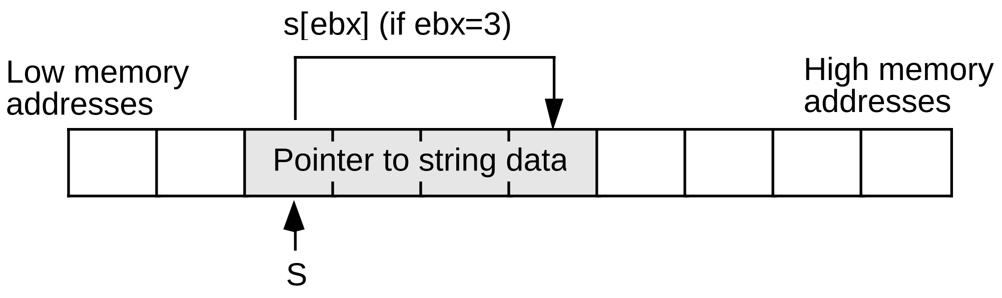

Figure 2.1 Incorrectly Indexing Off a String Variable

In Figure 2.1, assuming EBX contains three, "s[ebx]" does not access the fourth character in the string *s*, instead it fetches the fourth byte of the pointer to the string data. It is very unlikely that this is the desired effect you would want. Figure 2.2 shows the operation that is necessary to fetch a character from the string, assuming EBX contains the value of *s*:

4. Actually, there are limits on the maximum number of characters that stdin.a_gets will allocate. This is typically between 1,024 bytes and 4,096 bytes; See the HLA Standard Library source listings for the exact value.

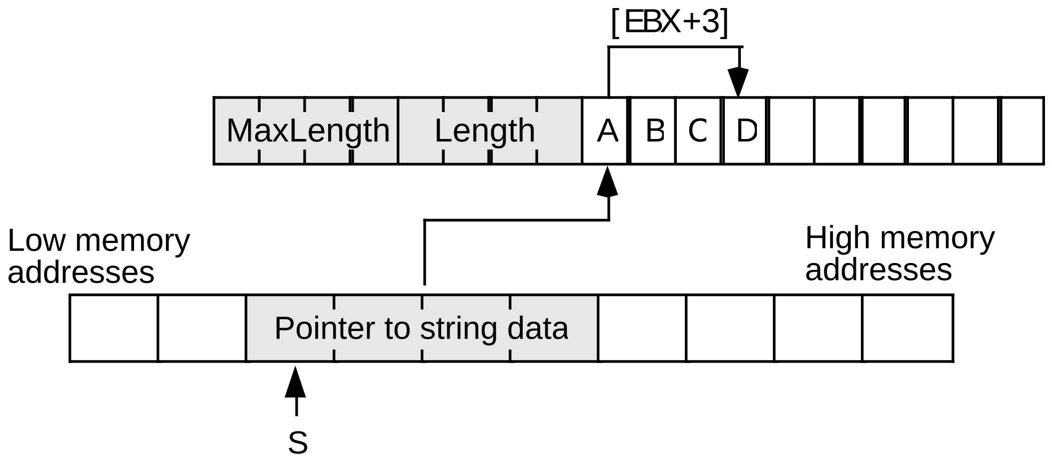

Figure 2.2 Correctly Indexing Off the Value of a String Variable

In Figure 2.2 EBX contains the value of string *s*. The value of *s* is a pointer to the actual string data in memory. Therefore, EBX will point at the first character of the string when you load the value of *s* into EBX. The following code demonstrates how to access the fourth character of string *s* in this fashion:

```
mov( s, ebx );          // Get pointer to string data into EBX.
mov( [ebx+3], al );     // Fetch the fourth character of the string.
```

If you want to load the character at a variable, rather than fixed, offset into the string, then you can use one of the 80x86's scaled indexed addressing modes to fetch the character. For example, if an *uns32* variable index contains the desired offset into the string, you could use the following code to access the character at *s[index]*:

```
mov( s, ebx );          // Get address of string data into EBX.
mov( index, ecx );      // Get desired offset into string.
mov( [ebx+ecx], al );   // Get the desired character into AL.
```

There is only one problem with the code above- it does not check to ensure that the character at offset *index* actually exists. If *index* is greater than the current length of the string, then this code will fetch a garbage byte from memory. Unless you can apriori determine that *index* is always less than the length of the string, code like this is dangerous to use. A better solution is to check the index against the string's current length before attempting to access the character. the following code provides one way to do this.

```
mov( s, ebx );
mov( index, ecx );
if( ecx < (type str.strRec [ebx]).Length ) then

    mov( [ebx+ecx], al );

else

    << error, string index is of bounds >>

endif;
```

In the ELSE portion of this IF statement you could take corrective action, print an error message, or raise an exception. If you want to explicitly raise an exception, you can use the HLA RAISE statement to accomplish this. The syntax for the RAISE statement is

```
raise( integer_constant );
raise( reg32 );
```

The value of the *integer_constant* or 32-bit register must be an exception number. Usually, this is one of the predefined constants in the excepts.hhf header file. An appropriate exception to raise when a string index is greater than the length of the string is *ex.StringIndexError*. The following code demonstrates raising this exception if the string index is out of bounds:

```
        mov( s, ebx );
        mov( index, ecx );
        if( ecx < (type str.strRec [ebx]).Length ) then

            mov( [ebx+ecx], al );

        else

            raise( ex.StringIndexError );

        endif;
```

2.6 The HLA String Module and Other String-Related Routines

Although HLA provides a powerful definition for string data, the real power behind HLA's string capabilities lies in the HLA Standard Library, not in the definition of HLA string data. HLA provides several dozen string manipulation routines that far exceed the capabilities found in standard HLLs like C/C++, Java, or Pascal; indeed, HLA's string handling capabilities rival those in string processing languages like Icon or SNOBOL4. While it is premature to introduce all of HLA's character string handling routines, this chapter will discuss many of the string facilities that HLA provides.

Perhaps the most basic string operation you will need is to assign one string to another. There are three different ways to assign strings in HLA: by reference, by copying a string, and by duplicating a string. Of these, assignment by reference is the fastest and easiest. If you have two strings and you wish to assign one string to the other, a simple and fast way to do this is to copy the string pointer. The following code fragment demonstrates this:

```
static
        string1:   string     := “Some String Data”;
        string2:   string;
            .
            .
            .
        mov( string1, eax );
        mov( eax, string2 );
            .
            .
            .
```

String assignment by reference is very efficient because it only involves two simple MOV instructions, regardless of the actual length of the string. Assignment by reference works great if you never modify the string data after the assignment operation. Do keep in mind, though, that both string variables (string1 and string2 in the example above) *wind up pointing at the same data*. So if you make a change to the data pointed at by one string variable, you will change the string data pointed at by the second string object since both objects point at the same data. The following program demonstrates this problem:

```
// Program to demonstrate the problem
// with string assignment by reference.

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

static
    string1:    string;
    string2:    string;

begin strRefAssignDemo;

    // Get a value into string1

    forever

        stdout.put( “Enter a string with at least three characters: “ );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( “Please enter a string with at least three chars.” nl );

    endfor;

    stdout.put( “You entered: ‘”, string1, “‘” nl );

    // Do the string assignment by copying the pointer

    mov( string1, ebx );
    mov( ebx, string2 );

    stdout.put( “String1= ‘”, string1, “‘” nl );
    stdout.put( “String2= ‘”, string2, “‘” nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
    // a string pointer always points at the first character
    // position in the string and we know we’ve got at least
    // three characters here).

    mov( ‘a’, (type char [ebx]) );
    mov( ‘b’, (type char [ebx+1]) );
    mov( ‘c’, (type char [ebx+2]) );

    // Okay, demonstrate the problem with assignment via
    // pointer copy.

    stdout.put
    (
        “After assigning ‘abc’ to the first three characters in string1:”
        nl
        nl
    );
    stdout.put( “String1= ‘”, string1, “‘” nl );
    stdout.put( “String2= ‘”, string2, “‘” nl );

    strfree( string1 );     // Don’t free string2 as well!


end strRefAssignDemo;
```

Program 2.6 Problem with String Assignment by Copying Pointers

Since both *string1* and *string2* point at the same string data in this example, any change you make to one string is reflected in the other. While this is sometimes acceptable, most programmers expect assignment to produce a different copy of a string; they expect the semantics of string assignment to produce two unique copies of the string data.

An important point to remember when using *copy by reference* (this term means copying a pointer rather than copying the actual data) is that you have created an *alias* to the string data. The term “alias” means that you have two names for the same object in memory (e.g., in the program above, *string1* and *string2* are two different names for the same string data). When you read a program it is reasonable to expect that different variables refer to different memory objects. Aliases violate this rule, thus making your program harder to read and understand because you’ve got to remember that aliases do not refer to different objects in memory. Failing to keep this in mind can lead to subtle bugs in your program. For instance, in the example above you have to remember that *string1* and *string2* are aliases so as not to free both objects at the end of the program. Worse still, you to remember that *string1* and *string2* are aliases so that you don’t continue to use *string2* after freeing *string1* in this code since *string2* would be a dangling reference at that point.

Since using copy by reference makes your programs harder to read and increases the possibility that you might introduce subtle defects in your programs, you might wonder why someone would use copy by reference at all. There are two reasons for this: first, copy by reference is very efficient; it only involves the execution of two MOV instructions. Second, some algorithms actually depend on copy by reference semantics. Nevertheless, you should carefully consider whether copying string pointers is the appropriate way to do a string assignment in your program before using this technique.

The second way to assign one string to another is to actually copy the string data. The HLA Standard Library *str.cpy* routine provides this capability. A call to the *str.cpy* procedure using the following form:

```
str.cpy( source_string, destination_string );
```

The source and destination strings must be string variables (pointers) or 32-bit registers containing the addresses of the string data in memory.

The *str.cpy* routine first checks the maximum length field of the destination string to ensure that it is at least as big as the current length of the source string. If it is not, then *str.cpy* raises the *ex.StringOverflow* exception. If the maximum string length field of the destination string is at least as big as the current string length of the source string, then *str.cpy* copies the string length, the characters, and the zero terminating byte from the source string to the data area at which the destination string points. When this process is complete, the two strings point at identical data, but they do not point at the same data in memory[5]. The following program is a rework of the previous example using *str.cpy* rather than copy by reference.

```
// Program to demonstrate string assignment using str.cpy.

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

static
    string1:    string;
    string2:    string;

begin strcpyDemo;
```

5. Unless, of course, both string pointers contained the same address to begin with, in which case str.cpy copies the string data over the top of itself.

```
    // Allocate storage for string2:

    stralloc( 64 );
    mov( eax, string2 );

    // Get a value into string1

    forever

        stdout.put( “Enter a string with at least three characters: “ );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( “Please enter a string with at least three chars.” nl );

    endfor;


    // Do the string assignment via str.cpy

    str.cpy( string1, string2 );

    stdout.put( “String1= ‘”, string1, “‘” nl );
    stdout.put( “String2= ‘”, string2, “‘” nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
    // a string pointer always points at the first character
    // position in the string and we know we’ve got at least
    // three characters here).

    mov( string1, ebx );
    mov( ‘a’, (type char [ebx]) );
    mov( ‘b’, (type char [ebx+1]) );
    mov( ‘c’, (type char [ebx+2]) );

    // Okay, demonstrate that we have two different strings
    // since we used str.cpy to copy the data:

    stdout.put
    (
        “After assigning ‘abc’ to the first three characters in string1:”
        nl
        nl
    );
    stdout.put( “String1= ‘”, string1, “‘” nl );
    stdout.put( “String2= ‘”, string2, “‘” nl );


    // Note that we have to free the data associated with both
    // strings since they are not aliases of one another.

    strfree( string1 );
    strfree( string2 );


end strcpyDemo;
```

Program 2.7 Copying Strings using str.cpy

There are two really important things to note about this program. First, note that this program begins by allocating storage for *string2*. Remember, the *str.cpy* routine does not allocate storage for the destination string, it assumes that the destination string already has storage allocated to it. Keep in mind that *str.cpy* does not initialize *string2*, it only copies data to the location where *string2* is pointing. It is the program's responsibility to initialize the string by allocating sufficient memory before calling *str.cpy*. The second thing to notice here is that the program calls *strfree* to free up the storage for both *string1* and *string2* before the program quits.

Allocating storage for a string variable prior to calling *str.cpy* is so common that the HLA Standard Library provides a routine that allocates and copies the string: *str.a_cpy*. This routine uses the following call syntax:

```
str.a_cpy( source_string );
```

Note that there is no destination string. This routine looks at the length of the source string, allocates sufficient storage, makes a copy of the string, and then returns a pointer to the new string in the EAX register. The following program demonstrates the current example using the *str.a_cpy* procedure.

```
// Program to demonstrate string assignment using str.a_cpy.

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

static
    string1:    string;
    string2:    string;

begin stra_cpyDemo;


    // Get a value into string1

    forever

        stdout.put( “Enter a string with at least three characters: “ );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( “Please enter a string with at least three chars.” nl );

    endfor;


    // Do the string assignment via str.a_cpy

    str.a_cpy( string1 );
    mov( eax, string2 );

    stdout.put( “String1= ‘”, string1, “’” nl );
    stdout.put( “String2= ‘”, string2, “’” nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
    // a string pointer always points at the first character
    // position in the string and we know we’ve got at least
    // three characters here).

    mov( string1, ebx );
    mov( ‘a’, (type char [ebx]) );
    mov( ‘b’, (type char [ebx+1]) );
    mov( ‘c’, (type char [ebx+2]) );

    // Okay, demonstrate that we have two different strings
    // since we used str.cpy to copy the data:

    stdout.put
    (
        “After assigning ‘abc’ to the first three characters in string1:”
        nl
        nl
    );
    stdout.put( “String1= ‘”, string1, “‘” nl );
    stdout.put( “String2= ‘”, string2, “‘” nl );


    // Note that we have to free the data associated with both
    // strings since they are not aliases of one another.

    strfree( string1 );
    strfree( string2 );


end stra_cpyDemo;
```

Program 2.8 Copying Strings using str.a_cpy

Warning: Whenever using copy by reference or *str.a_cpy* to assign a string, don’t forget to free the storage associated with the string when you are (completely) done with that string’s data. Failure to do so may produce a memory leak if you do not have another pointer to the previous string data laying around.

Obtaining the length of a character string is such a common need that the HLA Standard Library provides a *str.length* routine specifically for this purpose. Of course, you can fetch the length by using the *str.strRec* data type to access the length field directly, but constant use of this mechanism can be tiring since it involves a lot of typing. The *str.length* routine provides a more compact and convenient way to fetch the length information. You call *str.length* using one of the following two formats:

```
str.length( Reg32 );
str.length( string_variable );
```

This routine returns the current string length in the EAX register.

Another pair of useful string routines are the *str.cat* and *str.a_cat* procedures. They use the following calling sequence:

```
str.cat( srcStr, destStr );
str.a_cat( src1Str, src2Str );
```

These two routines concatenate two strings (that is, they create a new string by joining the two strings together). The *str.cat* procedure concatenates the source string to the end of the destination string. Before the concatenation actually takes place, *str.cat* checks to make sure that the destination string is large enough to hold the concatenated result, it raises the *ex.StringOverflow* exception if the destination string is too small.

The *str.a_cat*, as its name suggests, allocates storage for the resulting string before doing the concatenation. This routine will allocate sufficient storage to hold the concatenated result, then it will copy the *src1Str* to the allocated storage, finally it will append the string data pointed at by *src2Str* to the end of this new string and return a pointer to the new string in the EAX register.

Warning: note a potential source of confusion. The *str.cat* procedure concatenates its first operand to the end of the second operand. Therefore, *str.cat* follows the standard (src, dest) operand format present in many HLA statements. The *str.a_cat* routine, on the other hand, has two source operands rather than a source and destination operand. The *str.a_cat* routine concatenates its two operands in an intuitive left-to-right fashion. This is the opposite of *str.cat*. Keep this in mind when using these two routines.

The following program demonstrates the use of the *str.cat* and *str.a_cat* routines:

```
// Program to demonstrate str.cat and str.a_cat.

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

static
    UserName:   string;
    Hello:      string;
    a_Hello:    string;

begin strcatDemo;

    // Allocate storage for the concatenated result:

    stralloc( 1024 );
    mov( eax, Hello );

    // Get some user input to use in this example:

    stdout.put( “Enter your name: “ );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, UserName );

    // Use str.cat to combine the two strings:

    str.cpy( “Hello “, Hello );
    str.cat( UserName, Hello );

    // Use str.a_cat to combine the string strings:

    str.a_cat( “Hello “, UserName );
    mov( eax, a_Hello );

    stdout.put( “Concatenated string #1 is ‘”, Hello, “‘” nl );
    stdout.put( “Concatenated string #2 is ‘”, a_Hello, “‘” nl );

    strfree( UserName );
    strfree( a_Hello );
    strfree( Hello );

end strcatDemo;
```

Program 2.9 Demonstration of str.cat and str.a_cat Routines

The *str.insert* and *str.a_insert* routines are closely related to the string concatenation procedures. However, the *str.insert* and *str.a_insert* routines let you insert one string anywhere into another string, not just at the end of the string. The calling sequences for these two routines are

```
str.insert( src, dest, index );
str.a_insert( StrToInsert, StrToInsertInto, index );
```

These two routines insert the source string (*src* or *StrToInsert*) into the destination string (*dest* or *StrToInsertInto*) starting at character position *index*. The *str.insert* routine inserts the source string directly into the destination string; if the destination string is not large enough to hold both strings, *str.insert* raises an *ex.StringOverflow* exception. The s*tr.a_insert* routine first allocates a new string on the heap, copies the destination string (*StrToInsertInto*) to the new string, and then inserts the source string (*StrToInsert*) into this new string at the specified offset; *str.a_insert* returns a pointer to the new string in the EAX register.

Indexes into a string are zero-based. This means that if you supply the value zero as the index in *str.insert* or *str.a_insert*, then these routines will insert the source string before the first character of the destination string. Likewise, if the index is equal to the length of the string, then these routines will simply concatenate the source string to the end of the destination string. Note: if the index is greater than the length of the string, the *str.insert* and *str.a_insert* procedures will not raise an exception; instead, they will simply append the source string to the end of the destination string.

The *str.delete* and *str.a_delete* routines let you remove characters from a string. They use the following calling sequence:

```
 str.delete( str, StartIndex, Length );
str.a_delete( str, StartIndex, Length );
```

Both routines delete *Length* characters starting at character position *StartIndex* in string *str.* The difference between the two is that *str.delete* deletes the characters directly from *str* whereas *str.a_delete* first allocates storage and copies *str*, then deletes the characters from the new string (leaving *str* untouched). The *str.a_delete* routine returns a pointer to the new string in the EAX register.

The *str.delete* and *str.a_delete* routines are very forgiving with respect to the values you pass in *StartIndex* and *Length*. If *StartIndex* is greater than the current length of the string, these routines do not delete any characters from the string. If *StartIndex* is less than the current length of the string, but *StartIndex+Length* is greater than the length of the string, then these routines will delete all characters from *StartIndex* to the end of the string.

Another very common string operation is the need to copy a portion of a string to a different string without otherwise affecting the source string. The *str.substr* and *str.a_substr* routines provide this capability. These routines use the following calling sequence:

```
str.substr( src, dest, StartIndex, Length );
  str.a_substr( src, StartIndex, Length );
```

The *str.substr* routine copies length characters, starting at position *StartIndex*, from the *src* string to the *dest* string. The *dest* string must have sufficient storage allocated to hold the new string or *str.substr* will raise an *ex.StringOverflow* exception. If the *StartIndex* value is greater than the length of the string, then *str.substr* will raise an *ex.StringIndexError* exception. If *StartIndex+Length* is greater than the length of the source string, but *StartIndex* is less than the length of the string, then *str.substr* will extract only those characters from *StartIndex* to the end of the string.

The *str.a_substr* procedure behaves in a fashion nearly identical to *str.substr* except it allocates storage on the heap for the destination string. Other than overflow never occurs, *str.a_substr* handles exceptions the identically to *str.substr*[6]. As you can probably guess by now, *str.a_substr* returns a pointer to the newly allocated string in the EAX register.

After you begin working with string data for a little while, the need will invariably arise to compare two strings. A first attempt at string comparison, using the standard HLA relational operators, will compile but not necessarily produce the desired results:

```
mov( s1, eax );
if( eax = s2 ) then

    << code to execute if the strings are equal >>

else

    << code to execute if the strings are not equal >>

endif;
```

As stated above, this code will compile and execute just fine. However, it's probably not doing what you expect it to do. Remember *strings are pointers*. This code compares the two pointers to see if they are equal. If they are equal, clearly the two strings are equal (since both *s1* and *s2* point at the exact same string data). However, the fact that the two pointers are different doesn't necessarily mean that the strings are not equivalent. Both *s1* and *s2* could contain different values (that is, they point at different addresses in memory) yet the string data at those two different addresses could be identical. Most programmers expect a string comparison for equality to be true if the data for the two strings is the same. Clearly a pointer comparison does not provide this type of comparison. To overcome this problem, the HLA Standard Library provides a set of string comparison routines that will compare the string data, not just their pointers. These routines use the following calling sequences:

```
str.eq( src1, src2 );
str.ne( src1, src2 );
str.lt( src1, src2 );
str.le( src1, src2 );
str.gt( src1, src2 );
str.ge( src1, src2 );
```

Each of these routines compares the *src1* string to the *src2* string and return true (1) or false (0) in the EAX register depending on the comparison. For example, "str.eq(s1, s2);" returns true in EAX if *s1* is equal to *s2*. HLA provides a small extension that allows you to use the string comparison routines within an IF statement[7]. The following code demonstrates the use of some of these comparison routines within an IF statement:

```
stdout.put( "Enter a single word: " );
stdin.a_gets();
if( str.eq( eax, "Hello" )) then

    stdout.put( "You entered 'Hello'", nl );

endif;
strfree( eax );
```

Note that the string the user enters in this example must exactly match "Hello", including the use of an upper case "H" at the beginning of the string. When processing user input, it is best to ignore alphabetic case in string comparisons because different users have different ideas about when they should be pressing the shift key on the keyboard. An easy solution is to use the HLA case insensitive string comparison functions. These routines compare two strings ignoring any differences in alphabetic case. These routines use the following calling sequences:

```
str.ieq( src1, src2 );
```

6. Technically, *str.a_substr*, like all routines that call *malloc* to allocate storage, can raise an *ex.MemoryAllocationFailure* exception, but this is very unlikely to occur.
7. This extension is actually a little more general than this section describes. A later chapter will explain it fully.

```
str.ine( src1, src2 );
str.ilt( src1, src2 );
str.ile( src1, src2 );
str.igt( src1, src2 );
str.ige( src1, src2 );
```

Other than they treat upper case characters the same as their lower case equivalents, these routines behave exactly like the former routines, returning true or false in EAX depending on the result of the comparison.

Like most high level languages, HLA compares strings using *lexicographical ordering*. This means that two strings are equal if and only if their lengths are the same and the corresponding characters in the two strings are exactly the same. For less than or greater than comparisons, lexicographical ordering corresponds to the way words appear in a dictionary. That is, “a” is less than “b” is less than “c” etc. Actually, HLA compares the strings using the ASCII numeric codes for the characters, so if you are unsure whether “a” is less than a period, simply consult the ASCII character chart (incidentally, “a” is greater than a period in the ASCII character set, just in case you were wondering).

If two strings have different lengths, lexicographical ordering only worries about the length if the two strings exactly match up through the length of the shorter string. If this is the case, then the longer string is greater than the shorter string (and, conversely, the shorter string is less than the longer string). Note, however, that if the characters in the two strings do not match at all, then HLA’s string comparison routines ignore the length of the string; e.g., “z” is always greater than “aaaaa” even though it has a shorter length.

The *str.eq* routine checks to see if two strings are equal. Sometimes, however, you might want to know whether one string *contains* another string. For example, you may want to know if some string contains the substring “north” or “south” to determine some action to take in a game. The HLA *str.index* routine lets you check to see if one string is contained as a substring of another. The *str.index* routine uses the following calling sequence:

```
str.index( StrToSearch, SubstrToSearchFor );
```

This function returns, in EAX, the offset into *StrToSearch* where *SubstrToSearchFor* appears. This routine returns -1 in EAX if *SubstrToSearchFor* is not present in *StrToSearch*. Note that str.index will do a case sensitive search. Therefore the strings must exactly match. There is no case insensitive variant of str.index you can use[8].

The HLA strings module contains many additional routines besides those this section presents. Space limitations and prerequisite knowledge prevent the presentation of all the string functions here; however, this does not mean that the remaining string functions are unimportant. You should definitely take a look at the HLA Standard Library documentation to learn everything you can about the powerful HLA string library routines. The chapters on advanced string handling contain more information on HLA string and pattern matching routines.

2.7 In-Memory Conversions

The HLA Standard Library’s string module contains dozens of routines for converting between strings and other data formats. Although it’s a little premature in this text to present a complete description of those functions, it would be rather criminal not to discuss at least one of the available functions: the *str.put* routine. This one routine (which is actually a macro) encapsulates the capabilities of all the other string conversion functions, so if you learn how to use this one, you’ll have most of the capabilities of those other routines at your disposal. For more information on the other string conversions, see the chapters in the volume on Advanced String Handling.

8. However, HLA does provide routines that will convert all the characters in a string to one case or another. So you can make copies of the strings, convert all the characters in both copies to lower case, and then search using these converted strings. This will achieve the same result.

You use the *str.put* routine in a manner very similar to the *stdout.put* routine. The only difference is that the *str.put* routine "writes" its data to a string instead of the standard output device. A call to *str.put* has the following syntax:

```
str.put( destString, values_to_convert );
```

Example of a call to *str.put*:

```
str.put( destString, "I =", i:4, " J= ", j, " s=", s );
```

Note: generally you would not put a newline character seqeuence at the end of the string as you would if you were printing the string to the standard output device.

The *destString* parameter at the beginning of the *str.put* parameter list must be a string variable and it must already have storage associated with it. If *str.put* attempts to store more characters than allowed into the *destString* parameter, then this function raises the *ex.StringOverflow* exception.

Most of the time you won't know the length of the string that *str.put* will produce. In those instances, you should simply allocate sufficient storage for a really large string, one that is way larger than you expect, and use this string data as the first parameter of the *str.put* call. This will prevent an exception from crashing your program. Generally, if you expect to produce about one screen line of text, you should probably allocate at least 256 characters for the destination string. If you're creating longer strings, you should probably use a default of 1024 characters (or more, if you're going to produce *really* large strings).

Example:

```
static
    s: string;
        .
        .
        .
    mov( stralloc( 256 ), s );
        .
        .
        .
    str.put( s, "R: ", r:16:4, " strval: '", strval:-10, "'" );
```

You can use the *str.put* routine to convert any data to a string that you can print using *stdout.put*. You will probably find this routine invaluable for common value-to-string conversions.

At the time this is being written, there is no corresponding *str.get* routine that will read values from an input string (this routine will probably appear in a future version of the HLA Standard Library, so watch out for it). In the meantime, the HLA strings and conversions modules in the Standard Library do provide lots of stand-alone conversion functions you can use to convert string data to some other format. See the volume on "Advanced String Handling" for more details about these routines.

2.8 Putting It All Together

There are many different ways to represent character strings. This chapter began by discussing how the C/C++ and Pascal languages represent strings using zero-terminated and length prefixed strings. HLA uses a hybrid representation for its string. HLA strings consist of a pointer to a zero terminated sequence of character with a pair of prefix length values. HLA's format offers all the advantages of the other two forms with the slight disadvantage of a few extra bytes of overhead.

After discussing string formats, this chapter discussed how to operate on string data. In addition to accessing the characters in a string directly (which is easy, you just index off the pointer to the string data), this chapter described how to manipulate strings using several routines from the HLA Standard Library. This chapter provides a very basic introduction to string handling in HLA. To learn more about string manipulation in assembly language (and the use of the routines in the HLA Standard Library), see the separate volume on "Advanced String Handling" in this text.

Characters and Character Sets Chapter Three

3.1 Chapter Overview

This chapter completes the discussion of the character data type by describing several character translation and classification functions found in the HLA Standard Library. These functions provide most of the character operations that aren't trivial to realize using a few 80x86 machine instructions.

This chapter also introduces another composite data type based on the character – the character set data type. Character sets and their associated operations, let you quickly test characters to see if they belong to some set. These operations also let you manipulate sets of characters using familiar operations like set union, intersection, and difference.

3.2 The HLA Standard Library CHARS.HHF Module

The HLA Standard Library chars.hhf module provides a couple of routines that convert characters from one form to another and several routines that classify characters according to their graphic representation. These functions are especially useful for processing user input to verify that it is correct.

The first two routines we will consider are the translation/conversion functions. These functions are *chars.toUpper* and *chars.toLower.* These functions use the following syntax:

```
chars.toLower( characterValue ); // Returns converted character in AL.
chars.toUpper( characterValue ); // Returns converted character in AL.
```

These two functions require a byte-sized parameter (typically a register or a *char* variable). They check the character to see if it is an alphabetic character; if it is not, then these functions return the unmodified parameter value in the AL register. If the character is an alphabetic character, then these functions may translate the value depending on the particular function. The *chars.toUpper* function translates lower case alphabetic characters to upper case; it returns upper case character unmodified. The chars.toLower function does the converse – it translates upper case characters to lower case characters and leaves lower case characters alone.

These two functions are especially useful when processing user input containing alphabetic characters. For example, suppose you expect a "Y" or "N" answer from the user at some point in your program. You code might look like the following:

```
forever

    stdout.put( "Answer 'Y' or 'N':" );
    stdin.FlushInput();   // Force input of new line of text.
    stdin.getc();  // Read user input in AL.
    breakif( al = 'Y' );
    breakif( al = 'N' );
    stdout.put( "Illegal input, please reenter", nl );

endfor;
```

The problem with this program is that the user must answer exactly "Y" or "N" (using upper case) or the program will reject the user's input. This means that the program will reject "y" and "n" since the ASCII codes for these characters are different than "Y" and "N".

One way to solve this problem is to include two additional BREAKIF statements in the code above that test for "y" and "n" as well as "Y" and "N". The problem with this approach is that AL will still contain one of four different characters, complicating tests of AL once the program exits the loop. A better solution is to use either *chars.toUpper* or *chars.toLower* to translate all alphabetic characters to a single case. Then you can test AL for a single pair of characters, both in the loop and outside the loop. The resulting code would look like the following:

```
forever

    stdout.put( “Answer ‘Y’ or ‘N’:” );
    stdin.FlushInput();   // Force input of new line of text.
    stdin.getc();         // Read user input in AL.
    chars.toUpper( al );  // Convert “y” and “n” to “Y” and “N”.
    breakif( al = ‘Y’ );
    breakif( al = ‘N’ );
    stdout.put( “Illegal input, please reenter”, nl );

endfor;
<< test for “Y” or “N” down here to determine user input >>
```

As you can see from this example, the case conversion functions can be quite useful when processing user input. As a final example, consider a program that presents a menu of options to the user and the user selects an option using an alphabetic character. Once again, you can use *chars.toUpper* or *chars.toLower* to map the input character to a single case so that it is easier to process the user’s input:

```
stdout.put( “Enter selection (A-G):” );
stdin.FlushInput();
stdin.getc();
chars.toLower( al );
if( al = ‘a’ ) then

    << Handle Menu Option A >>

elseif( al = ‘b’ ) then

    << Handle Menu Option B >>

elseif( al = ‘c’ ) then

    << Handle Menu Option C >>

elseif( al = ‘d’ ) then

    << Handle Menu Option D >>

elseif( al = ‘e’ ) then

    << Handle Menu Option E >>

elseif( al = ‘f’ ) then

    << Handle Menu Option F >>

elseif( al = ‘g’ ) then

    << Handle Menu Option G >>

else

    stdout.put( “Illegal input!” nl );

endif;
```

The remaining functions in the chars.hhf module all return a boolean result depending on the type of the character you pass them as a parameter. These classification functions let you quickly and easily test a character to determine if it's type is valid for the some intended use. These functions expect a single byte (char) parameter and they return true (1) or false (0) in the EAX register. These functions use the following calling syntax:

```
chars.isAlpha( c );    // Returns true if c is alphabetic
chars.isUpper( c );    // Returns true if c is upper case alphabetic.
chars.isLower( c );    // Returns true if c is lower case alphabetic.
chars.isAlphaNum( c ); // Returns true if c is alphabetic or numeric.
chars.isDigit( c );    // Returns true if c is a decimal digit.
chars.isXDigit( c );   // Returns true if c is a hexadecimal digit.
chars.isGraphic( c );  // See notes below.
chars.isSpace( c );    // Returns true if c is a whitespace character.
chars.isASCII( c );    // Returns true if c is in the range #$00..#$7f.
chars.isCtrl( c );     // Returns true if c is a control character.
```

Notes: Graphic characters are the printable characters whose ASCII codes fall in the range $21..$7E. Note that a space is not considered a graphic character (nor are the control characters). Whitespace characters are the space, the tab, the carriage return, and the linefeed. Control characters are those characters whose ASCII code is in the range $00..$1F and $7F.

These classification functions are great for validating user input. For example, if you want to check to ensure that a user has entered nothing but numeric characters in a string you read from the standard input, you could use code like the following:

```
stdin.a_gets();  // Read line of text from the user.
mov( eax, ebx ); // save ptr to string in EBX.
mov( ebx, ecx ); // Another copy of string pointer to test each char.
while( (type char [ecx]) <> #0 ) do // Repeat while not at end of string.

    breakif( !chars.isDigit( (type char [ecx] )));
    inc( ecx ); // Move on to the next character;

endwhile;
if( (type char [ecx] ) = #0 ) then

    << Valid string, process it >>

else

    << invalid string >>

endif;
```

Although the chars.hhf module's classification functions handle many common situations, you may find that you need to test a character to see if it belongs in a class that the chars.hhf module does not handle. Fear not, checking for such characters is very easy. The next section will explain how to do this.

3.3 Character Sets

Character sets are another composite data type, like strings, built upon the character data type. A character set is a mathematical set of characters with the most important attribute being membership. That is, a character is either a member of a set or it is not a member of a set. The concept of sequence (e.g., whether one character comes before another, as in a string) is completely foreign to a character set. If two characters are members of a set, their order in the set is irrelevant. Also, membership is a binary relation; a character is either in the set or it is not in the set; you cannot have multiple copies of the same character in a character set. Finally, there are various operations that are possible on character sets including the mathematical set operations of union, intersection, difference, and membership test.

HLA implements a restricted form of character sets that allows set members to be any of the 128 standard ASCII characters (i.e., HLA's character set facilities do not support extended character codes in the range #128..#255). Despite this restriction, however, HLA's character set facilities are very powerful and are very handy when writing programs that work with string data. The following sections describe the implementation and use of HLA's character set facilities so you may take advantage of character sets in your own programs.

3.4 Character Set Implementation in HLA

There are many different ways to represent character sets in an assembly language program. HLA implements character sets by using an array of 128 boolean values. Each boolean value determines whether the corresponding character is or is not a member of the character set; i.e., a true boolean value indicates that the specified character is a member of the set, a false value indicates that the corresponding character is not a member of the set. To conserve memory, HLA allocates only a single bit for each character in the set; therefore, HLA character sets consume 16 bytes of memory since there are 128 bits in 16 bytes. This array of 128 bits is organized in memory as shown in Figure 3.1.

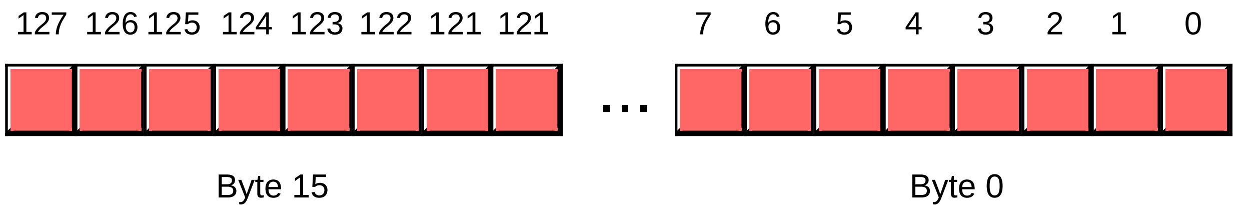

Figure 3.1 Bit Layout of a Character Set Object

Bit zero of byte zero corresponds to ASCII code zero (the NUL character). If this bit is one, then the character set contains the NUL character; if this bit contains false, then the character set does not contain the NUL character. Likewise, bit zero of byte one (the eighth bit in the 128-bit array) corresponds to the backspace character (ASCII code is eight). Bit one of byte eight corresponds to ASCII code 65, an upper case 'A'. Bit 65 will contain a one if 'A' is a current member of the character set, it will contain zero if 'A' is not a member of the set.

While there are other possible ways to implement character sets, this bit vector implementation has the advantage that it is very easy to implement set operations like union, intersection, difference comparison, and membership tests.

HLA supports character set variables using the *cset* data type. To declare a character set variable, you would use a declaration like the following:

```
static
    CharSetVar: cset;
```

This declaration will reserve 16 bytes of storage to hold the 128 bits needed to represent an ASCII character set.

Although it is possible to manipulate the bits in a character set using instructions like AND, OR, XOR, etc., the 80x86 instruction set includes several bit test, set, reset, and complement instructions that are nearly perfect for manipulating character sets. The BT (bit test) instruction, for example will copy a single bit in memory to the carry flag. The BT instruction allows the following syntactical forms:

```
bt( BitNumber, BitsToTest );

bt( reg16, reg16 );
```

```
        bt( reg$_{32}$, reg$_{32}$ );
        bt( constant, reg$_{16}$ );
        bt( constant, reg$_{32}$ );

        bt( reg$_{16}$, mem$_{16}$ );
        bt( reg$_{32}$, mem$_{32}$ );  //HLA treats cset objects as dwords within bt.
        bt( constant, mem$_{16}$ );
        bt( constant, mem$_{32}$ );  //HLA treats cset objects as dwords within bt.
```

The first operand holds a bit number, the second operand specifies a register or memory location whose bit should be copied into the carry flag. If the second operand is a register, the first operand must contain a value in the range 0..*n*-1, where *n* is the number of bits in the second operand. If the first operand is a constant and the second operand is a memory location, the constant must be in the range 0..255. Here are some examples of these instructions:

```
    bt( 7, ax );             // Copies bit #7 of AX into the carry flag (CF).
    mov( 20, eax );
    bt( eax, ebx );          // Copies bit #20 of EBX into CF.

// Copies bit #0 of the byte at CharSetVar+3 into CF.

    bt( 24, CharSetVar );

    // Copies bit #4 of the byte at DWmem+2 into CF.

    bt( eax, CharSetVar );
```

The BT instruction turns out to be quite useful for testing set membership. For example, to see if the character 'A' is a member of a character set, you could use a code sequence like the following:

```
        bt( 'A', CharSetVar );
        if( @c ) then

            << Do something if 'A' is a member of the set >>

        endif;
```

The BTS (bit test and set), BTR (bit test and reset), and BTC (bit test and complement) instructions are also quite useful for manipulating character set variables. Like the BT instruction, these instructions copy the specified bit into the carry flag; after copying the specified bit, these instructions will set, clear, or invert (respectively) the specified bit. Therefore, you can use the BTS instruction to add a character to a character set via set union (that is, it adds a character to the set if the character was not already a member of the set, otherwise the set is unaffected). You can use the BTR instruction to remove a character from a character set via set intersection (That is, it removes a character from the set if and only if it was previously in the set; otherwise it has no effect on the set). The BTC instruction lets you add a character to the set if it wasn't previously in the set, it removes the character from the set if it was previously a member (that is, it toggles the membership of that character in the set).

The HLA Standard Library provides lots of character set handling routines. See "Character Set Support in the HLA Standard Library" on page 445. for more details about HLA's character set facilities.

3.5 HLA Character Set Constants and Character Set Expressions

HLA supports literal character set constants. These *cset* constants make it easy to initialize *cset* variables at compile time and they make it very easy to pass character set constants as procedure parameters. An HLA character set constant takes the following form:

```
          { Comma_separated_list_of_characters_and_character_ranges }
```

The following is an example of a simple character set holding the numeric digit characters:

```
{ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' }
```

When specifying a character set literal that has several contiguous values, HLA lets you concisely specify the values using only the starting and ending values of the range thusly:

```
{ '0'..'9' }
```

You may combine characters and various ranges within the same character set constant. For example, the following character set constant is all the alphanumeric characters:

```
{ '0'..'9', 'a'..'z', 'A'..'Z' }
```

You can use these *cset* literal constants in the CONST and VAL sections. The following example demonstrates how to create the symbolic constant *AlphaNumeric* using the character set above:

```
const
    AlphaNumeric: cset := {'0'..'9', 'a'..'z', 'A'..'Z' };
```

After the above declaration, you can use the identifier *AlphaNumeric* anywhere the character set literal is legal.

You can also use character set literals (and, of course, character set symbolic constants) as the initializer field for a STATIC or READONLY variable. The following code fragment demonstrates this:

```
static
    Alphabetic: cset := { 'a'..'z', 'A'..'Z' };
```

Anywhere you can use a character set literal constant, a character set constant expression is also legal. HLA supports the following operators in character set constant expressions:

CSetConst + *CSetConst*	Computes the union of the two sets[1].
CSetConst * *CSetConst*	Computes the intersection of the two sets[2].
CSetConst - *CSetConst*	Computes the set difference of the two sets[3].
-*CSetConst*	Computes the set complement[4].

Note that these operators only produce compile-time results. That is, the expressions above are computed by the compiler during compilation, they do not emit any machine code. If you want to perform these operations on two different sets while your program is running, the HLA Standard Library provides routines you can call to achieve the results you desire. HLA also provides other compile-time character set operators. See the chapter on the compile-time language and macros for more details.

3.6 The IN Operator in HLA HLL Boolean Expressions

The HLA IN operator can dramatically reduce the logic in your HLA programs. This text has waited until now to discuss this operator because certain forms require a knowledge of character sets and character set constants. Now that you've seen character set constants, there is no need to delay the introduction of this important language feature.

In addition to the standard boolean expressions in IF, WHILE, REPEAT..UNTIL, and other statements, HLA also supports boolean expressions that take the following forms:

reg_8 in *CSetConstant*

1. The set union is the set of all characters that are in either set.
2. The set intersection is the set of all characters that appear in both operand sets.
3. The set difference is the set of characters that appear in the first set but do not appear in the second set.
4. The set complement is the set of all characters not in the set.

```
reg_8 not in CSetConstant
reg_8 in CSetVariable
reg_8 not in CSetVariable
```

These four forms of the IN and NOT IN operators check to see if a character in an eight-bit register is a member of a character set (either a character set constant or a character set variable). The following code fragment demonstrates these operators:

```
const
    Alphabetic: cset := {'a'..'z', 'A'..'Z'};
        .
        .
        .
    stdin.getc();
    if( al in Alphabetic ) then

        stdout.put( "You entered an alphabetic character" nl );

    elseif( al in {'0'..'9'} ) then

        stdout.put( "You entered a numeric character" nl );

    endif;
```

3.7 Character Set Support in the HLA Standard Library

As noted in the previous sections, the HLA Standard Library provides several routines that provide character set support. The character set support routines fall into four categories: standard character set functions, character set tests, character set conversions, and character set I/O. This section describes these routines in the HLA Standard Library.

To begin with, let's consider the Standard Library routines that help you construct character sets. These routines include: *cs.empty*, *cs.cpy*, *cs.charToCset*, *cs.unionChar*, *cs.removeChar*, *cs.rangeChar*, *cs.strToCset*, and *cs.unionStr*. These procedures let you build up character sets at run-time using character and string objects.

The *cs.empty* procedure initializes a character set variable to the empty set by setting all the bits in the character set to zero. This procedure call uses the following syntax (*CSvar* is a character set variable):

```
cs.empty( CSvar );
```

The *cs.cpy* procedure copies one character set to another, replacing any data previously held by the destination character set. The syntax for *cs.cpy* is

```
cs.cpy( srcCsetValue, destCsetVar );
```

The *cs.cpy* source character set can be either a character set constant or a character set variable. The destination character set must be a character set variable.

The *cs.unionChar* procedure adds a character to a character set. It uses the following calling sequence:

```
cs.unionChar( CharVar, CSvar );
```

This call will add the first parameter, a character, to the set via set union. Note that you could use the BTS instruction to achieve this same result although the *cs.unionChar* call is often more convenient (though slower).

The *cs.charToCset* function creates a singleton set (a set containing a single character). The calling format for this function is

```
cs.charToCset( CharValue, CSvar );
```

The first operand, the character value *CharValue*, can be an eight-bit register, a constant, or a character variable. The second operand (*CSvar*) must be a character set variable. This function clears the destination character set to all zeros and then adds the specified character to the character set.

The *cs.removeChar* procedure lets you remove a single character from a character set without affecting the other characters in the set. This function uses the same syntax as *cs.charToCset* and the parameters have the same attributes. The calling sequence is

```
cs.removeChar( CharValue, CSVar );
```

The *cs.rangeChar* constructs a character set containing all the characters between two characters you pass as parameters. This function sets all bits outside the range of these two characters to zero. The calling sequence is

```
cs.rangeChar( LowerBoundChar, UpperBoundChar, CSVar );
```

The *LowerBoundChar* and *UpperBoundChar* parameters can be constants, registers, or character variables. *CSVar*, the destination character set, must be a *cset* variable.

The *cs.strToCset* procedure creates a new character set containing the union of all the characters in a character string. This procedure begins by setting the destination character set to the empty set and then it unions in the characters in the string one by one until it exhausts all characters in the string. The calling sequence is

```
cs.strToCset( StringValue, CSVar );
```

Technically, the *StringValue* parameter can be a string constant as well as a string variable, however, it doesn't make any sense to call *cs.strToCset* in this fashion since *cs.cpy* is a much more efficient way to initialize a character set with a constant set of characters. As usual, the destination character set must be a *cset* variable. Typically, you'd use this function to create a character set based on a string input by the user.

The *cs.unionStr* procedure will add the characters in a string to an existing character set. Like *cs.strToCset*, you'd normally use this function to union characters into a set based on a string input by the user. The calling sequence for this is

```
cs.unionStr( StringValue, CSVar );
```

Standard set operations include union, intersection, and set difference. The HLA Standard Library routines *cs.setunion*, *cs.intersection*, and *cs.difference* provide these operations, respectively[5]. These routines all use the same calling sequence:

```
 cs.setunion( srcCset, destCset );
cs.intersection( srcCset, destCset );
 cs.difference( srcCset, destCset );
```

The first parameter can be a character set constant or a character set variable. The second parameter must be a character set variable. These procedures compute "destCset := destCset op srcCset" where *op* represents set union, intersection, or difference, depending on the function call.

The third category of character set routines test character sets in various ways. They typically return a boolean value indicating the result of the test. The HLA character set routines in this category include *cs.IsEmpty, cs.member, cs.subset, cs.psubset, cs.superset, cs.psuperset, cs.eq,* and *cs.ne*.

The *cs.IsEmpty* function tests a character set to see if it is the empty set. The function returns true or false in the EAX register. This function uses the following calling sequence:

```
cs.IsEmpty( CSetValue );
```

5. "cs.setunion" was used rather than "cs.union" because "union" is an HLA reserved word.

The single parameter may be a constant or a character set variable, although it doesn't make much sense to pass a character set constant to this procedure (since you would know at compile-time whether this set is empty or not empty).

The *cs.member* function tests to see if a character value is a member of a set. This function returns true in the EAX register if the supplied character is a member of the specified set. Note that you can use the BT instruction to (more efficiently) test this same condition. However, the *cs.member* function is probably a little more convenient to use. The calling sequence for *cs.member* is

```
cs.member( CharValue, CsetValue );
```

The first parameter is a register, character variable, or a constant. The second parameter is either a character set constant or a character set variable. It would be unusual for both parameters to be constants.

The *cs.subset*, *cs.psubset* (proper subset), *cs.superset*, and *cs.psuperset* (proper superset) functions let you check to see if one character set is a subset or superset of another. The calling sequence for these four routines is nearly identical, it is one of the following:

```
cs.subset( CsetValue1, CsetValue2 );
cs.psubset( CsetValue1, CsetValue2 );
cs.superset( CsetValue1, CsetValue2 );
cs.psuperset( CsetValue1, CsetValue2 );
```

These routines compare the first parameter against the second parameter and return true or false in the EAX register depending upon the result of the comparison. One set is a subset of another if all the members of the first character set can be found in the second character set. It is a proper subset if the second character set also contains characters not found in the first (left) character set. Likewise, one character set is a superset of another if it contains all the characters in the second (right) set (and, possibly, more). A proper superset contains additional characters above and beyond those found in the second set. The parameters can be either character set variables or character set constants; however, it would be unusual for both parameters to be character set constants (since you can determine this at compile time, there would be no need to call a run-time function to compute this).

The *cs.eq* and *cs.ne* check to see if two sets are equal or not equal. These functions return true or false in EAX depending upon the set comparison. The calling sequence is identical to the sub/superset functions above:

```
cs.eq( CsetValue1, CsetValue2 );
cs.ne( CsetValue1, CsetValue2 );
```

The *cs.extract* routine removes an arbitrary character from a character set and returns that character in the EAX register[6]. The calling sequence is the following:

```
cs.extract( CsetVar );
```

The single parameter must be a character set variable. Note that this function will modify the character set variable by removing some character from the character set. This function returns $FFFF_FFFF (-1) in EAX if the character set was empty prior to the call.

In addition to the routines found in the *cs* (character set) library module, the string and standard output modules also provide functions that allow or expect character set parameters. For example, if you supply a character set value as a parameter to *stdout.put*, the *stdout.put* routine will print the characters currently in the set. See the HLA Standard Library documentation for more details on character set handling procedures.

3.8 Using Character Sets in Your HLA Programs

Character sets are valuable for many different applications in your programs. For example, in the volume on Advanced String Handling you'll discover how to use character sets to match complex patterns.

6. This routine returns the character in AL and zeros out the H.O. three bytes of EAX.

However, such use of character sets is a little beyond the scope of this chapter, so at this point we'll concentrate on another common use of character sets: validating user input. This section will also present a couple of other applications for character sets to help you start thinking about how you could use them in your program.

Consider the following short code segment that gets a yes/no type answer from the user:

```
static
    answer: char;
        .
        .
        .
    repeat
            .
            .
            .
        stdout.put( "Would you like to play again? " );
        stdin.FlushInput();
        stdin.get( answer );

    until( answer = 'n' );
```

A major problem with this code sequence is that it will only stop if the user presses a lower case 'n' character. If they type anything other than 'n' (including upper case 'N') the program will treat this as an affirmative answer and transfer back to the beginning of the repeat..until loop. A better solution would be to validate the user input before the UNTIL clause above to ensure that the user has only typed "n", "N", "y", or "Y". The following code sequence will accomplish this:

```
    repeat
            .
            .
            .
        repeat

            stdout.put( "Would you like to play again? " );
            stdin.FlushInput();
            stdin.get( answer );

        until( cs.member( answer, { 'n', 'N', 'Y', 'y' } );
        if( answer = 'N' ) then

            mov( 'n', answer );

        endif;

    until( answer = 'n' );
```

While an excellent use for character sets is to validate user input, especially when you must restrict the user to a small set of non-contiguous input characters, you should not use the *cs.member* function to test to see if a character value is within literal set. For example, you should never do something like the following:

```
    repeat

        stdout.put( "Enter a character between 0..9: " );
        stdin.getc();

    until( cs.member( al, {'0'..'9' } );
```

While there is nothing logically wrong with this code, keep in mind that HLA run-time boolean expressions allow simple membership tests using the IN operator. You could write the code above far more efficiently using the following sequence:

```
    repeat

        stdout.put( “Enter a character between 0..9: “ );
        stdin.getc();

    until( al in ‘0’..’9’ );
```

The place where the *cs.member* function becomes useful is when you need to see if an input character is within a set of characters that you build at run time.

3.9 Low-level Implementation of Set Operations

Although the HLA Standard Library character set module simplifies the use of character sets within your assembly language programs, it is instructive to look at how all these functions operate so you know the cost associated with each function. Also, since many of these functions are quite trivial, you might want to implement them in-line for performance reasons. The following subsections describe how each of the functions operate.

3.9.1 Character Set Functions That Build Sets

The first group of functions we will look at in the Character Set module are those that construct or copy character sets. These functions are *cs.empty, cs.cpy, cs.charToCset, cs.unionChar, cs.removeChar, cs.rangeChar, cs.strToCset,* and *cs.unionStr.*

Creating an empty set is, perhaps, the easiest of all the operations. To create an empty set all we need to is zero out all 128 bits in the *cset* object. Program 3.1 provides the implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.empty function.

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

static
    csetDest: cset;
    csetSrc: cset := {‘a’..’z’, ‘A’..’Z’};

begin csEmpty;

    // How to create an empty set (cs.empty):
    // (Zero out all bits in the cset)

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    stdout.put( “Empty set = {“, csetDest, “}” nl );

end csEmpty;
```

Program 3.1 cs.empty Implementation

Note that cset objects are 16 bytes long. Therefore, this code zeros out those 16 bytes by storing EAX into the four consecutive double words that comprise the object. Note the use of type coercion in the MOV statements; this is necessary since *cset* objects are not the same size as *dword* objects.

To copy one character set to another is only a little more difficult than creating an empty set. All we have to do is copy the 16 bytes from the source character set to the destination character set. We can accomplish this with four pairs of double word MOV statements. Program 3.2 provides the sample implementation.

```
// Program that demonstrates the implmentation of
// the cs.empty function.

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

static
    csetDest: cset;
    csetSrc: cset := {‘a’..’z’, ‘A’..’Z’};

begin csCpy;

    // How to create an empty set (cs.empty):
    // (Zero out all bits in the cset)

    mov( (type dword csetSrc), eax );
    mov( eax, (type dword csetDest ));

    mov( (type dword csetSrc[4]), eax );
    mov( eax, (type dword csetDest[4] ));

    mov( (type dword csetSrc[8]), eax );
    mov( eax, (type dword csetDest[8] ));

    mov( (type dword csetSrc[12]), eax );
    mov( eax, (type dword csetDest[12] ));

    stdout.put( “Copied set = {“, csetDest, “}” nl );

end csCpy;
```

Program 3.2 cs.cpy Implementation

The *cs.charToCset* function creates a singleton set containing the specified character. To implement this function we first begin by creating an empty set (using the same code as cs.empty) and then we set the bit corresponding to the single character in the character set. We can use the BTS (bit test and set) instruction to easily set the specified bit in the cset object. Program 3.3 provides the implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.charToCset function.

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

static
```

```
    csetDest: cset;
    chrValue: char := ‘a’;

begin cscharToCset;

    // Begin by creating an empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    // Okay, use the BTS instruction to set the specified bit in
    // the character set.

    movzx( chrValue, eax );
    bts( eax, csetDest );

    stdout.put( “Singleton set = {“, csetDest, “}” nl );

end cscharToCset;
```

Program 3.3 cs.charToCset Implementation

If you study this code carefully, you will note an interesting fact: the BTS instruction’s operands are not the same size (*dword* and *cset*). Since programmers often use the BTx instructions to manipulate items in a character set, HLA allows you to specify a *cset* object as the destination operand of a BTx(reg_{32}, mem) instruction. Technically, the memory operand should be a double word object; HLA automatically coerces *cset* objects to *dword* for these instructions. Note that BTS requires a 16 or 32-bit register. Therefore, this code zero extends the character’s value into EAX prior to executing the BTS instruction. Note that the value in EAX must not exceed 127 or this code will manipulate data beyond the end of the character set in memory. The use of a BOUND instruction might be warranted here if you can’t ensure that the *chrValue* variable contains a value in the range 0..127.

The *cs.unionChar* adds a single character to the character set (if that character was not already present in the character set). This code is actually a bit simpler than the cs.charToCset function; the only difference is that the code does not clear the set to begin with – it simply sets the bit corresponding to the given character. Program 3.4 provides the implementation.

```
// Program that demonstrates the implmentation of
// the cs.unionChar function.

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

static
    csetDest: cset := {‘0’..’9’};
    chrValue: char := ‘a’;

begin csUnionChar;


    // Okay, use the BTS instruction to add the specified bit to
    // the character set.
    movzx( chrValue, eax );
    bts( eax, csetDest );

    stdout.put( “New set = {“, csetDest, “}” nl );

end csUnionChar;
```

Program 3.4 cs.unionChar Implementation

Once again, note that this code assumes that the character value is in the range 0..127. If it is possible for the character to fall outside this range, you should check the value before attempting to union the character into the character set. You can use an IF statement or the BOUND instruction for this check.

The *cs.removeChar* function removes a character from a character set, provided that character was a member of the set. If the character was not originally in the character set, then *cs.removeChar* does not affect the original character set. To accomplish this, the code must clear the bit associated with the character to remove from the set. Program 3.5 uses the BTR (bit test and reset) instruction to achieve this.

```
// Program that demonstrates the implmentation of
// the cs.removeChar function.

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

static
    csetDest: cset := {‘0’..’9’};
    chrVal1: char := ‘0’;
    chrVal2: char := ‘a’;

begin csRemoveChar;


    // Okay, use the BTC instruction to remove the specified bit from
    // the character set.

    movzx( chrVal1, eax );
    btr( eax, csetDest );

    stdout.put( “Set w/o ‘0’ = {“, csetDest, “}” nl );

    // Now remove a character not in the set to demonstrate
    // that removal of a non-existant character doesn’t affect
    // the set:

    movzx( chrVal2, eax );
    btr( eax, csetDest );
    stdout.put( “Final set = {“, csetDest, “}” nl );

end csRemoveChar;
```

Program 3.5 cs.removeChar Implementation

Don't forget to use a BOUND instruction or an IF statement in Program 3.5 if it is possible for the character's value to fall outside the range 0..127. This will prevent the code from manipulating memory beyond the end of the character set.

The *cs.rangeChar* function creates a set containing all the characters between two specified boundaries. This function begins by creating an empty set; then it loops over the range of character to insert, inserting each character into the set as appropriate. Program 3.6 provides an example implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.rangeChar function.

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

static
    csetDest: cset;
    startRange: char := 'a';
    endRange: char := 'z';

begin csRangeChar;


    // Begin by creating the empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    // Run the following loop for each character between
    // 'startRange' and 'endRange' and set the corresponding
    // bit in the cset for each character in the range.

    movzx( startRange, eax );
    while( al <= endRange ) do

        bts( eax, csetDest );
        inc( al );

    endwhile;
        stdout.put( "Final set = {", csetDest, "}" nl );

end csRangeChar;
```

Program 3.6 cs.rangeChar Implementation

One interesting thing to note about the code in Program 3.6 is how it takes advantage of the fact that AL contains the actual character index even though it has to use EAX with the BTS instruction. As usual, you should check the range of the two values if there is any possibility that they could be outside the range 0..127.

One problem with this particular implementation of the *cs.rangeChar* function is that it is not particularly efficient if you ask it to create a set with a lot of characters in it. As you can see by studying the code, the execution time of this function is proportional to the number of characters in the range. In particular, the loop in this function iterates once for each character in the range. So if the range is large the loop executes many more times than if the range is small. There is a more efficient solution to this problem using table lookups whose execution time is independent of the size of the character set it creates. For more details on using table lookups, see “Calculation Via Table Lookups” on page 647.

The *cs.strToCset* function scans through an HLA string character by character and creates a new character set by adding each character in the string to an empty character set.

```
// Program that demonstrates the implmentation of
// the cs.strToCset function.

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

static
    StrToAdd: string := “Hello_World”;
    csetDest: cset;

begin csStrToCset;


    // Begin by creating the empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest ));
    mov( eax, (type dword csetDest[4] ));
    mov( eax, (type dword csetDest[8] ));
    mov( eax, (type dword csetDest[12] ));

    // For each character in the source string, add that character
    // to the set.

    mov( StrToAdd, eax );
    while( (type char [eax]) <> #0 ) do  // While not at end of string.

        movzx( (type char [eax]), ebx );
        bts( ebx, csetDest );
        inc( eax );

    endwhile;
    stdout.put( “Final set = {“, csetDest, “}” nl );

end csStrToCset;
```

Program 3.7 cs.strToCset Implementation

This code begins by fetching the pointer to the first character in the string. The loop repeats for each character in the string up to the zero terminating byte of the string. For each character, this code uses the BTS instruction to set the corresponding bit in the destination character set. As usual, don’t forget to use an IF statement or BOUND instruction if it is possible for the characters in the string to have values outside the range 0..127.

The *cs.unionStr* function is very similar to the *cs.strToCset* function; in fact, the only difference is that it doesn’t create an empty character set prior to adding the characters in a string to the destination character set.

```
// Program that demonstrates the implmentation of
// the cs.unionStr function.

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

static
    StrToAdd: string := “Hello_World”;
    csetDest: cset := {‘0’..’9’};

begin csUnionStr;



    // For each character in the source string, add that character
    // to the set.

    mov( StrToAdd, eax );
    while( (type char [eax]) <> #0 ) do  // While not at end of string.

        movzx( (type char [eax]), ebx );
        bts( ebx, csetDest );
        inc( eax );

    endwhile;
    stdout.put( “Final set = {“, csetDest, “}” nl );

end csUnionStr;
```

Program 3.8 cs.unionStr Implementation

3.9.2 Traditional Set Operations

The previous section describes how to construct character sets from characters and strings. In this section we’ll take a look at how you can manipulate character sets using the traditional set operations of intersection, union, and difference.

The union of two sets A and B is the collection of all items that are in set A, set B, or both. In the bit array representation of a set, this means that a bit in the destination character set will be one if either or both of the corresponding bits in sets A or B are set. This of course, corresponds to the logical OR operation. Therefore, we can easily create the set union of two sets by logically ORing their bytes together. Program 3.9 provides the complete implementation of this function.

```
// Program that demonstrates the implmentation of
// the cs.setunion function.

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

static
    csetSrc1: cset := {‘a’..’z’};
    csetSrc2: cset := {‘A’..’Z’};
    csetDest: cset;

begin cssetUnion;
```

```
    // To compute the union of csetSrc1 and csetSrc2 all we have
    // to do is logically OR the two sets together.

    mov( (type dword csetSrc1), eax );
    or( (type dword csetSrc2), eax );
    mov( eax, (type dword csetDest));

    mov( (type dword csetSrc1[4]), eax );
    or( (type dword csetSrc2[4]), eax );
    mov( eax, (type dword csetDest[4]));

    mov( (type dword csetSrc1[8]), eax );
    or( (type dword csetSrc2[8]), eax );
    mov( eax, (type dword csetDest[8]));

    mov( (type dword csetSrc1[12]), eax );
    or( (type dword csetSrc2[12]), eax );
    mov( eax, (type dword csetDest[12]));

    stdout.put( “Final set = {“, csetDest, “}” nl );

end cssetUnion;
```

Program 3.9 cs.setunion Implementation

The intersection of two sets is those elements that are members of both sets. In the bit array representation of character sets that HLA uses, this means that a bit is set in the destination character set if the corresponding bit is set in both the source sets; this corresponds to the logical AND operation; therefore, to compute the set intersection of two character sets, all you need do is logically AND the 16 bytes of the two source sets together. Program 3.10 provides a sample implementation.

```
// Program that demonstrates the implmentation of
// the cs.intersection function.

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

static
    csetSrc1: cset := {‘a’..’z’};
    csetSrc2: cset := {‘A’..’z’};
    csetDest: cset;

begin csIntersection;

    // To compute the intersection of csetSrc1 and csetSrc2 all we have
    // to do is logically AND the two sets together.

    mov( (type dword csetSrc1), eax );
    and( (type dword csetSrc2), eax );
    mov( eax, (type dword csetDest));

    mov( (type dword csetSrc1[4]), eax );
    and( (type dword csetSrc2[4]), eax );
    mov( eax, (type dword csetDest[4]));
    mov( (type dword csetSrc1[8]), eax );
    and( (type dword csetSrc2[8]), eax );
    mov( eax, (type dword csetDest[8]));

    mov( (type dword csetSrc1[12]), eax );
    and( (type dword csetSrc2[12]), eax );
    mov( eax, (type dword csetDest[12]));

    stdout.put( " Set A = {", csetSrc1, "}" nl );
    stdout.put( " Set B = {", csetSrc2, "}" nl );
    stdout.put( "Intersection of A and B = {", csetDest, "}" nl );

end csIntersection;
```

Program 3.10 cs.intersection Implementation

The difference of two sets is all the elements in the first set that are not also present in the second set. To compute this result we must logically AND the values from the first set with the inverted values of the second set; i.e., to compute C := A - B we use the following expression:

```
C := A and (not B);
```

Program 3.11 provides the code to implement this operation.

```
// Program that demonstrates the implmentation of
// the cs.difference function.

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

static
    csetSrc1: cset := {'0'..'9', 'a'..'z'};
    csetSrc2: cset := {'A'..'z'};
    csetDest: cset;

begin csDifference;

    // To compute the difference of csetSrc1 and csetSrc2 all we have
    // to do is logically AND A and NOT B together.

    mov( (type dword csetSrc2), eax );
    not( eax );
    and( (type dword csetSrc1), eax );
    mov( eax, (type dword csetDest));

    mov( (type dword csetSrc2[4]), eax );
    not( eax );
    and( (type dword csetSrc1[4]), eax );
    mov( eax, (type dword csetDest[4]));

    mov( (type dword csetSrc2[8]), eax );
    not( eax );
    and( (type dword csetSrc1[8]), eax );
    mov( eax, (type dword csetDest[8]));

    mov( (type dword csetSrc2[12]), eax );
    not( eax );
    and( (type dword csetSrc1[12]), eax );
    mov( eax, (type dword csetDest[12]));

    stdout.put( " Set A = {", csetSrc1, "}" nl );
    stdout.put( " Set B = {", csetSrc2, "}" nl );
    stdout.put( "Difference of A and B = {", csetDest, "}" nl );

end csDifference;
```

Program 3.11 cs.difference Implementation

3.9.3 Testing Character Sets

In addition to manipulating the members of a character set, the need often arises to compare character sets, check to see if a character is a member of a set, and check to see if a set is empty. In this section we'll discuss how HLA implements the relational operations on character sets.

Occasionally you'll want to check a character set to see if it contains any members. Although you could achieve this by creating a static *cset* variable with no elements and comparing the set in question against this empty set, there is a more efficient way to do this – just check to see if all the bits in the set in question are zero. An easy way to do this, that uses, is to logically OR the four double words in a cset object together. If the result is zero, then all the bits in the *cset* variable are zero and, hence, the character set is empty.

```
// Program that demonstrates the implmentation of
// the cs.IisEmpty function.

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

static
    csetSrc1: cset := {};
    csetSrc2: cset := {'A'..'Z'};

begin csIsEmpty;

    // To see if a set is empty, simply OR all the dwords
    // together and see if the result is zero:

    mov( (type dword csetSrc1[0]), eax );
    or( (type dword csetSrc1[4]), eax );
    or( (type dword csetSrc1[8]), eax );
    or( (type dword csetSrc1[12]), eax );

    if( @z ) then

        stdout.put( "csetSrc1 is empty ({", csetSrc1, "})" nl );

    else

        stdout.put( "csetSrc1 is not empty ({", csetSrc1, "})" nl );

    endif;

    // Repeat the test for csetSrc2:
```

```
    mov( (type dword csetSrc2[0]), eax );
    or( (type dword csetSrc2[4]), eax );
    or( (type dword csetSrc2[8]), eax );
    or( (type dword csetSrc2[12]), eax );

    if( @z ) then

        stdout.put( “csetSrc2 is empty ({“, csetSrc2, “})” nl );

    else

        stdout.put( “csetSrc2 is not empty ({“, csetSrc2, “})” nl );

    endif;

end csIsEmpty;
```

Program 3.12 Implementation of cs.IsEmpty

Perhaps the most common check on a character set is set membership; that is, checking to see if some character is a member of a given character set. As you’ve seen already (see “Character Set Implementation in HLA” on page 442), the BT instruction is perfect for this. Since we’ve already discussed how to use the BT instruction (along with, perhaps, a MOVZX instruction), there is no need to repeat the implementation of this operation here.

Two sets are equal if and only if all the bits are equal in the two set objects. Therefore, we can implement the cs.ne and cs.eq (set inequality and set equality) functions by comparing the four double words in a *cset* object and noting if there are any differences. Program 3.13 demonstrates how you can do this.

```
// Program that demonstrates the implmentation of
// the cs.eq and cs.ne functions.

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

static
    csetSrc1: cset := {‘a’..’z’};
    csetSrc2: cset := {‘a’..’z’};
    csetSrc3: cset := {‘A’..’Z’};

begin cseqne;

    // To see if a set equal to another, check to make sure
    // all four dwords are equal.  One sneaky way to do this
    // is to use the XOR operator (XOR is “not equals” as you
    // may recall).

    mov( (type dword csetSrc1[0]), eax ); // Set EAX to zero if these
    xor( (type dword csetSrc2[0]), eax ); //  two dwords are equal
    mov( eax, ebx );  // Accumulate result here.

    mov( (type dword csetSrc1[4]), eax );
    xor( (type dword csetSrc2[4]), eax );
    or( eax, ebx );
```

```
        mov( (type dword csetSrc1[8]), eax );
        xor( (type dword csetSrc2[8]), eax );
        or( eax, ebx );

        mov( (type dword csetSrc1[12]), eax );
        xor( (type dword csetSrc2[12]), eax );
        or( eax, ebx );

        // At this point, EBX is zero if the two csets are equal
        // (also, the zero flag is set if they are equal).

        if( @z ) then

            stdout.put( “csetSrc1 is equal to csetSrc2” nl );

        else

            stdout.put( “csetSrc1 is not equal to csetSrc2” nl );

        endif;

        // Implementation of cs.ne:

        mov( (type dword csetSrc1[0]), eax ); // Set EAX to zero if these
        xor( (type dword csetSrc3[0]), eax ); //  two dwords are equal
        mov( eax, ebx );  // Accumulate result here.

        mov( (type dword csetSrc1[4]), eax );
        xor( (type dword csetSrc3[4]), eax );
        or( eax, ebx );

        mov( (type dword csetSrc1[8]), eax );
        xor( (type dword csetSrc3[8]), eax );
        or( eax, ebx );

        mov( (type dword csetSrc1[12]), eax );
        xor( (type dword csetSrc3[12]), eax );
        or( eax, ebx );

        // At this point, EBX is non-zero if the two csets are not equal
        // (also, the zero flag is clear if they are not equal).

        if( @nz ) then

            stdout.put( “csetSrc1 is not equal to csetSrc3” nl );

        else

            stdout.put( “csetSrc1 is equal to csetSrc3” nl );

        endif;


    end cseqne;
```

Program 3.13 Implementation of cs.ne and cs.eq

The remaining tests on character sets roughly correspond to tests for less than or greater than; though in set theory we refer to these as superset and subset. One set is a subset of another if the second set contains all the elements of the first set; the second set may contain additional elements. The subset relationship is roughly equivalent to "less than or equal." The proper subset relation of two sets states that the elements of one set are all present in a second set and the two sets are not equal (i.e., the second set contains additional elements). This is roughly equivalent to the "less than" relationship.

Testing for a subset is an easy task. All you have to do is take the set intersection of the two sets and verify that the intersection of the two is equal to the first set. That is, A <= B if and only if:

```
A == ( A * B )  "*" denotes set intersection
```

Testing for a proper subset is a little more work. The same relationship above must hold but the resulting inspection must not be equal to B. That is, A < B if and only if,

```
(A == ( A * B )) and (B <> ( A * B ))
```

The algorithms for superset and proper superset are nearly identical. They are:

```
B == ( A * B )                          A >= B
(B == ( A * B )) and (A <> ( A * B ))   A >  B
```

The implementation of these four relational operations is left as an exercise.

3.10 Putting It All Together

This chapter describes HLA's implementation of character sets. Character sets are a very useful tool for validating user input and for other character scanning and manipulation operations. HLA uses a bit array implementation for character set objects. HLA's implementation allows for 128 different character values in a character set.

The HLA Standard Library provides a wide set of functions that let you build, manipulate, and compare character sets. Although these functions are convenient to use, most of the character set operations are so simple that you can implement them directly using in-line code. This chapter provided the implementation of many of the HLA Standard Library character set functions.

Note that this chapter does not cover all the uses of character sets in an assembly language program. In the volume on "Advanced String Handling" you will see many more uses for character sets in your programs.

Arrays Chapter Four

4.1 Chapter Overview

This chapter discusses how to declare and use arrays in your assembly language programs. This is probably the most important chapter on composite data structures in this text. Even if you elect to skip the chapters on Strings, Character Sets, Records, and Dates and Times, be sure you read and understand the material in this chapter. Much of the rest of the text depends on your understanding of this material.

4.2 Arrays

Along with strings, arrays are probably the most commonly used composite data type. Yet most beginning programmers have a very weak understanding of how arrays operate and their associated efficiency trade-offs. It's surprising how many novice (and even advanced!) programmers view arrays from a completely different perspective once they learn how to deal with arrays at the machine level.

Abstractly, an array is an aggregate data type whose members (elements) are all the same type. Selection of a member from the array is by an integer index[1]. Different indices select unique elements of the array. This text assumes that the integer indices are contiguous (though this is by no means required). That is, if the number *x* is a valid index into the array and *y* is also a valid index, with $x < y$, then all *i* such that $x < i < y$ are valid indices into the array.

Whenever you apply the indexing operator to an array, the result is the specific array element chosen by that index. For example, *A[i]* chooses the i^{th} element from array *A*. Note that there is no formal requirement that element *i* be anywhere near element *i+1* in memory. As long as *A[i]* always refers to the same memory location and *A[i+1]* always refers to its corresponding location (and the two are different), the definition of an array is satisfied.

In this text, we will assume that array elements occupy contiguous locations in memory. An array with five elements will appear in memory as shown in Figure 4.1

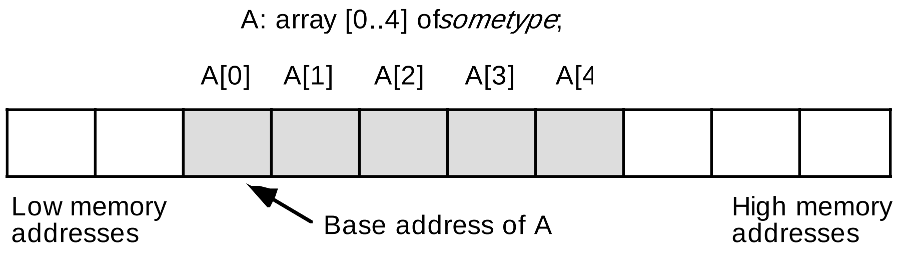

Figure 4.1 Array Layout in Memory

The *base address* of an array is the address of the first element on the array and always appears in the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, etc. Note that there is no requirement that the indices start at zero. They may start with any number as long as they are contiguous. However, for the purposes of discussion, it's easier to discuss accessing array elements if the first index is zero. This text generally begins most arrays at index zero unless

1. Or some value whose underlying representation is integer, such as character, enumerated, and boolean types.

there is a good reason to do otherwise. However, this is for consistency only. There is no efficiency benefit one way or another to starting the array index at zero.

To access an element of an array, you need a function that translates an array index to the address of the indexed element. For a single dimension array, this function is very simple. It is

```
Element_Address = Base_Address + ((Index - Initial_Index) * Element_Size)
```

where *Initial_Index* is the value of the first index in the array (which you can ignore if zero) and the value *Element_Size* is the size, in bytes, of an individual element of the array.

4.3 Declaring Arrays in Your HLA Programs

Before you access elements of an array, you need to set aside storage for that array. Fortunately, array declarations build on the declarations you've seen thus far. To allocate *n* elements in an array, you would use a declaration like the following in one of the variable declaration sections:

```
ArrayName: basetype[n];
```

ArrayName is the name of the array variable and *basetype* is the type of an element of that array. This sets aside storage for the array. To obtain the base address of the array, just use *ArrayName*.

The "[n]" suffix tells HLA to duplicate the object *n* times. Now let's look at some specific examples:

```
static

    CharArray: char[128];        // Character array with elements 0..127.
    IntArray: integer[ 8 ];      // “integer” array with elements 0..7.
    ByteArray: byte[10];         // Array of bytes with elements 0..9.
    PtrArray: dword[4];          // Array of double words with elements 0..3.
```

The second example, of course, assumes that you have defined the *integer* data type in the TYPE section of the program.

These examples all allocate storage for uninitialized arrays. You may also specify that the elements of the arrays be initialized to a single value using declarations like the following in the STATIC and READONLY sections:

```
RealArray: real32[8] := [ 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 ];
IntegerAry: integer[8] := [ 1, 1, 1, 1, 1, 1, 1, 1 ];
```

These definitions both create arrays with eight elements. The first definition initializes each four-byte real value to 1.0, the second declaration initializes each integer element to one. Note that the number of constants within the square brackets must match the size you declare for the array.

This initialization mechanism is fine if you want each element of the array to have the same value. What if you want to initialize each element of the array with a (possibly) different value? No sweat, just specify a different set of values in the list surrounded by the square brackets in the example above:

```
RealArray: real32[8] := [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 ];
IntegerAry: integer[8] := [ 1, 2, 3, 4, 5, 6, 7, 8 ];
```

4.4 HLA Array Constants

The last few examples in the last section demonstrate the use of HLA array constants. An HLA array constant is nothing more than a list of values (all the same time) surrounded by a pair of brackets. The following are all legal array constants:

```
                    [ 1, 2, 3, 4 ]
              [ 2.0, 3.14159, 1.0, 0.5 ]
```

```
[ 'a', 'b', 'c', 'd' ]
[ "Hello", "world", "of", "assembly" ]
```

(note that this last array constant contains four double word pointers to the four HLA strings appearing elsewhere in memory.)

As you saw in the previous section you can use array constants in the STATIC and READONLY sections to provide initial values for array variables. Of course, the number of comma separated items in an array constant must exactly match the number of array elements in the variable declaration. Likewise, the type of the array constant's elements must match the type of the elements in the array variable.

Using array constants to initialize small arrays is very convenient. Of course, if your array has several thousand elements in it, typing them all in will not be very much fun. Most arrays initialized this way have no more than a couple hundred entries, and generally far less than 100. It is reasonable to use an array constant to initialize such variables. However, at some point it will become far too tedious and error-prone to initialize arrays in this fashion. It is doubtful, for example, that you would want to manually initialize an array with 1,000 different elements using an array constant[2]. However, if you want to initialize all the elements of an array with the same value, HLA does provide a special array constant syntax for doing so. Consider the following declaration:

```
BigArray: uns32[ 1000 ] := 1000 dup [ 1 ];
```

This declaration creates a 1,000 element integer array initializing each element of the array with the value one. The "1000 dup [1]" expression tells HLA to create an array constant by duplicating the single value "[1]" one thousand times. You can even use the DUP operator to duplicate a series of values (rather than a single value) as the following example indicates:

```
SixteenInts: int32[16] := 4 dup [1,2,3,4];
```

This example initializes *SixteenInts* with four copies of the sequence "1, 2, 3, 4" yielding a total of sixteen different integers (i.e., 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4).

You will see some more possibilities with the DUP operator when looking at multidimensional arrays a little later.

4.5 Accessing Elements of a Single Dimension Array

To access an element of a zero-based array, you can use the simplified formula:

Element_Address = Base_Address + index * Element_Size

For the *Base_Address* entry you can use the name of the array (since HLA associates the address of the first element of an array with the name of that array). The *Element_Size* entry is the number of bytes for each array element. If the object is an array of bytes, the *Element_Size* field is one (resulting in a very simple computation). If each element of the array is a word (or other two-byte type) then *Element_Size* is two. And so on. To access an element of the *SixteenInts* array in the previous section, you'd use the formula:

```
Element_Address = SixteenInts + index*4
```

The 80x86 code equivalent to the statement "EAX:=SixteenInts[index]" is

```
mov( index, ebx );
shl( 2, ebx );                      //Sneaky way to compute 4*ebx
mov( SixteenInts[ ebx ], eax );
```

There are two important things to notice here. First of all, this code uses the SHL instruction rather than the INTMUL instruction to compute *4*index*. The main reason for choosing SHL is that it was more efficient. It turns out that SHL is a *lot* faster than INTMUL on many processors.

2. In the chapter on Macros and the HLA Run-Time Language you will learn how to automate the initialization of large array objects. So initializing large objects is not completely out of the question.

The second thing to note about this instruction sequence is that it does not explicitly compute the sum of the base address plus the index times two. Instead, it relies on the indexed addressing mode to implicitly compute this sum. The instruction “mov(SixteenInts[ebx], eax);” loads EAX from location *SixteenInts*+EBX which is the base address plus *index**4 (since EBX contains *index**4). Sure, you could have used

```
lea( eax, SixteenInts );
mov( index, ebx );
shl( 2, ebx );                    //Sneaky way to compute 4*ebx
add( eax, ebx );                  //Compute base address plus index*4
mov( SixteenInts[ ebx ], eax );
```

in place of the previous sequence, but why use five instructions where three will do the same job? This is a good example of why you should know your addressing modes inside and out. Choosing the proper addressing mode can reduce the size of your program, thereby speeding it up.

Of course, as long as we’re discussing efficiency improvements, it’s worth pointing out that the 80x86 scaled indexed addressing modes let you automatically multiply an index by one, two, four, or eight. Since this current example multiplies the index by four, we can simplify the code even farther by using the scaled indexed addressing mode:

```
mov( index, ebx );
mov( SixteenInts[ ebx*4 ], eax );
```

Note, however, that if you need to multiply by some constant other than one, two, four, or eight, then you cannot use the scaled indexed addressing modes. Similarly, if you need to multiply by some element size that is not a power of two, you will not be able to use the SHL instruction to multiply the index by the element size; instead, you will have to use INTMUL or some other instruction sequence to do the multiplication.

The indexed addressing mode on the 80x86 is a natural for accessing elements of a single dimension array. Indeed, it’s syntax even suggests an array access. The only thing to keep in mind is that you must remember to multiply the index by the size of an element. Failure to do so will produce incorrect results.

Before moving on to multidimensional arrays, a couple of additional points about addressing modes and arrays are in order. The above sequences work great if you only access a single element from the *SixteenInts* array. However, if you access several different elements from the array within a short section of code, and you can afford to dedicate another register to the operation, you can certainly shorten your code and, perhaps, speed it up as well. Consider the following code sequence:

```
lea( ebx, SixteenInts );
mov( index, esi );
mov( [ebx+esi*4], eax );
```

Now EBX contains the base address and ESI contains the *index* value. Of course, this hardly appears to be a good trade-off. However, when accessing additional elements if *SixteenInts* you do not have to reload EBX with the base address of *SixteenInts* for the next access. The following sequence is a little shorter than the comparable sequence that doesn’t load the base address into EBX:

```
lea( ebx, SixteenInts );
mov( index, esi );
mov( [ebx+esi*4], eax );
 .
 .                          //Assumption: EBX is left alone
 .                          //            through this code.
mov( index2, esi );
mov( [ebx+esi*4], eax );
```

This code is slightly shorter because the “mov([ebx+esi*4], eax);” instruction is slightly shorter than the “mov(SixteenInts[ebx*4], eax);” instruction. Of course the more accesses to *SixteenInts* you make without reloading EBX, the greater your savings will be. Tricky little code sequences such as this one sometimes pay off handsomely. However, the savings depend entirely on which processor you’re using. Code sequences that run faster on one 80x86 CPU might actually run *slower* on a different CPU. Unfortunately, if speed is what you're after there are no hard and fast rules. In fact, it is very difficult to predict the speed of most instructions on the 80x86 CPUs.

4.5.1 Sorting an Array of Values

Almost every textbook on this planet gives an example of a sort when introducing arrays. Since you've probably seen how to do a sort in high level languages already, it's probably instructive to take a quick look at a sort in HLA. The example code in this section will use a variant of the Bubble Sort which is great for short lists of data and lists that are nearly sorted, but horrible for just about everything else[3].

```
const
    NumElements:= 16;

static
    DataToSort: uns32[ NumElements ] :=
                    [
                        1, 2, 16, 14,
                        3, 9, 4,  10,
                        5, 7, 15, 12,
                        8, 6, 11, 13
                    ];

    NoSwap: boolean;

        .
        .
        .

    // Bubble sort for the DataToSort array:

    repeat

        mov( true, NoSwap );
        for( mov( 0, ebx ); ebx <= NumElements-2; inc( ebx )) do

            mov( DataToSort[ ebx*4], eax );
            if( eax > DataToSort[ ebx*4 + 4] ) then

                mov( DataToSort[ ebx*4 + 4 ], ecx );
                mov( ecx, DataToSort[ ebx*4 ] );
                mov( eax, DataToSort[ ebx*4 + 4 ] ); // Note: EAX contains
                mov( false, NoSwap );                // DataToSort[ ebx*4 ]

            endif;

        endfor;

    until( NoSwap );
```

The bubble sort works by comparing adjacent elements in an array. The interesting thing to note in this code fragment is how it compares adjacent elements. You will note that the IF statement compares EAX (which contains DataToSort[ebx*4]) against DataToSort[EBX*4 + 4]. Since each element of this array is four bytes (*uns32*), the index [EBX*4 + 4] references the next element beyond [EBX*4].

3. Fear not, you'll see some better sorting algorithms in the chapter on procedures and recursion.

As is typical for a bubble sort, this algorithm terminates if the innermost loop completes without swapping any data. If the data is already presorted, then the bubble sort is very efficient, making only one pass over the data. Unfortunately, if the data is not sorted (worst case, if the data is sorted in reverse order), then this algorithm is extremely inefficient. Indeed, although it is possible to modify the code above so that, on the average, it runs about twice as fast, such optimizations are wasted on such a poor algorithm. However, the Bubble Sort is very easy to implement and understand (which is why introductory texts continue to use it in examples). Fortunately, you will learn about more advanced sorts later in this text, so you won't be stuck with it for very long.

4.6 Multidimensional Arrays

The 80x86 hardware can easily handle single dimension arrays. Unfortunately, there is no magic addressing mode that lets you easily access elements of multidimensional arrays. That's going to take some work and lots of instructions.

Before discussing how to declare or access multidimensional arrays, it would be a good idea to figure out how to implement them in memory. The first problem is to figure out how to store a multi-dimensional object into a one-dimensional memory space.

Consider for a moment a Pascal array of the form "A:array[0..3,0..3] of char;". This array contains 16 bytes organized as four rows of four characters. Somehow you've got to draw a correspondence with each of the 16 bytes in this array and 16 contiguous bytes in main memory. Figure 4.2 shows one way to do this:

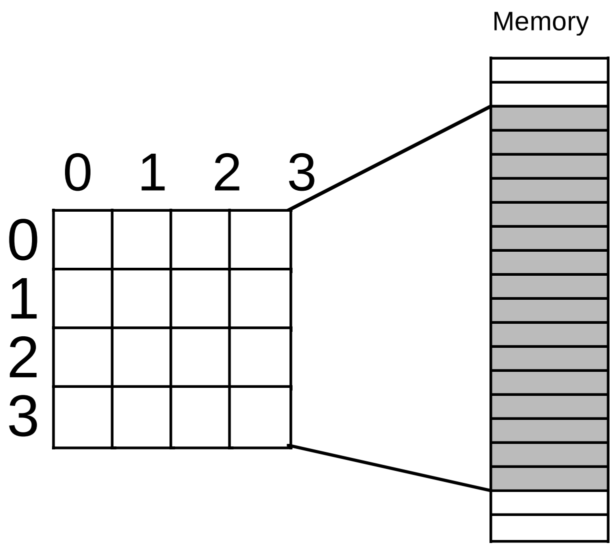

Figure 4.2 Mapping a 4x4 Array to Sequential Memory Locations

The actual mapping is not important as long as two things occur: (1) each element maps to a unique memory location (that is, no two entries in the array occupy the same memory locations) and (2) the mapping is consistent. That is, a given element in the array always maps to the same memory location. So what you really need is a function with two input parameters (row and column) that produces an offset into a linear array of sixteen memory locations.

Now any function that satisfies the above constraints will work fine. Indeed, you could randomly choose a mapping as long as it was unique. However, what you really want is a mapping that is efficient to compute at run time and works for any size array (not just 4x4 or even limited to two dimensions). While there are a large number of possible functions that fit this bill, there are two functions in particular that most programmers and most high level languages use: *row major ordering* and *column major ordering.*

4.6.1 Row Major Ordering

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. This mapping is demonstrated in Figure 4.3:

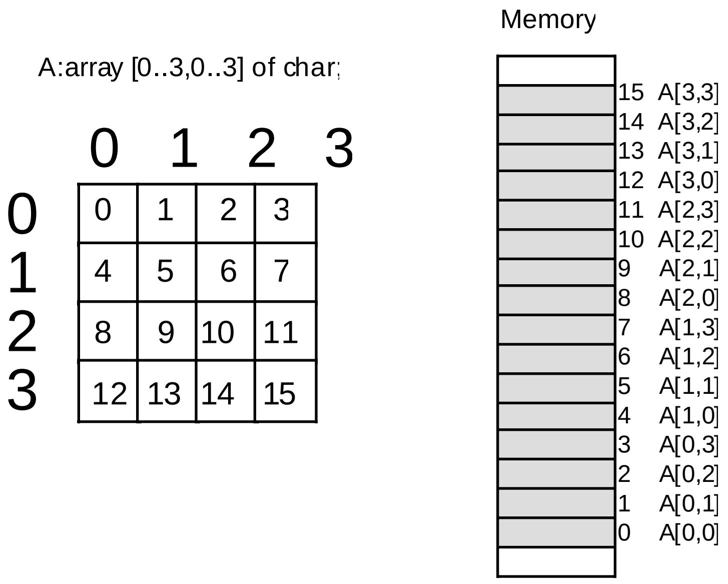

Figure 4.3 Row Major Array Element Ordering

Row major ordering is the method employed by most high level programming languages including Pascal, C/C++, Java, Ada, Modula-2, etc. It is very easy to implement and easy to use in machine language. The conversion from a two-dimensional structure to a linear array is very intuitive. You start with the first row (row number zero) and then concatenate the second row to its end. You then concatenate the third row to the end of the list, then the fourth row, etc. (see Figure 4.4).

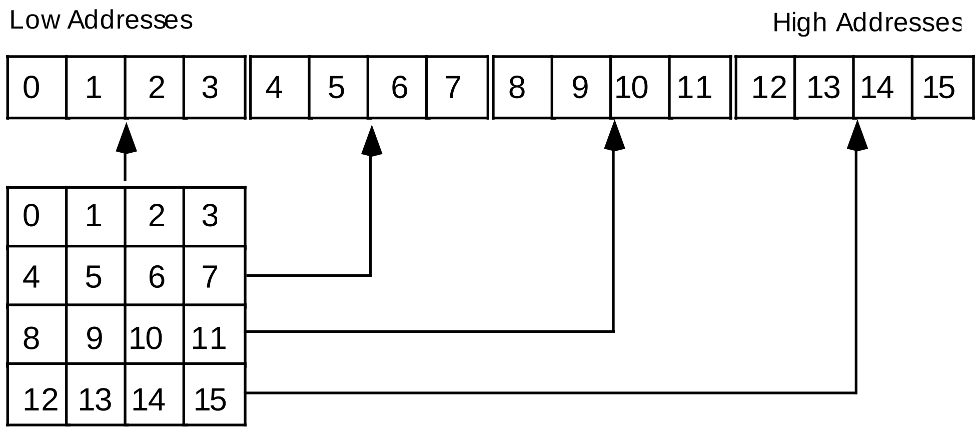

Figure 4.4 Another View of Row-Major Ordering for a 4x4 Array

For those who like to think in terms of program code, the following nested Pascal loop also demonstrates how row major ordering works:

```
index := 0;
for colindex := 0 to 3 do
    for rowindex := 0 to 3 do
    begin

        memory [index] := rowmajor [colindex][rowindex];
        index := index + 1;
    end;
```

The important thing to note from this code, that applies regardless of the number of dimensions, is that the rightmost index increases the fastest. That is, as you allocate successive memory locations you increment the rightmost index until you reach the end of the current row. Upon reaching the end, you reset the index back to the beginning of the row and increment the next successive index by one (that is, move down to the next row.). This works equally well for any number of dimensions[4]. The following Pascal segment demonstrates row major organization for a 4x4x4 array:

```
index := 0;
for depthindex := 0 to 3 do
    for colindex := 0 to 3 do
       for rowindex := 0 to 3 do begin

        memory [index] := rowmajor [depthindex][colindex][rowindex];
        index := index + 1;

       end;
```

The actual function that converts a list of index values into an offset doesn't involve loops or much in the way of fancy computations. Indeed, it's a slight modification of the formula for computing the address of an element of a single dimension array. The formula to compute the offset for a two-dimension row major ordered array declared in Pascal as "A:array [0..3,0..3] of integer" is

```
Element_Address = Base_Address + (colindex * row_size + rowindex) * Element_Size
```

As usual, *Base_Address* is the address of the first element of the array (*A[0][0]* in this case) and *Element_Size* is the size of an individual element of the array, in bytes. *Colindex* is the leftmost index, *rowindex* is the rightmost index into the array. *Row_size* is the number of elements in one row of the array (four, in

4. By the way, the number of dimensions of an array is its *arity*.

this case, since each row has four elements). Assuming *Element_Size* is one, this formula computes the following offsets from the base address:

Column index	Row Index	Offset into Array
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

For a three-dimensional array, the formula to compute the offset into memory is the following:

```
Address = Base + ((depthindex*col_size+colindex) * row_size + rowindex) * Element_Size
```

Col_size is the number of items in a column, *row_size* is the number of items in a row. In C/C++, if you've declared the array as "*type* A[i] [j] [k];" then *row_size* is equal to *k* and *col_size* is equal to *j*.

For a four dimensional array, declared in C/C++ as "type A[i] [j] [k] [m];" the formula for computing the address of an array element is

```
Address =
Base + (((LeftIndex * depth_size + depthindex)*col_size+colindex) * row_size +
rowindex) * Element_Size
```

Depth_size is equal to *j*, *col_size* is equal to *k*, and *row_size* is equal to *m*. *LeftIndex* represents the value of the leftmost index.

By now you're probably beginning to see a pattern. There is a generic formula that will compute the offset into memory for an array with *any* number of dimensions, however, you'll rarely use more than four.

Another convenient way to think of row major arrays is as arrays of arrays. Consider the following single dimension Pascal array definition:

```
A: array [0..3] of  sometype;
```

Assume that *sometype* is the type "sometype = array [0..3] of char;".

A is a single dimension array. Its individual elements happen to be arrays, but you can safely ignore that for the time being. The formula to compute the address of an element of a single dimension array is

```
Element_Address = Base + Index * Element_Size
```

In this case *Element_Size* happens to be four since each element of *A* is an array of four characters. So what does this formula compute? It computes the base address of each row in this 4x4 array of characters (see Figure 4.5):

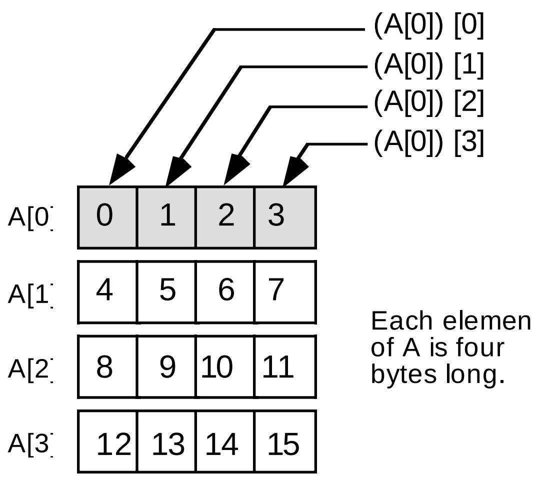

Figure 4.5 Viewing a 4x4 Array as an Array of Arrays

Of course, once you compute the base address of a row, you can reapply the single dimension formula to get the address of a particular element. While this doesn't affect the computation at all, conceptually it's probably a little easier to deal with several single dimension computations rather than a complex multidimensional array element address computation.

Consider a Pascal array defined as "A:array [0..3] [0..3] [0..3] [0..3] [0..3] of char;" You can view this five-dimension array as a single dimension array of arrays. The following Pascal code demonstrates such a definition:

```
type
        OneD = array [0..3] of char;
        TwoD = array [0..3] of OneD;
        ThreeD = array [0..3] of TwoD;
        FourD = array [0..3] of ThreeD;
var
        A : array [0..3] of FourD;
```

The size of *OneD* is four bytes. Since *TwoD* contains four *OneD* arrays, its size is 16 bytes. Likewise, *ThreeD* is four *TwoDs*, so it is 64 bytes long. Finally, *FourD* is four *ThreeDs*, so it is 256 bytes long. To compute the address of "A [b, c, d, e, f]" you could use the following steps:

- Compute the address of *A [b]* as "Base + *b* * size". Here size is 256 bytes. Use this result as the new base address in the next computation.
- Compute the address of *A [b, c]* by the formula "Base + *c**size", where *Base* is the value obtained immediately above and size is 64. Use the result as the new base in the next computation.
- Compute the address of *A [b, c, d]* by "Base + *d**size" with *Base* coming from the above computation and size being 16.
- Compute the address of *A [b, c, d, e]* with the formula "Base + *e**size" with Base from above and size being four. Use this value as the base for the next computation.
- Finally, compute the address of *A [b, c, d, e, f]* using the formula "Base + *f**size" where base comes from the above computation and size is one (obviously you can simply ignore this final multiplication). The result you obtain at this point is the address of the desired element.

Not only is this scheme easier to deal with than the fancy formulae given earlier, but it is easier to compute (using a single loop) as well. Suppose you have two arrays initialized as follows

A1 = [256, 64, 16, 4, 1] and A2 = [b, c, d, e, f]

then the Pascal code to perform the element address computation becomes:

```
        for i := 0 to 4 do
            base := base + A1[i] * A2[i];
```

Presumably *base* contains the base address of the array before executing this loop. Note that you can easily extend this code to any number of dimensions by simply initializing *A1* and *A2* appropriately and changing the ending value of the for loop.

As it turns out, the computational overhead for a loop like this is too great to consider in practice. You would only use an algorithm like this if you needed to be able to specify the number of dimensions at run time. Indeed, one of the main reasons you won't find higher dimension arrays in assembly language is that assembly language displays the inefficiencies associated with such access. It's easy to enter something like "*A [b,c,d,e,f]*" into a Pascal program, not realizing what the compiler is doing with the code. Assembly language programmers are not so cavalier – they see the mess you wind up with when you use higher dimension arrays. Indeed, good assembly language programmers try to avoid two dimension arrays and often resort to tricks in order to access data in such an array when its use becomes absolutely mandatory. But more on that a little later.

4.6.2 Column Major Ordering

Column major ordering is the other function frequently used to compute the address of an array element. FORTRAN and various dialects of BASIC (e.g., older versions of Microsoft BASIC) use this method to index arrays.

In row major ordering the rightmost index increased the fastest as you moved through consecutive memory locations. In column major ordering the leftmost index increases the fastest. Pictorially, a column major ordered array is organized as shown in Figure 4.6:

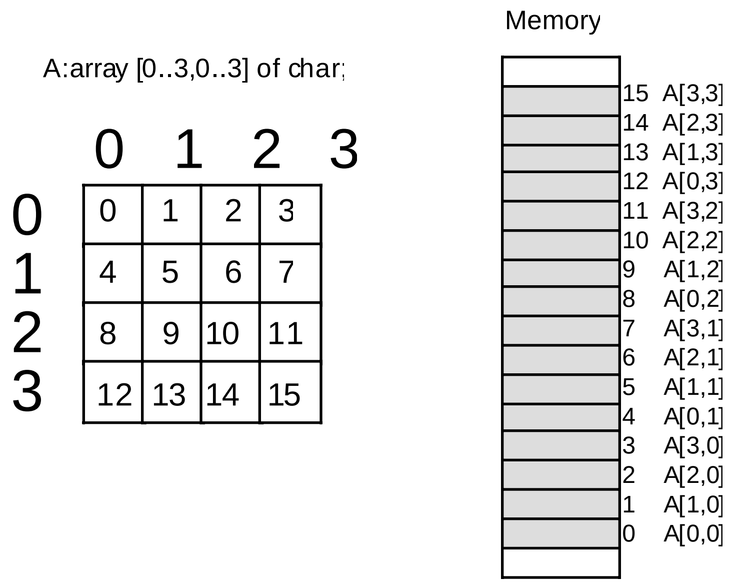

Figure 4.6 Column Major Array Element Ordering

The formulae for computing the address of an array element when using column major ordering is very similar to that for row major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimension column major array:

```
Element_Address = Base_Address + (rowindex * col_size + colindex) * Element_Size
```

For a three-dimension column major array:

```
Address = Base + ((rowindex*col_size+colindex) * depth_size + depthindex) *
Element_Size
```

For a four-dimension column major array:

```
Address =
    Base + (((rowindex * col_size + colindex)*depth_size + depthindex) *
        Left_size + Leftindex) * Element_Size
```

The single Pascal loop provided for row major access remains unchanged (to access A[b][c][d][e][f]):

```
for i := 0 to 4 do
    base := base + A1[i] * A2[i];
```

Likewise, the initial values of the *A1* array remain unchanged:

```
A1 = {256, 64, 16, 4, 1}
```

The only thing that needs to change is the initial values for the *A2* array, and all you have to do here is reverse the order of the indices:

```
A2 = {f, e, d, c, b}
```

4.7 Allocating Storage for Multidimensional Arrays

If you have an *m* x *n* array, it will have *m * n* elements and require *m*n*Element_Size* bytes of storage. To allocate storage for an array you must reserve this amount of memory. As usual, there are several different ways of accomplishing this task. Fortunately, HLA's array declaration syntax is very similar to high level language array declaration syntax, so C/C++, BASIC, and Pascal programmers will feel right at home. To declare a multidimensional array in HLA, you use a declaration like the following:

```
ArrayName: elementType [ comma_separated_list_of_dimension_bounds ];
```

For example, here is a declaration for a 4x4 array of characters:

```
GameGrid: char[ 4, 4 ];
```

Here is another example that shows how to declare a three dimensional array of strings:

```
NameItems: string[ 2, 3, 3 ];
```

Remember, string objects are really pointers, so this array declaration reserves storage for 18 double word pointers (2*3*3=18).

As was the case with single dimension arrays, you may initialize every element of the array to a specific value by following the declaration with the assignment operator and an array constant. Array constants ignore dimension information; all that matters is that the number of elements in the array constant correspond to the number of elements in the actual array. The following example shows the *GameGrid* declaration with an initializer:

```
GameGrid: char[ 4, 4 ] :=
    [
        'a', 'b', 'c', 'd',
        'e', 'f', 'g', 'h',
        'i', 'j', 'k', 'l',
        'm', 'n', 'o', 'p'
    ];
```

Note that HLA ignores the indentation and extra whitespace characters (e.g., newlines) appearing in this declaration. It was laid out to enhance readability (which is always a good idea). HLA does not interpret the four separate lines as representing rows of data in the array. Humans do, which is why it's good to lay out the initial data in this manner, but HLA completely ignores the physical layout of the declaration. All that matters is that there are 16 (4*4) characters in the array constant. You'll probably agree that this is much easier to read than

```
GameGrid: char[ 4,4 ] :=
    [ ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’, ‘l’, ‘m’,
      ‘n’, ‘o’, ‘p’ ];
```

Of course, if you have a large array, an array with really large rows, or an array with many dimensions, there is little hope for winding up with something reasonable. That's when comments that carefully explain everything come in handy.

As with single dimension arrays, you can use the DUP operator to initialize each element of a really large array with the same value. The following example initializes a 256x64 array of bytes so that each byte contains the value $FF:

```
StateValue: byte[ 256, 64 ] := 256*64 dup [$ff];
```

Note the use of a constant expression to compute the number of array elements rather than simply using the constant 16,384 (256*64). The use of the constant expression more clearly suggests that this code is initializing each element of a 256x64 element array than does the simple literal constant 16,384.

Another HLA trick you can use to improve the readability of your programs is to use *nested array constants*. The following is an example of an HLA nested array constant:

```
[ [0, 1, 2], [3, 4], [10, 11, 12, 13] ]
```

Whenever HLA encounters an array constant nested inside another array constant, it simply removes the brackets surrounding the nested array constant and treats the whole constant as a single array constant. For example, HLA converts the nested array constant above to the following:

```
[ 0, 1, 2, 3, 4, 10, 11, 12, 13 ]
```

You can take advantage of this fact to help make your programs a little more readable. For multidimensional array constants you can enclose each row of the constant in square brackets to denote that the data in each row is grouped and separate from the other rows. As an example, consider the following declaration for the *GameGrid* array that is identical (as far as HLA is concerned) to the previous declaration:

```
GameGrid: char[ 4, 4 ] :=
    [
        [ ‘a’, ‘b’, ‘c’, ‘d’ ],
        [ ‘e’, ‘f’, ‘g’, ‘h’ ],
        [ ‘i’, ‘j’, ‘k’, ‘l’ ],
        [ ‘m’, ‘n’, ‘o’, ‘p’ ]
    ];
```

This declaration makes it clearer that the array constant is a 4x4 array rather than just a 16-element one-dimensional array whose elements wouldn't fit all on one line of source code. Little aesthetic improvements like this are what separate mediocre programmers from good programmers.

4.8 Accessing Multidimensional Array Elements in Assembly Language

Well, you've seen the formulae for computing the address of an array element. You've even looked at some Pascal code you could use to access elements of a multidimensional array. Now it's time to see how to access elements of those arrays using assembly language.

The MOV, SHL, and INTMUL instructions make short work of the various equations that compute offsets into multidimensional arrays. Let's consider a two dimension array first:

```
static
```

```
    i:      int32;
    j:      int32;
    TwoD:   int32[ 4, 8 ];

          .
          .
          .

// To peform the operation TwoD[i,j] := 5; you'd use code like the following.
// Note that the array index computation is (i*8 + j)*4.

        mov( i, ebx );
        shl( 3, ebx );       // Multiply by eight (shl by 3 is a multiply by 8).
        add( j, ebx );
        mov( 5, TwoD[ ebx*4 ] );
```

Note that this code does *not* require the use of a two register addressing mode on the 80x86. Although an addressing mode like *TwoD[ebx][esi]* looks like it should be a natural for accessing two dimensional arrays, that isn't the purpose of this addressing mode.

Now consider a second example that uses a three dimension array:

```
static
    i:         int32;
    j:         int32;
    k:         int32;
    ThreeD:    int32[ 3, 4, 5 ];
        .
        .
        .

// To peform the operation ThreeD[i,j,k] := ESI; you'd use the following code
// that computes ((i*4 + j)*5 + k )*4 as the address of ThreeD[i,j,k].

        mov( i, ebx );
        shl( 2, ebx );       // Four elements per column.
        add( j, ebx );
        intmul( 5, ebx );    // Five elements per row.
        add( k, ebx );
        mov( esi, ThreeD[ ebx*4 ] );
```

Note that this code uses the INTMUL instruction to multiply the value in EBX by five. Remember, the SHL instruction can only multiply a register by a power of two. While there are ways to multiply the value in a register by a constant other than a power of two, the INTMUL instruction is more convenient[5].

4.9 Large Arrays and MASM

There is a defect in later versions of MASM v6.x that create some problems when you declare large static arrays in your programs. Now you may be wondering what this has to do with you since we're using HLA, but don't forget that HLA v1.x compiles to MASM assembly code and then runs MASM to assemble this output. Therefore, any defect in MASM is going to be a problem for HLA users.

The problem occurs when the total number of array elements you declare in a static section (STATIC, READONLY, or STORAGE) starts to get large. Large in this case is CPU dependent, but it falls somewhere between 128,000 and one million elements for most systems. MASM, for whatever reason, uses a very slow algorithm to emit array code to the object file; by the time you declare 64K array elements, MASM starts to produce a noticeable delay while compiling your code. After that point, the delay grows linearly with the

5. A full discussion of multiplication by constants other than a power of two appears in the chapter on arithmetic.

number of array elements (i.e., as you double the number of array elements you double the assembly time) until the data saturates MASM's internal buffers and the cache. Then there is a big jump in execution time. For example, on a 300 MHz Pentium II processor, compiling a program with an array with 256,000 elements takes about 30 seconds, compiling a program with an array having 512,000 element takes several minutes. Compiling a program with a one-megabyte array seems to take forever.

There are a couple of ways to solve this problem. First, of course, you can limit the size of your arrays in your program. Unfortunately, this isn't always an option available to you. The second possibility is to use MASM v6.11; the defect was introduced in MASM after this version. The problem with MASM v6.11 is that it doesn't support the MMX instruction set, so if you're going to compile MMX instructions (or other instructions that MASM v6.11 doesn't support) with HLA you will not be able to use this option. A third option is to put your arrays in a VAR section rather than a static declaration section; HLA processes arrays you declare in the VAR section so MASM never sees them. Hence, arrays you declare in the VAR section don't suffer from this problem.

4.10 Dynamic Arrays in Assembly Language

One problem with arrays up to this point is that their size is static. That is, the number of elements in all of the examples is chosen when writing the program, it is not set while the program is running (i.e., dynamically). Alas, sometimes you simply don't know how big an array needs to be when you're writing the program; you can only determine the size of the array while the program is running. This section describes how to allocate storage for arrays dynamically so you can set their size at run time.

Allocating storage for a single dimension array, and accessing elements of that array, is a nearly trivial task at run time. All you need to do is call the HLA Standard Library *malloc* routine specifying the size of the array, in bytes. *Malloc* will return a pointer to the base address of the new array in the EAX register. Typically, you would save this address in a pointer variable and use that value as the base address of the array in all future array accesses.

To access an element of a single dimensional dynamic array, you would generally load the base address into a register and compute the index in a second register. Then you could use the based indexed addressing mode to access elements of that array. This is not a whole lot more work than accessing elements of a statically allocated array. The following code fragment demonstrates how to allocate and access elements of a single dimension dynamic array:

```
static
    ArySize:        uns32;
    BaseAdrs:       pointer to uns32;
        .
        .
        .
    stdout.put( “How many elements do you want in your array? “ );
    stdin.getu32();
    mov( eax, ArySize;        // Save away the upper bounds on this array.
    shl( 2, eax );            // Multiply eax by four to compute the number of bytes.
    malloc( eax );            // Allocate storage for the array.
    mov( eax, BaseAdrs );     // Save away the base address of the new array.
        .
        .
        .

    // Zero out each element of the array:

    mov( BaseAdrs, ebx );
    mov( 0, eax );
    for( mov(0, esi); esi < ArySize; inc( esi )) do

        mov( eax, [ebx + esi*4 ]);
    endfor;
```

Dynamically allocating storage for a multidimensional array is fairly straight-forward. The number of elements in a multidimensional array is the product of all the dimension values; e.g., a 4x5 array has 20 elements. So if you get the bounds for each dimension from the user, all you need do is compute the product of all of these bound values and multiply the final result by the size of a single element. This computes the total number of bytes in the array, the value that *malloc* expects.

Accessing elements of multidimensional arrays is a little more problematic. The problem is that you need to keep the dimension information (that is, the bounds on each dimension) around because these values are needed when computing the row major (or column major) index into the array[6]. The conventional solution is to store these bounds into a static array (generally you know the *arity*, or number of dimensions, at compile-time, so it is possible to statically allocate storage for this array of dimension bounds). This array of dynamic array bounds is known as a *dope vector*. The following code fragment shows how to allocate storage for a two-dimensional dynamic array using a simple dope vector.

```
var
    ArrayPtr:   pointer to uns32;
    ArrayDims:  uns32[2];
        .
        .
        .
    // Get the array bounds from the user:

    stdout.put( "Enter the bounds for dimension #1: " );
    stdin.get( ArrayDims[0] );

    stdout.put( "Enter the bounds for dimension #2: " );
    stdin.get( ArrayDims[1*4] );

    // Allocate storage for the array:

    mov( ArrayDims[0], eax );
    intmul( ArrayDims[1*4], eax );
    shl( 2, eax );          // Multiply by four since each element is 4 bytes.
    malloc( eax );          // Allocate storage for the array and
    mov( eax, ArrayPtr );   //  save away the pointer to the array.


    // Initialize the array:

    mov( 0, edx );
    mov( ArrayPtr, edi );
    for( mov( 0, ebx ); ebx < ArrayDims[0]; inc( ebx )) do

        for( mov( 0, ecx ); ecx < ArrayDims[1*4]; inc( ecx )) do

            // Compute the index into the array
            // as esi := ( ebx * ArrayDims[1*4] + ecx ) * 4
            // (Note that the final multiplication by four is
            //  handled by the scaled indexed addressing mode below.)

            mov( ebx, esi );
            intmul( ArrayDims[1*4], esi );
            add( ecx, esi );

            // Initialize the current array element with edx.
```

6. Technically, you don't need the value of the left-most dimension bound to compute an index into the array, however, if you want to check the index bounds using the BOUND instruction (or some other technique), you will need this value around at run-time as well.

```
            mov( edx, [edi+esi*4] );
            inc( edx );

        endfor;

    endfor;
```

4.11 HLA Standard Library Array Support

The HLA Standard Library provides an array module that helps reduce the effort needed to support static and dynamic arrays in your program. The "arrays.hhf" library module provides code to declare and allocate dynamic arrays, compute the index into an array, copy arrays, perform row reductions, transpose arrays, and more. This section will explore some of the more useful features the arrays module provides.

One of the more interesting features of the HLA Standard Library arrays module is that most of the array manipulation procedures support both statically allocated and dynamically allocated arrays. In fact, the HLA array procedures can automatically figure out if an array is static or dynamic and generate the appropriate code for that array. There is one catch, however. In order for HLA to be able to differentiate statically and dynamically allocated arrays, you must use the dynamic array declarations found in the arrays package. This won't be a problem because HLA's dynamic array facilities are powerful and very easy to use.

To declare a dynamic array with the HLA arrays package, you use a variable declaration like the following:

```
variableName: array.dArray( elementType, Arity );
```

The *elementType* parameter is a regular HLA data type identifier (e.g., *int32* or some type identifier you've defined in the TYPE section). The *Arity* parameter is a constant that specifies the number of dimensions for the array (*arity* is the formal name for "number of dimensions"). Note that you do not specify the bounds of each dimension in this declaration. Storage allocation occurs later, at run time. The following is an example of a declaration for a dynamically allocated two-dimensional matrix:

```
ScreenBuffer: array.dArray( char, 2 );
```

The *array.dArray* data type is actually an HLA macro[7] that expands the above to the following:

```
ScreenBuffer: record
                  dataPtr:      dword;
                  dopeVector:   uns32[ 2 ];
                  elementType:  char;
              endrecord;
```

The *dataPtr* field will hold the base address of the array once the program allocates storage for it. The *dopeVector* array has one element for each array dimension (the macro uses the second parameter of the *array.dArray* type as the number of dimensions for the *dopeVector* array). The *elementType* field is a single object that has the same type as an element of the dynamic array. HLA provides a couple of built-in functions that you can use on these fields to extract important information. The *@Elements* function returns the number of elements in an array. Therefore, "@Elements(ScreenBuffer.dopeVector)" will return the number of elements (two) in the *ScreenBuffer.dopeVector* array. Since this array contains one element for each dimension in the dynamic array, you can use the *@Elements* function with the *dopeVector* field to determine the arity of the array. You can use the HLA *@Size* function on the *ScreenBuffer.elementType* field to determine the size of an array element in the dynamic array. Most of the time you will know the arity and type of your dynamic arrays (after all, you declared them), so you probably won't use these functions often until you start writing macros that process dynamic arrays.

7. See the chapter on Macros and the HLA Compile-Time Language for details on macros.

After you declare a dynamic array, you must initialize the dynamic array object before attempting to use the array. The HLA Standard Library *array.daAlloc* routine handles this task for you. This routine uses the following syntax:

```
array.daAlloc( arrayName, comma_separated_list_of_array_bounds );
```

To allocate storage for the *ScreenBuffer* variable in the previous example you could use a call like the following:

```
array.daAlloc( ScreenBuffer, 20, 40 );
```

This call will allocate storage for a 20x40 array of characters. It will store the base address of the array into the *ScreenBuffer.dataPtr* field. It will also initialize *ScreenBuffer.dopeVector[0]* with 20 and *ScreenBuffer.dopeVector[1*4]* with 40. To access elements of the *ScreenBuffer* array you can use the techniques of the previous section, or you could use the *array.index* function.

The *array.index* function automatically computes the address of an array element for you. This function uses the following call syntax:

```
array.index( reg32, arrayName, comma_separated_list_of_index_values );
```

The first parameter must be a 32-bit register. The *array.index* function will store the address of the specified array element in this register. The second *array.index* parameter must be the name of an array; this can be either a statically allocated array or an array you've declared with *array.dArray* and allocated dynamically with *array.daAlloc*. Following the array name parameter is a list of one or more array indices. The number of array indices must match the arity of the array. These array indices can be constants, *dword* memory variables, or registers (however, you must not specify the same register that appears in the first parameter as one of the array indices). Upon return from this function, you may access the specified array element using the register indirect addressing mode and the register appearing as the first parameter.

One last routine you'll want to know about when manipulating HLA dynamic arrays is the *array.daFree* routine. This procedure expects a single parameter that is the name of an HLA dynamic array. Calling *array.daFree* will free the storage associated with the dynamic array. The following code fragment is a rewrite of the example from the previous section that uses HLA dynamic arrays:

```
var
    da:     array.dArray( uns32, 2 );
    Bnd1:   uns32;
    Bnd2:   uns32;
        .
        .
        .
    // Get the array bounds from the user:

    stdout.put( "Enter the bounds for dimension #1: " );
    stdin.get( Bnd1 );

    stdout.put( "Enter the bounds for dimension #2: " );
    stdin.get( Bnd2 );

    // Allocate storage for the array:

    array.daAlloc( da, Bnd1, Bnd2 );


    // Initialize the array:

    mov( 0, edx );
    for( mov( 0, ebx ); ebx < Bnd1; inc( ebx )) do

        for( mov( 0, ecx ); ecx < Bnd2; inc( ecx )) do
```

```
            // Initialize the current array element with edx.
            // Use array.index to compute the address of the array element.

            array.index( edi, da, ebx, ecx );
            mov( edx, [edi] );
            inc( edx );

        endfor;

    endfor;
```

Another extremely useful library module is the *array.cpy* routine. This procedure will copy the data from one array to another. The calling syntax is:

```
array.cpy( sourceArrayName, destArrayName );
```

The source and destination arrays can be static or dynamic arrays. The *array.cpy* automatically adjusts and emits the proper code for each combination of parameters. With most of the array manipulation procedures in the HLA Standard Library, you pay a small performance penalty for the convenience of these library modules. Not so with *array.cpy*. This procedure is very, very fast; much faster than writing a loop to copy the data element by element.

4.12 Putting It All Together

Accessing elements of an array is a very common operation in assembly language programs. This chapter provides the basic information you need to efficiently access array elements. After mastering the material in this chapter you should know how to declare arrays in HLA and access elements of those arrays in your programs.

Records, Unions, and Name Spaces Chapter Five

5.1 Chapter Overview

This chapter discusses how to declare and use record (structures), unions, and name spaces in your programs. After strings and arrays, records are among the most commonly used composite data types; indeed, records are the mechanism you use to create user-defined composite data types. Many assembly language programmers never bother to learn how to use records in assembly language, yet would never consider not using them in high level language programs. This is somewhat inconsistent since records (structures) are just as useful in assembly language programs as in high level language programs. Given that you use records in assembly language (and especially HLA) in a manner quite similar to high level languages, there really is no reason for excluding this important tool from your programmer's tool chest. Although you'll use unions and name spaces far less often than records, their presence in the HLA language is crucial for many advanced applications. This brief chapter provides all the information you need to successfully use records, unions, and name spaces within your HLA programs.

5.2 Records

Another major composite data structure is the Pascal *record* or C/C++ *structure*[1]. The Pascal terminology is probably better, since it tends to avoid confusion with the more general term *data structure*. Since HLA uses the term "record" we'll adopt that term here.

Whereas an array is homogeneous, whose elements are all the same, the elements in a record can be of any type. Arrays let you select a particular element via an integer index. With records, you must select an element (known as a *field*) by name.

The whole purpose of a record is to let you encapsulate different, but logically related, data into a single package. The Pascal record declaration for a student is probably the most typical example:

```
student =
    record
        Name: string [64];
        Major: integer;
        SSN:   string[11];
        Midterm1: integer;
        Midterm2: integer;
        Final: integer;
        Homework: integer;
        Projects: integer;
    end;
```

Most Pascal compilers allocate each field in a record to contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name[2], the next two bytes hold the major code, the next 12 the Social Security Number, etc.

In HLA, you can also create structure types using the RECORD/ENDRECORD declaration. You would encode the above record in HLA as follows:

```
type
    student:   record
                   Name: char[65];
                   Major: int16;
                   SSN:   char[12];
```

1. It also goes by some other names in other languages, but most people recognize at least one of these names.
2. Strings require an extra byte, in addition to all the characters in the string, to encode the length.

```
            Midterm1: int16;
            Midterm2: int16;
            Final: int16;
            Homework: int16;
            Projects: int16;
        endrecord;
```

As you can see, the HLA declaration is very similar to the Pascal declaration. Note that, to be true to the Pascal declaration, this example uses character arrays rather than strings for the *Name* and *SSN* (U.S Social Security Number) fields. In a real HLA record declaration you'd probably use a string type for at least the name (keeping in mind that a string variable is only a four byte pointer).

The field names within the record must be unique. That is, the same name may not appear two or more times in the same record. However, all field names are local to that record. Therefore, you may reuse those field names elsewhere in the program.

The RECORD/ENDRECORD type declaration may appear in a variable declaration section (e.g., STATIC or VAR) or in a TYPE declaration section. In the previous example the *Student* declaration appears in the TYPE section, so this does not actually allocate any storage for a *Student* variable. Instead, you have to explicitly declare a variable of type *Student*. The following example demonstrates how to do this:

```
var
    John: Student;
```

This allocates 81 bytes of storage laid out in memory as shown in Figure 5.1.

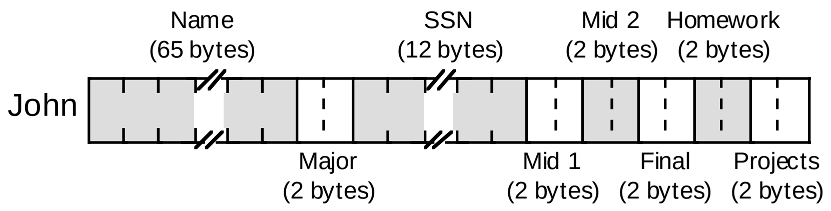

Figure 5.1 Student Data Structure Storage in Memory

If the label *John* corresponds to the *base address* of this record, then the *Name* field is at offset *John+0*, the *Major* field is at offset *John+65*, the *SSN* field is at offset *John+67*, etc.

To access an element of a structure you need to know the offset from the beginning of the structure to the desired field. For example, the *Major* field in the variable *John* is at offset 65 from the base address of *John*. Therefore, you could store the value in AX into this field using the instruction

```
mov( ax, (type word John[65]) );
```

Unfortunately, memorizing all the offsets to fields in a record defeats the whole purpose of using them in the first place. After all, if you've got to deal with these numeric offsets why not just use an array of bytes instead of a record?

Well, as it turns out, HLA lets you refer to field names in a record using the same mechanism C/C++ and Pascal use: the dot operator. To store AX into the *Major* field, you could use "mov(ax, John.Major);" instead of the previous instruction. This is much more readable and certainly easier to use.

Note that the use of the dot operator does *not* introduce a new addressing mode. The instruction "mov(ax, John.Major);" still uses the displacement only addressing mode. HLA simply adds the base address of *John* with the offset to the *Major* field (65) to get the actual displacement to encode into the instruction.

Like any type declaration, HLA requires all record type declarations to appear in the program before you use them. However, you don't have to define all records in the TYPE section to create record variables. You can use the RECORD/ENDRECORD declaration directly in a variable declaration section. This is convenient if you have only one instance of a given record object in your program. The following example demonstrates this:

```
storage

    OriginPoint:    record
                        x: uns8;
                        y: uns8;
                        z: uns8;
                    endrecord;
```

5.3 Record Constants

HLA lets you define record constants. In fact, HLA supports both symbolic record constants and literal record constants. Record constants are useful as initializers for static record variables. They are also quite useful as compile-time data structures when using the HLA compile-time language (see the chapters on Macros and the HLA Compile-Time Language). This section discusses how to create record constants.

A record literal constant takes the following form:

```
RecordTypeName:[ List_of_comma_separated_constants ]
```

The *RecordTypeName* is the name of a record data type you've defined in an HLA TYPE section prior to this point. To create a record constant you must have previously defined the record type in a TYPE section of your program.

The constant list appearing between the brackets are the data items for each of the fields in the specified record. The first item in the list corresponds to the first field of the record, the second item in the list corresponds to the second field, etc. The data types of each of the constants appearing in this list must match their respective field types. The following example demonstrates how to use a literal record constant to initialize a record variable:

```
type
    point: record
                x:int32;
                y:int32;
                z:int32;
            endrecord;

static
    Vector: point := point:[ 1, -2, 3 ];
```

This declaration initializes *Vector.x* with 1, *Vector.y* with -2, and *Vector.z* with 3.

You can also create symbolic record constants by declaring record objects in the CONST or VAL sections of your program. You access fields of these symbolic record constants just as you would access the field of a record variable, using the dot operator. Since the object is a constant, you can specify the field of a record constant anywhere a constant of that field's type is legal. You can also employ symbolic record constants as record variable initializers. The following example demonstrates this:

```
type
    point: record
                x:int32;
                y:int32;
                z:int32;
            endrecord;

const
    PointInSpace: point := point:[ 1, 2, 3 ];

static
    Vector: point := PointInSpace;
    XCoord: int32 := PointInSpace.x;
        .
        .
        .
    stdout.put( "Y Coordinate is ", PointInSpace.y, nl );
        .
        .
        .
```

5.4 Arrays of Records

It is a perfectly reasonable operation to create an array of records. To do so, you simply create a record type and then use the standard array declaration syntax when declaring an array of that record type. The following example demonstrates how you could do this:

```
type
    recElement:
        record
            << fields for this record >>
        endrecord;
        .
        .
        .
static
    recArray: recElement[4];
```

To access an element of this array you use the standard array indexing techniques found in the chapter on arrays. Since *recArray* is a single dimension array, you'd compute the address of an element of this array using the formula "baseAddress + index*@size(recElement)." For example, to access an element of *recArray* you'd use code like the following:

```
// Access element i of recArray:

    intmul( @size( recElement ), i, ebx );  // ebx := i*@size( recElement )
    mov( recArray.someField[ebx], eax );
```

Note that the index specification follows the entire variable name; remember, this is assembly not a high level language (in a high level language you'd probably use "recArray[i].someField").

Naturally, you can create multidimensional arrays of records as well. You would use the standard row or column major order functions to compute the address of an element within such records. The only thing that really changes (from the discussion of arrays) is that the size of each element is the size of the record object.

```
static
    rec2D: recElement[ 4, 6 ];
        .
        .
        .
    // Access element [i,j] of rec2D and load "someField" into EAX:

    intmul( 6, i, ebx );
    add( j, ebx );
    intmul( @size( recElement ), ebx );
    mov( rec2D.someField[ ebx ], eax );
```

5.5 Arrays/Records as Record Fields

Records may contain other records or arrays as fields. Consider the following definition:

```
type
   Pixel:
      record
         Pt:        point;
         color:     dword;
      endrecord;
```

The definition above defines a single point with a 32 bit color component. When initializing an object of type Pixel, the first initializer corresponds to the *Pt* field, *not the x-coordinate field*. **The following definition is incorrect:**

```
static
   ThisPt: Pixel := Pixel:[ 5, 10 ];   // Syntactically incorrect!
```

The value of the first field ("5") is not an object of type *point*. Therefore, the assembler generates an error when encountering this statement. HLA will allow you to initialize the fields of *Pixel* using declarations like the following:

```
static
   ThisPt: Pixel := Pixel:[ point:[ 1, 2, 3 ], 10 ];
   ThatPt: Pixel := Pixel:[ point:[ 0, 0, 0 ], 5 ];
```

Accessing *Pixel* fields is very easy. Like a high level language you use a single period to reference the *Pt* field and a second period to access the *x*, *y*, and *z* fields of *point*:

```
      stdout.put( "ThisPt.Pt.x = ", ThisPt.Pt.x, nl );
      stdout.put( "ThisPt.Pt.y = ", ThisPt.Pt.y, nl );
      stdout.put( "ThisPt.Pt.z = ", ThisPt.Pt.z, nl );
       .
       .
       .
   mov( eax, ThisPt.Color );
```

You can also declare *arrays* as record fields. The following record creates a data type capable of representing an object with eight points (e.g., a cube):

```
type
   Object8:
      record
         Pts:       point[8];
         Color:     dword;
      endrecord;
```

This record allocates storage for eight different points. Accessing an element of the *Pts* array requires that you know the size of an object of type *point* (remember, you must multiply the index into the array by the size of one element, 12 in this particular case). Suppose, for example, that you have a variable *CUBE* of type *Object8*. You could access elements of the *Pts* array as follows:

```
// CUBE.Pts[i].x := 0;

      mov( i, ebx );
      intmul( 12, ebx );
      mov( 0, CUBE.Pts.x[ebx] );
```

The one unfortunate aspect of all this is that you must know the size of each element of the *Pts* array. Fortunately, HLA provides a built-in function that will compute the size of an array element (in bytes) for you: the *@size* function. You can rewrite the code above using *@size* as follows:

```
// CUBE.Pts[i].x := 0;

      mov( i, ebx );
```

```
        intmul( @size( point ), ebx );
        mov( 0, CUBE.Pts.x[ebx] );
```

This solution is much better than multiplying by the literal constant 12. Not only does HLA figure out the size for you (so you don't have to), it automatically substitutes the correct size if you ever change the definition of the *point* record in your program. For this reason, you should always use the @size function to compute the size of array element objects in your programs.

Note in this example that the index specification ("[ebx]") follows the whole object name even though the array is *Pts*, not *x*. Remember, the "[ebx]" specification is an indexed addressing mode, not an array index. Indexes always follow the entire name, you do not attach them to the array component as you would in a high level language like C/C++ or Pascal. This produces the correct result because addition is commutative, and the dot operator (as well as the index operator) corresponds to addition. In particular, the expression "CUBE.Pts.x[ebx]" tells HLA to compute the sum of *CUBE* (the base address of the object) plus the offset to the *Pts* field, plus the offset to the *x* field plus the value of EBX. Technically, we're really computing offset(*CUBE*)+offset(*Pts*)+EBX+offset(*x*) but we can rearrange this since addition is commutative.

You can also define two-dimensional arrays within a record. Accessing elements of such arrays is no different than any other two-dimensional array other than the fact that you must specify the array's field name as the base address for the array. E.g.,

```
type
    RecW2DArray:
        record
            intField: int32;
            aField: int32[4,5];
                .
                .
                .
        endrecord;

static
    recVar: RecW2DArray;
        .
        .
        .
    // Access element [i,j] of the aField field using Row-major ordering:

    mov( i, ebx );
    intmul( 5, ebx );
    add( j, ebx );
    mov( recVar.aField[ ebx*4 ], eax );
        .
        .
        .
```

The code above uses the standard row-major calculation to index into a 4x5 array of double words. The only difference between this example and a stand-alone array access is the fact that the base address is *recVar.aField*.

There are two common ways to nest record definitions. As noted earlier in this section, you can create a record type in a TYPE section and then use that type name as the data type of some field within a record (e.g., the *Pt:point* field in the *Pixel* data type above). It is also possible to declare a record directly within another record without creating a separate data type for that record; the following example demonstrates this:

```
type
    NestedRecs:
        record
            iField: int32;
            sField: string;
            rField:
```

```
            record
                i:int32;
                u:uns32;
            endrecord;
        cField:char;
    endrecord;
```

Generally, it's a better idea to create a separate type rather than embed records directly in other records, but nesting them is perfectly legal and a reasonable thing to do on occasion.

If you have an array of records and one of the fields of that record type is an array, you must compute the indexes into the arrays independently of one another and then use the sum of these indexes as the ultimate index. The following example demonstrates how to do this:

```
type
    recType:
        record
            arrayField: dword[4,5];
            << Other Fields >>
        endrecord;

static
    aryOfRecs: recType[3,3];
        .
        .
        .
    // Access aryOfRecs[i,j].arrayField[k,l]:

    intmul( 5, i, ebx );               // Computes index into aryOfRecs
    add( j, ebx );                     //  as (i*5 +j)*@size( recType ).
    intmul( @size( recType ), ebx );

    intmul( 3, k, eax );               // Computes index into aryOfRecs
    add( l, eax );                     //  as (k*3 + j) (*4 handled later).

    mov( aryOfRecs.arrayField[ ebx + eax*4 ], eax );
```

Note the use of the base plus scaled indexed addressing mode to simplify this operation.

5.6 Controlling Field Offsets Within a Record

By default, whenever you create a record, HLA automatically assigns the offset zero to the first field of that record. This corresponds to records in a high level language and is the intuitive default condition. In some instances, however, you may want to assign a different starting offset to the first field of the record. HLA provides a mechanism that lets you set the starting offset of the first field in the record.

The syntax to set the first offset is

```
name:
    record := startingOffset;
        << Record Field Declarations >>
    endrecord;
```

Using the syntax above, the first field will have the starting offset specified by the *startingOffset int32* constant expression. Since this is an *int32* value, the starting offset value can be positive, zero, or negative.

One circumstance where this feature is invaluable is when you have a record whose base address is actually somewhere within the data structure. The classic example is an HLA string. An HLA string uses a record declaration similar to the following:

```
    record
```

```
        MaxStrLen: dword;
        length: dword;
        charData: char[xxxx];
    endrecord;
```

As you're well aware by now, HLA string pointers do not contain the address of the *MaxStrLen* field; they point at the *charData* field. The str.strRec record type found in the HLA Standard Library Strings module uses a record declaration similar to the following:

```
type
    strRec:
        record := -8;
            MaxStrLen: dword;
            length: dword;
            charData: char;
        endrecord;
```

The starting offset for the *MaxStrLen* field is -8. Therefore, the offset for the *length* field is -4 (four bytes later) and the offset for the *charData* field is zero. Therefore, if EBX points at some string data, then "(type str.strRec [ebx]).length" is equivalent to "[ebx-4]" since the *length* field has an offset of -4.

Generally, you will not use HLA's ability to specify the starting field offset when creating your own record types. Instead, this feature finds most of its use when you are mapping an HLA data type over the top of some other predefined data type in memory (strings are a good example, but there are many other examples as well).

5.7 Aligning Fields Within a Record

To achieve maximum performance in your programs, or to ensure that HLA's records properly map to records or structures in some high level language, you will often need to be able to control the alignment of fields within a record. For example, you might want to ensure that a *dword* field's offset is an even multiple of four. You use the ALIGN directive to do this, the same way you would use ALIGN in the STATIC declaration section of your program. The following example shows how to align some fields on important boundaries:

```
type
    PaddedRecord:
        record
            c:char;
            align(4);
            d:dword;
            b:boolean;
            align(2);
            w:word;
        endrecord;
```

Whenever HLA encounters the ALIGN directive within a record declaration, it automatically adjusts the following field's offset so that it is an even multiple of the value the ALIGN directive specifies. It accomplishes this by increasing the offset of that field, if necessary. In the example above, the fields would have the following offsets: *c*:0, *d*:4, *b*:8, *w*:10. Note that HLA inserts three bytes of padding between *c* and *d* and it inserts one byte of padding between *b* and *w*. It goes without saying that you should never assume that this padding is present. If you want to use those extra bytes, then declare fields for them.

Note that specifying alignment within a record declaration does not guarantee that the field will be aligned on that boundary in memory; it only ensures that the field's offset is aligned on the specified boundary. If a variable of type *PaddedRecord* starts at an odd address in memory, then the *d* field will also start at an odd address (since any odd address plus four is an odd address). If you want to ensure that the fields are aligned on appropriate boundaries in memory, you must also use the ALIGN directive before variable declarations of that record type, e.g.,

```
static
        .
        .
        .
    align(4);
    PRvar: PaddedRecord;
```

The value of the ALIGN operand should be an even value that is evenly divisible by the largest ALIGN expression within the record type (four is the largest value in this case, and it's already evenly divisible by two).

If you want to ensure that the record's size is a multiple of some value, then simply stick an ALIGN directive as the last item in the record declaration. HLA will emit an appropriate number of bytes of padding at the end of the record to fill it in to the appropriate size. The following example demonstrates how to ensure that the record's size is a multiple of four bytes:

```
type
    PaddedRec:
        record
            << some field declarations >>

            align(4);
        endrecord;
```

5.8 Pointers to Records

During execution, your program may refer to structure objects directly or indirectly using a pointer. When you use a pointer to access fields of a structure, you must load one of the 80x86's 32-bit registers with the address of the desired record. Suppose you have the following variable declarations (assuming the *Object8* structure from "Arrays/Records as Record Fields" on page 487):

```
static
    Cube:       Object8;
    CubePtr:    pointer to Object8 := &Cube;
```

CubePtr contains the address of (i.e., it is a pointer to) the *Cube* object. To access the *Color* field of the *Cube* object, you could use an instruction like "mov(Cube.Color, eax);". When accessing a field via a pointer you need to load the address of the object into a 32-bit register such as EBX. The instruction "mov(CubePtr EBX);" will do the trick. After doing so, you can access fields of the *Cube* object using the [EBX+offset] addressing mode. The only problem is "How do you specify which field to access?" Consider briefly, the following *incorrect* code:

```
        mov( CubePtr, ebx );
        mov( [ebx].Color, eax );      // This does not work!
```

There is one major problem with the code above. Since field names are local to a structure and it's possible to reuse a field name in two or more structures, how does HLA determine which offset *Color* represents? When accessing structure members directly (.e.g., "mov(Cube.Color, EAX);") there is no ambiguity since *Cube* has a specific type that the assembler can check. "[EBX]", on the other hand, can point at *anything*. In particular, it can point at any structure that contains a *Color* field. So the assembler cannot, on its own, decide which offset to use for the *Color* symbol.

HLA resolves this ambiguity by requiring that you explicitly supply a type. To do this, you must coerce "[EBX]" to type *Cube*. Once you do this, you can use the normal dot operator notation to access the *Color* field:

```
        mov( CubePtr, ebx );
        mov( (type Cube [ebx]).Color, eax );
```

By specifying the record name, HLA knows which offset value to use for the *Color* symbol.

If you have a pointer to a record and one of that record's fields is an array, the easiest way to access elements of that field is by using the base plus indexed addressing mode. To do so, you just load the pointer to the record into one register and compute the index into the array in a second register. Then you combine these two registers in the address expression. In the example above, the *Pts* field is an array of eight *point* objects. To access field *x* of the *i*th element of the *Cube.Pts* field, you'd use code like the following:

```
        mov( CubePtr, ebx );
        intmul( @size( point ), i, esi );   // Compute index into point array.
        mov( (type Object8 [ebx]).Pts.x[ esi*4 ], eax );
```

As usual, the index appears after all the field names.

If you use a pointer to a particular record type frequently in your program, typing a coercion operator like "(type Object8 [ebx])" can get old pretty quick. One way to reduce the typing needed to coerce EBX is to use a TEXT constant. For example, consider the following statement in a program:

```
const
    O8ptr: text := "(type Object8 [ebx])";
```

With this statement at the beginning of your program you can use *O8ptr* in place of the type coercion operator and HLA will automatically substitute the appropriate text. With a text constant like the above, the former example becomes a little more readable and writable:

```
        mov( CubePtr, ebx );
        intmul( @size( point ), i, esi );   // Compute index into point array.
        mov( O8Ptr.Pts.x[ esi*4 ], eax );
```

5.9 Unions

A record definition assigns different offsets to each field in the record according to the size of those fields. This behavior is quite similar to the allocation of memory offsets in a VAR or STATIC section. HLA provides a second type of structure declaration, the UNION, that does not assign different addresses to each object; instead, each field in a UNION declaration has the same offset – zero. The following example demonstrates the syntax for a UNION declaration:

```
type
    unionType:
        union
            << fields (syntactically identical to record declarations) >>
        endunion;
```

You access the fields of a UNION exactly the same way you access the fields of a record: using dot notation and field names. The following is a concrete example of a UNION type declaration and a variable of the UNION type:

```
type
    numeric:
        union
            i: int32;
            u: uns32;
            r: real64;
        endunion;
            .
            .
            .
static
    number: numeric;
            .
            .
```

```
              .
    mov( 55, number.u );
              .
              .
              .
    mov( -5, number.i );
              .
              .
              .
    stdout.put( “Real value = “, number.r, nl );
```

The important thing to note about UNION objects is that all the fields of a UNION have the same offset in the structure. In the example above, the *number.u*, *number.i*, and *number.r* fields all have the same offset: zero. Therefore, the fields of a UNION overlap one another in memory; this is very similar to the way the 80x86 eight, sixteen, and thirty-two bit registers overlap one another. Usually, access to the fields of a UNION are mutually exclusive; that is, you do not manipulate separate fields of a particular UNION variable concurrently because writing to one field overwrite's the other fields. In the example above, any modification of number.u would also change *number.i* and *number.r*.

Programmers typically use UNIONs for two different reasons: to conserve memory or to create aliases. Memory conservation is the intended use of this data structure facility. To see how this works, let's compare the *numeric* UNION above with a corresponding record type:

```
type
    numericRec:
        record
            i: int32;
            u: uns32;
            r: real64;
        endrecord;
```

If you declare a variable, say *n*, of type *numericRec*, you access the fields as *n.i*, *n,u*, and *n.r*; exactly as though you had declared the variable to be type *numeric*. The difference between the two is that *numericRec* variables allocate separate storage for each field of the record while numeric objects allocate the same storage for all fields. Therefore, *@size(numericRec)* is 16 since the record contains two double word fields and a quad word (*real64*) field. *@size(numeric)*, however, is eight. This is because all the fields of a UNION occupy the same memory locations and the size of a UNION object is the size of the largest field of that object (see Figure 5.2).

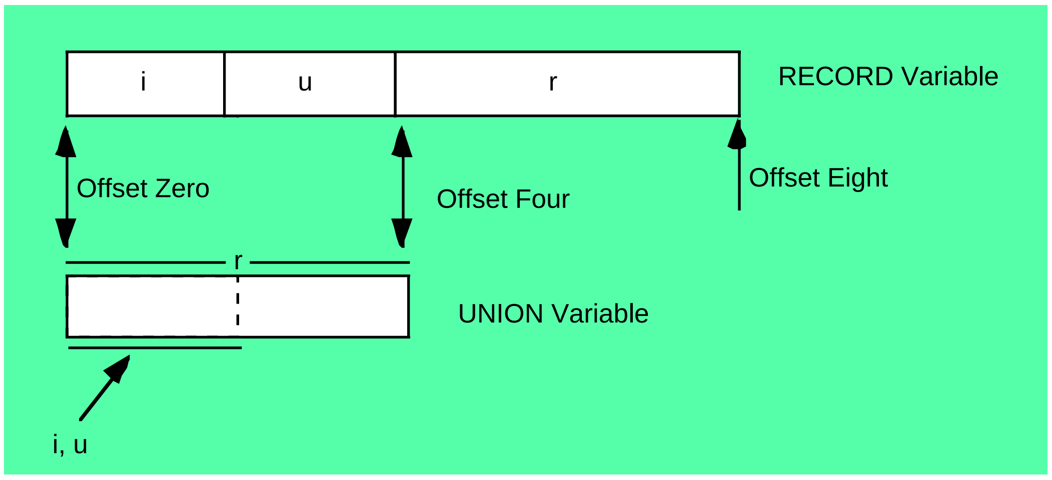

Figure 5.2 Layout of a UNION versus a RECORD Variable

In addition to conserving memory, programmers often use UNIONs to create aliases in their code. As you may recall, an alias is a different name for the same memory object. Aliases are often a source of confusion in a program so you should use them sparingly; sometimes, however, using an alias can be quite convenient. For example, in some section of your program you might need to constantly use type coercion to refer to an object using a different type. Although you can use an HLA TEXT constant to simplify this process, another way to do this is to use a UNION variable with the fields representing the different types you want to use for the object. As an example, consider the following code:

```
type
    CharOrUns:
        union
            c:char;
            u:uns32;
        endrecord;

static
    v:CharOrUns;
```

With a declaration like the above, you can manipulate an *uns32* object by accessing *v.u*. If, at some point, you need to treat the L.O. byte of this *uns32* variable as a character, you can do so by simply accessing the *v.c* variable, e.g.,

```
    mov( eax, v.u );
    stdout.put( "v, as a character, is '", v.c, "'" nl );
```

You can use UNIONs exactly the same way you use RECORDs in an HLA program. In particular, UNION declarations may appear as fields in RECORDs, RECORD declarations may appear as fields in UNIONs, array declarations may appear within UNIONs, you can create arrays of UNIONs, etc.

5.10 Anonymous Unions

Within a RECORD declaration you can place a UNION declaration without specifying a fieldname for the union object. The following example demonstrates the syntax for this:

```
type
    HasAnonUnion:
        record
```

```
            r:real64;
            union
                u:uns32;
                i:int32;
            endunion;
            s:string;
        endrecord;

static
    v: HasAnonUnion;
```

Whenever an anonymous union appears within an RECORD you can access the fields of the UNION as though they were direct fields of the RECORD. In the example above, for example, you would access *v's u* and *i* fields using the syntax "v.u" and "v.i", respectively. The *u* and *i* fields have the same offset in the record (eight, since they follow a *real64* object). The fields of *v* have the following offsets from *v's* base address:

```
    v.r     0
    v.u     8
    v.i     8
    v.s    12
```

@size(v) is 16 since the *u* and *i* fields only consume four bytes between them.

Warning: HLA gets confused if you attempt to create a record constant when that record has anonymous unions (HLA doesn't allow UNION constants). So don't create record constants of a record if that record contains anonymous unions as fields.

HLA also allows anonymous records within unions. Please see the HLA documentation for more details, though the syntax and usage is identical to anonymous unions within records.

5.11 Variant Types

One big use of UNIONs in programs is to create *variant* types. A variant variable can change its type dynamically while the program is running. A variant object can be an integer at one point in the program, switch to a string at a different part of the program, and then change to a real value at a later time. Many very high level language systems use a dynamic type system (i.e., variant objects) to reduce the overall complexity of the program; indeed, proponents of many very high level languages insist that the use of a dynamic typing system is one of the reasons you can write complex programs in so few lines. Of course, if you can create variant objects in a very high level language, you can certainly do it in assembly language. In this section we'll look at how we can use the UNION structure to create variant types.

At any one given instant during program execution a variant object has a specific type, but under program control the variable can switch to a different type. Therefore, when the program processes a variant object it must use an IF statement or SWITCH statement to execute a different sequence of instructions based on the object's current type. Very high level languages (VHLLs) do this transparently. In assembly language you will have to provide the code to test the type yourself. To achieve this, the variant type needs some additional information beyond the object's value. Specifically, the variant object needs a field that specifies the current type of the object. This field (often known as the *tag* field) is a small enumerated type or integer that specifies the type of the object at any given instant. The following code demonstrates how to create a variant type:

```
type
    VariantType:
        record
            tag:uns32;  // 0-uns32, 1-int32, 2-real64
            union
                u:uns32;
                i:int32;
                r:real64;
            endunion;
        endrecord;

static
    v:VariantType;
```

The program would test the *v.tag* field to determine the current type of the *v* object. Based on this test, the program would manipulate the v.*i*, *v.u*, or *v.r* field.

Of course, when operating on variant objects, the program's code must constantly be testing the tag field and executing a separate sequence of instructions for uns32, int32, or real64 values. If you use the variant fields often, it makes a lot of since to write procedures to handle these operations for you (e.g., *vadd*, *vsub*, *vmul*, and *vdiv*). Better yet, you might want to make a class out of your variant types. For details on this, see the chapter on Classes appearing later in this text.

5.12 Namespaces

One really nice feature of RECORDs and UNIONs is that the field names are local to a given RECORD or UNION declaration. That is, you can reuse field names in different RECORDs or UNIONs. This is an important feature of HLA because it helps avoid *name space pollution*. Name space pollution occurs when you use up all the "good" names within the current scope and you have to start creating non-descriptive names for some object because you've already used the most appropriate name for something else. Because you can reuse names in different RECORD/UNION definitions (and you can even reuse those names outside of the RECORD/UNION definitions) you don't have to dream up new names for the objects that have less meaning. We use the term *namespace* to describe how HLA associates names with a particular object. The field names of a RECORD have a namespace that is limited to objects of that record type. HLA provides a generalization of this namespace mechanism that lets you create arbitrary namespaces. These namespace objects let you shield the names of constants, types, variables, and other objects so their names do not interfere with other declarations in your program.

An HLA NAMESPACE section encapsulates a set of generic declarations in much the same way that a RECORD encapsulates a set of variable declarations. A NAMESPACE declaration takes the following form:

```
namespace name;

    << declarations >>

end name;
```

The *name* identifier provides the name for the NAMESPACE. The identifier after the END clause must exactly match the identifier after NAMESPACE. You may have several NAMESPACE declarations within a program as long as the identifiers for the name spaces are all unique. Note that a NAMESPACE declaration section is a section unto itself. It does not have to appear in a TYPE or VAR section. A NAMESPACE may appear anywhere one of the HLA declaration sections is legal. A program may contain any number of NAMESPACE declarations; in fact, the name space identifiers don't even have to be unique as you will soon see.

The declarations that appear between the NAMESPACE and END clauses are all the standard HLA declaration sections except that you cannot nest name space declarations. You may, however, put CONST, VAL, TYPE, STATIC, READONLY, STORAGE, and VAR sections within a namespace[3]. The following code provides an example of a typical NAMESPACE declaration in an HLA program:

```
namespace myNames;
```

3. Procedure and macro declarations, the subjects of later chapters, are also legal within a name space declaration section.

```
    type
        integer: int32;

    static
        i:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

To access the fields of a name space you use the same dot notation that records and unions use. For example, to access the fields of *myNames* outside of the name space you'd use the following identifiers:

```
myNames.integer - A type declaration equivalent to int32.
myNames.i - An integer variable (int32).
myNames.j - An uns32 variable.
myNames.pi - A real64 constant.
```

This example also demonstrates an important point about NAMESPACE declarations: within a name space you may reference other identifiers in that same NAMESPACE declaration without using the dot notation. For example, the *i* field above uses type *integer* from the *myNames* name space without the "mynames." prefix.

What is not obvious from the example above is that NAMESPACE declarations create a clean symbol table whenever you open up a declaration. The only external symbols that HLA recognizes in a NAMESPACE declaration are the predefined type identifiers (e.g., int32, uns32, and char). HLA does not recognize any symbols you've declared outside the NAMESPACE while it is processing your namespace declaration. This creates a problem if you want to use symbols outside the NAMESPACE when declaring other symbols inside the NAMESPACE. For example, suppose the type *integer* had been defined outside *myNames* as follows:

```
type
    integer: int32;

namespace myNames;


    static
        i:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

If you were to attempt to compile this code, HLA would complain that the symbol *integer* is undefined. Clearly *integer* is defined in this program, but HLA hides all external symbols when creating a name space so that you can reuse (and redefine) those symbols within the name space. Of course, this doesn't help much if you actually want to use a name that you've defined outside *myNames* within that name space. HLA provides a solution to this problem: the *@global:* operator. If, within a name space declaration section, you prefix a name with "@global:" then HLA will use the global definition of that name rather than the local definition (if a local definition even exists). To correct the problem in the previous example, you'd use the following code:

```
type
    integer: int32;
```

```
namespace myNames;


    static
        i:@global:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

With the *@global:* prefix, the *i* variable will be type *int32* even if a different declaration of integer appears within the *myNames* name space.

You cannot nest NAMESPACE declarations[4]. However, you can have multiple NAMESPACE declarations in the same program that use the same name space identifier, e.g.,

```
namespace ns;

    << declaration group #1 >>

end ns;
    .
    .
    .
namespace ns;

    << declaration group #2 >>

end ns;
```

When HLA encounters a second NAMESPACE declaration for a given identifier, it simply appends the declarations in the second group to the end of the symbol list it created for the first group. Therefore, after processing the two NAMESPACE declarations, the *ns* name space would contain the set of all symbols you've declared in both the name space blocks.

Perhaps the most common use of name spaces is in library modules. If you create a set of library routines to use in various projects or distribute to others, you have to be careful about the names you choose for your functions and other objects. If you use common names like *get* and *put*, the users of your module will complain when your names collide with theirs. An easily solution is to put all your code in a NAMESPACE block. Then the only name you have to worry about is the name of the NAMESPACE itself. This is the only name that will collide with other users' code. That can happen, but it's much less likely to happen than if you don't use a name space and your library module introduces dozens, if not hundreds, of new names into the global name space[5]. The HLA Standard Library provides many good examples of name spaces in use. The HLA Standard Library defines several name spaces like *stdout*, *stdin*, *str*, *cs*, and *chars*. You refer to functions in these name spaces using names like *stdout.put*, *stdin.get*, *cs.intersection*, *str.eq*, and *chars.toUpper*. The use of name spaces in the HLA Standard Library prevents conflicts with similar names in your own programs.

5.13 Putting It All Together

One of the more amazing facts about programmer psychology is the fact that a high level language programmer would refuse to use a high level language that doesn't support records or structures; then that same programmer won't bother to learn how to use them in assembly language (all the time, grumbling about their

4. There really doesn't seem to be a need to do this; hence its omission from HLA.
5. The global name space is the global section of your program.

absence). You use records in assembly language for the same reason you use them in high level languages. Given that most programmers consider records and structure essential in high level languages, it is surprising they aren't as concerned about using them in assembly language.

This short chapter demonstrates that it doesn't take much effort to master the concept of records in an assembly language program. Taken together with UNIONs and NAMESPACEs, RECORDs can help you write HLA programs that are far more readable and easier to understand. Therefore, you should use these language features as appropriate when writing assembly code.

Dates and Times Chapter Six

6.1 Chapter Overview

This chapter discusses dates and times as a data type. In particular, this chapter discusses the data/time data structures the HLA Standard Library defines and it also discusses date arithmetic and other operations on dates and times.

6.2 Dates

For the first 50 years, or so, of the computer's existence, programmers did not give much thought to date calculations. They either used a date/time package provided with their programming language, or they kludged together their own date processing libraries. It wasn't until the Y2K[1] problem came along that programmers began to give dates serious consideration in their programs. The purpose of this chapter is two-fold. First, this chapter teaches that date manipulation is not as trivial as most people would like to believe – it takes a lot of work to properly compute various date functions. Second, this chapter presents the HLA date and time formats found in the "datetime.hhf" library module. Hopefully this chapter will convince you that considerable thought has gone into the HLA datetime.hhf module so you'll be inclined to use it rather than trying to create your own date/time formats and routines.

Although date and time calculations may seem like they should be trivial, they are, in fact, quite complex. Just remember the Y2K problem to get a good idea of the kinds of problems your programs may create if they don't calculate date and time values correctly. Fortunately, you don't have to deal with the complexities of date and time calculations, the HLA Standard Library does the hard stuff for you.

The HLA Standard Library date routines produce valid results for dates between January 1, 1583 and December 31, 9999[2]. HLA represents dates using the following record definition (in the *date* namespace):

```
type
    daterec:
        record
            day:uns8;
            month:uns8;
            year:uns16;
        endrecord;
```

This format (*date.daterec*) compactly represents all legal dates using only four bytes. Note that this is the same date format that the chapter on Data Representation presents for the extended data format (see "Bit Fields and Packed Data" on page 81). You should use the *date.daterec* data type when declaring date objects in your HLA programs, e.g.,

```
static
    TodaysDate: date.daterec;
    Century21:  date.daterec := date.daterec:[ 1, 1, 2001 ]; // note: d, m ,y
```

As the second example above demonstrates, the first field is the day field and the second field is the month field if you use a *date.daterec* constant to initialize a static *date.daterec* object. Don't fall into the trap of using the mm/dd/yy or yy/mm/dd organization common in most countries.

1. For those who missed it, the Y2K (or Year 2000) problem occurred when programmers used two digits for the date and assumed that the H.O. two digits were "19". Clearly this code malfunctioned when the year 2000 came along.
2. The Gregorial Calendar came into existence in Oct, 1582, so any dates earlier than this are meaningless as far as date calculations are concerned. The last legal date, 9999, was chosen arbitrarily as a trap for wild dates entering the calculation. This means, of course, that code calling the HLA Standard Library Date/Time package will suffer from the Y10K problem. However, you'll probably not consider this a severe limitation!

The HLA *date.daterec* format has a couple of advantages. First, it is a trivial matter to convert between the internal and external representations of a date. All you have to do is extract the *d*, *m*, and *y* fields and manipulate them as integers of the appropriate sizes. Second, this format makes it very easy to compare two dates to see if one date follows another in time; all you've got to do is compare the *date.daterec* object as though it were a 32-bit unsigned integer and you'll get the correct result. The Standard Library *data.daterec* format does have a few disadvantages. Specifically, certain calculations like computing the number of days between two dates is a bit difficult. Fortunately, the HLA Standard Library Date module provides most of the functions you'll ever need for date calculations, so this won't prove to be much of a disadvantage.

A second disadvantage to the *date.daterec* format is that the resolution is only one day. Some calculations need to maintain the time of day (down to some fraction of a second) as well as the calendar date. The HLA Standard Library also provides a TIME data structure. By combining these two structures together you should be able handle any problem that comes along.

Before going on and discussing the functions available in the HLA Standard Library's Date module, it's probably worthwhile to briefly discuss some other date formats that find common use. Perhaps the most common date format is to use an integer value that specifies the number of days since an *epoch*, or starting, date. The advantage to this scheme is that it's very easy to do certain kinds of date arithmetic (e.g., to compute the number of days between two dates you simply subtract them) and it's also very easy to compare these dates. The disadvantages to this scheme include the fact that it is difficult to convert between the internal representation and an external representation like "xx/yy/zzzz." Another problem with this scheme, which it shares with the HLA scheme, is that the granularity is one day. You cannot represent time with any more precision than one day.

Another popular format combines dates and times into the same value. For example, the representation of time on most UNIX systems measures the number of seconds that have passed since Jan 1, 1970. Unfortunately, many UNIX systems only use a 32-bit signed integer; therefore, those UNIX systems will experience their own "Y2.038K" problem in the year 2038 when these signed integers roll over from 2,147,483,637 seconds to -2,147,483,638 seconds. Although this format does maintain time down to seconds, it does not handle fractions of a second very well. Most UNIX system include an extra field in their date/time format to handle milliseconds, but this extra field is a kludge. One could just as easily add a time field to an existing date format if you're willing to kludge.

For those who want to be able to accurately measure dates and times, a good solution is to use a 64-bit unsigned integer to count the number of microseconds since some epoch data. A 64-bit unsigned integer will provide microsecond accuracy for a little better than 278,000 years. Probably sufficient for most needs. If you need better than microsecond accuracy, you can get nanosecond accuracy that is good for about 275 years (beyond the epoch date) with a 64-bit integer. Of course, if you want to use such a date/time format, you will have to write the routines that manipulate such dates yourself; the HLA Standard Library's Date/Time module doesn't use that format.

6.3 A Brief History of the Calendar

Man has been interested in keeping track of time since the time man became interested in keeping track of history. To understand why we need to perform various calculations, you'll need to know a little bit about the history of the calendar. So this section will digress a bit from computers and discuss that history.

What exactly is time? Time is a concept that we are all intuitively familiar with, but try and state a concrete definition that does not define time in terms of itself. Before you run off and grab a dictionary, you should note that many of the definitions of time in a typical dictionary contain a circular reference (that is, they define time in terms of itself). The *American Heritage Dictionary of the English Language* provides the following definition:

> **A nonspatial continuum in which events occur in apparently irreversible succession from the past through the present to the future.**

As horrible as this definition sounds, it is one of the few that doesn't define time by how we measure it or by a sequence of observable events.

Why are we so obsessed with keeping track of time? This question is much more easily answered. We need to keep track of time so we can predict certain future events. Historically, important events the human race has needed to predict include the arrival of spring (for planting), the observance of religious anniversaries (e.g., Christmas, Passover), or the gestation period for livestock (or even humans). Of course, modern life may seem much more complex and tracking time more important, but we track time for the same reasons the human race always has, to predict the future. Today, we predict business meetings, when a department store will open to the public, the start of a college lecture, periods of high traffic on the highways, and the start of our favorite television shows by using time. The better we are able to measure time, the better we will be able to predict when certain types of events will occur (e.g., the start of spring so we can begin planting).

To measure time, we need some predictable, periodic, event. Since ancient times, there have been three celestial events that suit this purpose: the solar *day*, the lunar *month*, and the solar *year.* The solar day (or *tropical* day) consists of one complete rotation of the Earth on its axis. The lunar month consists of one complete set of moon phases. The solar year is one complete orbit of the Earth around the Sun. Since these periodic events are easy to measure (crudely, at least), they have become the primary basis by which we measure time.

Since these three celestial events were obvious even in prehistoric times, it should come as no surprise that one society would base their measurement of time on one cyclic standard such as the lunar month while another social group would base their time unit on a different cycle such as the solar year. Clearly, such fundamentally different time keeping schemes would complicate business transactions between the two societies effectively erecting an artificial barrier between them. Nevertheless, until about the year 46 BC (by our modern calendar), most countries used their own system for time keeping.

One major problem with reconciling the different calendars is that the celestial cycles are not integral. That is, there are not an even number of solar days in a lunar month, there are not an integral number of solar days in a solar year, and there are not an integral number of lunar months in a solar year. Indeed, there are approximately 365.2422 days in a solar year and approximately 29.5 days in a lunar month. Twelve lunar months are 354 days, a little over a week short of a full year. Therefore, it is very difficult to reconcile these three periodic events if you want to use two of them or all three of them in your calendar.

In 46 BC (or BCE, for *Before Common Era,* as it is more modernly written) Julius Caesar introduced the calendar upon which our modern calendar is based. He decreed that each year would be exactly 365 $^1/_4$ days long by having three successive years having 365 days each and every fourth year having 366 days. He also abolished reliance upon the lunar cycle from the calendar. However, 365 $^1/_4$ is just a little bit more than 365.2422, so Julius Caesar’s calendar lost a day every 128 years or so.

Around 700 AD (or CE, for *Common Era*, as it is more modernly written) it was common to use the birth of Jesus Christ as the *Epoch* year. Unfortunately, the equinox kept losing a full day every 128 years and by the year 1500 the equinoxes occurred on March 12th, and September 12th. This was of increasing concern to the Church since it was using the Calendar to predict Easter, the most important Christian holiday[3]. In 1582 CE, Pope Gregory XIII dropped ten days from the Calendar so that the equinoxes would fall on March 21st and September 21st, as before, and as advised by Christoph Clavius, he dropped three leap years every 400 years. From that point forward, century years were leap years only if divisible by 400. Hence 1700, 1800, 1900 are *not* leap years, but 2000 *is* a leap year. This new calendar is known as the Gregorian Calendar (named after Pope Gregory XIII) and with the exception of the change from BC/AD to BCE/CE is, essentially, the calendar in common use today[4].

The Gregorian Calendar wasn’t accepted universally until well into the twentieth century. Largely Roman Catholic countries (e.g., Spain and France) adopted the Gregorian Calendar the same year as Rome. Other countries followed later. For example, portions of Germany did not adopt the Gregorian Calendar until the year 1700 AD while England held out until 1750. For this reason, many of the American founding fathers have *two* birthdates listed. The first date is the date in force at the time of their birth, the second date

3. Easter is especially important since the Church computed all other holidays relative to Easter. If the date of Easter was off, then all holidays would be off.
4. One can appreciate that non-Christian cultures might be offended at by the abbreviations BC (Before Christ) and AD (Anno Domini [day of our Lord]).

is their birthdate using the Gregorian Calendar. For example, George Washington was actually born on February 11th by the English Calendar, but after England adopted the Gregorian Calendar, this date changed to February 22nd. Note that George Washington's birthday didn't actually change, only the calendar used to measure dates at the time changed.

The Gregorian Calendar still isn't correct, though the error is very small. After approximately 3323 years it will be off by a day. Although there has been some proposals thrown around to adjust for this in the year 4000, that is such a long time off that it's hardly worth contemporary concern (with any luck, mankind will be a spacefaring race by then and the concept of a year, month, or day, may be a quaint anachronism).

There is one final problem with the calendar- the length of the solar day is constantly changing. Ocean tidal forces, meteors burning up in our atmosphere, and other effects are slowing down the Earth's rotation resulting in longer days. The effect is small, but compared to the length of a day, but it amounts to a loss of one to three milliseconds (that is, about 1/500th of a second) every 100 years since the defining Epoch (Jan 1, 1900). That means that Jan 1, 2000 is about two seconds longer than Jan 1, 1900. Since there are 86,400 seconds in a day, it will probably take on the order of 100,000 years before we lose a day due to the Earth's rotation slowing down. However, those who want to measure especially small time intervals have a problem: hours and seconds have been defined as submultiples of a single day. If the length of a day is constantly changing, that means that the definition of a second is constantly changing as well. In other words, two very precise measurements of equivalent events taken 10 years apart may show measurable differences.

To solve this problem scientists have developed the *Cesium-155 Atomic Clock*, the most accurate timing device ever invented. The Cesium atom, under special conditions, vibrates at exactly 9,192,631,770 cycles per second, for the year 1900. Because the clock is so accurate, it has to be adjusted periodically (about every 500 days, currently) so that its time (known as Universal Coordinated Time or UTC) matches that of the Earth (UT1). A high-quality Cesium Clock (like the one at the National Institute of Standards and Technology in Boulder, Colorado, USA) is very large (about the size of a large truck) and can keep accurate time to about one second in a million and a half years. Commercial units (about the size of a large suitcase) are available and they keep time accurate to about one second every 5-10,000 years.

The wall calendar you purchase each year is a device that is very similar to the Cesium Atomic Clock- it lets you measure time. The Cesium clock, clearly, lets time two discrete events that are very close to one another, but either device will probably let you predict that you start two week's vacation in Mexico starting next Monday (and the wall calendar does it for a whole lot less money). Most people don't think of a calendar as a time keeping device, but the only difference between it and a watch is the *granularity*, that is, the finest amount of time one can measure with the device. With a typical electronic watch, you can probably measure (accurately) to as little as $^1/_{100}$ seconds. With a calendar, the minimum interval you can measure is one day. While the watch is appropriate for measuring the 100 meter dash, it is inappropriate for measuring the duration of the Second World War; the calendar, however, is perfect for this latter task.

Time measurement devices, be they a Cesium Clock, a wristwatch, or a Calendar, do not measure time in an absolute sense. Instead, these devices measure time between two events. For the Gregorian Calendar, the (intended) Epoch event that marks year one was the birth of Christ. Unfortunately in 1582, the use of negative numbers was not widespread and even the use of zero was not common. Therefore, 1 AD was (supposed to be) the first year of Christ's life. The year prior to that point was considered 1BC. This unfortunate choice created some mathematical problems that tend to bother people 2,000 years later. For example, the first decade was the first 10 years of Christ's life, that is, 1 AD through 10 AD. Likewise, the first century was considered the first 100 years after Christ's birth, that is, 1 AD through 100 AD. Likewise, the first millennium was the first 1,000 years after Christ's birth, specifically 1 AD through 1000 AD. Similarly, the second millennium is the next 1,000 years, specifically 1001 AD through 2000 AD. The third, millennium, contrary to popular belief, began on January 1, 2001 (Hence the title of Clark's book: "2001: A Space Odyssey"). It is an unfortunately accident of human psychology that people attach special significance to round numbers; there were many people mistakenly celebrating the turn of the millennium on December 31st, 1999 when, in fact, the actual date was still a year away.

Now you're probably wondering what this has to do with computers and the representation of dates in the computer... The reason for taking a close look at the history of the Calendar is so that you don't misuse the date and time representations found in the HLA Standard Library. In particular, note that the HLA date format is based on the Gregorian Calendar. Since the Gregorian Calendar was "born" in October of 1582, it makes absolutely no sense to represent any date earlier than about Jan 1, 1583 using the HLA date format. Granted, the data type can represent earlier dates numerically, but any date computations would be severely off if one or both of the dates in the computation are pre-1583 (remember, Pope Gregory droped 10 days from the calendar; right off the bat your "days between two dates" computation would be off by 10 real days if the two dates crossed the date that Rome adopted the Gregorian Calendar).

In fact, you should be wary of any dates prior to about January 1, 1800. Prior to this point there were a couple of different (though similar) calendars in use in various countries. Unless you're a historian and have the appropriate tables to convert between these dates, you should not use dates prior to this point in calculations. Fortunately, by the year 1800, most countries that had a calendar based on Juilus Caesar's calendar fell into line and adopted the Gregorian Calendar. Some other calendars (most notably, the Chinese Calendar) were in common use into the middle of the 20th century. However, it is unlikely you would ever confuse a Chinese date with a Gregorian date.

6.4 HLA Date Functions

HLA provides a wide array of date functions you can use to manipulate date objects. The following subsections describe many of these functions and how you use them.

6.4.1 date.IsValid and date.validate

When storing data directly into the fields of a *date.daterec* object, you must be careful to ensure that the resulting date is correct. The HLA date procedures will raise an *ex.InvalidDate* exception if the date values are out of range. The *date.IsValid* and *date.validate* procedures provide some handy code to check the validity of a date object. These two routines use either of the following calling seqeuences:

```
date.IsValid( dateVar ); // dateVar is type date.daterec
date.IsValid( m, d, y ); // m, d, y are uns8, uns8, uns16, respectively

date.validate( dateVar ); // See comments above.
date.validate( m, d, y );
```

The *date.IsValid* procedure checks the date to see if it is a valid date. This procedure returns true or false in the AL register to indicate whether the date is valid. The *date.validate* procedure also checks the validity of the date; however, it raises the *ex.InvalidDate* exception if the date is invalid. The following sample program demonstrates the use of these two routines:

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

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

    theDate:    date.daterec;


begin DateTimeDemo;

    try
```

```
            stdout.put( “Enter the month (1-12):” );
            stdin.get( m );

            stdin.flushInput();
            stdout.put( “Enter the day (1-31):” );
            stdin.get( d );

            stdin.flushInput();
            stdout.put( “Enter the year (1583-9999): “ );
            stdin.get( y );

            if( date.isValid( m, d, y )) then

                stdout.put( m, “/”, d, “/”, y, “ is a valid date.” nl );

            endif;

            // Assign the fields to a date variable.

            mov( m, al );
            mov( al, theDate.month );
            mov( d, al );
            mov( al, theDate.day );
            mov( y, ax );
            mov( ax, theDate.year );

            // Force an exception if the date is illegal.

            date.validate( theDate );

          exception( ex.ConversionError )

            stdout.put
            (
                “One of the input values contained illegal characters” nl
            );

          exception( ex.ValueOutOfRange )

            stdout.put
            (
                “One of the input values was too large” nl
            );

          exception( ex.InvalidDate )

            stdout.put
            (
                “The input date (“, m, “/”, d, “/”, y, “) was invalid” nl
            );

        endtry;


    end DateTimeDemo;
```

Program 6.1 Date Validation Example

6.4.2 Checking for Leap Years

Determining whether a given year is a leap year is somewhat complex. The exact algorithm is "any year that is evenly divisible by four and is not evenly divisible by 100 or is evenly divisible by 400 is a leap year[5]." The HLA "datetime.hhf" module provides a convenient function, *date.IsLeapYear*, that efficiently determines whether a given year is a leap year. There are two different ways you can call this function; either of the following will work:

```
date.IsLeapYear( dateVar );    // dateVar is a date.dateRec variable.
date.IsLeapYear( y );          // y is a word value.
```

The following code demonstrates the use of this routine.

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

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

    theDate:    date.daterec;


begin DemoIsLeapYear;

    try

        stdout.put( "Enter the month (1-12):" );
        stdin.get( m );

        stdin.flushInput();
        stdout.put( "Enter the day (1-31):" );
        stdin.get( d );

        stdin.flushInput();
        stdout.put( "Enter the year (1583-9999): " );
        stdin.get( y );

        // Assign the fields to a date variable.

        mov( m, al );
        mov( al, theDate.month );
        mov( d, al );
        mov( al, theDate.day );
        mov( y, ax );
        mov( ax, theDate.year );

        // Force an exception if the date is illegal.

        date.validate( theDate );
```

5. The Gregorian Calendar does not account for the fact that sometime between the years 3,000 and 4,000 we will have to add an extra leap day to keep the Calendar in sync with the Earth's rotation around the Sun. The HLA date.IsLeapYear does not handle this situation either. Keep this in mind if you are doing date calculations that involve dates after the year 3,000. This is a defect in the current definition of the Gregorian Calendar, which HLA's routines faithfully reproduce.

```
            // Okay, report whether this is a leap year:

            if( date.isLeapYear( theDate )) then

                stdout.put( “The year “, y, “ is a leap year.” nl );

            else

                stdout.put( “The year “, y, “ is not a leap year.” nl );

            endif;

            // Technically, the leap day is Feb 25, but most people don’t
            // realize this, so use the following output to keep them happy:

            if( date.isLeapYear( y )) then

                if( m = 2 ) then

                    if( d = 29 ) then

                        stdout.put( m, “/”, d, “/”, y, “ is the leap day.” nl );

                    endif;

                endif;

            endif;


          exception( ex.ConversionError )

            stdout.put
            (
                “One of the input values contained illegal characters” nl
            );

          exception( ex.ValueOutOfRange )

            stdout.put
            (
                “One of the input values was too large” nl
            );

          exception( ex.InvalidDate )

            stdout.put
            (
                “The input date (“, m, “/”, d, “/”, y, “) was invalid” nl
            );

        endtry;


    end DemoIsLeapYear;
```

Program 6.2 Calling the date.IsLeapYear Function

6.4.3 Obtaining the System Date

The *date.today* function returns the current system date in the *date.daterec* variable you pass as a parameter[6]. The following program demonstrates how to call this routine:

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

static
    TodaysDate: date.daterec;


begin DemoToday;


    date.today( TodaysDate );

    stdout.put
    (
        “Today is “,
        (type uns8 TodaysDate.month), “/”,
        (type uns8 TodaysDate.day), “/”,
        (type uns16 TodaysDate.year),
        nl
    );


    // Okay, report whether this is a leap year:

    if( date.isLeapYear( TodaysDate )) then

        stdout.put( “This is a leap year.” nl );

    else

        stdout.put( “This is not a leap year.” nl );

    endif;

end DemoToday;
```

Program 6.3 Reading the System Date

Linux users should be aware that *date.today* returns the current date based on Universal Coordinated Time (UTC). Depending upon your time zone, date.today may return yesterday’s or tomorrow’s date within your particular timezone.

6. This function was not available in the Linux version of the HLA Standard Library as this was written. It may have been added by the time you read this, however.

6.4.4 Date to String Conversions and Date Output

The HLA date module provides a set of routines that will convert a *date.daterec* object to the string representation of that date. HLA provides a mechanism that lets you select from one of several different conversion formats when translating dates to strings. The date package defines an enumerated data type, *date.OutputFormat*, that specifies the different conversion mechanisms. The possible conversions are (these examples assume you are converting the date January 2, 2033):

```
date.mdyy         - Outputs date as 1/2/33.
date.mdyyyy       - Outputs date as 1/2/2033.
date.mmddyy       - Outputs date as 01/02/33.
date.mmddyyyy     - Outputs date as 01/02/2033.
date.yymd         - Outputs date as 33/1/2.
date.yyyymd       - Outputs date as 2033/1/2.
date.yymmdd       - Outputs date as 33/01/02.
date.yyyymmdd     - Outputs date as 2033/01/02.
date.MONdyyyy     - Outputs date as Jan 1, 2033.
date.MONTHdyyyy   - Outputs date as January 1, 2033.
```

To set the conversion format, you must call the *date.SetFormat* procedure and pass one of the above values as the single parameter[7]. For all but the last two formats above, the default month/day/year separator is the slash ("/") character. You can call the *date.SetSeparator* procedure, passing it a single character parameter, to change the separator character.

The *date.toString* and *date.a_toString* procedures convert a date to string data. Like the other string routines this chapter discusses, the difference between the *date.toString* and *date.a_toString* procedures is that *date.a_toString* automatically allocates storage for the string whereas you must supply a string with sufficient storage to the *date.toString* procedure. Note that a string containing 20 characters is sufficient for all the different date formats. The *date.toString* and *date.a_toString* procedures use the following calling sequences:

```
date.toString( m, d, y, s );
date.toString( dateVar, s );

date.a_toString( m, d, y );
date.a_toString( dateVar );
```

Note that *m* and *d* are byte values, *y* is a word value, *dateVar* is a *date.dateRec* value, and *s* is a string variable that must point at a string that holds at least 20 characters.

The *date.Print* procedure uses the *date.toString* function to print a date to the standard output device. This is a convenient function to use to display a date after some date calculation.

The following program demonstrates the use of the procedures this section discusses:

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

static
    TodaysDate: date.daterec;
    s:          string;


begin DemoStrConv;
```

7. the *date.SetFormat* routine raises the *ex.InvalidDateFormat* exception if the parameter is not one of these values.

```
    date.today( TodaysDate );
    stdout.put( “Today’s date is “ );
    date.print( TodaysDate );
    stdout.newln();

    // Convert the date using various formats
    // and display the results:

    date.setFormat( date.mdyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( “Date in mdyy format: ‘”, s, “‘” nl );
    strfree( s );

    date.setFormat( date.mmddyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( “Date in mmddyy format: ‘”, s, “‘” nl );
    strfree( s );

    date.setFormat( date.mdyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( “Date in mdyyyy format: ‘”, s, “‘” nl );
    strfree( s );

    date.setFormat( date.mmddyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( “Date in mmddyyyy format: ‘”, s, “‘” nl );
    strfree( s );


    date.setFormat( date.MONdyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( “Date in MONdyyyy format: ‘”, s, “‘” nl );
    strfree( s );


    date.setFormat( date.MONTHdyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( “Date in MONTHdyyyy format: ‘”, s, “‘” nl );
    strfree( s );


end DemoStrConv;
```

Program 6.4 Date <-> String Conversion and Date Output Routines

6.4.5 date.unpack and data.pack

The *date.pack* and *date.unpack* functions pack and unpack date data. The calling syntax for these functions is the following:

```
date.pack( y, m, d, dr );
date.unpack( dr, y, m, d );
```

Note: *y*, *m*, *d* must be uns32 or dword variables; *dr* must be a *date.daterec* object.

The *date.pack* function takes the *y*, *m*, and *d* values and packs them into a *date.daterec* format and stores the result into *dr*. The *date.unpack* function does just the opposite. Neither of these routines check their parameters for proper range. It is the caller's resposibility to ensure that *d's* value is in the range 1..31 (as appropriate for the month and year), *m's* value is in the range 1..12, and *y's* value is in the range 1583..9999.

6.4.6 date.Julian, date.fromJulian

These two functions convert a Gregorian date to and from a Julian day number[8]. Julian day numbers specify January 1, 4713 BCE as day zero and number the days consecutively from that point[9]. One nice thing about Julian day numbers is that date calculations are very easy. You can compute the number of days between two dates by simply subtracting them, you can compute new dates by adding an integer number of days to a Julian day number, etc. The biggest problem with Julian day numbers is converting them to and from the Gregorian Calendar with which we're familiar. Fortunately, these two functions handle that chore. The syntax for calling these two functions is:

```
date.fromJulian( julian, dateRecVar );
date.Julian( m, d, y );
date.Julian( dateRecVar );
```

The first call above converts the Julian day number that you pass in the first parameter to a Gregorian date and stores the result into the *date.daterec* variable you pass as the second parameter. Keep in mind that Julian day numbers that correspond to dates before Jan 1, 1582, will not produce accurate calendar dates since the Gregorian calendar did not exist prior to that point.

The second two calls above compute the Julian day number and return the value in the EAX register. They differ only in the types of parameters they expect. The first call to date.Julian above expects three parameters, *m* and *b* being byte values and *y* being a word value. The second call expects a *date.daterec* parameter; it extracts those three fields and converts them to the Julian day number.

6.4.7 date.datePlusDays, date.datePlusMonths, and date.daysBetween

These two functions provide some simple date arithmetic.operations. The compute a new date by adding some number of days or months to an existing date. The calling syntax for these functions is

```
date.datePlusDays( numDays, dateRecVar );
date.datePlusMonths( numMonths, dateRecVar );
```

Note: *numDays* and *numMonths* are *uns32* values, *dateRecVar* must be a *date.daterec* variable.

The *date.datePlusDays* function computes a new date that is *numDays* days beyond the date that *dateRecVar* specifies. This function leaves the resulting date in *dateRecVar.* This function automatically compensates for the differing number of days in each month as well as the differing number of days in leap years. The *date.datePlusMonths* function does a similar calculation except it adds *numMonths* months, rather than days to *dateRecVar.*

The *date.datePlusDays* function is not particularly efficient if the *numDays* parameter is large. There is a more efficient way to calculate a new date if *numDays* exceeds 1,000: convert the date to a Julian Day Number, add the *numDays* value directly to the Julian Number, and then convert the result back to a date.

8. Note that a Julian date and a Julian day number are not the same thing. Julian dates are based on the Julian Calendar, commisioned by Julius Caesar, which is very similar to the Gregorian Calendar; Julian day numbers were invented in the 1800's and are primarily used by astronomers.
9. Jan 1, 4713 BCE was chosen as a date that predates recorded history.

The *date.daysBetween* function computes the number of days between two dates. Like *date.datePlus-Days*, this function is not particularly efficient if the two dates are more than about three years apart; it is more efficient to compute the Julian day numbers of the two dates and subtract those values. For spans of less than three years, this function is probably more efficient. The calling sequence for this function is the following:

```
date.daysBetween( m1, d1, y1, m2, d2, y2 );
date.daysBetween( m1, d1, y1, dateRecVar2 );
date.daysBetween( dateRecVar1, m2, d2, y2 );
date.daysBetween( dateRecVar1, dateRecVar2 );
```

The four different calls allow you to specify either date as a m/d/y value or as a *date.daterec* value. The *m* and *d* parameters in these calls must be byte values and the *y* parameter must be a word value. The *dateRecVar1* and *dateRecVar2* parameters must, obviously, be *date.daterec* values. These functions return the number of days between the two dates in the EAX register. Note that the dates must be valid, but there is no requirement that the first date be less than the second date.

6.4.8 date.dayNumber, date.daysLeft, and date.dayOfWeek

The *date.dayNumber* function computes the day number into the current year (with Jan 1 being day number one) and returns this value in EAX. This value is always in the range 1..365 (or 1..366 for leap years). A call to this function uses the following syntax:

```
date.dayNumber( m, d, y );
date.dayNumber( dateRecVar );
```

The two forms differ only in the way you pass the date. The first call above expects two byte values (*m* and *d*) and a word value (*y*). The second form above expects a *date.daterec* value.

The *date.daysLeft* function computes the number of days left in a year. This function returns the number of days left in a year *counting the date you pass as a parameter*. Therefore, this function returns one for Dec 31st. Like *date.dayNumber*, this function always returns a value in the range 1..365/366 (regular/leap year). The calling syntax for this function is similar to *date.dayNumber*, it is

```
date.daysLeft( m, d, y );
date.daysLeft( dateRecVar );
```

The parameters have the same meaning as for *date.dayNumber*.

The *date.dayOfWeek* function accepts a date and returns a day of week value in the EAX register. A call to this function uses the following syntax:

```
date.dayOfWeek( m, d, y );
date.dayOfWeek( dateRecVar );
```

The parameters have their usual meanings.

These function calls return a value in the range 0..7 (in EAX) as follows:

0:	Sunday
1:	Monday
2:	Tuesday
3:	Wednesday
4:	Thursday
5:	Friday
6:	Saturday

6.5 Times

The HLA Standard Library provides a simple time module that lets you manipulate times in an HHMMSS (hours/minutes/seconds) format. The *time* namespace in the date/time module defines the time data type as follows:

```
type
    timerec:
        record
            secs:uns8;
            mins:uns8;
            hours:uns16;
        endrecord;
```

This format easily handles 60 seconds per minute and 60 minutes per hour. It also handles up to 65,535 hours (just over 2730 days or about 7-$^1/_2$ years).

The advantages to this time format parallel the advantages of the date format: it is easy to convert the time format to/from external representation (i.e., HH:MM:SS) and the storage format lets you compare times by treating them as *uns32* objects. Another advantage to this format is that it supports more than 24 hours, so you can use it to maintain timings for events that are not calendar based (up to seven years).

There are a couple of disadvantages to this format. The primary disadvantage is that the minimum granularity is one second; if you want to work with fractions of a second then you will need to use a different format and you will have to write the functions for that format. Another disadvantage is that time calculations are somewhat inconvenient. It is difficult to add *n* seconds to a time variable.

Before discussing the HLA Standard Library Time functions, a quick discussion of other possible time formats is probably wise. The only reasonable alternative to the HH:MM:SS format that the HLA Standard Library Time module uses is to use an integer value to represent some number of time units. The only question is "what time units do you want to use?" Whatever time format you use, you should be able to represent at least 86,400 seconds (24 hours) with the format. Furthermore, the granularity should be one second or less. This effectively means you will need at least 32 bits since 16 bits only provides for 65,536 seconds at one second granularity (okay, 24 bits would work, but it's much easier to work with four-byte objects than three-byte objects).

With 32-bits, we can easily represent more than 24 hours' worth of milliseconds (in fact, you can represent almost 50 days before the format rolls over). We could represent five days with a $^1/_{10,000}$ second granularity, but this is not a common timing to use (most people want microseconds if they need better than millisecond granularity), so millisecond granularity is probably the best choice for a 32-bit format. If you need better than millisecond granularity, you should use a combined date/time 64-bit format that measures microseconds since Julian Day Number zero (Jan 1, 4713 BCE). That's good for about a half million years. If you need finer granularity than microseconds, well, you're own your own! You'll have to carefully weigh the issues of granularity vs. years covered vs. the size of your data.

6.5.1 time.curTime

This function returns the current time as read from the system's time of day clock. The calling syntax for this function is the following:

```
time.curTime( timeRecVar );
```

This function call stores the current system time in the *time.timerec* variable you pass as a parameter. On Windows systems, the current time is the wall clock time for your particular time zone; under Linux, the current time is always given in UTC (Universal Coordinated Time) and you must adjust according to your particular time zone to get the local time. Keep this difference in mind when porting programs between Windows and Linux.

6.5.2 time.hmsToSecs and time.secstoHMS

These two functions convert between the HLA internal time format and a pure seconds format. Generally, when doing time arithmetic (e.g., time plus seconds, minutes, or hours), it's easiest to convert your times to seconds, do the calculations with seconds, and then translate the time back to the HLA internal format. This lets you avoid the headaches of modulo-60 arithmetic.

The calling sequences for the *time.hmsToSecs* function are

```
time.hmsToSecs( timeRecValue );
time.hmsToSecs( h, m, s );
```

Both functions return the number of seconds in the EAX register. They differ only in the type of parameters they expect. The first form above expects an HLA *time.timerec* value. The second call above lets you directly specify the hours, minutes, and seconds as separate parameters. The *h* parameter must be a word value, the *m* and *s* parameters must be byte values.

The *time.secsToHMS* function uses the following calling sequence:

```
time.secsToHMS( seconds, timeRecVar );
```

The first parameter must be an uns32 value specifying some number of seconds less than 235,939,600 seconds (which corresponds to 65,536 hours). The second parameter in this call must be a *time.timerec* variable. This function converts the seconds parameter to the HLA internal time format and stores the value into the *timeRecVar* variable.

6.5.3 Time Input/Output

The HLA Standard Library doesn't provide any specific I/O routines for time data. However, reading and writing time data in ASCII form is a fairly trivial process. This section will provide some examples of time I/O using the HLA Standard Input and Standard Output modules.

To output time in a standard HH:MM:SS format, just use the stdout.putisize routines with a width value of two and a fill character of '0' for the three fields of the HLA *time.timerec* data type. The following code demonstrates this:

```
static
    t:time.timerec;
        .
        .
        .
    stdout.putisize( t.h, 2, '0' );
    stdout.put( ':' );
    stdout.putisize( t.m, 2, '0' );
    stdout.put( ':' );
    stdout.putisize( t.s, 2, '0' );
```

If this seems like too much typing, well fear not; in a later chapter you will learn how to create your own functions and you can put this code into a function that will print the time with a single function call.

Time input is only a little more complicated. As it turns out, HLA accepts the colon (":") character as a delimiter when reading numbers from the user. Therefore, reading a time value is no more difficult than reading any other three integer values; you can do it with a single call like the following:

```
    stdin.get( t.hours, t.mins, t.secs );
```

There is one remaining problem with the time input code: it does not validate the input. To do this, you must manually check the seconds and minutes fields to ensure they are values in the range 0..59. If you wish to enforce a limit on the hours field, you should check that value as well. The following code offers one possible solution:

```
stdin.get( t.hours, t.mins, t.secs );
if( t.m >= 60 ) then

    raise( ex.ValueOutOfRange );

endif;
if( t.s >= 60 ) then

    raise( ex.ValueOutOfRange );

endif;
```

6.6 Putting It All Together

Date and time data types do not get anywhere near the consideration they deserve in modern programs. To help ensure that you calculate dates properly in your HLA programs, the HLA Standard Library provides a set of date and time functions that ease the use of dates and times in your programs.

Files Chapter Seven

7.1 Chapter Overview

In this chapter you will learn about the *file* persistent data type. In most assembly languages, file I/O is a major headache. Not so in HLA with the HLA Standard Library. File I/O is no more difficult than writing data to the standard output device or reading data from the standard input device. In this chapter you will learn how to create and manipulate sequential and random-access files.

7.2 File Organization

A file is a collection of data that the system maintains in persistent storage. Persistent means that the storage is non-volatile – that is, the system maintains the data even after the program terminates; indeed, even if you shut off system power. For this reason, plus the fact that different programs can access the data in a file, applications typically use files to maintain data across executions of the application and to share data with other applications.

The operating system typically saves file data on a disk drive or some other form of secondary storage device. As you may recall from the chapter on the memory hierarchy (see "The Memory Hierarchy" on page 303), secondary storage (disk drives) is much slower than main memory. Therefore, you generally do not store data that a program commonly accesses in files during program execution unless that data is far too large to fit into main memory (e.g., a large database).

Under Linux and Windows, a standard file is simply a stream of bytes that the operating system does not interpret in any way. It is the responsibility of the application to interpret this information, much the same as it is your application's responsibility to interpret data in memory. The stream of bytes in a file could be a sequence of ASCII characters (e.g., a text file) or they could be pixel values that form a 24-bit color photograph.

Files generally take one of two different forms: *sequential files* or *random access files*. Sequential files are great for data you read or write all at once; random access files work best for data you read and write in pieces (or rewrite, as the case may be). For example, a typical text file (like an HLA source file) is usually a sequential file. Usually your text editor will read or write the entire file at once. Similarly, the HLA compiler will read the data from the file in a sequential fashion without skipping around in the file. A database file, on the other hand, requires random access since the application can read data from anywhere in the file in response to a query.

7.2.1 Files as Lists of Records

A good view of a file is as a list of records. That is, the file is broken down into a sequential string of records that share a common structure. A list is simply an open-ended single dimensional array of items, so we can view a file as an array of records. As such, we can index into the file and select record number zero, record number one, record number two, etc. Using common file access operations, it is quite possible to skip around to different records in a file. Under Windows and Linux, the principle difference between a sequential file and a random access file is the organization of the records and how easy it is to locate a specific record within the file. In this section we'll take a look at the issues that differentiate these two types of files.

The easiest file organization to understand is the random access file. A random access file is a list of records whose lengths are all identical (i.e., random access files require fixed length records). If the record length is *n* bytes, then the first record appears at byte offset zero in the file, the second record appears at byte offset *n* in the file, the third record appears at byte offset *n*2* in the file, etc. This organization is virtually identical to that of an array of records in main memory; you use the same computation to locate an "element" of this list in the file as you would use to locate an element of an array in memory; the only difference is that a file doesn't have a "base address" in memory, you simply compute the zero-based offset of the record in the file. This calculation is quite simple, and using some file I/O functions you will learn about a little later, you can quickly locate and manipulate any record in a random access file.

Sequential files also consist of a list of records. However, these records do not all have to be the same length[1]. If a sequential file does not use fixed length records then we say that the file uses variable-length records. If a sequential file uses variable-length records, then the file must contain some kind of marker or other mechanism to separate the records in the file. Typical sequential files use one of two mechanisms: a length prefix or some special terminating value. These two schemes should sound quite familiar to those who have read the chapter on strings. Character strings use a similar scheme to determine the bounds of a string in memory.

A text file is the best example of a sequential file that uses variable-length records. Text files use a special marker at the end of each record to delineate the records. In a text file, a record corresponds to a single line of text. Under Windows, the carriage return/line feed character sequence marks the end of each record. Other operating systems may use a different sequence; e.g., Linux uses a single line feed character while the Mac OS uses a single carriage return. Since we're working with Windows or Linux here, we'll adopt the carriage return/line feed or single line feed convention.

Accessing records in a file containing variable-length records is problematic. Unless you have an array of offsets to each record in a variable-length file, the only practical way to locate record *n* in a file is to read the first *n-1* records. This is why variable-length files are sequential-access – you have the read the file sequentially from the start in order to locate a specific record in the file. This will be much slower than accessing the file in a random access fashion. Generally, you would not use a variable-length record organization for files you need to access in a random fashion.

At first blush it would seem that fixed-length random access files offer all the advantages here. After all, you can access records in a file with fixed-length records much more rapidly than files using the variable-length record organization. However, there is a cost to this: your fixed-length records have to be large enough to hold the largest possible data object you want to store in a record. To store a sequence of lines in a text file, for example, your record sizes would have to be large enough to hold the longest possible input line. This could be quite large (for example, HLA allows lines up to 256 characters). Each record in the file will consume this many bytes even if the record uses substantially less data. For example, an empty line only requires one or two bytes (for the line feed [Linux] or carriage return/line feed [Windows] sequence). If your record size is 256 bytes, then you're wasting 255 or 254 bytes for that blank line in your file. If the average line length is around 60 characters, then each line wastes an average of about 200 characters. This problem, known as *internal fragmentation*, can waste a tremendous amount of space on your disk, especially as your files get larger or you create lots of files. File organizations that use variable-length records generally don't suffer from this problem.

7.2.2 Binary vs. Text Files

Another important thing to realize about files is that they don't all contain human readable text. Object and executable files are good examples of files that contain binary information rather than text. A text file is a very special kind of variable-length sequential file that uses special end of line markers (carriage returns/line feeds) at the end of each record (line) in the file. Binary files are everything else.

Binary files are often more compact than text files and they are usually more efficient to access. Consider a text file that contains the following set of two-byte integer values:

```
1234
543
3645
32000
```

1. There is nothing preventing a sequential file from using fixed length records. However, they don't require fixed length records.

```
1
87
0
```

As a text file, this file consumes at least 34 bytes (assuming a two-byte end of line marker on each line). However, were we to store the data in a fixed-record length binary file, with two bytes per integer value, this file would only consume 14 bytes – less than half the space. Furthermore, since the file now uses fixed-length records (two bytes per record) we can efficiently access it in a random fashion. Finally, there is one additional, though hidden, efficiency aspect to the binary format: when a program reads and writes binary data it doesn't have to convert between the binary and string formats. This is an expensive process (with respect to computer time). If a human being isn't going to read this file with a separate program (like a text editor) then converting to and from text format on every I/O operation is a wasted effort.

Consider the following HLA record type:

```
type
    person:
        record
            name:string;
            age:int16;
            ssn:char[11];
            salary:real64;
        endrecord;
```

If we were to write this record as text to a text file, a typical record would take the following form (<nl> indicates the end of line marker, a line feed or carriage return/line feed pair):

```
Hyde, Randall<nl>
45<nl>
555-55-5555<nl>
123456.78<nl>
```

Presumably, the next *person* record in the file would begin with the next line of text in the text file.

The binary version of this file (using a fixed length record, reserving 64 bytes for the *name* string) would look, schematically, like the following:

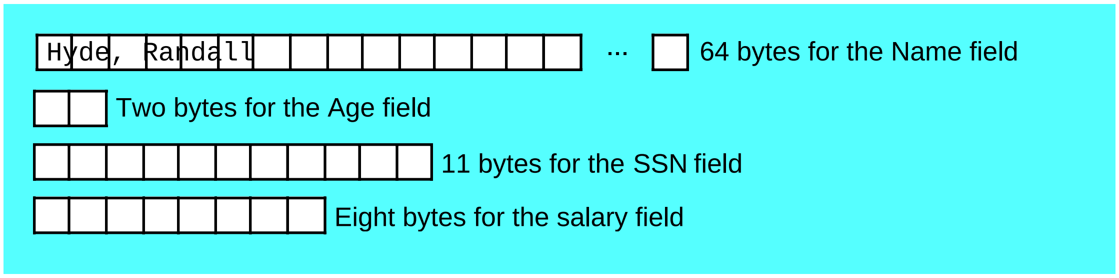
Hyde, Randall
...
64 bytes for the Name field
Two bytes for the Age field
11 bytes for the SSN field
Eight bytes for the salary field

Figure 7.1 Fixed-lengthFormat for Person Record

Don't get the impression that binary files must use fixed length record sizes. We could create a variable-length version of this record by using a zero byte to terminate the string, as follows:

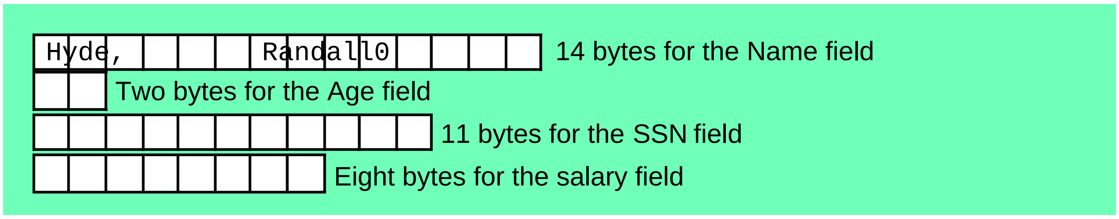
Hyde, Randall0
14 bytes for the Name field
Two bytes for the Age field
11 bytes for the SSN field
Eight bytes for the salary field

Figure 7.2 Variable-length Format for Person Record

In this particular record format the *age* field starts at offset 14 in the record (since the name field and the "end of field" marker [the zero byte] consume 14 bytes). If a different name were chosen, then the *age* field would begin at a different offset in the record. In order to locate the age, ssn, and salary fields of this record, the program would have to scan past the name and find the zero terminating byte. The remaining fields would follow at fixed offsets from the zero terminating byte. As you can see, it's a bit more work to process this variable-length record than the fixed-length record. Once again, this demonstrates the performance difference between random access (fixed-length) and sequential access (variable length, in this case) files.

Although binary files are often more compact and more efficient to access, they do have their drawbacks. In particular, only applications that are aware of the binary file's record format can easily access the file. If you're handed an arbitrary binary file and asked to decipher its contents, this could be very difficult. Text files, on the other hand, can be read by just about any text editor or filter program out there. Hence, your data files will be more interchangeable with other programs if you use text files. Furthermore, it is easier to debug the output of your programs if they produce text files since you can load a text file into the same editor you use to edit your source files.

7.3 Sequential Files

Sequential files are perfect for three types of persistent data: ASCII text files, "memory dumps", and stream data. Since you're probably familiar with ASCII text files, we'll skip their discussion. The other two methods of writing sequential files deserve more explanation.

A "memory dump" is a file that consists of data you transfer from data structures in memory directly to a file. Although the term "memory dump" suggests that you sequentially transfer data from consecutive memory locations to the file, this isn't necessarily the case. Memory access can, an often does, occur in a random access fashion. However, once the application constructs a record to write to the file, it writes that record in a sequential fashion (i.e., each record is written in order to the file). A "memory dump" is what most applications do when you request that they save the program's current data to a file or read data from a file into application memory. When writing, they gather all the important data from memory and write it to the file in a sequential fashion; when reading (loading) data from a file, they read the data from the file in a sequential fashion and store the data into appropriate memory-based data structures. Generally, when loading or saving file data in this manner, the program opens a file, reads/writes data from/to the file, and then it closes the file. Very little processing takes place during the data transfer and the application does not leave the file open for any length of time beyond what is necessary to read or write the file's data.

Stream data on input is like data coming from a keyboard. The program reads the data at various points in the application where it needs new input to continue. Similarly, stream data on output is like a write to the console device. The application writes data to the file at various points in the program after important computations have taken place and the program wishes to report the results of the calculation. Note that when reading data from a sequential file, once the program reads a particular piece of data, that data is no longer available in future reads (unless, of course, the program closes and reopens the file). When writing data to a sequential file, once data is written, it becomes a permanent part of the output file. When processing this kind of data the program typically opens a file and then continues execution. As program execution continues, the application can read or write data in the file. At some point, typically towards the end of the application's execution, the program closes the file and commits the data to disk.

Although disk drives are generally thought of as random access devices, the truth is that they are only pseudo-random access; in fact, they perform much better when writing data sequentially on the disk surface. Therefore, sequential access files tend to provide the highest performance (for sequential data) since they match the highest performance access mode of the disk drive.

Working with sequential files in HLA is very easy. In fact, you already know most of the functions you need in order to read or write sequential files. All that's left to learn is how to open and close files and perform some simple tests (like "have we reached the end of a file when reading data from the file?").

The file I/O functions are nearly identical to the *stdin* and *stdout* functions. Indeed, *stdin* and *stdout* are really nothing more than special file I/O functions that read data from the standard input device (a file) or write data to the standard output device (which is also a file). You use the file I/O functions in a manner analogous to *stdin* and *stdout* except you use the *fileio* prefix rather than stdin or stdout. For example, to write a string to an output file, you could use the *fileio.puts* function almost the same way you use the *stdout.puts* routine. Similarly, if you wanted to read a string from a file, you would use *fileio.gets*. The only real difference between these function calls and their *stdin* and *stdout* counterparts is that you must supply an extra parameter to tell the function what file to use for the transfer. This is a double word value known as the *file handle*. You'll see how to initialize this file handle in a moment, but assuming you have a dword variable that holds a file handle value, you can use calls like the following to read and write data to sequential files:

```
    fileio.get( inputHandle, i, j, k ); // Reads i, j, k, from file inputHandle.
    fileio.put( outputHandle, “I = “, i, “J = “, j, “ K = “, k, nl );
```

Although this example only demonstrates the use of *get* and *put*, be aware that almost all of the *stdin* and *stdout* functions are available as *fileio* functions, as well (in fact, most of the *stdin* and *stdout* functions simply call the appropriate *fileio* function to do the real work).

There is, of course, the issue of this file handle variable. You're probably wondering what a file handle is and how you tell the *fileio* routines to work with data in a specific file on your disk. Well, the definition of the file handle object is the easiest to explain – it's just a dword variable that the operating system initializes and uses to keep track of your file. To declare a file handle, you'd just create a dword variable, e.g.,

```
static
    myFileHandle:dword;
```

You should never explicitly manipulate the value of a file handle variable. The operating system will initialize this variable for you (via some calls you'll see in a moment) and the OS expects you to leave this value alone as long as you're working with the file the OS associates with that handle. If you're curious, both Linux and Windows store small integer values into the handle variable. Internally, the OS uses this value as an index into an array that contains pertinent information about open files. If you mess with the file handle's value, you will confuse the OS greatly the next time you attempt to access the file. Moral of the story – leave this value alone while the file is open.

Before you can read or write a file you must open that file and associate a filename with it. The HLA Standard Library provides a couple of functions that provide this service: *fileio.open* and *fileio.openNew*. The *fileio.open* function opens an existing file for reading, writing, or both. Generally, you open sequential files for reading or writing, but not both (though there are some special cases where you can open a sequential file for reading and writing). The syntax for the call to this function is

```
                    fileio.open( “filename”, access );
```

The first parameter is a string value that specifies the filename of the file to open. This can be a string constant, a register that contains the address of a string value, or a string variable. The second parameter is a constant that specifies how you want to open the file. You may use any of the three predefined constants for the second parameter:

```
            fileio.r
```

```
fileio.w
fileio.rw
```

fileio.r obviously specifies that you want to open an existing file in order to read the data from that file; likewise, *fileio.w* says that you want to open an existing file and overwrite the data in that file. The *fileio.rw* option lets you open a file for both reading and writing.

The *fileio.open* routine, if successful, returns a *file handle* in the EAX register. Generally, you will want to save the return value into a double word variable for use by the other HLA *fileio* routines (i.e., the *MyFileHandle* variable in the earlier example).

If the OS cannot open the file, *fileio.open* will raise an *ex.FileOpenFailure* exception. This usually means that it could not find the specified file on the disk.

The *fileio.open* routine requires that the file exist on the disk or it will raise an exception. If you want to create a new file, that might not already exist, the *fileio.openNew* function will do the job for you. This function uses the following syntax:

```
fileio.openNew( "filename" );
```

Note that this call has only a single parameter, a string specifying the filename. When you open a file with *fileio.openNew*, the file is always opened for writing. If a file by the specified filename already exists, then this function will delete the existing file and the new data will be written over the top of the old file (*so be careful*!).

Like *fileio.open*, *fileio.openNew* returns a file handle in the EAX register if it successfully opens the file. You should save this value in a file handle variable. This function raises the *ex.FileOpenFailure* exception if it cannot open the file.

Once you open a sequential file with *fileio.open* or *fileio.openNew* and you save the file handle value away, you can begin reading data from an input file (*fileio.r*) or writing data to an output file (*fileio.w*). To do this, you would use functions like *fileio.put* as noted above.

When the file I/O is complete, you must close the file to commit the file data to the disk. You should always close all files you open as soon as you are through with them so that the program doesn't consume excess system resources. The syntax for *fileio.close* is very simple, it takes a single parameter, the file handle value returned by *fileio.open* or *fileio.openNew*:

```
fileio.close( file_handle );
```

If there is an error closing the file, *fileio.close* will raise the *ex.FileCloseError* exception. Note that Linux and Windows automatically close all open files when an application terminates; however, it is very bad programming style to depend on this feature. If the system crashes (or the user turns off the power) before the application terminates, file data may be lost. So you should always close your files as soon as you are done accessing the data in that file.

The last function of interest to us right now is the *fileio.eof* function. This function returns true (1) or false (0) in the AL register depending on whether the current file pointer is at the end of the file. Generally you would use this function when reading data from an input file to determine if there is more data to read from the file. You would not normally call this function for output files; it always returns false[2]. Since the fileio routines will raise an exception if the disk is full, there is no need to waste time checking for end of file (EOF) when writing data to a file. The syntax for *fileio.eof* is

```
fileio.eof( file_handle );
```

The following program example demonstrates a complete program that opens and writes a simple text file:

```
program SimpleFileOutput;
```

2. Actually, it will return true under Windows if the disk is full.

```
#include( “stdlib.hhf” )

static
    outputHandle:dword;

begin SimpleFileOutput;

    fileio.openNew( “myfile.txt” );
    mov( eax, outputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.put( outputHandle, (type uns32 ebx ), nl );

    endfor;
    fileio.close( outputHandle );

end SimpleFileOutput;
```

Program 7.1 A Simple File Output Program

The following sample program reads the data that Program 7.1 produces and writes the data to the standard output device:

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

static
    inputHandle:dword;
    u:uns32;

begin SimpleFileInput;

    fileio.open( “myfile.txt”, fileio.r );
    mov( eax, inputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.get( inputHandle, u );
        stdout.put( “ebx=”, ebx, “ u=”, u, nl );

    endfor;
    fileio.close( inputHandle );

end SimpleFileInput;
```

Program 7.2 A Sample File Input Program

There are a couple of interesting functions that you can use when working with sequential files. They are the following:

```
fileio.rewind( fileHandle );
fileio.append( fileHandle );
```

The *fileio.rewind* function resets the “file pointer” (the cursor into the file where the next read or write will take place) back to the beginning of the file. This name is a carry-over from the days of files on tape drives when the system would rewind the tape on the tape drive to move the read/write head back to the beginning of the file.

If you’ve opened a file for reading, then *fileio.rewind* lets you begin reading the file from the start (i.e., make a second pass over the data). If you’ve opened the file for writing, then *fileio.rewind* will cause future writes to overwrite the data you’ve previously written; you won’t normally use this function with files you’ve opened only for writing. If you’ve opened the file for reading and writing (using the *fileio.rw* option) then you can write the data after you’ve first opened the file and then rewind the file and read the data you’ve written. The following is a modification to Program 7.2 that reads the data file twice. This program also demonstrates the use of *fileio.eof* to test for the end of the file (rather than just counting the records).

```
program SimpleFileInput2;
#include( “stdlib.hhf” )

static
    inputHandle:dword;
    u:uns32;

begin SimpleFileInput2;

    fileio.open( “myfile.txt”, fileio.r );
    mov( eax, inputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.get( inputHandle, u );
        stdout.put( “ebx=”, ebx, “ u=”, u, nl );

    endfor;
    stdout.newln();

    // Rewind the file and reread the data from the beginning.
    // This time, use fileio.eof() to determine when we’ve
    // reached the end of the file.

    fileio.rewind( inputHandle );
    while( fileio.eof( inputHandle ) = false ) do

        // Read and display the next item from the file:

        fileio.get( inputHandle, u );
        stdout.put( “u=”, u, nl );

        // Note: after we read the last numeric value, there is still
        // a newline sequence left in the file, if we don’t read the
        // newline sequence after each number then EOF will be false
        // at the start of the loop and we’ll get an EOF exception
        // when we try to read the next value.  Calling fileio.ReadLn
        // “eats” the newline after each number and solves this problem.

        fileio.readLn( inputHandle );


    endwhile;
    fileio.close( inputHandle );

end SimpleFileInput2;
```

Program 7.3   Another Sample File Input Program

The *fileio.append* function moves the file pointer to the end of the file. This function is really only useful for files you've opened for writing (or reading and writing). After executing *fileio.append*, all data you write to the file will be written after the data that already exists in the file (i.e., you use this call to append data to the end of a file you've opened). The following program demonstrates how to use this program to append data to the file created by Program 7.1:

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

static
    fileHandle:dword;
    u:uns32;

begin AppendDemo;

    fileio.open( "myfile.txt", fileio.rw );
    mov( eax, fileHandle );
    fileio.append( eax );

    for( mov( 10, ecx ); ecx < 20; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;


    // Okay, let's rewind to the beginning of the file and
    // display all the data from the file, including the
    // new data we just wrote to it:

    fileio.rewind( fileHandle );
    while( !fileio.eof( fileHandle )) do

        // Read and display the next item from the file:

        fileio.get( fileHandle, u );
        stdout.put( "u=", u, nl );
        fileio.readLn( fileHandle );

    endwhile;
    fileio.close( fileHandle );

end AppendDemo;
```

Program 7.4   Demonstration of the fileio.Append Routine

Another function, similar to *fileio.eof*, that will prove useful when reading data from a file is the *fileio.eoln* function. This function returns true if the next character(s) to be read from the file are the end of line sequence (carriage return, linefeed, or the sequence of these two characters under Windows, just a line feed under Linux). This function returns true or false in the EAX register if it detects an end of line sequence. The calling sequence for this function is

```
fileio.eoln( fileHandle );
```

If *fileio.eoln* detects an end of line sequence, it will read those characters from the file (so the next read from the file will not read the end of line characters). If *fileio.eoln* does not detect the end of line sequence, it does not modify the file pointer position. The following sample program demonstrates the use of *fileio.eoln* in the AppendDemo program, replacing the call to *fileio.readLn* (since *fileio.eoln* reads the end of line sequence, there is no need for the call to *fileio.readLn*):

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

static
    fileHandle:dword;
    u:uns32;

begin EolnDemo;

    fileio.open( "myfile.txt", fileio.rw );
    mov( eax, fileHandle );
    fileio.append( eax );

    for( mov( 10, ecx ); ecx < 20; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;


    // Okay, let's rewind to the beginning of the file and
    // display all the data from the file, including the
    // new data we just wrote to it:

    fileio.rewind( fileHandle );
    while( !fileio.eof( fileHandle )) do

        // Read and display the next item from the file:

        fileio.get( fileHandle, u );
        stdout.put( "u=", u, nl );
        if( !fileio.eoln( fileHandle )) then

            stdout.put( "Hmmm, expected the end of the line", nl );

        endif;

    endwhile;
    fileio.close( fileHandle );

end EolnDemo;
```

Program 7.5 fileio.eoln Demonstration Program.

7.4 Random Access Files

The problem with sequential files is that they are, well, sequential. They are great for dumping and retrieving large blocks of data all at once, but they are not suitable for applications that need to read, write, and rewrite the same data in a file multiple times. In those situations random access files provide the only reasonable alternative.

Windows and Linux don't differentiate sequential and random access files anymore than the CPU differentiates byte and character values in memory; it's up to your application to treat the files as sequential or random access. As such, you use many of the same functions to manipulate random access files as you use to manipulate sequential access files; you just use them differently is all.

You still open files with *fileio.open* and *fileio.openNew*. Random access files are generally opened for reading or reading and writing. You rarely open a random access file as write-only since a program typically needs to read data if it's jumping around in the file.

You still close the files with *fileio.close*.

You can read and write the files with *fileio.get* and *fileio.put*, although you would not normally use these functions for random access file I/O because each record you read or write has to be exactly the same length and these functions aren't particularly suited for fixed-length record I/O. Most of the time you will use one of the following functions to read and write fixed-length data:

```
fileio.write( fileHandle, buffer, count );
fileio.read( fileHandle, buffer, count );
```

The *fileHandle* parameter is the usual file handle value (a dword variable). The *count* parameter is an uns32 object that specifies how many bytes to read or write. The *buffer* parameter must be an array object with at least *count* bytes. This parameter supplies the address of the first byte in memory where the I/O transfer will take place. These functions return the number of bytes read or written in the EAX register. For *fileio.read*, if the return value in EAX does not equal *count's* value, then you've reached the end of the file. For *fileio.write*, if EAX does not equal *count* then the disk is full.

Here is a typical call to the *fileio.read* function that will read a record from a file:

```
fileio.read( myHandle, myRecord, @size( myRecord ) );
```

If the return value in EAX does not equal @size(myRecord) and it does not equal zero (indicating end of file) then there is something seriously wrong with the file since the file should contain an integral number of records.

Writing data to a file with *fileio.write* uses a similar syntax to *fileio.read*.

You can use *fileio.read* and *fileio.write* to read and write data from/to a sequential file, just as you can use routines like *fileio.get* and *fileio.put* to read/write data from/to a random access file. You'd typically use these routines to read and write data from/to a binary sequential file.

The functions we've discussed to this point don't let you randomly access records in a file. If you call *fileio.read* several times in a row, the program will read those records sequentially from the text file. To do true random access I/O we need the ability to jump around in the file. Fortunately, the HLA Standard Library's file module provides several functions you can use to accomplish this.

The *fileio.position* function returns the current offset into the file in the EAX register. If you call this function immediately before reading or writing a record to a file, then this function will tell you the exact position of that record. You can use this value to quickly locate that record for a future access. The calling sequence for this function is

```
fileio.position( fileHandle ); // Returns current file position in EAX.
```

The *fileio.seek* function repositions the file pointer to the offset you specify as a parameter. The following is the calling sequence for this function:

```
fileio.seek( fileHandle, offset ); // Repositions file to specified offset.
```

The function call above will reposition the file pointer to the byte offset specified by the *offset* parameter. If you feed this function the value returned by *fileio.position*, then the next read or write operation will access the record written (or read) immediately after the *fileio.position* call.

You can pass any arbitrary offset value as a parameter to the *fileio.seek* routine; this value does not have to be one that the *fileio.position* function returns. For random access file I/O you would normally compute this offset file by specifying the index of the record you wish to access multiplied by the size of the record. For example, the following code computes the byte offset of record *index* in the file, repositions the file pointer to that record, and then reads the record:

```
intmul( @size( myRecord ), index, ebx );
fileio.seek( fileHandle, ebx );
fileio.read( fileHandle, (type byte myRecord), @size( myRecord ) );
```

You can use essentially this same code sequence to select a specific record in the file for writing.

Note that it is not an error to seek beyond the current end of file and then write data. If you do this, the OS will automatically fill in the intervening records with uninitialized data. Generally, this isn’t a great way to create files, but it is perfectly legal. On the other hand, be aware that if you do this by accident, you may wind up with garbage in the file and no error to indicate that this has happened.

The *fileio* module provides another routine for repositioning the file pointer: *fileio.rSeek*. This function’s calling sequence is very similar to *fileio.seek*, it is

```
fileio.rSeek( fileHandle, offset );
```

The difference between this function and the regular *fileio.seek* function is that this function repositions the file pointer offset bytes from the end of the file (rather than offset bytes from the start of the file). The “r” in “rSeek” stands for “reverse” seek.

Repositioning the file pointer, especially if you reposition it a fair distance from its current location, can be a time-consuming process. If you reposition the file pointer and then attempt to read a record from the file, the system may need to reposition a disk arm (a very slow process) and wait for the data to rotate underneath the disk read/write head. This is why random access I/O is much less efficient than sequential I/O.

The following program demonstrates random access I/O by writing and reading a file of records:

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

type
    fileRec:
        record
            x:int16;
            y:int16;
            magnitude:uns8;
        endrecord;

const

    // Some arbitrary data we can use to initialize the file:
```

```
    fileData:=
        [
            fileRec:[ 2000, 1, 1 ],
            fileRec:[ 1000, 10, 2 ],
            fileRec:[ 750, 100, 3 ],
            fileRec:[ 500, 500, 4 ],
            fileRec:[ 100, 1000, 5 ],
            fileRec:[ 62, 2000, 6 ],
            fileRec:[ 32, 2500, 7 ],
            fileRec:[ 10, 3000, 8 ]
        ];


static
    fileHandle:         dword;
    RecordFromFile:     fileRec;
    InitialFileData:    fileRec[ 8 ] := fileData;



begin RandomAccessDemo;

    fileio.openNew( “fileRec.bin” );
    mov( eax, fileHandle );

    // Okay, write the initial data to the file in a sequential fashion:

    for( mov( 0, ebx ); ebx < 8; inc( ebx )) do

        intmul( @size( fileRec ), ebx, ecx );   // Compute index into fileData
        fileio.write
        (
            fileHandle,
            (type byte InitialFileData[ecx]),
            @size( fileRec )
        );

    endfor;

    // Okay, now let’s demonstrate a random access of this file
    // by reading the records from the file backwards.

    stdout.put( “Reading the records, backwards:” nl );
    for( mov( 7, ebx ); (type int32 ebx) >= 0; dec( ebx )) do

        intmul( @size( fileRec ), ebx, ecx );   // Compute file offset
        fileio.seek( fileHandle, ecx );
        fileio.read
        (
            fileHandle,
            (type byte RecordFromFile),
            @size( fileRec )
        );
        if( eax = @size( fileRec )) then

            stdout.put
            (
                “Read record #”,
                (type uns32 ebx),
                “, values:” nl
                “  x: “, RecordFromFile.x, nl
```

```
                    "  y: ", RecordFromFile.y, nl
                    "  magnitude: ", RecordFromFile.magnitude, nl nl
                );

            else

                stdout.put( "Error reading record number ", (type uns32 ebx), nl );

            endif;

        endfor;
        fileio.close( fileHandle );

    end RandomAccessDemo;
```

Program 7.6 Random Access File I/O Example

7.5 ISAM (Indexed Sequential Access Method) Files

ISAM is a trick that attempts to allow random access to variable-length records in a sequential file. This is a technique employed by IBM on their mainframe data bases in the 1960's and 1970's. Back then, disk space was very precious (remember why we wound up with the Y2K problem?) and IBM's engineers did everything they could to save space. At that time disks held about five megabytes, or so, were the size of washing machines, and cost tens of thousands of dollars. You can appreciate why they wanted to make every byte count. Today, data base designers have disk drives with hundreds of gigabytes per drive and RAID[3] devices with dozens of these drives installed. They don't bother trying to conserve space at all ("Heck, I don't know how big the person's name can get, so I'll allocate 256 bytes for it!"). Nevertheless, even with large disk arrays, saving space is often a wise idea. Not everyone has a terabyte (1,000 gigabytes) at their disposal and a user of your application may not appreciate your decision to waste their disk space. Therefore, techniques like ISAM that can reduce disk storage requirements are still important today.

ISAM is actually a very simple concept. Somewhere, the program saves the offset to the start of every record in a file. Since offsets are four bytes long, an array of dwords will work quite nicely[4]. Generally, as you construct the file you fill in the list (array) of offsets and keep track of the number of records in the file. For example, if you were creating a text file and you wanted to be able to quickly locate any line in the file, you would save the offset into the file of each line you wrote to the file. The following code fragment shows how you could do this:

```
static
    outputLine: string;
    ISAMarray: dword[ 128*1024 ]; // allow up to 128K records.
            .
            .
            .
    mov( 0, ecx );          // Keep record count here.
    forever

        << create a line of text in "outputLine" >>

        fileio.position( fileHandle );
```

3. Redundant array of inexpensive disks. RAID is a mechanism for combining lots of cheap disk drives together to form the equivalent of a really large disk drive.
4. This assumes, of course, that your files have a maximum size of four gigabytes.

```
        mov( eax, ISAMarray[ecx*4] );  // Save away current record offset.
        fileio.put( fileHandle, outputLine, nl ); // Write the record.
        inc( ecx );  // Advance to next element of ISAMarray.

        << determine if we’re done and BREAK if we are >>

    endfor;

    << At this point, ECX contains the number of records and >>
    << ISAMarray[0]..ISAMarray[ecx-1] contain the offsets to >>
    << each of the records in the file.                      >>
```

After building the file using the code above, you can quickly jump to an arbitrary line of text by fetching the index for that line from the *ISAMarray* list. The following code demonstrates how you could read line *recordNumber* from the file:

```
    mov( recordNumber, ebx );
    fileio.seek( fileHandle, ISAMarray[ ebx*4 ] );
    fileio.a_gets( fileHandle, inputString );
```

As long as you’ve precalculated the *ISAMarray* list, accessing an arbitrary line in this text file is a trivial matter.

Of course, back in the days when IBM programmers were trying to squeeze every byte from their databases as possible so they would fit on a five megabyte disk drive, they didn’t have 512 kilobytes of RAM to hold 128K entries in the *ISAMarray* list. Although a half a megabyte is no big deal today, there are a couple of reasons why keeping the *ISAMarray* list in a memory-based array might not be such a good idea. First, databases are much larger these days. Some databases have hundreds of millions of entries. While setting aside a half a megabyte for an ISAM table might not be a bad thing, few people are willing to set aside a half a gigabyte for this purpose. Even if your database isn’t amazingly big, there is another reason why you might not want to keep your *ISAMarray* in main memory – it’s the same reason you don’t keep the file in memory – memory is volatile and the data is lost whenever the application quits or the user removes power from the system. The solution is exactly the same as for the file data: you store the *ISAMarray* data in its own file. A program that builds the ISAM table while writing the file is a simple modification to the previous ISAM generation program. The trick is to open two files concurrently and write the ISAM data to one file while you’re writing the text to the other file:

```
static
    fileHandle: dword;     // file handle for the text file.
    outputLine: string;    // file handle for the ISAM file.
    CurrentOffset: dword;  // Holds the current offset into the text file.
            .
            .
            .
    forever

        << create a line of text in “outputLine” >>

        // Get the offset of the next record in the text file
        // and write this offset (sequentially) to the ISAM file.

        fileio.position( fileHandle );
        mov( eax, CurrentOffset );
        fileio.write( isamHandle, (type byte CurrentOffset), 4 );

        // Okay, write the actual text data to the text file:

        fileio.put( fileHandle, outputLine, nl ); // Write the record.

        << determine if we’re done and BREAK if we are >>
    endfor;
```

If necessary, you can count the number of records as before. You might write this value to the first record of the ISAM file (since you know the first record of the text file is always at offset zero, you can use the first element of the ISAM list to hold the count of ISAM/text file records).

Since the ISAM file is just a sequence of four-byte integers, each record in the file (i.e., an integer) has the same length. Therefore, we can easily access any value in the ISAM file using the random access file I/O mechanism. In order to read a particular line of text from the text file, the first task is to read the offset from the ISAM file and then use that offset to read the desired line from the text file. The code to accomplish this is as follows:

```
    // Assume we want to read the line specified by the “lineNumber” variable.

    if( lineNumber <> 0 ) then

        // If not record number zero, then fetch the offset to the desired
        // line from the ISAM file:

        intmul( 4, lineNumber, eax );   // Compute the index into the ISAM file.
        fileio.seek( isamHandle, eax );
        fileio.read( isamHandle, (type byte CurrentOffset), 4 ); // Read offset

    else

        mov( 0, eax );  // Special case for record zero because the file
                        // contains the record count in this position.

    endif;
    fileio.seek( fileHandle, CurrentOffset ); // Set text file position.
    fileio.a_gets( fileHandle, inputLine );   // Read the line of text.
```

This operation runs at about half the speed of having the ISAM array in memory (since it takes four file accesses rather than two to read the line of text from the file), but the data is non-volatile and is not limited by the amount of available RAM.

If you decide to use a memory-based array for your ISAM table, it’s still a good idea to keep that data in a file somewhere so you don’t have to recompute it (by reading the entire file) every time your application starts. If the data is present in a file, all you’ve got to do is read that file data into your *ISAMarray* list. Assuming you’ve stored the number of records in element number zero of the ISAM array, you could use the following code to read your ISAM data into the *ISAMarray* variable:

```
static
    isamSize: uns32;
    isamHandle: dword;
    fileHandle: dword;
    ISAMarray: dword[ 128*1024 ];
        .
        .
        .
    // Read the first record of the ISAM file into the isamSize variable:

    fileio.read( isamHandle, (type byte isamSize), 4 );

    // Now read the remaining data from the ISAM file into the ISAMarray
    // variable:

    if( isamSize >= 128*1024 ) then

        raise( ex.ValueOutOfRange );
    endif;
    intmul( 4, isamSize, ecx );  // #records * 4 is number of bytes to read.
    fileio.read( isamHandle, (type byte ISAMarray), ecx );

    // At this point, ISAMarray[0]..ISAMarray[isamSize-1] contain the indexes
    // into the text file for each line of text.
```

7.6 Truncating a File

If you open an existing file (using *fileio.open*) for output and write data to that file, it overwrites the existing data from the start of the file. However, if the new data you write to the file is shorter than the data originally appearing in the file, the excess data from the original file, beyond the end of the new data you've written, will still appear at the end of the new data. Sometimes this might be desirable, but most of the time you'll want to delete the old data after writing the new data.

One way to delete the old data is to use the *fileio.openNew* function to open the file. The *fileio.openNew* function automatically deletes any existing file so only the data you write to the file will be present in the file. However, there may be times when you may want to read the old data first, rewind the file, and then overwrite the data. In this situation, you'll need a function that will *truncate* the old data at the end of the file after you've written the new data. The *fileio.truncate* function accomplishes this task. This function uses the following calling syntax:

```
fileio.truncate( fileHandle );
```

Note that this function does not close the file. You still have to call *fileio.close* to commit the data to the disk.

The following sample program demonstrates the use of the *fileio.truncate* function:

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

static
    fileHandle:dword;
    u:uns32;

begin TruncateDemo;

    fileio.openNew( "myfile.txt" );
    mov( eax, fileHandle );
    for( mov( 0, ecx ); ecx < 20; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;


    // Okay, let's rewind to the beginning of the file and
    // rewrite the first ten lines and then truncate the
    // file at that point.

    fileio.rewind( fileHandle );
    for( mov( 0, ecx ); ecx < 10; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );
    endfor;
    fileio.truncate( fileHandle );


    // Rewind and display the file contents to ensure that
    // the file truncation has worked.

    fileio.rewind( fileHandle );
    while( !fileio.eof( fileHandle )) do

        // Read and display the next item from the file:

        fileio.get( fileHandle, u );
        stdout.put( “u=”, u, nl );
        fileio.readLn( fileHandle );

    endwhile;
    fileio.close( fileHandle );

end TruncateDemo;
```

Program 7.7 Using fileio.truncate to Eliminate Old Data From a File

7.7 File Utility Routines

The following subsections describe *fileio* functions that manipulate files or return meta-information about files (e.g., the file size and attributes).

7.7.1 Copying, Moving, and Renaming Files

Some very useful file utilities are copying, moving, and renaming files. For example, you might want to copy a file an application has created in order to make a backup copy. Moving files from one subdirectory to another, or even from one disk to another is another common operation. Likewise, the need to change the name of a file arises all the time. In this section we’ll take a look at the HLA Standard Library routines that accomplish these operations.

Copying a file is a nearly trivial process under Windows[5]. All you’ve got to do is open a source file, open a destination file, then read the bytes from the source file and write them to the destination file until you hit end of file. Unfortunately, this simple approach to copying a file can suffer from performance problems. Windows provides an internal function to copy files using a high performance algorithm (Linux does not provide this call). The HLA Standard Library *fileio.copy* function provides an interface to this copy operation. The copy a file using the *fileio.copy* procedure, you’d use the following call sequence:

```
fileio.copy( sourcefileName, destFileName, failIfExists );
```

The *sourceFileName* and *destFileName* parameters are strings that specify the pathnames of the source and destination files. These can be string constants or variables. The last parameter is a boolean variable that specifies what should happen if the destination file exists. If this parameter contains true and the file already exists, then the function will fail; if *failIfExists* is false, the *fileio.copy* routine will replace the existing destination file with a copy of the source file. In either case, of course, the source file must exist or this function

5. Sorry, the *fileio.copy* function is not available under Linux.

will fail. This function returns a boolean success/failure result in the EAX register. It returns true if the function returns TRUE in EAX.

Program 7.8 demonstrates the use of this function to copy a file:

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

begin CopyDemo;

    // Make a copy of myfile.txt to itself to demonstrate
    // a true “failsIfExists” parameter.

    if( !fileio.copy( “myfile.txt”, “myfile.txt”, true )) then

        stdout.put( “Did not copy ‘myfile.txt’ over itself” nl );

    else

        stdout.put( “Whoa!  The failsIfExists parameter didn’t work.” nl );

    endif;

    // Okay, make a copy of the file to a different file, to verify
    // that this works properly:

    if( fileio.copy( “myfile.txt”, “copyOfMyFile.txt”, false )) then

        stdout.put( “Successfully copied the file” nl );

    else

        stdout.put( “Failed to copy the file (maybe it doesn’t exist?)” nl );

    endif;

end CopyDemo;
```

Program 7.8 Demonstration of a fileio.copy Operation

To move a file from one location to another might seem like another trivial task – all you’ve got to do is copy the file to the destination and then delete the original file. However, these scheme is quite inefficient in most situations. Copying the file can be an expensive process if the file is large; Worse, the move operation may fail if you’re moving the file to a new location on the same disk and there is insufficient space for a second copy of the file. A much better solution is to simply move the file’s *directory entry* from one location to another on the disk. Win32’s disk directory entries are quite small, so moving a file to a different location on the same disk by simply moving its directory entry is very fast and efficient. Unfortunately, if you move a file from one file system (disk) to another, you will have to first copy the file and then delete the original file. Once again, you don’t have to bother with the complexities of this operation because Windows has a built-in function that automatically moves files for you. The HLA Standard Library’s *fileio.move* procedure provides a direct interface to this function (available only under Windows). The calling sequence is

```
fileio.move( source, dest );
```

The two parameters are strings providing the source and destination filenames. This function returns true or false in EAX to denote the success or failure of the operation.

Not only can the *fileio.move* procedure move a file around on the disk, it can also move subdirectories around. The only catch is that you cannot move a subdirectory from one volume (file system/disk) to another.

If both the destination and source filenames are simple filenames, not a pathnames, then the *fileio.move* function moves the source file from the current directory back to the current directory. Although this seems rather weird, this is a very common operation; this is how you rename a file. The HLA Standard Library does not have a separate "fileio.rename" function. Instead, you use the fileio.move function to rename files by moving them to the same directory but with a different filename. Program 7.9 demonstrates how to use *fileio.move* in this capacity.

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

begin FileMoveDemo;

    // Rename the "myfile.txt" file to the name "renamed.txt".

    if( !fileio.move( "myfile.txt", "renamed.txt" )) then

        stdout.put
        (
            "Could not rename 'myfile.txt' (maybe it doesn't exist?)" nl
        );

    else

        stdout.put( "Successfully renamed the file" nl );

    endif;


end FileMoveDemo;
```

Program 7.9 Using fileio.move to Rename a File

7.7.2 Computing the File Size

Another useful function to have is one that computes the size of an existing file on the disk. The *fileio.size* function provides this capability. The calling sequences for this function are

```
fileio.size( filenameString );
fileio.size( fileHandle );
```

The first form above expects you to pass the filename as a string parameter. The second form expects a handle to a file you've opened with *fileio.open* or *fileio.openNew*. These two calls return the size of the file in EAX. If an error occurs, these functions return -1 ($FFFF_FFFF) in EAX. Note that the files must be less than four gigabytes in length when using this function (if you need to check the size of larger files, you will have to call the appropriate OS function rather than these functions; however, since files larger than four gigabytes are rather rare, you probably won't have to worry about this problem).

One interesting use for this function is to determine the number of records in a fixed-length-record random access file. By getting the size of the file and dividing by the size of a record, you can determine the number of records in the file.

Another use for this function is to allow you to determine the size of a (smaller) file, allocate sufficient storage to hold the entire file in memory (by using *malloc*), and then read the entire file into memory using the *fileio.read* function. This is generally the fastest way to read data from a file into memory.

Program 7.10 demonstrates the use of the two forms of the *fileio.size* function by displaying the size of the “myfile.txt” file created by other sample programs in this chapter.

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

static
    handle:dword;

begin FileSizeDemo;

    // Display the size of the “FileSizeDemo.hla” file:

    fileio.size( “FileSizeDemo.hla” );
    if( eax <> -1 ) then

        stdout.put( “Size of file: “, (type uns32 eax), nl );

    else

        stdout.put( “Error calculating file size” nl );

    endif;

    // Same thing, using the file handle as a parameter:

    fileio.open( “FileSizeDemo.hla”, fileio.r );
    mov( eax, handle );
    fileio.size( handle );
    if( eax <> -1 ) then

        stdout.put( “Size of file(2): “, (type uns32 eax), nl );

    else

        stdout.put( “Error calculating file size” nl );

    endif;
    fileio.close( handle );



end FileSizeDemo;
```

Program 7.10 Sample Program That Demonstrates the fileio.size Function

7.7.3 Deleting Files

Another useful file utility function is the fileio.delete function. As its name suggests, this function deletes a file that you specify as the function's parameter. The calling sequence for this function is

```
fileio.delete( filenameToDelete );
```

The single parameter is a string containing the pathname of the file you wish to delete. This function returns true/false in the EAX register to denote success/failure.

Program 7.11 provides an example of the use of the *fileio.delete* function.

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

static
    handle:dword;

begin DeleteFileDemo;

    // Delete the “myfile.txt” file:

    fileio.delete( “xyz” );
    if( eax ) then

        stdout.put( “Deleted the file”, nl );

    else

        stdout.put( “Error deleting the file” nl );

    endif;


end DeleteFileDemo;
```

Program 7.11 Example Usage of the fileio.delete Procedure

7.8 Directory Operations

In addition to manipulating files, you can also manipulate directories with some of the *fileio* functions. The HLA Standard Library includes several functions that let you create and use subdirectories. These functions are *fileio.cd* (change directory), *fileio.gwd* (get working directory), and *fileio.mkdir* (make directory). Their calling sequences are

```
fileio.cd( pathnameString );
fileio.gwd( stringToHoldPathname );
fileio.mkdir( newDirectoryName );
```

The *fileio.cd* and *fileio.mkdir* functions return success or failure (true or false, respectively) in the EAX register. For the *fileio.gwd* function, the string parameter is a destination string where the system will store the pathname to the current directory. You must allocate sufficient storage for the string prior to passing the string to this function (260 characters[6] is a good default amount if you're unsure how long the pathname could be). If the actual pathname is too long to fit in the destination string you supply as a parameter, the *fileio.gwd* function will raise the *ex.StringOverflow* exception.

The *fileio.cd* function sets the current working directory to the pathname you specify. After calling this function, the OS will assume that all future “unadorned” file references (those without any “\” or “/” characters in the pathname) will default to the directory you specify as the *fileio.cd* parameter. Proper use of this function can help make your program much more convenient to use by your program’s users since they won’t have to enter full pathnames for every file they manipulate.

The *fileio.gwd* function lets you query the system to determine the current working directory. After a call to *fileio.cd*, the string that *fileio.gwd* returns should be the same as *fileio.cd’s* parameter. Typically, you would use this function to keep track of the default directory when your program first starts running. You program will exhibit good manners by switching back to this default directory when your program terminates.

The *fileio.mkdir* function lets your program create a new subdirectory. If your program creates data files and stores them in a default directory somewhere, it’s good etiquette to let the user specify the subdirectory where your program should put these files. If you do this, you should give your users the option to create a new directory (in case they want the data placed in a brand-new directory). You can use *fileio.mkdir* for this purpose.

7.9 Putting It All Together

This chapter began with a discussion of the basic file operations. That section was rather short because you’ve already learned most of what you need to know about file I/O when learning the *stdout* and *stdin* functions. So the introductory material concentrated on a file general file concepts (like the differences between sequential and random access files and the differences between binary and text files). After teaching you the few extra routines you need in order to open and close files, the remainder of this chapter simply concentrated on providing a few examples (like ISAM) of file access and a discussion of the *fileio* routines available in the HLA Standard Library.

While this chapter demonstrates the mechanics of file I/O, how you efficiently use files is well beyond the scope of this chapter. In future volumes you will see how to search for data in files, sort data in files, and even create databases. So keep on reading if you’re interested in more information about file operations.

6. This is the default MAX_PATH value in Windows. This is probably sufficient for most Linux applications, too.

Introduction to Procedures — Chapter Eight

8.1 Chapter Overview

In a procedural programming language the basic unit of code is the *procedure*. A procedure is a set of instructions that compute some value or take some action (such as printing or reading a character value). The definition of a procedure is very similar to the definition of an *algorithm*. A procedure is a set of rules to follow which, if they conclude, produce some result. An algorithm is also such a sequence, but an algorithm is guaranteed to terminate whereas a procedure offers no such guarantee.

This chapter discusses how HLA implements procedures. This is actually the first of three chapters on this subject in this text. This chapter presents HLA procedures from a high level language perspective. A later chapter, Intermediate Procedures, discusses procedures at the machine language level. A whole volume in this sequence, Advanced Procedures, covers advanced programming topics of interest to the very serious assembly language programmer. This chapter, however, provides the foundation for all that follows.

8.2 Procedures

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[1]:

```
call stdout.newln;
```

The *stdout.newln* procedure prints a newline sequence to the console device and returns control to the instruction immediately following the "call stdout.newln;" instruction.

Alas, the HLA Standard Library does not supply all the routines you will need. Most of the time you'll have to write your own procedures. To do this, you will use HLA's procedure declaration facilities. A basic HLA procedure declaration takes the following form:

```
procedure ProcName;
    << Local declarations >>
begin ProcName;
    << procedure statements >>
end ProcName;
```

Procedure declarations appear in the declaration section of your program. That is, anywhere you can put a STATIC, CONST, TYPE, or other declaration section, you may place a procedure declaration. In the syntax example above, *ProcName* represents the name of the procedure you wish to define. This can be any valid HLA identifier. Whatever identifier follows the PROCEDURE reserved word must also follow the BEGIN and END reserved words in the procedure. As you've probably noticed, a procedure declaration looks a whole lot like an HLA program. In fact, the only difference (so far) is the use of the PROCEDURE reserved word rather than the PROGRAM reserved word.

Here is a concrete example of an HLA procedure declaration. This procedure stores zeros into the 256 double words that EBX points at upon entry into the procedure:

```
procedure zeroBytes;
begin zeroBytes;

    mov( 0, eax );
```

1. Normally you would call newln using the "newln();" statement, but the CALL instruction works as well.

```
    mov( 256, ecx );
    repeat

        mov( eax, [ebx] );
        add( 4, ebx );
        dec( ecx );

    until( @z );  // That is, until ECX=0.

end zeroBytes;
```

You can use the 80x86 CALL instruction to call this procedure. When, during program execution, the code falls into the “end zeroBytes;” statement, the procedure returns to whomever called it and begins executing the first instruction beyond the CALL instruction. The following program provides an example of a call to the *zeroBytes* routine:

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


    procedure zeroBytes;
    begin zeroBytes;

        mov( 0, eax );
        mov( 256, ecx );
        repeat

            mov( eax, [ebx] );  // Zero out current dword.
            add( 4, ebx );      // Point ebx at next dword.
            dec( ecx );         // Count off 256 dwords.

        until( ecx = 0 );       // Repeat for 256 dwords.

    end zeroBytes;

static
    dwArray: dword[256];

begin zeroBytesDemo;

    lea( ebx, dwArray );
    call zeroBytes;

end zeroBytesDemo;
```

Program 8.1 Example of a Simple Procedure

As you may have noticed when calling HLA Standard Library procedures, you don’t always need to use the CALL instruction to call HLA procedures. There is nothing special about the HLA Standard Library procedures versus your own procedures. Although the formal 80x86 mechanism for calling procedures is to use the CALL instruction, HLA provides a HLL extension that lets you call a procedure by simply specifying that procedure’s name followed by an empty set of parentheses[2]. For example, either of the following statements will call the HLA Standard Library *stdout.newln* procedure:

2. This assumes that the procedure does not have any parameters.

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

Likewise, either of the following statements will call the *zeroBytes* procedure in Program 8.1:

```
call zeroBytes;
 zeroBytes();
```

The choice of calling mechanism is strictly up to you. Most people, however, find the HLL syntax easier to read.

8.3 Saving the State of the Machine

Take a look at the following program:

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


    procedure PrintSpaces;
    begin PrintSpaces;

        mov( 40, ecx );
        repeat

            stdout.put( ‘ ‘ );  // Print 1 of 40 spaces.
            dec( ecx );         // Count off 40 spaces.

        until( ecx = 0 );

    end PrintSpaces;

begin nonWorkingProgram;

    mov( 20, ecx );
    repeat

        PrintSpaces();
        stdout.put( ‘*’, nl );
        dec( ecx );

    until( ecx = 0 );

end nonWorkingProgram;
```

Program 8.2 Program with an Unintended Infinite Loop

This section of code attempts to print 20 lines of 40 spaces and an asterisk. Unfortunately, there is a subtle bug that causes it to print 40 spaces per line and an asterisk in an infinite loop. The main program uses the REPEAT..UNTIL loop to call *PrintSpaces* 20 times. *PrintSpaces* uses ECX to count off the 40 spaces it prints. *PrintSpaces* returns with ECX containing zero. The main program then prints an asterisk, a newline, decrements ECX, and then repeats because ECX isn’t zero (it will always contain $FFFF_FFFF at this point).

The problem here is that the *PrintSpaces* subroutine doesn't preserve the ECX register. Preserving a register means you save it upon entry into the subroutine and restore it before leaving. Had the *PrintSpaces* subroutine preserved the contents of the ECX register, the program above would have functioned properly.

Use the 80x86's PUSH and POP instructions to preserve register values while you need to use them for something else. Consider the following code for *PrintSpaces*:

```
    procedure PrintSpaces;
    begin PrintSpaces;

        push( eax );
        push( ecx );
        mov( 40, ecx );
        repeat

            stdout.put( ' ' );  // Print 1 of 40 spaces.
            dec( ecx );         // Count off 40 spaces.

        until( ecx = 0 );
        pop( ecx );
        pop( eax );

    end PrintSpaces;
```

Note that *PrintSpaces* saves and restores EAX and ECX (since this procedure modifies these registers). Also, note that this code pops the registers off the stack in the reverse order that it pushed them. The last-in, first-out, operation of the stack imposes this ordering.

Either the caller (the code containing the CALL instruction) or the callee (the subroutine) can take responsibility for preserving the registers. In the example above, the callee preserved the registers. The following example shows what this code might look like if the caller preserves the registers:

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


    procedure PrintSpaces;
    begin PrintSpaces;

        mov( 40, ecx );
        repeat

            stdout.put( ‘ ‘ );  // Print 1 of 40 spaces.
            dec( ecx );         // Count off 40 spaces.

        until( ecx = 0 );

    end PrintSpaces;

begin callerPreservation;

    mov( 20, ecx );
    repeat

        push( eax );
        push( ecx );
        PrintSpaces();
        pop( ecx );
        pop( eax );
        stdout.put( ‘*’, nl );
        dec( ecx );
    until( ecx = 0 );

end callerPreservation;
```

Program 8.3 Demonstration of Caller Register Preservation

There are two advantages to callee preservation: space and maintainability. If the callee preserves all affected registers, then there is only one copy of the PUSH and POP instructions, those the procedure contains. If the caller saves the values in the registers, the program needs a set of PUSH and POP instructions around every call. Not only does this make your programs longer, it also makes them harder to maintain. Remembering which registers to push and pop on each procedure call is not something easily done.

On the other hand, a subroutine may unnecessarily preserve some registers if it preserves all the registers it modifies. In the examples above, the code needn't save EAX. Although *PrintSpaces* changes AL, this won't affect the program's operation. If the caller is preserving the registers, it doesn't have to save registers it doesn't care about:

```
program callerPreservation2;
#include( "stdlib.hhf" );


    procedure PrintSpaces;
    begin PrintSpaces;

        mov( 40, ecx );
        repeat

            stdout.put( ' ' );  // Print 1 of 40 spaces.
            dec( ecx );         // Count off 40 spaces.

        until( ecx = 0 );

    end PrintSpaces;

begin callerPreservation2;

    mov( 10, ecx );
    repeat

        push( ecx );
        PrintSpaces();
        pop( ecx );
        stdout.put( '*', nl );
        dec( ecx );

    until( ecx = 0 );


    mov( 5, ebx );
    while( ebx > 0 ) do

        PrintSpaces();

        stdout.put( ebx, nl );
        dec( ebx );
    endwhile;


    mov( 110, ecx );
    for( mov( 0, eax );  eax < 7; inc( eax )) do

        PrintSpaces();

        stdout.put( eax, " ", ecx, nl );
        dec( ecx );

    endfor;

end callerPreservation2;
```

Program 8.4 Demonstrating that Caller Preservation Need not Save All Registers

This example provides three different cases. The first loop (REPEAT..UNTIL) only preserves the ECX register. Modifying the AL register won't affect the operation of this loop. Immediately after the first loop, this code calls *PrintSpaces* again in the WHILE loop. However, this code doesn't save EAX or ECX because it doesn't care if *PrintSpaces* changes them. Since the final loop (FOR) uses EAX and ECX, it saves them both.

One big problem with having the caller preserve registers is that your program may change. You may modify the calling code or the procedure so that they use additional registers. Such changes, of course, may change the set of registers that you must preserve. Worse still, if the modification is in the subroutine itself, you will need to locate *every* call to the routine and verify that the subroutine does not change any registers the calling code uses.

Preserving registers isn't all there is to preserving the environment. You can also push and pop variables and other values that a subroutine might change. Since the 80x86 allows you to push and pop memory locations, you can easily preserve these values as well.

8.4 Prematurely Returning from a Procedure

The HLA EXIT and EXITIF statements let you return from a procedure without having to fall into the corresponding END statement in the procedure. These statements behave a whole lot like the BREAK and BREAKIF statements for loops, except they transfer control to the bottom of the procedure rather than out of the current loop. These statements are quite useful in many cases.

The syntax for these two statements is the following:

```
exit procedurename;
exitif( boolean_expression ) procedurename;
```

The *procedurename* operand is the name of the procedure you wish to exit. If you specify the name of your main program, the EXIT and EXITIF statements will terminate program execution (even if you're currently inside a procedure rather than the body of the main program.

The EXIT statement immediately transfers control out of the specified procedure or program. The conditional exit, EXITIF, statement first tests the boolean expression and exits if the result is true. It is semantically equivalent to the following:

```
if( boolean_expression ) then

    exit procedurename;
```

```
        endif;
```

Although the EXIT and EXITIF statements are invaluable in many cases, you should try to avoid using them without careful consideration. If a simple IF statement will let you skip the rest of the code in your procedure, by all means use the IF statement. Procedures that contain lots of EXIT and EXITIF statements will be harder to read, understand, and maintain that procedures without these statements (after all, the EXIT and EXITIF statements are really nothing more than GOTO statements and you've probably heard already about the problems with GOTOs). EXIT and EXITIF are convenient when you got to return from a procedure inside a sequence of nested control structures and slapping an IF..ENDIF around the remaining code in the procedure is not possible.

8.5 Local Variables

HLA procedures, like procedures and functions in most high level languages, let you declare *local variables*. Local variables are generally accessible only within the procedure, they are not accessible by the code that calls the procedure. Local variable declarations are identical to variable declarations in your main program except, of course, you declare the variables in the procedure's declaration section rather than the main program's declaration section. Actually, you may declare anything in the procedure's declaration section that is legal in the main program's declaration section, including constants, types, and even other procedures[3]. In this section, however, we'll concentrate on local variables.

Local variables have two important attributes that differentiate them from the variables in your main program (i.e., *global* variables): *lexical scope* and *lifetime*. Lexical scope, or just *scope*, determines when an identifier is usable in your program. Lifetime determines when a variable has memory associated with it and is capable of storing data. Since these two concepts differentiate local and global variables, it is wise to spend some time discussing these two attributes.

Perhaps the best place to start when discussing the scope and lifetimes of local variables is with the scope and lifetimes of global variables -- those variables you declare in your main program. Until now, the only rule you've had to follow concerning the declaration of your variables has been "you must declare all variables that you use in your programs." The position of the HLA declaration section with respect to the program statements automatically enforces the other major rule which is "you must declare all variables before their first use." With the introduction of procedures, it is now possible to violate this rule since (1) procedures may access global variables, and (2) procedure declarations may appear anywhere in a declaration section, even before some variable declarations. The following program demonstrates this source code organization:

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

static
    AccessibleInProc: char;


    procedure aProc;
    begin aProc;

        mov( 'a', AccessibleInProc );

    end aProc;
```

3. The chapter on Advanced Procedures discusses the concept of local procedures in greater detail.

```
static
    InaccessibleInProc: char;


begin demoGlobalScope;


    mov( 'b', InaccessibleInProc );
    aProc();
    stdout.put
    (
        "AccessibleInProc   = '", AccessibleInProc,   "'" nl
        "InaccessibleInProc = '", InaccessibleInProc, "'" nl
    );


end demoGlobalScope;
```

Program 8.5 Demonstration of Global Scope

This example demonstrates that a procedure can access global variables in the main program as long as you declare those global variables before the procedure. In this example, the *aProc* procedure cannot access the *InaccessibleInProc* variable because its declaration appears after the procedure declaration. However, *aProc* may reference *AccessibleInProc* since it's declaration appears before the *aProc* procedure in the source code.

A procedure can access any STATIC, STORAGE, or READONLY object exactly the same way the main program accesses such variables -- by simply referencing the name. Although a procedure may access global VAR objects, a different syntax is necessary and you need to learn a little more before you will understand the purpose of the additional syntax. Therefore, we'll defer the discussion of accessing VAR objects until the chapters dealing with Advanced Procedures.

Accessing global objects is convenient and easy. Unfortunately, as you've probably learned when studying high level language programming, accessing global objects makes your programs harder to read, understand, and maintain. Like most introductory programming texts, this text will discourage the use of global variables within procedures. Accessing global variables within a procedure is sometimes the best solution to a given problem. However, such (legitimate) access typically occurs only in advanced programs involving multiple threads of execution or in other complex systems. Since it is unlikely you would be writing such code at this point, it is equally unlikely that you will absolutely need to access global variables in your procedures so you should carefully consider your options before accessing global variables within your procedures[4].

Declaring local variables in your procedures is very easy, you use the same declaration sections as the main program: STATIC, READONLY, STORAGE, and VAR. The same rules and syntax for the declaration sections and the access of variables you declare in these sections applies in your procedure. The following example code demonstrates the declaration of a local variable.

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

4. Note that this argument against accessing global variables does not apply to other global symbols. It is perfectly reasonable to access global constants, types, procedures, and other objects in your programs.

```
    // Simple procedure that displays 0..9 using
    // a local variable as a loop control variable.

    procedure CntTo10;
    var
        i: int32;

    begin CntTo10;

        for( mov( 0, i ); i < 10; inc( i )) do

            stdout.put( “i=”, i, nl );

        endfor;

    end CntTo10;


begin demoLocalVars;

    CntTo10();

end demoLocalVars;
```

Program 8.6 Example of a Local Variable in a Procedure

Local variables you declare in a procedure are accessible only within that procedure[5]. Therefore, the variable *i* in procedure *CntTo10* in Program 8.6 is not accessible in the main program.

HLA relaxes, somewhat, the rule that identifiers must be unique in a program for local variables. In an HLA program, all identifiers must be unique within a given *scope*. Therefore, all global names must be unique with respect to one another. Similarly, all local variables within a given procedure must have unique names *but only with respect to other local symbols in that procedure*. In particular, a local name may be the same as a global name. When this occurs, HLA creates two separate variables for the two objects. Within the scope of the procedure any reference to the common name accesses the local variable; outside that procedure, any reference to the common name references the global identifier. Although the quality of the resultant code is questionable, it is perfectly legal to have a global identifier named *MyVar* with the same local name in two or more different procedures. The procedures each have their own local variant of the object which is independent of *MyVar* in the main program. Program 8.7 provides an example of an HLA program that demonstrates this feature.

```
program demoLocalVars2;
#include( “stdlib.hhf” );

static
    i:  uns32 := 10;
    j:  uns32 := 20;


    // The following procedure declares “i” and “j”
    // as local variables, so it does not have access
    // to the global variables by the same name.
```

5. Strictly speaking, this is not true. The chapter on Advanced Procedures will present an exception.

```
    procedure First;
    var
        i: int32;
        j:uns32;

    begin First;

        mov( 10, j );
        for( mov( 0, i ); i < 10; inc( i )) do

            stdout.put( “i=”, i,” j=”, j, nl );
            dec( j );

        endfor;

    end First;

    // This procedure declares only an “i” variable.
    // It cannot access the value of the global “i”
    // variable but it can access the value of the
    // global “j” object since it does not provide
    // a local variant of “j”.

    procedure Second;
    var
        i:uns32;

    begin Second;

        mov( 10, j );
        for( mov( 0, i ); i < 10; inc( i )) do

            stdout.put( “i=”, i,” j=”, j, nl );
            dec( j );

        endfor;

    end Second;


begin demoLocalVars2;

    First();
    Second();

    // Since the calls to First and Second have not
    // modified variable “i”, the following statement
    // should print “i=10”.  However, since the Second
    // procedure manipulated global variable “j”, this
    // code will print “j=0” rather than “j=20”.

    stdout.put(  “i=”, i, “ j=”, j, nl );

end demoLocalVars2;
```

Program 8.7 Local Variables Need Not Have Globally Unique Names

There are good and bad points to be made about reusing global names within a procedure. On the one hand, there is the potential for confusion. If you use a name like *ProfitsThisYear* as a global symbol and you reuse that name within a procedure, someone reading the procedure might think that the procedure refers to the global symbol rather than the local symbol. On the other hand, simple names like *i, j*, and *k* are nearly meaningless (almost everyone expects the program to use them as loop control variables or for other local uses), so reusing these names as local objects is probably a good idea. From a software engineering perspective, it is probably a good idea to keep all variables names that have a very specific meaning (like *ProfitsThisYear*) unique throughout your program. General names, that have a nebulous meaning (like *index, counter*, and names like *i, j,* or *k*) will probably be okay to reuse as global variables

There is one last point to make about the scope of identifiers in an HLA program: variables in separate procedures (that is, two procedures where one procedure is not declared in the declaration section of the second procedure) are separate, even if they have the same name. The *First* and *Second* procedures in Program 8.7, for example, share the same name (*i*) for a local variable. However, the *i* in *First* is a completely different variable than the *i* in *Second*.

The second major attribute that differentiates (certain) local variables from global variables is *lifetime*. The lifetime of a variable spans from the point the program first allocates storage for a variable to the point the program deallocates the storage for that variable. Note that lifetime is a dynamic attribute (controlled at run time) whereas scope is a static attribute (controlled at compile time). In particular, a variable can actually have several lifetimes if the program repeatedly allocates and then deallocates the storage for that variable.

Global variables always have a single lifetime that spans from the moment the main program first begins execution to the point the main program terminates. Likewise, all static objects have a single lifetime that spans the execution of the program (remember, static objects are those you declare in the STATIC, READONLY, or STORAGE sections). This is true even for procedures. So there is no difference between the lifetime of a local static object and the lifetime of a global static object. Variables you declare in the VAR section, however, are a different matter. VAR objects use *automatic storage allocation*. Automatic storage allocation means that the procedure automatically allocates storage for a local variable upon entry into a procedure. Similarly, the program deallocates storage for automatic objects when the procedure returns to its caller. Therefore, the lifetime of an automatic object is from the point the procedure is first called to the point it returns to its caller.

Perhaps the most important thing to note about automatic variables is that you cannot expect them to maintain their values between calls to the procedure. Once the procedure returns to its caller, the storage for the automatic variable is lost and, therefore, the value is lost as well. Therefore, *you must always assume that a local VAR object is uninitialized upon entry into a procedure*; even if you know you've called the procedure before and the previous procedure invocation initialized that variable. Whatever value the last call stored into the variable was lost when the procedure returned to its caller. If you need to maintain the value of a variable between calls to a procedure, you should use one of the static variable declaration types.

Given that automatic variables cannot maintain their values across procedure calls, you might wonder why you would want to use them at all. However, there are several benefits to automatic variables that static variables do not have. The biggest disadvantage to static variables is that they consume memory even when the (only) procedure that references them is not running. Automatic variables, on the other hand, only consume storage while there associated procedure is executing. Upon return, the procedure returns any automatic storage it allocated back to the system for reuse by other procedures. You'll see some additional advantages to automatic variables later in this chapter.

8.6 Other Local and Global Symbol Types

As mentioned in the previous section, HLA lets you declare constants, values, types, and anything else legal in the main program's declaration section within a procedure's declaration section. The same rules for scope apply to these identifiers. Therefore, you can reuse constant names, procedure names, type names, etc. in local declarations (although this is almost always a bad idea).

Referencing global constants, values, and types, does not present the same software engineering problems that occur when you reference global variables. The problem with referencing global variable is that a procedure can change the value of a global variable in a non-obvious way. This makes programs more difficult to read, understand, and maintain since you can't often tell that a procedure is modifying memory by looking only at the call to that procedure. Constants, values, types, and other non-variable objects, don't suffer from this problem because you cannot change them at run-time. Therefore, the pressure to avoid global objects at nearly all costs doesn't apply to non-variable objects.

Having said that it's okay to access global constants, types, etc., it's also worth pointing out that you should declare these objects locally within a procedure if the only place your program references such objects is within that procedure. Doing so will make your programs a little easier to read since the person reading your code won't have to search all over the place for the symbol's definition.

8.7 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. In straight assembly language, passing parameters can be a real chore. Fortunately, HLA provides a HLL-like syntax for procedure declarations and for procedure calls involving parameters. This chapter will present HLA's HLL parameter syntax. Later chapters on Intermediate Procedures and Advanced Procedures will deal with the low-level mechanisms for passing parameters in pure assembly code.

The first thing to consider when discussing parameters is *how* we pass them to a procedure. If you are familiar with Pascal or C/C++ you've probably seen two ways to pass parameters: pass by value and pass by reference. HLA certainly supports these two parameter passing mechanisms. However, HLA also supports pass by value/result, pass by result, pass by name, and pass by lazy evaluation. Of course, HLA is assembly language so it is possible to pass parameters in HLA using any scheme you can dream up (at least, any scheme that is possible at all on the CPU). However, HLA provides special HLL syntax for pass by value, reference, value/result, result, name, and lazy evaluation.

Because pass by value/result, result, name, and lazy evaluation are somewhat advanced, this chapter will not deal with those parameter passing mechanisms. If you're interested in learning more about these parameter passing schemes, see the chapters on Intermediate and Advanced Procedures.

Another concern you will face when dealing with parameters is *where* you pass them. There are lots of different places to pass parameters; the chapter on Intermediate Procedures will consider these places in greater detail. In this chapter, since we're using HLA's HLL syntax for declaring and calling procedures, we'll wind up passing procedure parameters on the stack. You don't really need to concern yourself with the details since HLA abstracts them away for you; however, do keep in mind that procedure calls and procedure parameters make use of the stack. Therefore, something you push on the stack immediately before a procedure call is not going to be immediately on the top of the stack upon entry into the procedure.

8.7.1 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 them. In HLA the idea of a pass by value parameter being an input only parameter makes a lot of sense. Given the HLA procedure call:

```
CallProc(I);
```

If you pass *I* by value, then *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 records by value is very inefficient (since you must create and pass a copy of the object to the procedure).

HLA, like Pascal and C/C++, passes parameters by value unless you specify otherwise. Here's what a typical function looks like with a single pass by value parameter:

```
    procedure PrintNSpaces( N:uns32 );
    begin PrintNSpaces;

        push( ecx );
        mov( N, ecx );
        repeat

            stdout.put( ' ' );  // Print 1 of N spaces.
            dec( ecx );         // Count off N spaces.

        until( ecx = 0 );
        pop( ecx );

    end PrintNSpaces;
```

The parameter *N* in *PrintNSpaces* is known as a formal parameter. Anywhere the name *N* appears in the body of the procedure the program references the value passed through *N* by the caller.

The calling sequence for PrintNSpaces can be any of the following:

```
              PrintNSpaces( constant );
              PrintNSpaces( reg32 );
              PrintNSpaces( uns32_variable );
```

Here are some concrete examples of calls to *PrintNSpaces*:

```
    PrintNSpaces( 40 );
    PrintNSpaces( EAX );
    PrintNSpaces( SpacesToPrint );
```

The parameter in the calls to *PrintNSpaces* is known as an *actual parameter.* In the examples above, 40, EAX, and *SpacesToPrint* are the actual parameters.

Note that pass by value parameters behave exactly like local variables you declare in the VAR section with the single exception that the procedure's caller initializes these local variables before it passes control to the procedure.

HLA uses positional parameter notation just like most high level languages. Therefore, if you need to pass more than one parameter, HLA will associate the actual parameters with the formal parameters by their position in the parameter list. The following *PrintNChars* procedure demonstrates a simple procedure that has two parameters.

```
    procedure PrintNChars( N:uns32; c:char );
    begin PrintNChars;

        push( ecx );
        mov( N, ecx );
        repeat

            stdout.put( c );    // Print 1 of N characters.
            dec( ecx );         // Count off N characters.

        until( ecx = 0 );
        pop( ecx );

    end PrintNChars;
```

The following is an invocation of the PrintNChars procedure that will print 20 asterisk characters:

```
PrintNChars( 20, ‘*’ );
```

Note that HLA uses semicolons to separate the formal parameters in the procedure declaration and it uses commas to separate the actual parameters in the procedure invocation (Pascal programmers should be comfortable with this notation). Also note that each HLA formal parameter declaration takes the following form:

parameter_identifier : *type_identifier*

In particular, note that the parameter type has to be an identifier. None of the following are legal parameter declarations because the data type is not a single identifier:

```
PtrVar: pointer to uns32
ArrayVar: uns32[10]
recordVar: record i:int32; u:uns32; endrecord
DynArray: array.dArray( uns32, 2 )
```

However, don’t get the impression that you cannot pass pointer, array, record, or dynamic array variables as parameters. The trick is to declare a data type for each of these types in the TYPE section. Then you can use a single identifier as the type in the parameter declaration. The following code fragment demonstrates how to do this with the four data types above:

```
type
    uPtr:      pointer to uns32;
    uArray10:  uns32[10];
    recType:   record i:int32; u:uns32; endrecord
    dType:     array.dArray( uns32, 2 );

    procedure FancyParms
    (
        PtrVar: uPtr;
        ArrayVar:uArray10;
        recordVar:recType;
        DynArray: dtype
    );
    begin FancyParms;
        .
        .
        .
    end FancyParms;
```

By default, HLA assumes that you intend to pass a parameter by value. HLA also lets you explicitly state that a parameter is a value parameter by prefacing the formal parameter declaration with the VAL keyword. The following is a version of the *PrintNSpaces* procedure that explicitly states that *N* is a pass by value parameter:

```
procedure PrintNSpaces( val N:uns32 );
begin PrintNSpaces;

    push( ecx );
    mov( N, ecx );
    repeat

        stdout.put( ' ' );  // Print 1 of N spaces.
        dec( ecx );         // Count off N spaces.

    until( ecx = 0 );
    pop( ecx );

end PrintNSpaces;
```

Explicitly stating that a parameter is a pass by value parameter is a good idea if you have multiple parameters in the same procedure declaration that use different passing mechanisms.

When you pass a parameter by value and call the procedure using the HLA high level language syntax, HLA will automatically generate code that will make a copy of the actual parameter's value and copy this data into the local storage for that parameter (i.e., the formal parameter). For small objects pass by value is probably the most efficient way to pass a parameter. For large objects, however, HLA must generate code that copies each and every byte of the actual parameter into the formal parameter. For large arrays and records this can be a very expensive operation[6]. Unless you have specific semantic concerns that require you to pass an array or record by value, you should use pass by reference or some other parameter passing mechanism for arrays and records.

When passing parameters to a procedure, HLA checks the type of each actual parameter and compares this type to the corresponding formal parameter. If the types do not agree, HLA then checks to see if either the actual or formal parameter is a byte, word, or dword object and the other parameter is one, two, or four bytes in length (respectively). If the actual parameter does not satisfy either of these conditions, HLA reports a parameter type mismatch error. If, for some reason, you need to pass a parameter to a procedure using a different type than the procedure calls for, you can always use the HLA type coercion operator to override the type of the actual parameter.

8.7.2 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.

To declare a pass by reference parameter you must preface the formal parameter declaration with the VAR keyword. The following code fragment demonstrates this:

```
procedure UsePassByReference( var PBRvar: int32 );
begin UsePassByReference;
    .
    .
    .
end UsePassByReference;
```

Calling a procedure with a pass by reference parameter uses the same syntax as pass by value except that the parameter has to be a memory location; it cannot be a constant or a register. Furthermore, the type of the memory location must exactly match the type of the formal parameter. The following are legal calls to the procedure above (assuming *i32* is an *int32* variable):

```
UsePassByReference( i32 );
UsePassByReference( (type int32 [ebx] ) );
```

The following are all illegal *UsePassbyReference* invocations (assumption: *charVar* is of type *char*):

```
UsePassByReference( 40 );           // Constants are illegal.
UsePassByReference( EAX );          // Bare registers are illegal.
UsePassByReference( charVar );      // Actual parameter type must match
                                    //  the formal parameter type.
```

Unlike the high level languages Pascal and C++, HLA does not completely hide the fact that you are passing a pointer rather than a value. In a procedure invocation, HLA will automatically compute the

6. Note to C/C++ programmers: HLA does not automatically pass arrays by reference. If you specify an array type as a formal parameter, HLA will emit code that makes a copy of each and every byte of that array when you call the associated procedure.

address of a variable and pass that address to the procedure. Within the procedure itself, however, you cannot treat the variable like a value parameter (as you could in most HLLs). Instead, you treat the parameter as a dword variable containing a pointer to the specified data. You must explicitly dereference this pointer when accessing the parameter's value. The following example provides a simple demonstration of this:

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

var
    i:  int32;
    j:  int32;

    procedure pbr( var a:int32; var b:int32 );
    const
        aa: text := “(type int32 [ebx])”;
        bb: text := “(type int32 [ebx])”;

    begin pbr;

        push( eax );
        push( ebx );        // Need to use EBX to dereference a and b.

        // a = -1;

        mov( a, ebx );      // Get ptr to the “a” variable.
        mov( -1, aa );      // Store -1 into the “a” parameter.

        // b = -2;

        mov( b, ebx );      // Get ptr to the “b” variable.
        mov( -2, bb );      // Store -2 into the “b” parameter.

        // Print the sum of a+b.
        // Note that ebx currently contains a pointer to “b”.

        mov( bb, eax );
        mov( a, ebx );      // Get ptr to “a” variable.
        add( aa, eax );
        stdout.put( “a+b=”, (type int32 eax), nl );

    end pbr;

begin PassByRefDemo;

    // Give i and j some initial values so
    // we can see that pass by reference will
    // overwrite these values.

    mov( 50, i );
    mov( 25, j );

    // Call pbr passing i and j by reference

    pbr( i, j );

    // Display the results returned by pbr.

    stdout.put
    (
        “i=  “, i, nl,
        “j=  “, j, nl
    );

end PassByRefDemo;
```

Program 8.8 Accessing Pass by Reference Parameters

Passing parameters by reference can produce some peculiar results in some rare circumstances. Consider the *pbr* procedure in Program 8.8. Were you to modify the call in the main program to be “pbr(i,i)” rather than “pbr(i,j);” the program would produce the following non-intuitive output:

```
a+b=-4
i=  -2;
j=  25;
```

The reason this code displays “a+b=-4” rather than the expected “a+b=-3” is because the “pbr(i,i);” call passes the same actual parameter for *a* and *b*. As a result, the *a* and *b* reference parameters both contain a pointer to the same memory location- that of the variable *i*. In this case, *a* and *b* are *aliases* of one another. Therefore, when the code stores -2 at the location pointed at by *b*, it overwrites the -1 stored earlier at the location pointed at by *a*. When the program fetches the value pointed at by *a* and *b* to compute their sum, both *a* and *b* point at the same value, which is -2. Summing -2 + -2 produces the -4 result that the program displays. This non-intuitive behavior is possible anytime you encounter aliases in a program. Passing the same variable as two different parameters probably isn’t very common. But you could also create an alias if a procedure references a global variable and you pass that same global variable by reference to the procedure (this is a good example of yet one more reason why you should avoid referencing global variables in a procedure).

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 since it typically requires at least two instructions. 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. Of course, you’d probably need to access elements of that large data structure (e.g., an array) using a pointer, so very little efficiency is lost when you pass large arrays by reference.

8.8 Functions and Function Results

Functions are procedures that return a result. In assembly language, there are very few syntactical differences between a procedure and a function which is why HLA doesn’t provide a specific declaration for a function. Nevertheless, although there is very little *syntactical* difference between assembly procedures and functions, there are considerable *semantic* differences. That is, although you can declare them the same way in HLA, you use them differently.

Procedures are a sequence of machine instructions that fulfill some activity. The end result of the execution of a procedure is the accomplishment of that activity. Functions, on the other hand, execute a sequence of machine instructions specifically to compute some value to return to the caller. Of course, a function can perform some activity as well and procedures can undoubtedly compute some values, but the main difference is that the purpose of a function is to return some computed result; procedures don’t have this requirement.

A good example of a procedure is the *stdout.puti32* procedure. This procedure requires a single *int32* parameter. The purpose of this procedure is to print the decimal conversion of this integer value to the standard output device. Note that *stdout.puti32* doesn’t return any kind of value that is usable by the calling program.

A good example of a function is the *cs.member* function. This function expects two parameters: the first is a character value and the second is a character set value. This function returns true (1) in EAX if the character is a member of the specified character set. It returns false if the character parameter is not a member of the character set.

Logically, the fact that *cs.member* returns a usable value to the calling code (in EAX) while *stdout.puti32* does not is a good example of the main difference between a function and a procedure. So, in general, a procedure becomes a function by virtue of the fact that you explicitly decide to return a value somewhere upon procedure return. No special syntax is needed to declare and use a function. You still write the code as a procedure.

8.8.1 Returning Function Results

The 80x86's registers are the most popular place to return function results. The *cs.member* routine in the HLA Standard Library is a good example of a function that returns a value in one of the CPU's registers. It returns true (1) or false (0) in the EAX register. By convention, programmers try to return eight, sixteen, and thirty-two bit (non-real) results in the AL, AX, and EAX registers, respectively[7]. For example, this is where most high level languages return these types of results.

Of course, there is nothing particularly sacred about the AL/AX/EAX register. You could return function results in any register if it is more convenient to do so. However, if you don't have a good reason for not using AL/AX/EAX, then you should follow the convention. Doing so will help others understand your code better since they will generally assume that your functions return small results in the AL/AX/EAX register set.

If you need to return a function result that is larger than 32 bits, you obviously must return it somewhere besides in EAX (which can hold values 32 bits or less). For values slightly larger than 32 bits (e.g., 64 bits or maybe even as many as 128 bits) you can split the result into pieces and return those parts in two or more registers. For example, it is very common to see programs returning 64-bit values in the EDX:EAX register pair (e.g., the HLA Standard Library *stdin.geti64* function returns a 64-bit integer in the EDX:EAX register pair).

If you need to return a really large object as a function result, say an array of 1,000 elements, you obviously are not going to be able to return the function result in the registers. There are two common ways to deal with really large function return results: either pass the return value as a reference parameter or allocate storage on the heap (using *malloc*) for the object and return a pointer to it in a 32-bit register. Of course, if you return a pointer to storage you've allocated on the heap, the calling program must free this storage when it is done with it.

8.8.2 Instruction Composition in HLA

Several HLA Standard Library functions allow you to call them as operands of other instructions. For example, consider the following code fragment:

```
if( cs.member( al, {'a'..'z'}) ) then
    .
    .
    .
endif;
```

As your high level language experience (and HLA experience) should suggest, this code calls the *cs.member* function to check to see if the character in AL is a lower case alphabetic character. If the *cs.member* function returns true then this code fragment executes the then section of the IF statement; however, if *cs.member* returns false, this code fragment skips the IF..THEN body. There is nothing spectacular here

7. In the next chapter you'll see where most programmers return real results.

except for the fact that HLA doesn't support function calls as boolean expressions in the IF statement (look back at Chapter Two in Volume One to see the complete set of allowable expressions). How then, does this program compile and run producing the intuitive results?

The very next section will describe how you can tell HLA that you want to use a function call in a boolean expression. However, to understand how this works, you need to first learn about *instruction composition* in HLA.

Instruction composition lets you use one instruction as the operand of another. For example, consider the MOV instruction. It has two operands, a source operand and a destination operand. Instruction composition lets you substitute a valid 80x86 machine instruction for either (or both) operands. The following is a simple example:

```
mov( mov( 0, eax ), ebx );
```

Of course the immediate question is "what does this mean?" To understand what is going on, you must first realize that most instructions "return" a value to the compiler while they are being compiled. For most instructions, the value they "return" is their destination operand. Therefore, "mov(0, eax);" returns the string "eax" to the compiler during compilation since EAX is the destination operand. Most of the time, specifically when an instruction appears on a line by itself, the compiler ignores the string result the instruction returns. However, HLA uses this string result whenever you supply an instruction in place of some operand; specifically, HLA uses that string in place of the instruction as the operand. Therefore, the MOV instruction above is equivalent to the following two instruction sequence:

```
mov( 0, eax );     // HLA compiles interior instructions first.
mov( eax, ebx );
```

When processing composed instructions (that is, instruction sequences that have other instructions as operands), HLA always works in an " left-to-right then depth-first (inside-out)" manner. To make sense of this, consider the following instructions:

```
add( sub( mov( i, eax ), mov( j, ebx )), mov( k, ecx ));
```

To interpret what is happening here, begin with the source operand. It consists of the following:

```
sub( mov( i, eax ), mov( j, ebx ))
```

The source operand for this instruction is "mov(i, eax)" and this instruction does not have any composition, so HLA emits this instruction and returns its destination operand (EAX) for use as the source to the SUB instruction. This effectively gives us the following:

```
sub( eax, mov( j, ebx ))
```

Now HLA compiles the instruction that appears as the destination operand ("mov(j, ebx)") and returns its destination operand (EBX) to substitute for this MOV in the SUB instruction. This yields the following:

```
sub( eax, ebx )
```

This is a complete instruction, without composition, that HLA can compile. So it compiles this instruction and returns its destination operand (EBX) as the string result to substitute for the SUB in the original ADD instruction. So the original ADD instruction now becomes:

```
add( ebx, mov(i, ecx ));
```

HLA next compiles the MOV instruction appearing in the destination operand. It returns its destination operand as a string that HLA substitutes for the MOV, finally yielding the simple instruction:

```
add( ebx, ecx );
```

The compilation of the original ADD instruction, therefore, yields the following instruction sequence:

```
mov( i, eax );
mov( j, ebx );
sub( eax, ebx );
```

```
mov( k, ecx );
add( ebx, ecx );
```

Whew! It's rather difficult to look at the original instruction and easily see that this sequence is the result. As you can easily see in this example, *overzealous use of instruction composition can produce nearly unreadable programs*. You should be very careful about using instruction composition in your programs. With only a few exceptions, writing a composed instruction sequence makes your program harder to read.

Note that the excessive use of instruction composition may make errors in your program difficult to decipher. Consider the following HLA statement:

```
add( mov( eax, i ), mov( ebx, j ) );
```

This instruction composition yields the 80x86 instruction sequence:

```
mov( eax, i );
mov( ebx, j );
add( i, j );
```

Of course, the compiler will complain that you're attempting to add one memory location to another. However, the instruction composition effectively masks this fact and makes it difficult to comprehend the cause of the error message. Moral of the story: avoid using instruction composition unless it really makes your program easier to read. The few examples in this section demonstrate how *not* to use instruction composition.

There are two main areas where using instruction composition can help make your programs more readable. The first is in HLA's high level language control structures. The other is in procedure parameters. Although instruction composition is useful in these two cases (and probably a few others as well), this doesn't give you a license to use extremely convoluted instructions like the ADD instruction in the previous example. Instead, most of the time you will use a single instruction or a function call in place of a single operand in a high level language boolean expression or in a procedure/function parameter.

While we're on the subject, exactly what does a procedure call return as the string that HLA substitutes for the call in an instruction composition? For that matter, what do statements like IF..ENDIF return? How about instructions that don't have a destination operand? Well, function return results are the subject of the very next section so you'll read about that in a few moments. As for all the other statements and instructions, you should check out the HLA reference manual. It lists each instruction and its "RETURNS" value. The "RETURNS" value is the string that HLA will substitute for the instruction when it appears as the operand to another instruction. Note that many HLA statements and instructions return the empty string as their "RETURNS" value (by default, so do procedure calls). If an instruction returns the empty string as its composition value, then HLA will report an error if you attempt to use it as the operand of another instruction. For example, the IF..ENDIF statement returns the empty string as its "RETURNS" value, so you may not bury an IF..ENDIF inside another instruction.

8.8.3 The HLA RETURNS Option in Procedures

HLA procedure declarations allow a special option that specifies the string to use when a procedure invocation appears as the operand of another instruction: the RETURNS option. The syntax for a procedure declaration with the RETURNS option is as follows:

```
procedure ProcName ( optional parameters );  RETURNS( string_constant );
    << Local declarations >>
begin ProcName;
    << procedure statements >>
end ProcName;
```

If the RETURNS option is not present, HLA associates the empty string with the RETURNS value for the procedure. This effectively makes it illegal to use that procedure invocation as the operand to another instruction.

The RETURNS option requires a single string parameter surrounded by parentheses. This must be a string constant[8]. HLA will substitute this string constant for the procedure call if it ever appears as the operand of another instruction. Typically this string constant is a register name; however, any text that would be legal as an instruction operand is okay here. For example, you could specify memory address or constants. For purposes of clarity, you should always specify the location of a function's return value in the RETURNS parameter.

As an example, consider the following boolean function that returns true or false in the EAX register if the single character parameter is an alphabetic character[9]:

```
procedure IsAlphabeticChar( c:char ); RETURNS( "EAX" );
begin IsAlphabeticChar;

    // Note that cs.member returns true/false in EAX

    cs.member( c, {'a'..'z', 'A'..'Z'} );

end IsAlphabeticChar;
```

Once you tack the RETURNS option on the end of this procedure declaration you can legally use a call to *IsAlphabeticChar* as an operand to other HLA statements and instructions:

```
    mov( IsAlphabeticChar( al ), EBX );
        .
        .
        .
    if( IsAlphabeticChar( ch ) ) then
        .
        .
        .
    endif;
```

The last example above demonstrates that, via the RETURNS option, you can embed calls to your own functions in the boolean expression field of various HLA statements. Note that the code above is equivalent to

```
    IsAlphabeticChar( ch );
    if( EAX ) then
        .
        .
        .
    endif;
```

Not all HLA high level language statements expand composed instructions before the statement. For example, consider the following WHILE statement:

```
    while( IsAlphabeticChar( ch ) ) do
        .
        .
        .
    endwhile;
```

This code does not expand to the following:

```
    IsAlphabeticChar( ch );
    while( EAX ) do
        .
        .
        .
```

8. Do note, however, that it doesn't have to be a string literal constant. A CONST string identifier or even a constant string expression is legal here.

9. Before you run off and actually use this function in your own programs, note that the HLA Standard Library provides the *char.isAlpha* function that provides this test. See the HLA documentation for more details.

```
    endwhile;
```

Instead, the call to *IsAlphabeticChar* expands inside the WHILE's boolean expression so that the program calls this function on each iteration of the loop.

You should exercise caution when entering the RETURNS parameter. HLA does not check the syntax of the string parameter when it is compiling the procedure declaration (other than to verify that it is a string constant). Instead, HLA checks the syntax when it replaces the function call with the RETURNS string. So if you had specified "EAZ" instead of "EAX" as the RETURNS parameter for *IsAlphabeticChar* in the previous examples, HLA would not have reported an error until you actually used *IsAlphabeticChar* as an operand. Then of course, HLA complains about the illegal operand and it's not at all clear what the problem is by looking at the *IsAlphabeticChar* invocation. So take special care not to introduce typographical errors in the RETURNS string; figuring out such errors later can be very difficult.

8.9 Side Effects

A *side effect* is any computation or operation by a procedure that isn't the primary purpose of that procedure. For example, if you elect not to preserve all affected registers within a procedure, the modification of those registers is a side effect of that procedure. Side effect programming, that is, the practice of using a procedure's side effects, is very dangerous. All too often a programmer will rely on a side effect of a procedure. Later modifications may change the side effect, invalidating all code relying on that side effect. This can make your programs hard to debug and maintain. Therefore, you should avoid side effect programming.

Perhaps some examples of side effect programming will help enlighten you to the difficulties you may encounter. The following procedure zeros out an array. For efficiency reasons, it makes the caller responsible for preserving necessary registers. As a result, one side effect of this procedure is that the EBX and ECX registers are modified. In particular, the ECX register contains zero upon return.

```
procedure ClrArray;
begin ClrArray;

    lea( ebx, array );
    mov( 32, ecx );
    while( ecx > 0 ) do

        mov( 0, (type dword [ebx]));
        add( 4, ebx );
        dec( ecx );

    endwhile;

end ClrArray;
```

If your code expects ECX to contain zero after the execution of this subroutine, you would be relying on a side effect of the *ClrArray* procedure. The main purpose behind this code is zeroing out an array, not setting the ECX register to zero. Later, if you modify the *ClrArray* procedure to the following, your code that depends upon ECX containing zero would no longer work properly:

```
procedure ClrArray;
begin ClrArray;

    mov( 0, ebx );
    while( ebx < 32 ) do

        mov( 0, array[ebx*4] );
        inc( ebx );
    endwhile;

end ClrArray;
```

So how can you avoid the pitfalls of side effect programming in your procedures? By carefully structuring your code and paying close attention to exactly how your calling code and the subservient procedures interface with one another. These rules can help you avoid problems with side effect programming:

- Always properly document the input and output conditions of a procedure. Never rely on any other entry or exit conditions other than these documented operations.
- Partition your procedures so that they compute a single value or execute a single operation. Subroutines that do two or more tasks are, by definition, producing side effects unless every invocation of that subroutine requires all the computations and operations.
- When updating the code in a procedure, make sure that it still obeys the entry and exit conditions. If not, either modify the program so that it does or update the documentation for that procedure to reflect the new entry and exit conditions.
- Avoid passing information between routines in the CPU's flag register. Passing an error status in the carry flag is about as far as you should ever go. Too many instructions affect the flags and it's too easy to foul up a return sequence so that an important flag is modified on return.
- Always save and restore all registers a procedure modifies.
- Avoid passing parameters and function results in global variables.
- Avoid passing parameters by reference (with the intent of modifying them for use by the calling code).

These rules, like all other rules, were meant to be broken. Good programming practices are often sacrificed on the altar of efficiency. There is nothing wrong with breaking these rules as often as you feel necessary. However, your code will be difficult to debug and maintain if you violate these rules often. But such is the price of efficiency[10]. Until you gain enough experience to make a judicious choice about the use of side effects in your programs, you should avoid them. More often than not, the use of a side effect will cause more problems than it solves.

8.10 Recursion

Recursion occurs when a procedure calls itself. The following, for example, is a recursive procedure:

```
procedure Recursive;
begin Recursive;

    Recursive();

end Recursive;
```

Of course, the CPU will never return from this procedure. Upon entry into *Recursive*, this procedure will immediately call itself again and control will never pass to the end of the procedure. In this particular case, run away recursion results in an infinite loop[11].

Like a looping structure, recursion requires a termination condition in order to stop infinite recursion. *Recursive* could be rewritten with a termination condition as follows:

```
procedure Recursive;
begin Recursive;

    dec( eax );
    if( @nz ) then
```

10. This is not just a snide remark. Expert programmers who have to wring the last bit of performance out of a section of code often resort to poor programming practices in order to achieve their goals. They are prepared, however, to deal with the problems that are often encountered in such situations and they are a lot more careful when dealing with such code.
11. Well, not really infinite. The stack will overflow and Windows or Linux will raise an exception at that point.

```
            Recursive();

    endif;

end Recursive;
```

This modification to the routine causes *Recursive* to call itself the number of times appearing in the EAX register. On each call, *Recursive* decrements the EAX register by one and calls itself again. Eventually, *Recursive* decrements EAX to zero and returns. Once this happens, each successive call returns back to *Recursive* until control returns to the original call to *Recursive*.

So far, however, there hasn't been a real need for recursion. After all, you could efficiently code this procedure as follows:

```
procedure Recursive;
begin Recursive;

    repeat

        dec( eax );

    until( @z );

end Recursive;
```

Both examples would repeat the body of the procedure the number of times passed in the EAX register[12]. As it turns out, there are only a few recursive algorithms that you cannot implement in an iterative fashion. However, many recursively implemented algorithms are more efficient than their iterative counterparts and most of the time the recursive form of the algorithm is much easier to understand.

The quicksort algorithm is probably the most famous algorithm that usually appears in recursive form (see a "Data Structures and Algorithms" textbook for a discussion of this algorithm). An HLA implementation of this algorithm follows:

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

type
    ArrayType:  uns32[ 10 ];

static
    theArray:   ArrayType := [1,10,2,9,3,8,4,7,5,6];


    procedure quicksort( var a:ArrayType; Low:int32; High: int32 );
    const
        i:      text := "(type int32 edi)";
        j:      text := "(type int32 esi)";
        Middle: text := "(type uns32 edx)";
        ary:    text := "[ebx]";

    begin quicksort;

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

12. Although the latter version will do it considerably faster since it doesn't have the overhead of the CALL/RET instructions.

```
push( esi );
push( edi );

mov( a, ebx );      // Load BASE address of “a” into EBX

mov( Low, edi);     // i := Low;
mov( High, esi );   // j := High;

// Compute a pivotal element by selecting the
// physical middle element of the array.

mov( i, eax );
add( j, eax );
shr( 1, eax );
mov( ary[eax*4], Middle );  // Put middle value in EDX

// Repeat until the EDI and ESI indicies cross one
// another (EDI works from the start towards the end
// of the array, ESI works from the end towards the
// start of the array).

repeat

    // Scan from the start of the array forward
    // looking for the first element greater or equal
    // to the middle element).

    while( Middle > ary[i*4] ) do

        inc( i );

    endwhile;

    // Scan from the end of the array backwards looking
    // for the first element that is less than or equal
    // to the middle element.

    while( Middle < ary[j*4] ) do

        dec( j );

    endwhile;

    // If we’ve stopped before the two pointers have
    // passed over one another, then we’ve got two
    // elements that are out of order with respect
    // to the middle element.  So swap these two elements.

    if( i <= j ) then

        mov( ary[i*4], eax );
        mov( ary[j*4], ecx );
        mov( eax, ary[j*4] );
        mov( ecx, ary[i*4] );
        inc( i );
        dec( j );

    endif;

until( i > j );
```

```
            // We have just placed all elements in the array in
            // their correct positions with respect to the middle
            // element of the array.  So all elements at indicies
            // greater than the middle element are also numerically
            // greater than this element.  Likewise, elements at
            // indicies less than the middle (pivotal) element are
            // now less than that element.  Unfortunately, the
            // two halves of the array on either side of the pivotal
            // element are not yet sorted.  Call quicksort recursively
            // to sort these two halves if they have more than one
            // element in them (if they have zero or one elements, then
            // they are already sorted).

            if( Low < j ) then

                quicksort( a, Low, j );

            endif;
            if( i < High ) then

                quicksort( a, i, High );

            endif;

            pop( edi );
            pop( esi );
            pop( edx );
            pop( ecx );
            pop( ebx );
            pop( eax );

        end quicksort;

    begin QSDemo;

        stdout.put( "Data before sorting: " nl );
        for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

            stdout.put( theArray[ebx*4]:5 );

        endfor;
        stdout.newln();

        quicksort( theArray, 0, 9 );

        stdout.put( "Data after sorting: " nl );
        for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

            stdout.put( theArray[ebx*4]:5 );

        endfor;
        stdout.newln();

    end QSDemo;
```

Program 8.9 Recursive Quicksort Program

Note that this quicksort procedure uses registers for all non-parameter local variables. Also note how Quicksort uses TEXT constant definitions to provide more readable names for the registers. This technique can often make an algorithm easier to read; however, one must take care when using this trick not to forget that those registers are being used.

8.11 Forward Procedures

As a general rule HLA requires that you declare all symbols before their first use in a program[13]. Therefore, you must define all procedures before their first call. There are two reasons this isn't always practical: mutual recursion (two procedures call each other) and source code organization (you prefer to place a procedure in your code after the point you've first called it). Fortunately, HLA lets you use a *forward procedure definition* to declare a procedure *prototype*. Forward declarations let you define a procedure before you actually supply the code for that procedure.

A forward procedure declaration is a familiar procedure declaration that uses the reserved word FORWARD in place of the procedure's declaration section and body. The following is a forward declaration for the quicksort procedure appearing in the last section:

```
procedure quicksort( var a:ArrayType; Low:int32; High: int32 ); forward;
```

A forward declaration in an HLA program is a promise to the compiler that the actual procedure declaration will appear, exactly as stated in the forward declaration, at a later point in the source code. "Exactly as stated" means exactly that. The forward declaration must have the same parameters, they must be passed the same way, and they must all have the same types as the formal parameters in the procedure[14].

Routines that are mutually recursive (that is, procedure A calls procedure B and procedure B calls procedure A) require at least one forward declaration since only one of procedure A or B can be declared before the other. In practice, however, mutual recursion (direct or indirect) doesn't occur very frequently, so the need for forward declarations is not that great.

In the absence of mutual recursion, it is always possible to organize your source code so that each procedure declaration appears before its first invocation. What's possible and what's desired are two different things, however. You might want to group a related set of procedures at the beginning of your source code and a different set of procedures towards the end of your source code. This logical grouping, by function rather than by invocation, may make your programs much easier to read and understand. However, this organization may also yield code that attempts to call a procedure before its declaration. No sweat. Just use a forward procedure definition to resolve the problem.

One major difference between the forward definition and the actual procedure declaration has to do with the procedure options. Some options, like RETURNS may appear only in the forward declaration (if a FORWARD declaration is present). Other options may only appear in the actual procedure declaration (we haven't covered any of the other procedure options yet, so don't worry about them just yet). If your procedure requires a RETURNS option, the RETURNS option must appear before the FORWARD reserved word. E.g.,

```
procedure IsItReady( valueToTest: dword ); returns( "EAX" ); forward;
```

The RETURNS option must not also appear in the actual procedure declaration later in your source file.

8.12 Putting It All Together

This chapter has filled in one of the critical elements missing from your assembly language knowledge: how to create user-defined procedures in an HLA program. This chapter discussed HLA's high level proce-

13. There are a few minor exceptions to this rule, but it is certainly true for procedure calls.

14. Actually, "exactly" is too strong a word. You will see some exceptions in a moment.

dure declaration and calling syntax. It also described how to pass parameters by value and by reference as well as the use of local variables in HLA procedures. This chapter also provided information about instruction composition and the RETURNS option for procedures. Finally, this chapter explained recursion and the use of forward procedure declarations (prototypes).

The one thing this chapter did not discuss was how procedures are written in “pure” assembly language. This chapter presents just enough information to let you start using procedures in your HLA programs. The “real stuff” will have to wait for a few chapters. Fear not, however; later chapters will teach you far more than you probably care to know about procedures in assembly language programs.

Managing Large Programs Chapter Nine

9.1 Chapter Overview

When writing larger HLA programs you do not typically write the whole program as a single source file. This chapter discusses how to break up a large project into smaller pieces and assemble the pieces separately. This radically reduces development time on large projects.

9.2 Managing Large Programs

Most assembly language programs are not totally stand alone programs. In general, you will call various standard library or other routines that are not defined in your main program. For example, you've probably noticed by now that the 80x86 doesn't provide any machine instructions like "read", "write", or "printf" for doing I/O operations. Of course, you can write your own procedures to accomplish this. Unfortunately, writing such routines is a complex task, and beginning assembly language programmers are not ready for such tasks. That's where the HLA Standard Library comes in. This is a package of procedures you can call to perform simple I/O operations like *stdout.put*.

The HLA Standard Library contains tens of thousands of lines of source code. Imagine how difficult programming would be if you had to merge these thousands of lines of code into your simple programsl imagine how slow compiling your programs would be if you had to compile those tens of thousands of lines with each program you write. Fortunately, you don't have to.

For small programs, working with a single source file is fine. For large programs this gets very cumbersome (consider the example above of having to include the entire HLA Standard Library into each of your programs). Furthermore, once you've debugged and tested a large section of your code, continuing to assemble that same code when you make a small change to some other part of your program is a waste of time. The HLA Standard Library, for example, takes several minutes to assemble, even on a fast machine. Imagine having to wait five or ten minutes on a fast Pentium machine to assemble a program to which you've made a one line change!

As with high level languages, the solution is *separate compilation* . First, you break up your large source files into manageable chunks. Then you compile the separate files into object code modules. Finally, you link the object modules together to form a complete program. If you need to make a small change to one of the modules, you only need to reassemble that one module, you do not need to reassemble the entire program.

The HLA Standard Library works in precisely this way. The Standard Library is already compiled and ready to use. You simply call routines in the Standard Library and link your code with the Standard Library using a *linker* program. This saves a tremendous amount of time when developing a program that uses the Standard Library code. Of course, you can easily create your own object modules and link them together with your code. You could even add new routines to the Standard Library so they will be available for use in future programs you write.

"Programming in the large" is a term software engineers have coined to describe the processes, methodologies, and tools for handling the development of large software projects. While everyone has their own idea of what "large" is, separate compilation, and some conventions for using separate compilation, are among the more popular techniques that support "programming in the large." The following sections describe the tools HLA provides for separate compilation and how to effectively employ these tools in your programs.

9.3 The #INCLUDE Directive

The #INCLUDE directive, when encountered in a source file, switches program input from the current file to the file specified in the parameter list of the include directive. This allows you to construct text files containing common constants, types, source code, and other HLA items, and include such a file into the assembly of several separate programs. The syntax for the include directive is

```
#include( “filename” )
```

Filename must be a valid filename. HLA merges the specified file into the compilation at the point of the #INCLUDE directive. Note that you can nest #INCLUDE statements inside files you include. That is, a file being included into another file during assembly may itself include a third file. In fact, the “stdlib.hhf” header file you see in most example programs contains the following[1]:

```
#include( "hla.hhf" )
#include( "x86.hhf" )
#include( "misctypes.hhf" )
#include( "hll.hhf" )

#include( "excepts.hhf" )
#include( "memory.hhf" )

#include( "args.hhf" )
#include( "conv.hhf" )
#include( "strings.hhf" )
#include( "cset.hhf" )
#include( "patterns.hhf" )
#include( "tables.hhf" )
#include( "arrays.hhf" )
#include( "chars.hhf" )

#include( "math.hhf" )
#include( "rand.hhf" )

#include( "stdio.hhf" )
#include( “stdin.hhf” )
#include( “stdout.hhf” )
```

Program 9.1 The stdlib.hhf Header File, as of 01/01/2000

By including “stdlib.hhf” in your source code, you automatically include all the HLA library modules. It’s often more efficient (in terms of compile time and size of code generated) to provide only those #INCLUDE statements for the modules you actually need in your program. However, including “stdlib.hhf” is extremely convenient and takes up less space in this text, which is why most programs appearing in this text use “stdlib.hhf”.

Note that the #INCLUDE directive does not need to end with a semicolon. If you put a semicolon after the #INCLUDE, that semicolon becomes part of the source file and is the first character following the included file during compilation. HLA generally allows spare semicolons in various parts of the program, so you will often see a #INCLUDE statement ending with a semicolon that produces no harm. In general,

1. Note that this file changes over time as new library modules appear in the HLA Standard Library, so this file is probably not up to date. Furthermore, there are some minor differences between the Linux and Windows version of this file. The OS-specific entries do not appear in this example.

though, you should not get in the habit of putting semicolons after #INCLUDE statements because there is the slight possibility this could create a syntax error in certain circumstances.

Using the #include directive by itself does not provide separate compilation. You *could* use the include directive to break up a large source file into separate modules and join these modules together when you compile your file. The following example would include the PRINTF.HLA and PUTC.HLA files during the compilation of your program:

```
#include( “printf.hla” )
#include( “putc.hla” )
```

Now your program *will* benefit from the modularity gained by this approach. Alas, you will not save any development time. The #INCLUDE directive inserts the source file at the point of the #INCLUDE during compilation, exactly as though you had typed that code in yourself. HLA still has to compile the code and that takes time. Were you to include all the files for the Standard Library routines in this manner, your compilations would take *forever*.

In general, you should *not* use the include directive to include source code as shown above[2]. Instead, you should use the #INCLUDE directive to insert a common set of constants, types, external procedure declarations, and other such items into a program. Typically an assembly language include file does *not* contain any machine code (outside of a macro, see the chapter on Macros and the Compile-Time Language for details). The purpose of using #INCLUDE files in this manner will become clearer after you see how the external declarations work.

9.4 Ignoring Duplicate Include Operations

As you begin to develop sophisticated modules and libraries, you eventually discover a big problem: some header files will need to include other header files (e.g., the stdlib.hhf header file includes all the other Standard Library Header files). Well, this isn’t actually a big problem, but a problem will occur when one header file includes another, and that second header file includes another, and that third header file includes another, and ..., and that last header file includes the first header file. Now *this* is a big problem.

There are two problems with a header file indirectly including itself. First, this creates an infinite loop in the compiler. The compiler will happily go on about its business including all these files over and over again until it runs out of memory or some other error occurs. Clearly this is not a good thing. The second problem that occurs (usually before the problem above) is that the second time HLA includes a header file, it starts complaining bitterly about duplicate symbol definitions. After all, the first time it reads the header file it processes all the declarations in that file, the second time around it views all those symbols as duplicate symbols.

HLA provides a special include directive that eliminates this problem: #INCLUDEONCE. You use this directive exactly like you use the #include directive, e.g.,

```
#includeonce( “myHeaderFile.hhf” )
```

If myHeaderFile.hhf directly or indirectly includes itself (with a #INCLUDEONCE directive), then HLA will ignore the new request to include the file. Note, however, that if you use the #INCLUDE directive, rather than #INCLUDEONCE, HLA will include the file a second name. This was done in case you really do need to include a header file twice, for some reason (though it is hard to imagine needing to do this).

The bottom line is this: you should always use the #INCLUDEONCE directive to include header files you’ve created. In fact, you should get in the habit of always using #INCLUDEONCE, even for header files created by others (the HLA Standard Library already has provisions to prevent recursive includes, so you don’t have to worry about using #INCLUDEONCE with the Standard Library header files).

There is another technique you can use to prevent recursive includes – using conditional compilation. For details on this technique, see the chapter on the HLA Compile-Time Language in a later volume.

2. There is nothing wrong with this, other than the fact that it does not take advantage of separate compilation.

9.5 UNITs and the EXTERNAL Directive

Technically, the #INCLUDE directive provides you with all the facilities you need to create modular programs. You can create several modules, each containing some specific routine, and include those modules, as necessary, in your assembly language programs using #INCLUDE. However, HLA provides a better way: external and public symbols.

One major problem with the include mechanism is that once you've debugged a routine, including it into a compilation still wastes a lot of time since HLA must recompile bug-free code every time you assemble the main program. A much better solution would be to preassemble the debugged modules and link the object code modules together rather than reassembling the entire program every time you change a single module. This is what the EXTERNAL directive allows you to do.

To use the external facilities, you must create at least two source files. One file contains a set of variables and procedures used by the second. The second file uses those variables and procedures without knowing how they're implemented. The only problem is that if you create two separate HLA programs, the linker will get confused when you try to combine them. This is because both HLA programs have their own main program. Which main program does the OS run when it loads the program into memory? To resolve this problem, HLA uses a different type of compilation module, the UNIT, to compile programs without a main program. The syntax for an HLA UNIT is actually simpler than that for an HLA program, it takes the following form:

```
unit unitname;

    << declarations >>

end unitname;
```

With one exception (the VAR section), anything that can go in the declaration section of an HLA program can go into the declaration section of an HLA unit. Notice that a unit does not have a BEGIN clause and there are no program statements in the unit[3]; a unit only contains declarations.

In addition to the fact that a unit does not contain any executable statements, there is one other difference between units and programs. Units cannot have a VAR section. This is because the VAR section declares variables that are local to the main program's source code. Since there is no source code associated with a unit, VAR sections are illegal[4]

To demonstrate, consider the following two modules:

```
unit Number1;

static
    Var1:   uns32;
    Var2:   uns32;

    procedure Add1and2;
    begin Add1and2;

        push( eax );
        mov( Var2, eax );
        add( eax, Var1 );

    end Add1and2;
```

3. Of course, units may contain procedures and those procedures may have statements, but the unit itself does not have any executable instructions associated with it.
4. Of course, procedures in the unit may have their own VAR sections, but the procedure's declaration section is separate from the unit's declaration section.

```
end Number1;
```

Program 9.2 Example of a Simple HLA Unit

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

begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Add1and2();
    stdout.put( "Var1=", Var1, nl );

end main;
```

Program 9.3 Main Program that References External Objects

The main program references Var1, Var2, and Add1and2, yet these symbols are external to this program (they appear in unit *Number1*). If you attempt to compile the main program as it stands, HLA will complain that these three symbols are undefined.

Therefore, you must declare them external with the EXTERNAL option. An external procedure declaration looks just like a forward declaration except you use the reserved word EXTERNAL rather than FORWARD. To declare external static variables, simply follow those variables' declarations with the reserved word EXTERNAL. The following is a modification to the previous main program that includes the external declarations:

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

    procedure Add1and2; external;

static
    Var1: uns32; external;
    Var2: uns32; external;

begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Add1and2();
    stdout.put( "Var1=", Var1, nl );

end main;
```

Program 9.4 Modified Main Program with EXTERNAL Declarations

If you attempt to compile this second version of *main*, using the typical HLA compilation command "HLA main2.hla" you will be somewhat disappointed. This program will actually compile without error. However, when HLA attempts to link this code it will report that the symbols *Var1*, *Var2*, and *Add1and2* are undefined. This happens because you haven't compiled and linked in the associated unit with this main program. Before you try that, and discover that it still doesn't work, you should know that all symbols in a unit, by default, are *private* to that unit. This means that those symbols are inaccessible in code outside that unit unless you explicitly declare those symbols as *public* symbols. To declare symbols as public, you simply put external declarations for those symbols in the unit before the actual symbol declarations. If an external declaration appears in the same source file as the actual declaration of a symbol, HLA assumes that the name is needed externally and makes that symbol a public (rather than private) symbol. The following is a correction to the *Number1* unit that properly declares the external objects:

```
unit Number1;

static
    Var1:   uns32; external;
    Var2:   uns32; external;

    procedure Add1and2; external;

static
    Var1:   uns32;
    Var2:   uns32;


    procedure Add1and2;
    begin Add1and2;

        push( eax );
        mov( Var2, eax );
        add( eax, Var1 );

    end Add1and2;

end Number1;
```

Program 9.5 Correct Number1 Unit with External Declarations

It may seem redundant declaring these symbols twice as occurs in Program 9.5, but you'll soon seen that you don't normally write the code this way.

If you attempt to compile the *main* program or the *Number1* unit using the typical HLA statement, i.e.,

```
HLA main2.hla
HLA unit2.hla
```

You'll quickly discover that the linker still returns errors. It returns an error on the compilation of main2.hla because you still haven't told HLA to link in the object code associated with unit2.hla. Likewise, the linker complains if you attempt to compile unit2.hla by itself because it can't find a main program. The simple solution is to compile both of these modules together with the following single command:

```
HLA main2.hla unit2.hla
```

This command will properly compile both modules and link together their object code.

Unfortunately, the command above defeats one of the major benefits of separate compilation. When you issue this command it will compile both main2 and unit2 prior to linking them together. Remember, a major reason for separate compilation is to reduce compilation time on large projects. While the above command is convenient, it doesn't achieve this goal.

To separately compile the two modules you must run HLA separately on them. Of course, we saw earlier that attempting to compile these modules separately produced linker errors. To get around this problem, you need to compile the modules without linking them. The "-c" (compile-only) HLA command line option achieves this. To compile the two source files without running the linker, you would use the following commands:

```
HLA -c main2.hla
HLA -c unit2.hla
```

This produces two object code files, main2.obj and unit2.obj, that you can link together to produce a single executable. You could run the linker program directly, but an easier way is to use the HLA compiler to link the object modules together for you:

```
HLA main2.obj unit2.obj
```

Under Windows, this command produces an executable file named *main2.exe*[5]; under Linux, this command produces a file named *main2*. You could also type the following command to compile the main program and link it with a previously compiled unit2 object module:

```
HLA main2.hla unit2.obj
```

In general, HLA looks at the suffixes of the filenames following the HLA commands. If the filename doesn't have a suffix, HLA assumes it to be ".HLA". If the filename has a suffix, then HLA will do the following with the file:

- If the suffix is ".HLA", HLA will compile the file with the HLA compiler.
- If the suffix is ".ASM", HLA will assemble the file with MASM.
- If the suffix is ".OBJ" or ".LIB"(Windows), or ".o" or ".a" (Linux), then HLA will link that module with the rest of the compilation.

9.5.1 Behavior of the EXTERNAL Directive

Whenever you declare a symbol EXTERNAL using the external directive, keep in mind several limitations of EXTERNAL objects:

- Only one EXTERNAL declaration of an object may appear in a given source file. That is, you cannot define the same symbol twice as an EXTERNAL object.
- Only PROCEDURE, STATIC, READONLY, and STORAGE variable objects can be external. VAR and parameter objects cannot be external.
- External objects must be at the global declaration level. You cannot declarare EXTERNAL objects within a procedure or other nested structure.
- EXTERNAL objects publish their name globally. Therefore, you must carefully choose the names of your EXTERNAL objects so they do not conflict with other symbols.

This last point is especially important to keep in mind. As this text is being written, the HLA compiler translates your HLA source code into assembly code. HLA assembles the output by using MASM (the Microsoft Macro Assembler), Gas (Gnu's as), or some other assembler. Finally, HLA links your modules using a linker. At each step in this process, your choice of external names could create problems for you.

5. If you want to explicitly specify the name of the output file, HLA provides a command-line option to achieve this. You can get a menu of all legal command line options by entering the command "HLA -?".

Consider the following HLA external/public declaration:

```
static
        extObj:         uns32; external;
        extObj:         uns32;
        localObject:    uns32;
```

When you compile a program containing these declarations, HLA automatically generates a "munged" name for the *localObject* variable that probably isn't ever going to have any conflicts with system-global external symbols[6]. Whenever you declare an external symbol, however, HLA uses the object's name as the default external name. This can create some problems if you inadvertently use some global name as your variable name. Worse still, the assembler will not be able to properly process HLA's output if you happen to choose an identifier that is legal in HLA but is one of the assembler's reserved word. For example, if you attempt to compile the following code fragment as part of an HLA program (producing MASM output), it will compile properly but MASM will not be able to assemble the code:

```
static
    c: char; external;
    c: char;
```

The reason MASM will have trouble with this is because HLA will write the identifier "c" to the assembly language output file and it turns out that "c" is a MASM reserved word (MASM uses it to denote C-language linkage).

To get around the problem of conflicting external names, HLA supports an additional syntax for the EXTERNAL option that lets you explicitly specify the external name. The following example demonstrates this extended syntax:

```
static
    c: char; external( "var_c" );
    c: char;
```

If you follow the EXTERNAL keyword with a string constant enclosed by parentheses, HLA will continue to use the declared name (*c* in this example) as the identifier within your HLA source code. Externally (i.e., in the assembly code) HLA will substitute the name *var_c* whenever you reference *c*. This features helps you avoid problems with the misuse of assembler reserved words, or other global symbols, in your HLA programs.

You should also note that this feature of the EXTERNAL option lets you create *aliases*. For example, you may want to refer to an object by the name *StudentCount* in one module while refer to the object as *PersonCount* in another module (you might do this because you have a general library module that deals with counting people and you want to use the object in a program that deals only with students). Using a declaration like the following lets you do this:

```
static
    StudentCount: uns32; external( "PersonCount" );
```

Of course, you've already seen some of the problems you might encounter when you start creating aliases. So you should use this capability sparingly in your programs. Perhaps a more reasonable use of this feature is to simplify certain OS APIs. For example, Win32 uses some really long names for certain procedure calls. You can use the EXTERNAL directive to provide a more meaningful name than the standard one supplied by the operating system.

9.5.2 Header Files in HLA

HLA's technique of using the same EXTERNAL declaration to define public as well as external symbols may seem somewhat counter-intuitive. Why not use a PUBLIC reserved word for public symbols and

6. Typically, HLA creates a name like *?001A_localObject* out of *localObject*. This is a legal MASM identifier but it is not likely it will conflict with any other global symbols when HLA compiles the program with MASM.

the EXTERNAL keyword for external definitions? Well, as counter-intuitive as HLA's external declarations may seem, they are founded on decades of solid experience with the C/C++ programming language that uses a similar approach to public and external symbols[7]. Combined with a *header file*, HLA's external declarations make large program maintenance a breeze.

An important benefit of the EXTERNAL directive (versus separate PUBLIC and EXTERNAL directives) is that it lets you minimize duplication of effort in your source files. Suppose, for example, you want to create a module with a bunch of support routines and variables for use in several different programs (e.g., the HLA Standard Library). In addition to sharing some routines and some variables, suppose you want to share constants, types, and other items as well.

The #INCLUDE file mechanism provides a perfect way to handle this. You simply create a #INCLUDE file containing the constants, macros, and external definitions and include this file in the module that implements your routines and in the modules that use those routines (see Figure 9.1).

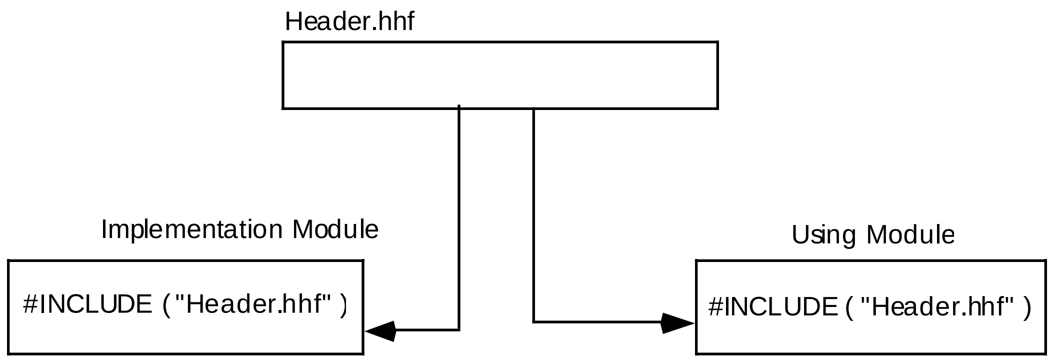

Figure 9.1 Using Header Files in HLA Programs

A typical header file contains only CONST, VAL, TYPE, STATIC, READONLY, STORAGE, and procedure prototypes (plus a few others we haven't look at yet, like macros). Objects in the STATIC, READONLY, and STORAGE sections, as well as all procedure declarations, are always EXTERNAL objects. In particular, you generally should not put any VAR objects in a header file, nor should you put any non-external variables or procedure bodies in a header file. If you do, HLA will make duplicate copies of these objects in the different source files that include the header file. Not only will this make your programs larger, but it will cause them to fail under certain circumstances. For example, you generally put a variable in a header file so you can share the value of that variable amongst several different modules. However, if you fail to declare that symbol as external in the header file and just put a standard variable declaration there, each module that includes the source file will get its own separate variable - the modules will not share a common variable.

If you create a standard header file, containing CONST, VAL, and TYPE declarations, and external objects, you should always be sure to include that file in the declaration section of all modules that need the definitions in the header file. Generally, HLA programs include all their header files in the first few statements after the PROGRAM or UNIT header.

This text adopts the HLA Standard Library convention of using an ".hhf" suffix for HLA header files ("HHF" stands for HLA Header File).

7. Actually, C/C++ is a little different. All global symbols in a module are assumed to be public unless explicitly declared private. HLA's approach (forcing the declaration of public items via EXTERNAL) is a little safer.

9.6 Make Files

Although using separate compilation reduces assembly time and promotes code reuse and modularity, it is not without its own drawbacks. Suppose you have a program that consists of two modules: pgma.hla and pgmb.hla. Also suppose that you've already compiled both modules so that the files pgma.obj and pgmb.obj exist. Finally, you make changes to pgma.hla and pgmb.hla and compile the pgma.hla file *but forget to compile the pgmb.hla file.* Therefore, the pgmb.obj file will be *out of date* since this object file does not reflect the changes made to the pgmb.hla file. If you link the program's modules together, the resulting executable file will only contain the changes to the pgma.hla file, it will not have the updated object code associated with pgmb.hla. As projects get larger they tend to have more modules associated with them, and as more programmers begin working on the project, it gets very difficult to keep track of which object modules are up to date.

This complexity would normally cause someone to recompile *all* modules in a project, even if many of the object files are up to date, simply because it might seem too difficult to keep track of which modules are up to date and which are not. Doing so, of course, would eliminate many of the benefits that separate compilation offers. Fortunately, there is a tool that can help you manage large projects: *make*[8]. The make program, with a little help from you, can figure out which files need to be reassemble and which files have up to date .obj files. With a properly defined *make file*, you can easily assemble only those modules that absolutely must be assembled to generate a consistent program.

A make file is a text file that lists compile-time dependencies between files. An .exe file, for example, is *dependent* on the source code whose assembly produce the executable. If you make any changes to the source code you will (probably) need to reassemble or recompile the source code to produce a new executable file[9].

Typical dependencies include the following:

- An executable file generally depends only on the set of object files that the linker combines to form the executable.
- A given object code file depends on the assembly language source files that were assembled to produce that object file. This includes the assembly language source files (.hla) and any files included during that assembly (generally .hhf files).
- The source files and include files generally don't depend on anything.

A make file generally consists of a dependency statement followed by a set of commands to handle that dependency. A dependency statement takes the following form:

`dependent-file` `:` *`list of files`*

Example :

```
pgm.exe: pgma.obj pgmb.obj             --Windows/nmake example
```

This statement says that "pgm.exe" is dependent upon pgma.obj and pgmb.obj. Any changes that occur to pgma.obj or pgmb.obj will require the generation of a new pgm.exe file. This example is Windows-specific, here's the same makefile statement in a Linux-friendly form:

Example :

```
pgm: pgma.o pgmb.o             --Linux/make example
```

The make program uses a *time/date stamp* to determine if a dependent file is out of date with respect to the files it depends upon. Any time you make a change to a file, the operating system will update a *modification time and date* associated with the file. The make program compares the modification date/time stamp of the dependent file against the modification date/time stamp of the files it depends upon. If the dependent

8. Under Windows, Microsoft calls this program *nmake*. This text will use the more generic name "make" when refering to this program. If you are using Microsoft tools under Windows, just substitute "nmake" for "make" throughout this chapter.
9. Obviously, if you only change comments or other statements in the source file that do not affect the executable file, a recompile or reassembly will not be necessary. To be safe, though, we will assume *any* change to the source file will require a reassembly.

file's modification date/time is earlier than one or more of the files it depends upon, or one of the files it depends upon is not present, then make assumes that some operation must be necessary to update the dependent file.

When an update is necessary, make executes the set of commands following the dependency statement. Presumably, these commands would do whatever is necessary to produce the updated file.

The dependency statement *must* begin in column one. Any commands that must execute to resolve the dependency must start on the line immediately following the dependency statement and each command must be indented one tabstop. The pgm.exe statement above (the Windows example) would probably look something like the following:

```
pgm.exe: pgma.obj pgmb.obj
    hla -opgm.exe pgma.obj pgmb.obj
```

(The "-opgm.exe" option tells HLA to name the executable file "pgm.exe.") Here's the same example for Linux users:

```
pgm: pgma.o pgmb.o
    hla -opgm pgma.obj pgmb.obj
```

If you need to execute more than one command to resolve the dependencies, you can place several commands after the dependency statement in the appropriate order. Note that you must indent all commands one tab stop. The make program ignores any blank lines in a make file. Therefore, you can add blank lines, as appropriate, to make the file easier to read and understand.

There can be more than a single dependency statement in a make file. In the example above, for example, executable (pgm or pgm.exe) depends upon the object files (pgma.obj or pgma.o and pgmb.obj or pgmb.o). Obviously, the object files depend upon the source files that generated them. Therefore, before attempting to resolve the dependencies for the executable, make will first check out the rest of the make file to see if the object files depend on anything. If they do, make will resolve those dependencies first. Consider the following (Windows) make file:

```
pgm.exe: pgma.obj pgmb.obj
    hla -opgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.hla
    hla -c pgma.hla

pgmb.obj: pgmb.hla
    hla -c pgmb.hla
```

The make program will process the first dependency line it finds in the file. However, the files that pgm.exe depends upon themselves have dependency lines. Therefore, make will first ensure that pgma.obj and pgmb.obj are up to date before attempting to execute HLA to link these files together. Therefore, if the only change you've made has been to pgmb.hla, make takes the following steps (assuming pgma.obj exists and is up to date).

1. The make program processes the first dependency statement. It notices that dependency lines for pgma.obj and pgmb.obj (the files on which pgm.exe depends) exist. So it processes those statements first.
2. the make program processes the pgma.obj dependency line. It notices that the pgma.obj file is newer than the pgma.hla file, so it does *not* execute the command following this dependency statement.
3. The make program processes the pgmb.obj dependency line. It notes that pgmb.obj is older than pgmb.hla (since we just changed the pgmb.hla source file). Therefore, make executes the command following on the next line. This generates a new pgmb.obj file that is now up to date.
4. Having processed the pgma.obj and pgmb.obj dependencies, make now returns its attention to the first dependency line. Since make just created a new pgmb.obj file, its date/time stamp will be newer than pgm.exe's. Therefore, make will execute the HLA command that links pgma.obj and pgmb.obj together to form the new pgm.exe file.

Note that a properly written make file will instruct the make program to assemble only those modules absolutely necessary to produce a consistent executable file. In the example above, make did not bother to assemble pgma.hla since its object file was already up to date.

There is one final thing to emphasize with respect to dependencies. Often, object files are dependent not only on the source file that produces the object file, but any files that the source file includes as well. In the previous example, there (apparently) were no such include files. Often, this is not the case. A more typical make file might look like the following (Linux example):

```
pgm: pgma.o pgmb.o
    hla -opgm pgma.o pgmb.o

pgma.o: pgma.hla pgm.hhf
    hla -c pgma.hla

pgmb.o: pgmb.hla pgm.hhf
    hla -c pgmb.hla
```

Note that any changes to the pgm.hhf file will force the make program to recompile *both* pgma.hla and pgmb.hla since the pgma.o and pgmb.o files both depend upon the pgm.hhf include file. Leaving include files out of a dependency list is a common mistake programmers make that can produce inconsistent executable files.

Note that you would not normally need to specify the HLA Standard Library include files nor the Standard Library ".lib" (Windows) or ".a" (Linux) files in the dependency list. True, your resulting exectuable file does depend on this code, but the Standard Library rarely changes, so you can safely leave it out of your dependency list. Should you make a modification to the Standard Library, simply delete any old executable and object files to force a reassembly of the entire system.

The make program, by default, assumes that it will be processing a make file named "makefile". When you run the make program, it looks for "makefile" in the current directory. If it doesn't find this file, it complains and terminates[10]. Therefore, it is a good idea to collect the files for each project you work on into their own subdirectory and give each project its own makefile. Then to create an executable, you need only change into the appropriate subdirectory and run the make program.

Although this section discusses the make program in sufficient detail to handle most projects you will be working on, keep in mind that the make program provides considerable functionality that this chapter does not discuss. To learn more about the nmake.exe program, consult the the appropriate documentation. Note that several versions of MAKE exist. Microsoft produces nmake.exe, Borland has their own MAKE.EXE program and various versions of MAKE have been ported to Windows from UNIX systems (e.g., GMAKE). Linux users will typically employ the GNU make program. While these various make programs are not equivalent, they all do a pretty good job of handling the simple make syntax that this chapter describes.

9.7 Code Reuse

One of the principle goals of Software Engineering is to reduce program development time. Although the techniques we've studied in this chapter will certainly reduce development effort, there are bigger prizes to be had here. Consider for a moment a simple program that reads an integer from the user and then displays the value of that integer on the standard output device. You can easily write a trivial version of this program with about eight lines of HLA code. That's not too difficult. However, suppose you did not have the HLA Standard Library at your disposal. Now, instead of an eight line program, you'd be faced with writing a program that hundreds if not thousands of lines long. Obviously, this program will take a lot longer to write than the original eight-line version. The difference between these two applications is the fact that in the first version of this program you got to reuse some code that was already written; in the second version of the program you had to write everything from scratch. This concept of code reuse is very important when

10. There is a command line option that lets you specify the name of the makefile. See the nmake documentation in the MASM manuals for more details.

writing large programs – you can get large programs working much more quickly if you reuse code from previous projects.

The idea behind code reuse is that many code sequences you write will be usable in future programs. As time passes and you write more code, progress on your projects will be faster since you can reuse code you've written (or others have written) on previous projects. The HLA Standard Library functions are the classic example, somebody had to write those functions so you could use them. And use them you do. As of this writing, the Standard Library represented about 50,000 lines of HLA source code. Imagine having to write a fair portion of that everytime you wanted to write an HLA program!

Although the HLA Standard Library contains lots of very useful routines and functions, this code base cannot possible predict the type of code you will need in every future project. The HLA Standard Library provides some of the more common routines you'll need when writing programs, but you're certainly going to have need for routines that the HLA Standard Library cannot satisfy. Unless you can find a source for the code you need from some third party, you're probably going to have to write the new routines yourself.

The trick when writing a program is to try and figure out which routines are general purpose and could be used in future programs; once you make this determination, you should write such routines separately from the rest of your application (i.e., put them in a separate source file for compilation). By keeping them separate, you can use them in future projects. If "try and figure out which routines are general purpose..." sounds a bit difficult, well, you're right it is. Even after 30 years of Software Engineering research, no one has really figured out how to effectively reuse code. There are some obvious routines we can reuse (that's why there are "standard libraries") but it is quite difficult for the practicing engineer to successfully predict which routines s/he will need in the future and write these as separate modules.

Attempting to teach you how to decide which routines are worthy of saving for future programs and which are specific to your current application is well beyond the scope of this text. There are several Software Engineering texts out there that try to explain how to do this, but keep in mind that even after the publication of these texts, practicing engineers still have problems picking the right routines to save. Hopefully, as you gain experience, you will begin to recognize those routines that are worth keeping for future programs and those that aren't worth bothering with. This text will take the easy way out and assume that you know which routines you want to keep and which you don't.

9.8 Creating and Managing Libraries

Imagine that you've created a few hundred routines over the past couple of years and you would like to have the object code ready to link with any new projects you begin. You could move all this code into a single source file, stick in a bunch of EXTERNAL declarations, and then link the resulting object file with any new programs you write that can use the routines in your "library". Unfortunately, there are a couple of problems with this approach. Let's take a look at some of these problems.

Problem number one is that your library will grow to a fairly good size with time; if you put the source code to every routine in a single source file, small additions or changes to the file will require a complete recompilation of the whole library. That's clearly not what we want to do, based on what you've learned from this chapter.

Another problem with this "solution" is that whenever you link this object file to your new applications, you link in the entire library, not just the routines you want to use. This makes your applications unnecessarily large, especially if your library has grown. Were you to link your simple projects with the entire HLA Standard library, for example, the result would be positively huge.

A solution to both of the above problems is to compile each routine in a separate file and produce a unique object file for it. Unfortunately, with hundreds of routines you're going to wind up with hundreds of object files; any time you want to call a dozen or so library routines, you'd have to link your main application with a dozen or so object modules from your library. Clearly, this isn't acceptable either.

You may have noticed by now that when you link your applications with the HLA Standard Library, you only link with a single file: *hlalib.lib (Windows)* or *hlalib.a (Linux)*. .LIB (library) and ".a" (archive) files are a collection of object files. When the linker processes a library file, it pulls out only the object files it needs, it does not link the entire file with your application. Hence you get to work with a single file and your applications don't grow unnecessarily large.

Linux provids the "ar" (archiver) program to manage library files. To use this program to combine several object files into a single ".a" file, you'd use a command line like the following:

```
ar -q library.a list_of_.o_files
```

For more information on this command, check out the man page on the "ar" program ("man ar").

To collect your library object files into a single library file, you need to use a library manager program. This is actually built into Microsoft's linker program, although Microsoft provides a LIB.EXE program that acts as a front end to LINK.EXE and processes command line parameter specifically for creating library files. In this section we'll discuss how to use LIB.EXE to construct library files.

> Warning: section describes the use of Microsoft's LIB program version 6.00.8168. Microsoft has a history of changing the user (command line) interface to their tools between even minor revisions of their products. Please be aware that the specific examples in this section may need to be modified for the version of LIB.EXE that you are actually using. The basic principles, however, will be the same. See the Microsoft manuals if you have any questions about the use of the LIB.EXE program. You should also be able to type "LIB /?" from the command line prompt to get an idea of the LIB.EXE invocation syntax.

The LIB.EXE program uses the following general syntax (from a command window prompt):

```
lib  {options} {files}

 options:

    /CONVERT
    /DEBUGTYPE:CV
    /DEF[:filename]
    /EXPORT:symbol
    /EXTRACT:membername
    /INCLUDE:symbol
    /LIBPATH:dir
    /LINK50COMPAT
    /LIST[:filename]
    /MACHINE:{ALPHA|ARM|IX86|MIPS|MIPS16|MIPSR41XX|PPC|SH3|SH4}
    /NAME:filename
    /NODEFAULTLIB[:library]
    /NOLOGO
    /OUT:filename
    /REMOVE:membername
    /SUBSYSTEM:{NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX}[,#[.##]]
    /VERBOSE
```

Most of these options are not of interest to us, but there are a few important ones. First of all, you should always use the "/out:*filename*" to specify the name of your library file. For example, you would often begin a LIB.EXE command line with the following:

```
lib /out:mylib.lib ....  (or whatever library name you want to use)
```

The second option you're probably going to use all the time is the "/subsystem:xxxxx" option. For console mode (i.e., text-based command window) programs you would use "/subsystem:console". For Windows programs you write that use the graphical user interface, you'd probably use the "/subsystem:windows" option. The HLA Standard Library, since it's mostly intended for console applications, uses the "/subsystem:console" option. You probably won't need to use most of the other options since LIB.EXE uses appropriate default values for them. If you're doing some advanced stuff, or if you're just curious, check out Microsoft's documentation on the LIB.EXE program for more details.

Following any options on the command line come the OBJ filenames that you want to merge together when creating the library file. A typical LIB.EXE command line might look something like the following:

```
lib /out:mylib.lib /subsystem:console  file1.obj file2.obj file3.obj
```

If you want to merge dozens or hundreds of files into a single library, you're going to run into a problem. The command window command line doesn't allow aribtrarily long lines. This will severely limit the number of files you can add to your library at one time. The easiest way to handle this problem is to create a text file containing the commands and filenames for LIB.EXE. Here's the file from HLA's "bits.lib" library module:

```
/out:..\bits.lib
/subsystem:console
cnt.obj
reverse.obj
merge.obj
```

(This file was chosen because it's really short.)

The first line tells LIB.EXE to create a library module named "bits.lib" in the parent subdirectory. Like most HLA Standard Library modules, this module is intended for use in console applications (not that it makes any difference to this module), hence the "/subsystem:console" option. The next three lines of the file contain the files to merge together when creating the bits.lib file. A separate call to LIB.EXE elsewhere combines bits.lib, strings.lib, chars.lib, etc., into a single library module: hlalib.lib. To specify that LIB.EXE should read this file instead of extracting this information from the command line, you use "@responsefile" in place of the operands on the LIB.EXE command line. For example, if the file above is named "bits.txt" (which it is, by the way), then you could tell LIB.EXE to read its commands from this file using the following comand line:

```
lib @bits.txt
```

Once you've created a library file, you can tell HLA to automatically extract any important files from the file by specifying its name on the HLA command line, e.g.,

```
hla mypgm.hla mylib.lib
```

Assuming all the necessary modules you need are present in mylib.lib, the command line above will compile mypgm and link in the appropriate OBJ modules found in the mylib.lib library file. Note that HLA automatically links in hlalib.lib, so you don't have to specify this on the command line when compiling your programs.

For more examples of using the LIB.EXE program, take a look at the makefiles in the HLA library source file directories. These makefiles contain several calls to LIB.EXE that build the hlalib.lib file. Hopefully you can see how to use LIB.EXE by looking at these files.

9.9 Name Space Pollution

One problem with creating libraries with lots of different modules is name space pollution. A typical library module will have a #INCLUDE file associated with it that provides external definitions for all the routines, constants, variables, and other symbols provided in the library. Whenever you want to use some routines or other objects from the library, you would typically #INCLUDE the library's header file in your project. As your libraries get larger and you add more declarations in the header file, it becomes more and more likely that the names you've chosen for your library's identifiers will conflict with names you want to use in your current project. This conflict is what is meant by name space pollution: library header files pollute the name space with many names you typically don't need in order to gain easy access to the few routines in the library you actually use. Most of the time those names don't harm anything – unless you want to use those names yourself in your program.

HLA requires that you declare all external symbols at the global (PROGRAM/UNIT) level. You cannot, therefore, include a header file with external declarations within a procedure[11]. As such, there will be no naming conflicts between external library symbols and symbols you declare locally within a procedure; the conflicts will only occur between the external symbols and your global symbols. While this is a good argument for avoiding global symbols as much as possible in your program, the fact remains that most symbols in an assembly language program will have global scope. So another solution is necessary.

HLA's solution, which it certainly uses in the Standard Library, is to put most of the library names in a NAMESPACE declaration section. A NAMESPACE declaration encapsulates all declarations and exposes only a single name (the NAMESPACE identifier) at the global level. You access the names within the NAMESPACE by using the familiar dot-notation (see "Namespaces" on page 496). This reduces the effect of namespace pollution from many dozens or hundreds of names down to a single name.

Of course, one disadvantage of using a NAMESPACE declaration is that you have to type a longer name in order to reference a particular identifier in that name space (i.e., you have to type the NAMESPACE identifier, a period, and then the specific identifier you wish to use). For a few identifiers you use frequently, you might elect to leave those identifiers outside of any NAMESPACE declaration. For example, the HLA Standard Library does not define the symbols *malloc, free,* or *nl* (among others) within a NAMESPACE. However, you want to minimize such declarations in your libraries to avoid conflicts with names in your own programs. Often, you can choose a NAMESPACE identifier to complement your routine names. For example, the HLA Standard Libraries string copy routine was named after the equivalent C Standard Library function, *strcpy*. HLA's version is *str.cpy*. The actual function name is *cpy*; it happens to be a member of the *str* NAMESPACE, hence the full name *str.cpy* which is very similar to the comparable C function. The HLA Standard Library contains several examples of this convention. The *arg.c* and *arg.v* functions are another pair of such identifiers (corresponding to the C identifiers *argc* and *argv*).

Using a NAMESPACE in a header file is no different than using a NAMESPACE in a PROGRAM or UNIT. Here's an example of a typical header file containing a NAMESPACE declaration:

```
// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file.

namespace myLib;

    procedure func1; external;
    procedure func2; external;
    procedure func3; external;

end myLib;
```

Typically, you would compile each of the functions (func1..func3) as separate units (so each has its own object file and linking in one function doesn't link them all). Here's what a sample UNIT declaration for one of these functions:

```
unit func1Unit;
#includeonce( "myHeader.hhf" )

procedure myLib.func1;
begin func1;

    << code for func1 >>

end func1;

end func1Unit;
```

You should notice two important things about this unit. First, you do not put the actual *func1* procedure code within a NAMESPACE declaration block. By using the identifier *myLib.func1* as the procedure's name, HLA automatically realizes that this procedure declaration belongs in a name space. The second thing to note is that you do not preface *func1* with "myLib." after the BEGIN and END clauses in the procedure.

11. Or within an Iterator or Method, as you will see in later chapters.

HLA automatically associates the BEGIN and END identifiers with the PROCEDURE declaration, so it knows that these identifiers are part of the *myLib* name space and it doesn't make you type the whole name again.

Important note: when you declare external names within a name space, as was done in *func1Unit* above, HLA uses only the function name (*func1* in this example) as the external name. This creates a name space pollution problem in the external name space. For example, if you have two different name spaces, *myLib* and *yourLib* and they both define a *func1* procedure, the linker will complain about a duplicate definition for *func1* if you attempt to use functions from both these library modules. There is an easy work-around to this problem: use the extended form of the EXTERNAL directive to explicitly supply an external name for all external identifiers appearing in a NAMESPACE declaration. For example, you could solve this problem with the following simple modification to the myHeader.hhf file above:

```
// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file.

namespace myLib;

    procedure func1; external( “myLib_func1” );
    procedure func2; external( “myLib_func2” );
    procedure func3; external( “myLib_func3” );

end myLib;
```

This example demonstrates an excellent convention you should adopt: when exporting names from a name space, always supply an explicit external name and construct that name by concatenating the NAMESPACE identifier with an underscore and the object's internal name.

The use of NAMESPACE declarations does not completely eliminate the problems of name space pollution (after all, the name space identifier is still a global object, as anyone who has included stdlib.hhf and attempted to define a "cs" variable can attest), but NAMESPACE declarations come pretty close to eliminating this problem. Therefore, you should use NAMESPACE everywhere practical when creating your own libraries.

9.10 Putting It All Together

Managing large projects is considerably easier if you break your program up into separate modules and work on them independently. In this chapter you learned about HLA's UNITs, include files, and the EXTERNAL directive. These provide the tools you need to break a program up into smaller modules. In addition to HLA's facilities, you'll also use a separate tool, nmake.exe, to automatically compile and link only those files that are necessary in a large project.

This chapter provided a very basic introduction to the use of makefiles and the make utility. Note that the MAKE programs are quite sophisticated. The presentation of the make program in this chapter barely scratches the surface. If you're interested in more information about MAKE facilities you should consult one of the excellent texts available on this subject. Lots of good information is also available on the Internet (just use the usual search tools).

This chapter also presented a brief introduction to the Library Manager programs (i.e., LIB.EXE and ar) that let you combine several different object files into a single library file. A single library file is much easier to work with than dozens or hundreds of object files. If you write lots of little subroutines and compile them as separate modules for use with future projects, you will definitely want to create a library file from those object files.

In addition to breaking up large HLA projects, UNITs are also the basis for letting you write assembly language functions that you can call from high level languages like C/C++ and Delphi/Kylix. A later volume in this text will describe how you can use UNITs for this purpose.

Integer Arithmetic Chapter Ten

10.1 Chapter Overview

This chapter discusses the implementation of arithmetic computation in assembly language. By the conclusion of this chapter you should be able to translate (integer) arithmetic expressions and assignment statements from high level languages like Pascal and C/C++ into 80x86 assembly language.

10.2 80x86 Integer Arithmetic Instructions

Before describing how to encode arithmetic expressions in assembly language, it would be a good idea to first discuss the remaining arithmetic instructions in the 80x86 instruction set. Previous chapters have covered most of the arithmetic and logical instructions, so this section will cover the few remaining instructions you'll need.

10.2.1 The MUL and IMUL Instructions

The multiplication instructions provide you with another taste of irregularity in the 80x86's instruction set. Instructions like ADD, SUB, and many others in the 80x86 instruction set support two operands. Unfortunately, there weren't enough bits in the 80x86's opcode byte to support all instructions, so the 80x86 treats the MUL (unsigned multiply) and IMUL (signed integer multiply) instructions as single operand instructions, like the INC, DEC, and NEG instructions.

Of course, multiplication *is* a two operand function. To work around this fact, the 80x86 always assumes the accumulator (AL,AX, or EAX) is the destination operand. This irregularity makes using multiplication on the 80x86 a little more difficult than other instructions because one operand has to be in the accumulator. Intel adopted this unorthogonal approach because they felt that programmers would use multiplication far less often than instructions like ADD and SUB.

Another problem with the MUL and IMUL instructions is that you cannot multiply the accumulator by a constant using these instructions. Intel quickly discovered the need to support multiplication by a constant and added the INTMUL instruction to overcome this problem. Nevertheless, you must be aware that the basic MUL and IMUL instructions do not support the full range of operands that INTMUL does.

There are two forms of the multiply instruction: unsigned multiplication (MUL) and signed multiplication (IMUL). Unlike addition and subtraction, you need separate instructions for these two operations.

The multiply instructions take the following forms:

Unsigned Multiplication:

```
mul( reg8 );        // returns "ax"
mul( reg16 );       // returns "dx:ax"
mul( reg32 );       // returns "edx:eax"

mul( mem8 );        // returns "ax"
mul( mem16 );       // returns "dx:ax"
mul( mem32 );       // returns "edx:eax"
```

Signed (Integer) Multiplication:

```
imul( reg8 );       // returns "ax"
imul( reg16 );      // returns "dx:ax"
imul( reg32 );      // returns "edx:eax"
```

```
        imul( mem8 );       // returns "ax"
        imul( mem16 );      // returns "dx:ax"
        imul( mem32 );      // returns "edx:eax"
```

The "returns" values above are the strings these instructions return for use with instruction composition in HLA (see "Instruction Composition in HLA" on page 558).

(I)MUL, available on all 80x86 processors, multiplies eight, sixteen, or thirty-two bit operands. Note that when multiplying two n-bit values, the result may require as many as 2*n bits. Therefore, if the operand is an eight bit quantity, the result could require sixteen bits. Likewise, a 16 bit operand produces a 32 bit result and a 32 bit operand requires 64 bits to hold the result.

The (I)MUL instruction, with an eight bit operand, multiplies the AL register by the operand and leaves the 16 bit product in AX. So

```
        mul( operand8 );
or      imul( operand8 );
```

computes:

```
        AX := AL * operand8
```

"*" represents an unsigned multiplication for MUL and a signed multiplication for IMUL.

If you specify a 16 bit operand, then MUL and IMUL compute:

```
        DX:AX := AX * operand16
```

"*" has the same meanings as above and DX:AX means that DX contains the H.O. word of the 32 bit result and AX contains the L.O. word of the 32 bit result. If you're wondering why Intel didn't put the 32-bit result in EAX, just note that Intel introduced the MUL and IMUL instructions in the earliest 80x86 processors, before the advent of 32-bit registers in the 80386 CPU.

If you specify a 32 bit operand, then MUL and IMUL compute the following:

```
        EDX:EAX := EAX * operand32
```

"*" has the same meanings as above and EDX:EAX means that EDX contains the H.O. double word of the 64 bit result and EAX contains the L.O. double word of the 64 bit result.

If an 8x8, 16x16, or 32x32 bit product requires more than eight, sixteen, or thirty-two bits (respectively), the MUL and IMUL instructions set the carry and overflow flags. MUL and IMUL scramble the sign, and zero flags. **Especially note that the sign and zero flags do not contain meaningful values after the execution of these two instructions**.

To help reduce some of the problems with the use of the MUL and IMUL instructions, HLA provides an extended syntax that allows the following two-operand forms:

Unsigned Multiplication:

```
        mul( reg8, al );
        mul( reg16, ax );
        mul( reg32, eax );

        mul( mem8, al );
        mul( mem16, ax );
        mul( mem32, eax );

        mul( constant8, al );
        mul( constant16, ax );
        mul( constant32, eax );
```

Signed (Integer) Multiplication:

```
imul( reg8, al );
imul( reg16, ax );
imul( reg32, eax );

imul( mem8, al );
imul( mem16, ax );
imul( mem32, eax );

imul( constant8, al );
imul( constant16, ax );
imul( constant32, eax );
```

The two operand forms let you specify the (L.O.) destination register. The instructions whose first operand is a register or memory location are completely identical to the instructions above. By specifying the destination register, however, you can make your programs easier to read; therefore, it's probably a good idea to go ahead and specify the destination register. Note that just because HLA allows two operands here, you can't specify an arbitrary register. The destination operand must always be AL, AX, or EAX, depending on the source operand.

Note that HLA allows a form that lets you specify a constant. The 80x86 doesn't actually support a MUL or IMUL instruction that has a constant operand. HLA will take the constant you specify and create a "variable" in the special "const" segment in memory and initialize that variable with this value. Then HLA converts the instruction to the "(I)MUL(memory);" instruction. Generally, you won't need to use this special form since the INTMUL instruction will multiply a register by a constant.

You'll use the MUL and IMUL instructions quite a bit when you learn about extended precision arithmetic in the chapter on Advanced Arithmetic. Until you get to that chapter, you'll probably just want to use the INTMUL instruction in place of the MUL or IMUL since it is more general. However, INTMUL is not a complete replacement for these two instructions. Besides the number of operands, there are several differences between the INTMUL instruction you've learned about earlier and the MUL and IMUL instructions. Specifically for the INTMUL instruction:

- There isn't an 8x8 bit INTMUL instruction available (the immediate$_8$ operands simply provide a shorter form of the instruction. Internally, the CPU sign extends the operand to 16 or 32 bits as necessary).
- The INTMUL instruction does not produce a 2*n bit result. That is, a 16x16 multiply produces a 16 bit result. Likewise, a 32x32 bit multiply produces a 32 bit result. These instructions set the carry and overflow flags if the result does not fit into the destination register.

10.2.2 The DIV and IDIV Instructions

The 80x86 divide instructions perform a 64/32 division, a 32/16 division or a 16/8 division. These instructions take the form:

```
div( reg8 );              // returns "al"
div( reg16 );             // returns "ax"
div( reg32 );             // returns "eax"

div( reg8, AX );          // returns "al"
div( reg16, DX:AX );
div( reg32, EDX:EAX );

div( mem8 );              // returns "al"
div( mem16 );             // returns "ax"
div( mem32 );             // returns "eax"
div( mem8, AX );          // returns "al"
div( mem16, DX:AX );      // returns "ax"
div( mem32, EDX:EAX );    // returns "eax"

div( constant8, AX );         // returns "al"
div( constant16, DX:AX );     // returns "ax"
div( constant32, EDX:EAX );   // returns "eax"

idiv( reg8 );                 // returns "al"
idiv( reg16 );                // returns "ax"
idiv( reg32 );                // returns "eax"

idiv( reg8, AX );             // returns "al"
idiv( reg16, DX:AX );         // returns "ax"
idiv( reg32, EDX:EAX );       // returns "eax"

idiv( mem8 );                 // returns "al"
idiv( mem16 );                // returns "ax"
idiv( mem32 );                // returns "eax"

idiv( mem8, AX );             // returns "al"
idiv( mem16, DX:AX );         // returns "ax"
idiv( mem32, EDX:EAX );       // returns "eax"

idiv( constant8, AX );       // returns "al"
idiv( constant16, DX:AX );   // returns "ax"
idiv( constant32, EDX:EAX ); // returns "eax"
```

The DIV instruction computes an unsigned division. If the operand is an eight bit operand, DIV divides the AX register by the operand leaving the quotient in AL and the remainder (modulo) in AH. If the operand is a 16 bit quantity, then the DIV instruction divides the 32 bit quantity in DX:AX by the operand leaving the quotient in AX and the remainder in DX. With 32 bit operands DIV divides the 64 bit value in EDX:EAX by the operand leaving the quotient in EAX and the remainder in EDX.

You cannot, on the 80x86, simply divide one eight bit value by another. If the denominator is an eight bit value, the numerator must be a sixteen bit value. If you need to divide one unsigned eight bit value by another, you must zero extend the numerator to sixteen bits. You can accomplish this by loading the numerator into the AL register and then moving zero into the AH register. Then you can divide AX by the denominator operand to produce the correct result. *Failing to zero extend AL before executing DIV may cause the 80x86 to produce incorrect results!*

When you need to divide two 16 bit unsigned values, you must zero extend the AX register (which contains the numerator) into the DX register. To do this, just load zero into the DX register. If you need to divide one 32-bit value by another, you must zero extend the EAX register into EDX (by loading a zero into EDX) before the division.

When dealing with signed integer values, you will need to sign extend AL into AX, AX into DX or EAX into EDX before executing IDIV. To do so, use the CBW, CWD, CDQ, or MOVSX instructions. If the H.O. byte or word does not already contain significant bits, then you must sign extend the value in the accumulator (AL/AX/EAX) before doing the IDIV operation. Failure to do so may produce incorrect results.

There is one other catch to the 80x86's divide instructions: you can get a fatal error when using this instruction. First, of course, you can attempt to divide a value by zero. Second, the quotient may be too large to fit into the EAX, AX, or AL register. For example, the 16/8 division "$8000 / 2" produces the quotient $4000 with a remainder of zero. $4000 will not fit into eight bits. If this happens, or you attempt to divide by zero, the 80x86 will generate an *ex.DivisionError* exception or integer overflow error (*ex.IntoInstr*). This usually means your program will display the appropriate dialog box and abort your program. If this happens to you, chances are you didn't sign or zero extend your numerator before executing the division operation. Since this error will cause your program to crash, you should be very careful about the values you select when using division. Of course, you can use the TRY..ENDTRY block with the *ex.DivisionError* and *ex.IntoInstr* to trap this problem in your program.

The carry, overflow, sign, and zero flags are undefined after a division operation. Like MUL and IMUL, HLA provides special syntax to allow the use of constant operands even though these instructions don't really support them.

The 80x86 does not provide a separate instruction to compute the remainder of one number divided by another. The DIV and IDIV instructions automatically compute the remainder at the same time they compute the quotient. HLA, however, provides mnemonics (instructions) for the MOD and IMOD instructions. These special HLA instructions compile into the exact same code as their DIV and IDIV counterparts. The only difference is the "returns" value for the instruction (since these instructions return the remainder in a different location than the quotient). The MOD and IMOD instructions that HLA supports are

```
mod( reg8 );                 // returns "ah"
mod( reg16 );                // returns "dx"
mod( reg32 );                // returns "edx"

mod( reg8, AX );             // returns "ah"
mod( reg16, DX:AX );         // returns "dx"
mod( reg32, EDX:EAX );       // returns "edx"

mod( mem8 );                 // returns "ah"
mod( mem16 );                // returns "dx"
mod( mem32 );                // returns "edx"

mod( mem8, AX );             // returns "ah"
mod( mem16, DX:AX );         // returns "dx"
mod( mem32, EDX:EAX );       // returns "edx"

mod( constant8, AX );        // returns "ah"
mod( constant16, DX:AX );    // returns "dx"
mod( constant32, EDX:EAX );  // returns "edx"

imod( reg8 );                // returns "ah"
imod( reg16 );               // returns "dx"
imod( reg32 );               // returns "edx"

imod( reg8, AX );            // returns "ah"
imod( reg16, DX:AX );        // returns "dx"
imod( reg32, EDX:EAX );      // returns "edx"

imod( mem8 );                // returns "ah"
imod( mem16 );               // returns "dx"
imod( mem32 );               // returns "edx"

imod( mem8, AX );            // returns "ah"
imod( mem16, DX:AX );        // returns "dx"
imod( mem32, EDX:EAX );      // returns "edx"

imod( constant8, AX );       // returns "ah"
imod( constant16, DX:AX );   // returns "dx"
imod( constant32, EDX:EAX ); // returns "edx"
```

10.2.3 The CMP Instruction

The CMP (compare) instruction is identical to the SUB instruction with one crucial difference – it does not store the difference back into the destination operand. The syntax for the CMP instruction is similar to SUB (though the operands are reversed so it reads better), the generic form is

```
cmp( LeftOperand, RightOperand );
```

This instruction computes "LeftOperand - RightOperand" (note the reversal from SUB). The specific forms are

```
cmp( reg, reg );      // Registers must be the same size (8, 16, or 32 bits)
cmp( reg, mem );      // Sizes must match.
cmp( reg, constant );
cmp( mem, constant );
```

Note that both operands are "source" operands, so the fact that a constant appears as the second operand is okay.

The CMP instruction updates the 80x86's flags according to the result of the subtraction operation (LeftOperand - RightOperand). The flags are generally set in an appropriate fashion so that we can read this instruction as "compare LeftOperand to RightOperand". You can test the result of the comparison by checking the appropriate flags in the flags register using the conditional set instructions (see the next section) or the conditional jump instructions.

Probably the first place to start when exploring the CMP instruction is to take a look at exactly how the CMP instruction affects the flags. Consider the following CMP instruction:

```
cmp( ax, bx );
```

This instruction performs the computation AX - BX and sets the flags depending upon the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if AX = BX. This is the only time AX - BX produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative. At first glance, you might think that this flag would be set if AX is less than BX but this isn't always the case. If AX=$7FFF and BX= -1 ($FFFF) subtracting AX from BX produces $8000, which is negative (and so the sign flag will be set). So, for signed comparisons anyway, the sign flag doesn't contain the proper status. For unsigned operands, consider AX=$FFFF and BX=1. AX is greater than BX but their difference is $FFFE which is still negative. As it turns out, the sign flag and the overflow flag, taken together, can be used for comparing two signed values.

O: The overflow flag is set after a CMP operation if the difference of AX and BX produced an overflow or underflow. As mentioned above, the sign flag and the overflow flag are both used when performing signed comparisons.

C: The carry flag is set after a CMP operation if subtracting BX from AX requires a borrow. This occurs only when AX is less than BX where AX and BX are both unsigned values.

Given that the CMP instruction sets the flags in this fashion, you can test the comparison of the two operands with the following flags:

```
cmp( Left, Right );
```

Table 1: Condition Code Settings After CMP

Unsigned operands:	Signed operands:
Z: equality/inequality	Z: equality/inequality
C: Left < Right (C=1) Left >= Right (C=0)	C: no meaning
S: no meaning	S: see below
O: no meaning	O: see below

For signed comparisons, the S (sign) and O (overflow) flags, taken together, have the following meaning:

If ((S=0) and (O=1)) or ((S=1) and (O=0)) then Left < Right when using a signed comparison.

If ((S=0) and (O=0)) or ((S=1) and (O=1)) then Left >= Right when using a signed comparison.

Note that (S xor O) is one if the left operand is less than the right operand. Conversely, (S xor O) is zero if the left operand is greater or equal to the right operand.

To understand why these flags are set in this manner, consider the following examples:

```
Left           minus     Right          S    O
------                   ------         -    -

$FFFF (-1)       -       $FFFE (-2)     0    0
$8000            -       $0001          0    1
$FFFE (-2)       -       $FFFF (-1)     1    0
$7FFF (32767)    -       $FFFF (-1)     1    1
```

Remember, the CMP operation is really a subtraction, therefore, the first example above computes (-1)-(-2) which is (+1). The result is positive and an overflow did not occur so both the S and O flags are zero. Since (S xor O) is zero, Left is greater than or equal to Right.

In the second example, the CMP instruction would compute (-32768)-(+1) which is (-32769). Since a 16-bit signed integer cannot represent this value, the value wraps around to $7FFF (+32767) and sets the overflow flag. The result is positive (at least as a 16 bit value) so the CPU clears the sign flag. (S xor O) is one here, so *Left* is less than *Right*.

In the third example above, CMP computes (-2)-(-1) which produces (-1). No overflow occurred so the O flag is zero, the result is negative so the sign flag is one. Since (S xor O) is one, Left is less than Right.

In the fourth (and final) example, CMP computes (+32767)-(-1). This produces (+32768), setting the overflow flag. Furthermore, the value wraps around to $8000 (-32768) so the sign flag is set as well. Since (S xor O) is zero, Left is greater than or equal to Right.

10.2.4 The SETcc Instructions

The *set on condition* (or SET*cc*) instructions set a single byte operand (register or memory location) to zero or one depending on the values in the flags register. The general formats for the SET*cc* instructions are

```
setcc( reg8 );
setcc( mem8 );
```

SET*cc* represents a mnemonic appearing in the following tables. These instructions store a zero into the corresponding operand if the condition is false, they store a one into the eight bit operand if the condition is true.

Table 2: SETcc Instructions That Test Flags

Instruction	Description	Condition	Comments
SETC	Set if carry	Carry = 1	Same as SETB, SETNAE
SETNC	Set if no carry	Carry = 0	Same as SETNB, SETAE
SETZ	Set if zero	Zero = 1	Same as SETE
SETNZ	Set if not zero	Zero = 0	Same as SETNE
SETS	Set if sign	Sign = 1	
SETNS	Set if no sign	Sign = 0	
SETO	Set if overflow	Ovrflw=1	
SETNO	Set if no overflow	Ovrflw=0	
SETP	Set if parity	Parity = 1	Same as SETPE
SETPE	Set if parity even	Parity = 1	Same as SETP
SETNP	Set if no parity	Parity = 0	Same as SETPO
SETPO	Set if parity odd	Parity = 0	Same as SETNP

The SET*cc* instructions above simply test the flags without any other meaning attached to the operation. You could, for example, use SETC to check the carry flag after a shift, rotate, bit test, or arithmetic operation. You might notice the SETP, SETPE, and SETNP instructions above. They check the *parity* flag. These instructions appear here for completeness, but this text will not consider the uses of the parity flag.

The CMP instruction works synergistically with the SET*cc* instructions. Immediately after a CMP operation the processor flags provide information concerning the relative values of those operands. They allow you to see if one operand is less than, equal to, greater than, or any combination of these.

There are two additional groups of SET*cc* instructions that are very useful after a CMP operation. The first group deals with the result of an *unsigned* comparison, the second group deals with the result of a *signed* comparison.

Table 3: SETcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Comments
SETA	Set if above (>)	Carry=0, Zero=0	Same as SETNBE
SETNBE	Set if not below or equal (not <=)	Carry=0, Zero=0	Same as SETA
SETAE	Set if above or equal (>=)	Carry = 0	Same as SETNC, SETNB
SETNB	Set if not below (not <)	Carry = 0	Same as SETNC, SETAE
SETB	Set if below (<)	Carry = 1	Same as SETC, SETNAE
SETNAE	Set if not above or equal (not >=)	Carry = 1	Same as SETC, SETB
SETBE	Set if below or equal (<=)	Carry = 1 or Zero = 1	Same as SETNA
SETNA	Set if not above (not >)	Carry = 1 or Zero = 1	Same as SETBE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The corresponding table for signed comparisons is

Table 4: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETG	Set if greater (>)	Sign = Ovrflw and Zero=0	Same as SETNLE
SETNLE	Set if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	Same as SETG
SETGE	Set if greater than or equal (>=)	Sign = Ovrflw	Same as SETNL
SETNL	Set if not less than (not <)	Sign = Ovrflw	Same as SETGE
SETL	Set if less than (<)	Sign ≠ Ovrflw	Same as SETNGE

Table 4: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETNGE	Set if not greater or equal (not >=)	Sign ≠ Ovrflw	Same as SETL
SETLE	Set if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	Same as SETNG
SETNG	Set if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	Same as SETLE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The SET*cc* instructions are particularly valuable because they can convert the result of a comparison to a boolean value (false/true or 0/1). This is especially important when translating statements from a high level language like Pascal or C/C++ into assembly language. The following example shows how to use these instructions in this manner:

```
// Bool := A <= B

        mov( A, eax );
        cmp( eax, B );
        setle( bool );    // bool is a boolean or byte variable.
```

Since the SET*cc* instructions always produce zero or one, you can use the results with the AND and OR instructions to compute complex boolean values:

```
// Bool := ((A <= B) and (D = E))

        mov( A, eax );
        cmp( eax, B );
        setle( bl );
        mov( D, eax );
        cmp( eax, E );
        sete( bh );
        and( bl, bh );
        mov( bh, Bool );
```

For more examples, see “Logical (Boolean) Expressions” on page 604.

10.2.5 The TEST Instruction

The 80x86 TEST instruction is to the AND instruction what the CMP instruction is to SUB. That is, the TEST instruction computes the logical AND of its two operands and sets the condition code flags based on the result; it does not, however, store the result of the logical AND back into the destination operand. The syntax for the TEST instruction is similar to AND, it is

```
test( operand1, operand2 );
```

The TEST instruction sets the zero flag if the result of the logical AND operation is zero. It sets the sign flag if the H.O. bit of the result contains a one. TEST always clears the carry and overflow flags.

The primary use of the TEST instruction is to check to see if an individual bit contains a zero or a one. Consider the instruction “test(1, AL);” This instruction logically ANDs AL with the value one; if bit one of AL contains zero, the result will be zero (setting the zero flag) since all the other bits in the constant one are zero. Conversely, if bit one of AL contains one, then the result is not zero so TEST clears the zero flag. Therefore, you can test the zero flag after this TEST instruction to see if bit zero contains a zero or a one.

The TEST instruction can also check to see if all the bits in a specified set of bits contain zero. The instruction "test($F, AL);" sets the zero flag if and only if the L.O. four bits of AL all contain zero.

One very important use of the TEST instruction is to check to see if a register contains zero. The instruction "TEST(reg, reg);" where both operands are the same register will logically AND that register with itself. If the register contains zero, then the result is zero and the CPU will set the zero flag. However, if the register contains a non-zero value, logically ANDing that value with itself produces that same non-zero value, so the CPU clears the zero flag. Therefore, you can test the zero flag immediately after the execution of this instruction (e.g., using the SETZ or SETNZ instructions) to see if the register contains zero. E.g.,

```
test( eax, eax );
setz( bl );          // BL is set to one if EAX contains zero.
```

10.3 Arithmetic Expressions

Probably the biggest shock to beginners facing assembly language for the very first time is the lack of familiar arithmetic expressions. Arithmetic expressions, in most high level languages, look similar to their algebraic equivalents, e.g.,

```
X:=Y*Z;
```

In assembly language, you'll need several statements to accomplish this same task, e.g.,

```
mov( y, eax );
intmul( z, eax );
mov( eax, x );
```

Obviously the HLL version is much easier to type, read, and understand. This point, more than any other, is responsible for scaring people away from assembly language.

Although there is a lot of typing involved, converting an arithmetic expression into assembly language isn't difficult at all. By attacking the problem in steps, the same way you would solve the problem by hand, you can easily break down any arithmetic expression into an equivalent sequence of assembly language statements. By learning how to convert such expressions to assembly language in three steps, you'll discover there is little difficulty to this task.

10.3.1 Simple Assignments

The easiest expressions to convert to assembly language are the simple assignments. Simple assignments copy a single value into a variable and take one of two forms:

```
variable := constant
```

or

```
variable := variable
```

Converting the first form to assembly language is trivial, just use the assembly language statement:

```
mov( constant, variable );
```

This MOV instruction copies the constant into the variable.

The second assignment above is slightelly more complicated since the 80x86 doesn't provide a memory–to-memory MOV instruction. Therefore, to copy one memory variable into another, you must move the data through a register. By convention (and for slight efficiency reasons), most programmers tend to use AL/AX/EAX for this purpose. If the AL, AX, or EAX register is available, you should use it for this operation. For example,

```
        var1 := var2;
```

becomes

```
        mov( var2, eax );
        mov( eax, var1 );
```

This is assuming, of course, that *var1* and *var2* are 32-bit variables. Use AL if they are eight bit variables, use AX if they are 16-bit variables.

Of course, if you're already using AL, AX, or EAX for something else, one of the other registers will suffice. Regardless, you must use a register to transfer one memory location to another.

Although the 80x86 does not support a memory-to-memory move, HLA does provide an extended syntax for the MOV instruction that allows two memory operands. However, both operands have to be 16-bit or 32-bit values; eight-bit values won't work. Assuming you want to copy the value of a word or dword object to another variable, you can use the following syntax:

```
    mov( var2, var1 );
```

HLA translates this "instruction" into the following two instruction sequence:

```
    push( var2 );
    pop( var1 );
```

Although this is slightly slower than the two MOV instructions, it is convenient.

10.3.2 Simple Expressions

The next level of complexity up from a simple assignment is a simple expression. A simple expression takes the form:

        var$_1$:= term$_1$ op term$_2$;

Var1 is a variable, *term1* and *term2* are variables or constants, and *op* is some arithmetic operator (addition, subtraction, multiplication, etc.).

As simple as this expression appears, most expressions take this form. It should come as no surprise then, that the 80x86 architecture was optimized for just this type of expression.

A typical conversion for this type of expression takes the following form:

        mov(term$_1$, eax);
        op(term$_2$, eax);
        mov(eax, var$_1$)

Op is the mnemonic that corresponds to the specified operation (e.g., "+" = add, "-" = sub, etc.).

There are a few inconsistencies you need to be aware of. Of course, when dealing with the multiply and divide instructions on the 80x86, you must use the AL/AX/EAX and DX/EDX registers. You cannot use arbitrary registers as you can with other operations. Also, don't forget the sign extension instructions if you're performing a division operation and you're dividing one 16/32 bit number by another. Finally, don't forget that some instructions may cause overflow. You may want to check for an overflow (or underflow) condition after an arithmetic operation.

Examples of common simple expressions:

```
x := y + z;

        mov( y, eax );
        add( z, eax );
        mov( eax, x );

x := y - z;
```

```
        mov( y, eax );
        sub( z, eax );
        mov( eax, x );

x := y * z; {unsigned}

        mov( y, eax );
        mul( z, eax );     // Don't forget this wipes out EDX.
        mov( eax, x );

x := y div z; {unsigned div}

        mov( y, eax );
        mov( 0, edx );       // Zero extend EAX into EDX.
        div( z, edx:eax );
        mov( eax, x );

x := y idiv z; {signed div}

        mov( y, eax );
        cdq();               // Sign extend EAX into EDX.
        idiv( z, edx:eax );
        mov( eax, z );

x := y mod z; {unsigned remainder}

        mov( y, eax );
        mov( 0, edx );       // Zero extend EAX into EDX.
        mod( z, edx:eax );
        mov( edx, x );       // Note that remainder is in EDX.

x := y imod z; {signed remainder}

        mov( y, eax );
        cdq();               // Sign extend EAX into EDX.
        imod( z, edx:eax );
        mov( edx, x );       // Remainder is in EDX.
```

Certain unary operations also qualify as simple expressions. A good example of a unary operation is negation. In a high level language negation takes one of two possible forms:

var := -var or var_1 := -var_2

Note that var := -constant is really a simple assignment, not a simple expression. You can specify a negative constant as an operand to the MOV instruction:

```
mov( -14, var );
```

To handle “var = -var;” use the single assembly language statement:

```
// var = -var;

neg( var );
```

If two different variables are involved, then use the following:

```
// var1 = -var2;

mov( var2, eax );
neg( eax );
mov( eax, var1 );
```

10.3.3 Complex Expressions

A complex expression is any arithmetic expression involving more than two terms and one operator. Such expressions are commonly found in programs written in a high level language. Complex expressions may include parentheses to override operator precedence, function calls, array accesses, etc. While the conversion of some complex expressions to assembly language is fairly straight-forward, others require some effort. This section outlines the rules you use to convert such expressions.

A complex expression that is easy to convert to assembly language is one that involves three terms and two operators, for example:

```
        w := w - y - z;
```

Clearly the straight-forward assembly language conversion of this statement will require two SUB instructions. However, even with an expression as simple as this one, the conversion is not trivial. There are actually *two ways* to convert this from the statement above into assembly language:

```
        mov( w, eax );
        sub( y, eax );
        sub( z, eax );
        mov( eax, w );
and
        mov( y, eax );
        sub( z, eax );
        sub( eax, w );
```

The second conversion, since it is shorter, looks better. However, it produces an incorrect result (assuming Pascal-like semantics for the original statement). Associativity is the problem. The second sequence above computes W := W - (Y - Z) which is not the same as W := (W - Y) - Z. How we place the parentheses around the subexpressions can affect the result. Note that if you are interested in a shorter form, you can use the following sequence:

```
        mov( y, eax );
        add( z, eax );
        sub( eax, w );
```

This computes W:=W-(Y+Z). This is equivalent to W := (W - Y) - Z.

Precedence is another issue. Consider the Pascal expression:

```
        X := W * Y + Z;
```

Once again there are two ways we can evaluate this expression:

```
        X := (W * Y) + Z;
or
        X := W * (Y + Z);
```

By now, you're probably thinking that this text is crazy. Everyone knows the correct way to evaluate these expressions is the second form provided in these two examples. However, you're wrong to think that way. The APL programming language, for example, evaluates expressions solely from right to left and does not give one operator precedence over another.

Most high level languages use a fixed set of precedence rules to describe the order of evaluation in an expression involving two or more different operators. Such programming languages usually compute multiplication and division before addition and subtraction. Those that support exponentiation (e.g., FORTRAN and BASIC) usually compute that before multiplication and division. These rules are intuitive since almost everyone learns them before high school. Consider the expression:

$$X\ op_1\ Y\ op_2\ Z$$

If op_1 takes precedence over op_2 then this evaluates to (X op_1 Y) op_2 Z otherwise if op_2 takes precedence over op_1 then this evaluates to X op_1 (Y op_2 Z). Depending upon the operators and operands involved, these two computations could produce different results.

When converting an expression of this form into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique:

```
// w := x + y * z;

       mov( x, ebx );
       mov( y, eax );      // Must compute y*z first since "*"
       intmul( z, eax );   //  has higher precedence than "+".
       add( ebx, eax );
       mov( eax, w );
```

If two operators appearing within an expression have the same precedence, then you determine the order of evaluation using *associativity* rules. Most operators are *left associative* meaning that they evaluate from left to right. Addition, subtraction, multiplication, and division are all left associative. A *right associative* operator evaluates from right to left. The exponentiation operator in FORTRAN and BASIC is a good example of a right associative operator:

```
       2^2^3 is equal to 2^(2^3) not (2^2)^3
```

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

```
// w := x - y - z

       mov( x, eax );   // All the same operator, so we need
       sub( y, eax );   //  to evaluate from left to right
       sub( z, eax );   //  because they all have the same
       mov( eax, w );   //  precedence and are left associative.

// w := x + y * z

       mov( y, eax );      // Must compute Y * Z first since
       intmul( z, eax );   // multiplication has a higher
       add( x, eax );      // precedence than addition.
       mov( eax, w );


// w := x / y - z

       mov( x, eax );      // Here we need to compute division
       cdq();              //  first since it has the highest
       idiv( y, edx:eax ); //  precedence.
       sub( z, eax );
       mov( eax, w );

// w := x * y * z

       mov( y, eax );      // Addition and multiplication are
       intmul( z, eax );   // commutative, therefore the order
       intmul( x, eax );   // of evaluation does not matter
       mov( eax, w );
```

There is one exception to the associativity rule. If an expression involves multiplication and division it is generally better to perform the multiplication first. For example, given an expression of the form:

`W := X/Y * Z        // Note: this is` $\frac{x}{y} \times z$ `not` $\frac{x}{y \times z}$ `!`

It is usually better to compute X*Z and then divide the result by Y rather than divide X by Y and multiply the quotient by Z. There are two reasons this approach is better. First, remember that the IMUL instruction always produces a 64 bit result (assuming 32 bit operands). By doing the multiplication first, you automatically *sign extend* the product into the EDX register so you do not have to sign extend EAX prior to the division. This saves the execution of the CDQ instruction. A second reason for doing the multiplication first is to increase the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute 5/2 you will get the value two, not 2.5. Computing (5/2)*3 produces six. However, if you compute (5*3)/2 you get the value seven which is a little closer to the real quotient (7.5). Therefore, if you encounter an expression of the form:

```
w := x/y*z;
```

You can usually convert it to the assembly code:

```
mov( x, eax );
imul( z, eax );       // Note the use of IMUL, not INTMUL!
idiv( y, edx:eax );
mov( eax, w );
```

Of course, if the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. Moral of the story: always make sure you fully understand any expression you are converting to assembly language. Obviously if the semantics dictate that you must perform the division first, do so.

Consider the following Pascal statement:

```
w := x - y * x;
```

This is similar to a previous example except it uses subtraction rather than addition. Since subtraction is not commutative, you cannot compute *y* * *z* and then subtract *x* from this result. This tends to complicate the conversion a tiny amount. Rather than a straight forward multiply and addition sequence, you'll have to load x into a register, multiply *y* and *z* leaving their product in a different register, and then subtract this product from *x*, e.g.,

```
mov( x, ebx );
mov( y, eax );
intmul( x, eax );
sub( eax, ebx );
mov( ebx, w );
```

This is a trivial example that demonstrates the need for *temporary variables* in an expression. This code uses the EBX register to temporarily hold a copy of *x* until it computes the product of *y* and *z*. As your expressions increase in complexity, the need for temporaries grows. Consider the following Pascal statement:

```
w := (a + b) * (y + z);
```

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (i.e., the two subexpressions with the highest precedence) first and set their values aside. When you computed the values for both subexpressions you can compute their sum. One way to deal with complex expressions like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, we can convert the single expression above into the following sequence:

$Temp_1$:= a + b;
$Temp_2$:= y + z;
w := $Temp_1$ * $Temp_2$;

Since converting simple expressions to assembly language is quite easy, it's now a snap to compute the former, complex, expression in assembly. The code is

```
mov( a, eax );
add( b, eax );
mov( eax, Temp1 );
mov( y, eax );
add( z, eax );
mov( eax, Temp2 );
mov( Temp1, eax );
intmul( Temp2, eax );
mov( eax, w );
```

Of course, this code is grossly inefficient and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results this code becomes:

```
mov( a, eax );
add( b, eax );
mov( y, ebx );
add( z, ebx );
intmul( ebx, eax );
mov( eax, w );
```

Yet another example:

```
x := (y+z) * (a-b) / 10;
```

This can be converted to a set of four simple expressions:

```
Temp1 := (y+z)
Temp2 := (a-b)
Temp1 := Temp1 * Temp2
X := Temp1 / 10
```

You can convert these four simple expressions into the assembly language statements:

```
mov( y, eax );      // Compute eax = y+z
add( z, eax );
mov( a, ebx );      // Compute ebx = a-b
sub( b, ebx );
imul( ebx, eax );   // This also sign extends eax into edx.
idiv( 10, edx:eax );
mov( eax, x );
```

The most important thing to keep in mind is that temporary values, if possible, should be kept in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers to use.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the results. Since you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly trivial. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

10.3.4 Commutative Operators

If "@" represents some operator, that operator is *commutative* if the following relationship is always true:

```
(A @ B) = (B @ A)
```

As you saw in the previous section, commutative operators are nice because the order of their operands is immaterial and this lets you rearrange a computation, often making that computation easier or more efficient. Often, rearranging a computation allows you to use fewer temporary variables. Whenever you encounter a commutative operator in an expression, you should always check to see if there is a better sequence you can use to improve the size or speed of your code. The following tables list the commutative and non-commutative operators you typically find in high level languages:

Table 5: Some Common Commutative Binary Operators

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
AND	&& or &	Logical or bitwise AND
OR	|| or |	Logical or bitwise OR
XOR	^	(Logical or) Bitwise exclusive-OR
=	==	Equality
<>	!=	Inequality

Table 6: Some Common Noncommutative Binary Operators

Pascal	C/C++	Description
-	-	Subtraction
/ or DIV	/	Division
MOD	%	Modulo or remainder
<	<	Less than
<=	<=	Less than or equal
>	>	Greater than
>=	>=	Greater than or equal

10.4 Logical (Boolean) Expressions

Consider the following expression from a Pascal program:

```
B := ((X=Y) and (A <= C)) or ((Z-A) <> 5);
```

`B` is a boolean variable and the remaining variables are all integers.

How do we represent boolean variables in assembly language? Although it takes only a single bit to represent a boolean value, most assembly language programmers allocate a whole byte or word for this purpose (as such, HLA also allocates a whole byte for a BOOLEAN variable). With a byte, there are 256 possible values we can use to represent the two values *true* and *false*. So which two values (or which two sets of values) do we use to represent these boolean values? Because of the machine's architecture, it's much easier to test for conditions like zero or not zero and positive or negative rather than to test for one of two particular boolean values. Most programmers (and, indeed, some programming languages like "C") choose zero to represent false and anything else to represent true. Some people prefer to represent true and false with one and zero (respectively) and not allow any other values. Others select all one bits ($FFFF_FFFF, $FFFF, or $FF) for true and 0 for false. You could also use a positive value for true and a negative value for false. All these mechanisms have their own advantages and drawbacks.

Using only zero and one to represent false and true offers two very big advantages: (1) The SETcc instructions produce these results so this scheme is compatible with those instructions; (2) the 80x86 logical instructions (AND, OR, XOR and, to a lesser extent, NOT) operate on these values exactly as you would expect. That is, if you have two boolean variables A and B, then the following instructions perform the basic logical operations on these two variables:

```
// c = a AND b;

    mov( a, al );
    and( b, al );
    mov( al, c );

// c = a OR b;

        mov( a, al );
        or( b, al );
        mov( al, c );

// c = a XOR b;

        mov( a, al );
        xor( b, al );
        mov( al, c );

// b = not a;

        mov( a, al );      // Note that the NOT instruction does not
        not( al );         // properly compute al = not al by itself.
        and( 1, al );      // I.e., (not 0) does not equal one.  The AND
        mov( al, b );      // instruction corrects this problem.

        mov( a, al );      // Another way to do b = not a;
        xor( 1, al );      // Inverts bit zero.
        mov( al, b );
```

Note, as pointed out above, that the NOT instruction will not properly compute logical negation. The bitwise not of zero is $FF and the bitwise not of one is $FE. Neither result is zero or one. However, by ANDing the result with one you get the proper result. Note that you can implement the NOT operation more efficiently using the "xor(1, ax);" instruction since it only affects the L.O. bit.

As it turns out, using zero for false and anything else for true has a lot of subtle advantages. Specifically, the test for true or false is often implicit in the execution of any logical instruction. However, this mechanism suffers from a very big disadvantage: you cannot use the 80x86 AND, OR, XOR, and NOT instructions to implement the boolean operations of the same name. Consider the two values $55 and $AA. They're both non-zero so they both represent the value true. However, if you logically AND $55 and $AA together using the 80x86 AND instruction, the result is zero. True AND true should produce true, not false. Although you can account for situations like this, it usually requires a few extra instructions and is somewhat less efficient when computing boolean operations.

A system that uses non-zero values to represent true and zero to represent false is an *arithmetic logical system*. A system that uses the two distinct values like zero and one to represent false and true is called a *boolean logical system*, or simply a boolean system. You can use either system, as convenient. Consider again the boolean expression:

```
B := ((X=Y) and (A <= D)) or ((Z-A) <> 5);
```

The simple expressions resulting from this expression might be:

```
        mov( x, eax );
        cmp( y, eax );
        sete( al );       // AL := x = y;

        mov( a, ebx );
        cmp( ebx, d );
        setle( bl );     // BL := a <= d;
        and( al, bl );   // BL := (x=y) and (a <= d);

        mov( z, eax );
        sub( a, eax );
        cmp( eax, 5 );
        setne( al );
        or( bl, al );     // AL := ((X=Y) and (A <= D)) or ((Z-A) <> 5);
        mov( al, b );
```

When working with boolean expressions don't forget the that you might be able to optimize your code by simplifying those boolean expressions. You can use algebraic transformations (especially DeMorgan's theorems) to help reduce the complexity of an expression. In the chapter on low-level control structures you'll also see how to use control flow to calculate a boolean result. This is generally quite a bit more efficient than using *complete boolean evaluation* as the examples in this section teach.

10.5 Machine and Arithmetic Idioms

An idiom is an idiosyncrasy. Several arithmetic operations and 80x86 instructions have idiosyncrasies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as "tricky programming" that you should always avoid in well written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well-known and commonly found in assembly language programs. Some of them can be really tricky, but a good number of them are simply "tricks of the trade." This text cannot even begin to present all of the idioms in common use today; they are too numerous and the list is constantly changing. Nevertheless, there are some very important idioms that you will see all the time, so it makes sense to discuss those.

10.5.1 Multiplying without MUL, IMUL, or INTMUL

If you take a quick look at the timing for the multiply instruction, you'll notice that the execution time for this instruction is often long[1]. When multiplying by a constant, you can sometimes avoid the performance penalty of the MUL, IMUL, and INTMUL instructions by using shifts, additions, and subtractions to perform the multiplication.

Remember, a SHL instruction computes the same result as multiplying the specified operand by two. Shifting to the left two bit positions multiplies the operand by four. Shifting to the left three bit positions multiplies the operand by eight. In general, shifting an operand to the left n bits multiplies it by 2^n. Any value can be multiplied by some constant using a series of shifts and adds or shifts and subtractions. For example, to multiply the AX register by ten, you need only multiply it by eight and then add in two times the original value. That is, 10*AX = 8*AX + 2*AX. The code to accomplish this is

```
        shl( 1, ax );     // Multiply AX by two.
        mov( ax, bx);     // Save 2*AX for later.
        shl( 2, ax );     // Multiply ax by eight (*4 really, but it contains *2).
        add( bx, ax );    // Add in AX*2 to AX*8 to get AX*10.
```

The AX register (or just about any register, for that matter) can often be multiplied by many constant values much faster using SHL than by using the MUL instruction. This may seem hard to believe since it only takes one instruction to compute this product:

1. Actually, this is specific to a given processor. Some processors execute the INTMUL instruction fairly fast.

```
intmul( 10, ax );
```

However, if you look at the timings, the shift and add example above requires fewer clock cycles on many processors in the 80x86 family than the MUL instruction. Of course, the code is somewhat larger (by a few bytes), but the performance improvement is usually worth it. Of course, on the later 80x86 processors, the multiply instructions are quite a bit faster than the earlier processors, but the shift and add scheme is often faster on these processors as well.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by seven:

```
mov( eax, ebx );     // Save EAX * 1
shl( 3, eax );       // EAX = EAX * 8
sub( ebx, eax );     // EAX*8 - EAX*1 is EAX*7
```

This follows directly from the fact that EAX*7 = (EAX*8)-EAX.

A common error made by beginning assembly language students is subtracting or adding one or two rather than EAX*1 or EAX*2. The following does *not* compute eax*7:

```
shl( 3, eax );
sub( 1, eax );
```

It computes (8*EAX)-1, something entirely different (unless, of course, EAX = 1). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

You can also use the LEA instruction to compute certain products. The trick is to use the scaled index addressing modes. The following examples demonstrate some simple cases:

```
lea( eax, [ecx][ecx] );       // EAX := ECX * 2
lea( eax, [eax]eax*2] );      // EAX := EAX * 3
lea( eax, [eax*4] );          // EAX := EAX * 4
lea( eax, [ebx][ebx*4] );     // EAX := EBX * 5
lea( eax, [eax*8] );          // EAX := EAX * 8
lea( eax, [edx][edx*8] );     // EAX := EDX * 9
```

10.5.2 Division Without DIV or IDIV

Much as the SHL instruction can be used for simulating a multiplication by some power of two, you may use the SHR and SAR instructions to simulate a division by a power of two. Unfortunately, you cannot use shifts, additions, and subtractions to perform a division by an arbitrary constant as easily as you can use these instructions to perform a multiplication operation.

Another way to perform division is to use the multiply instructions. You can divide by some value by multiplying by its reciprocal. Since the multiply instruction is faster than the divide instruction; multiplying by a reciprocal is usually faster than division.

Now you're probably wondering "how does one multiply by a reciprocal when the values we're dealing with are all integers?" The answer, of course, is that we must cheat to do this. If you want to multiply by one tenth, there is no way you can load the value 1/10th into an 80x86 register prior to performing the multiplication. However, we could multiply 1/10th by 10, perform the multiplication, and then divide the result by ten to get the final result. Of course, this wouldn't buy you anything at all, in fact it would make things worse since you're now doing a multiplication by ten as well as a division by ten. However, suppose you multiply 1/10th by 65,536 (6553), perform the multiplication, and then divide by 65,536. This would still perform the correct operation and, as it turns out, if you set up the problem correctly, you can get the division operation for free. Consider the following code that divides AX by ten:

```
mov( 6554, dx );     // 6554 = round( 65,536/10 ).
mul( dx, ax );
```

This code leaves AX/10 in the DX register.

To understand how this works, consider what happens when you multiply AX by 65,536 ($10000). This simply moves AX into DX and sets AX to zero (a multiply by $10000 is equivalent to a shift left by sixteen bits). Multiplying by 6,554 (65,536 divided by ten) puts AX divided by ten into the DX register. Since MUL is faster than DIV , this technique runs a little faster than using a straight division.

Multiplying by the reciprocal works well when you need to divide by a constant. You could even use it to divide by a variable, but the overhead to compute the reciprocal only pays off if you perform the division many, many times (by the same value).

10.5.3 Implementing Modulo-N Counters with AND

If you want to implement a counter variable that counts up to 2^n-1 and then resets to zero, simply using the following code:

```
inc( CounterVar );
and( nBits, CounterVar );
```

where *nBits* is a binary value containing n one bits right justified in the number. For example, to create a counter that cycles between zero and fifteen, you could use the following:

```
inc( CounterVar );
and( %00001111, CounterVar );
```

10.5.4 Careless Use of Machine Idioms

One problem with using machine idioms is that the machines change over time. The DOS/16-bit version of this text recommends the use of several machine idioms in addition to those this chapter presents. Unfortunately, as time passed Intel improved the processor and tricks that used to provide a performance benefit are actually slower on the newer processors. Therefore, you should be careful about employing common "tricks" you pick up; they may not actually improve the code.

10.6 The HLA (Pseudo) Random Number Unit

The HLA rand.hhf module provides a set of pseudo-random generators that returns seemingly random values on each call. These pseudo-random number generator functions are great for writing games and other simulations that require a sequence of values that the user can not easily guess. These functions return a 32-bit value in the EAX register. You can treat the result as a signed or unsigned value as appropriate for your application.

The rand.hhf library module includes the following functions:

```
procedure rand.random; returns( "eax" );
procedure rand.range( startRange:dword; endRange:dword ); returns( "eax" );

procedure rand.uniform; returns( "eax" );
procedure rand.urange( startRange:dword; endRange:dword ); returns( "eax" );

procedure rand.randomize;
```

The *rand.random* and *rand.uniform* procedures are both functions that return a 32-bit pseudo-random number in the EAX register. They differ only in the algorithm they use to compute the random number sequence (*rand.random* uses a standard linear congruential generator, *rand.uniform* uses an additive generator. See Knuth's "The Art of Computer Programming" Volume Two for details on these two algorithms).

The *rand.range* and *rand.urange* functions return a pseudo-random number that falls between two values passed as parameters (inclusive). These routines use better algorithms than the typical "mod the result by the range of values and add the starting value" algorithm that naive users often employ to limit random numbers to a specific range (that naive algorithm generally produces a stream of numbers that is somewhat less than random).

By default, the random number generators in the HLA Standard Library generate the same sequence of numbers every time you run a program. While this may not seem random at all (and statistically, it certainly is not random), this is generally what you want in a random number generator. The numbers should appear to be random but you usually need to be able to generate the same sequence over and over again when testing your program. After all, a defect you encounter with one random sequence may not be apparent when using a different random number sequence. By emitting the same sequence over and over again, your programs become deterministic so you can properly test them across several runs of the program.

Once your program is tested and operational, you might want your random number generator to generate a different sequence every time you run the program. For example, if you write a game and that game uses a pseudo-random sequence to control the action, the end user may detect a pattern and play the game accordingly if the random number generator always returns the same sequence of numbers.

To alleviate this problem, the HLA Standard Library rand module provides the *rand.randomize* procedure. This procedure reads the current date and time (in milliseconds) and, on processors that support it, reads the CPU's timestamp counter to generate an almost random set of bits as the starting random number generator value. Calling the *rand.randomize* procedure at the beginning of your program essentially guarantees that different executions of the program will produce a different sequence of random numbers.

Note that you cannot make the sequence "more random" by calling *rand.randomize* multiple times. In fact, since *rand.randomize* generates a new seed based on the date and time, calling *rand.randomize* multiple times in your program will actually generate a less random sequence (since time is an ever increasing value, not a random value). So make at most one call to *rand.randomize* and leave it up to the random number generators to take it from there.

Note that *rand.randomize* will randomize both the *rand.random* and *rand.uniform* random number generators. You do not need separate calls for the two different generators nor can you randomize one without randomizing the other.

One attribute of a random number generator is "how uniform are the results the generator returns." A uniform random number generator[2] that produces a 32-bit result returns a sequence of values that are evenly distributed throughout the 32-bit range of values. That is, any return result is as equally likely as any other return result. Good random number generators don't tend to bunch numbers up in groups.

The following program code provides a simple test of the random number generators by plotting asterisks at random positions on the screen. This program works by choosing two random numbers, one between zero and 79, the other between zero and 23. Then the program uses the *console.puts* function to print a single asterisk at the (X,Y) coordinate on the screen specified by these two random numbers (therefore, this code runs only under Windows). After 10,000 iterations of this process the program stops and lets you observe the result. **Note**: since random number generators generate random numbers, you should not expect this program to fill the entire screen with asterisks in only 10,000 iterations.

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

begin testRandom;

    console.cls();
    mov( 10_000, ecx );
    repeat

        // Generate a random X-coordinate
```

2. Despite their names, both *rand.uniform* and *rand.random* generate a uniformly distributed set of pseudo-random numbers.

```
            // using rand.range.

            rand.range( 0, 79 );
            mov( eax, ebx );            // Save the X-coordinate for now.

            // Generate a random Y-coordinate
            // using rand.urange.

            rand.urange( 0, 23 );

            // Print an asterisk at
            // the specified coordinate on the screen.

            console.puts( ax, bx, "*" );

            // Repeat this 10,000 times to get
            // a good distribution of values.

            dec( ecx );

        until( @z );

        // Position the cursor at the bottom of the
        // screen so we can observe the results.

        console.gotoxy( 24, 0 );

    end testRandom;
```

Program 10.1 Screen Plot Test of the HLA Random Number Generators

The rand.hhf module also provides an *iterator* that generates a random sequence of value in the range 0..n-1. However, a discussion of this function must wait until we cover iterators in a later chapter.

10.7 Putting It All Together

This chapter finished the presentation of the integer arithmetic instructions on the 80x86. Then it demonstrated how to convert expressions from a high level language syntax into assembly language. This chapter concluded by teaching you a few assembly language tricks you will commonly find in programs. By the conclusion of this chapter you are (hopefully) in a position where you can easily evaluate arithmetic expressions in your assembly language programs.

Real Arithmetic Chapter Eleven

11.1 Chapter Overview

This chapter discusses the implementation of floating point arithmetic computation in assembly language. By the conclusion of this chapter you should be able to translate arithmetic expressions and assignment statements involving floating point operands from high level languages like Pascal and C/C++ into 80x86 assembly language.

11.2 Floating Point Arithmetic

When the 8086 CPU first appeared in the late 1970's, semiconductor technology was not to the point where Intel could put floating point instructions directly on the 8086 CPU. Therefore, they devised a scheme whereby they could use a second chip to perform the floating point calculations – the floating point unit (or FPU)[1]. They released their original floating point chip, the 8087, in 1980. This particular FPU worked with the 8086, 8088, 80186, and 80188 CPUs. When Intel introduced the 80286 CPU, they released a redesigned 80287 FPU chip to accompany it. Although the 80287 was compatible with the 80386 CPU, Intel designed a better FPU, the 80387, for use in 80386 systems. The 80486 CPU was the first Intel CPU to include an on-chip floating point unit. Shortly after the release of the 80486, Intel introduced the 80486sx CPU that was an 80486 without the built-in FPU. To get floating point capabilities on this chip, you had to add an 80487 chip, although the 80487 was really nothing more than a full-blown 80486 which took over for the "sx" chip in the system. Intel's Pentium chips provide a high-performance floating point unit directly on the CPU. There is no (Intel) floating point coprocessor available for the Pentium chip.

Collectively, we will refer to all these chips as the 80x87 FPU. Given the obsolescence of the 8086, 80286, 8087, 80287, 80387, and 80487 chips, this text will concentrate on the Pentium and later chips. There are some differences between the Pentium floating point units and the earlier FPUs. If you need to write code that will execute on those earlier machines, you should consult the appropriate Intel documentation for those devices.

11.2.1 FPU Registers

The 80x86 FPUs add 13 registers to the 80x86 and later processors: eight floating point data registers, a control register, a status register, a tag register, an instruction pointer, and a data pointer. The data registers are similar to the 80x86's general purpose register set insofar as all floating point calculations take place in these registers. The control register contains bits that let you decide how the FPU handles certain degenerate cases like rounding of inaccurate computations, it contains bits that control precision, and so on. The status register is similar to the 80x86's flags register; it contains the condition code bits and several other floating point flags that describe the state of the FPU. The tag register contains several groups of bits that determine the state of the value in each of the eight general purpose registers. The instruction and data pointer registers contain certain state information about the last floating point instruction executed. We will not consider the last three registers in this text, see the Intel documentation for more details.

1. Intel has also referred to this device as the Numeric Data Processor (NDP), Numeric Processor Extension (NPX), and math coprocessor.

11.2.1.1 FPU Data Registers

The FPUs provide eight 80 bit data registers organized as a stack. This is a significant departure from the organization of the general purpose registers on the 80x86 CPU that comprise a standard general-purpose register set. HLA refers to these registers as ST0, ST1, …, ST7.

The biggest difference between the FPU register set and the 80x86 register set is the stack organization. On the 80x86 CPU, the AX register is always the AX register, no matter what happens. On the FPU, however, the register set is an eight element stack of 80 bit floating point values (see Figure 11.1).

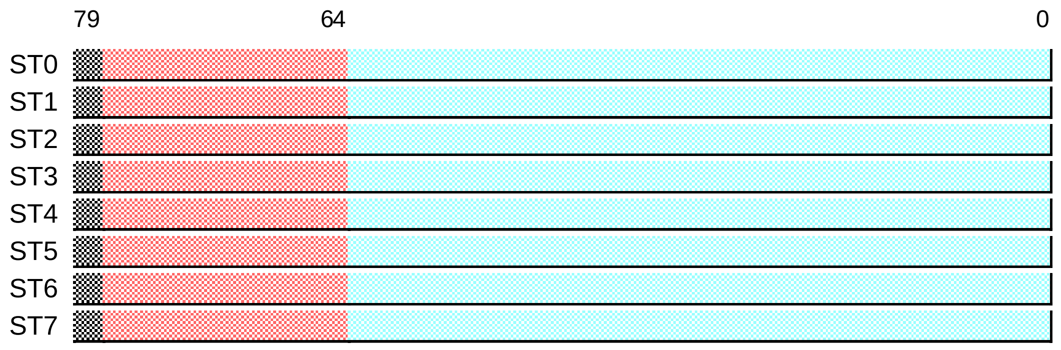

Figure 11.1 FPU Floating Point Register Stack

ST0 refers to the item on the top of the stack, ST1 refers to the next item on the stack, and so on. Many floating point instructions push and pop items on the stack; therefore, ST1 will refer to the previous contents of ST0 after you push something onto the stack. It will take some thought and practice to get used to the fact that the registers are changing under you, but this is an easy problem to overcome.

11.2.1.2 The FPU Control Register

When Intel designed the 80x87 (and, essentially, the IEEE floating point standard), there were no standards in floating point hardware. Different (mainframe and mini) computer manufacturers all had different and incompatible floating point formats. Unfortunately, much application software had been written taking into account the idiosyncrasies of these different floating point formats. Intel wanted to design an FPU that could work with the majority of the software out there (keep in mind, the IBM PC was three to four years away when Intel began designing the 8087, they couldn't rely on that "mountain" of software available for the PC to make their chip popular). Unfortunately, many of the features found in these older floating point formats were mutually incompatible. For example, in some floating point systems rounding would occur when there was insufficient precision; in others, truncation would occur. Some applications would work with one floating point system but not with the other. Intel wanted as many applications as possible to work with as few changes as possible on their 80x87 FPUs, so they added a special register, the FPU *control register*, that lets the user choose one of several possible operating modes for their FPU.

The 80x87 control register contains 16 bits organized as shown in Figure 11.2.

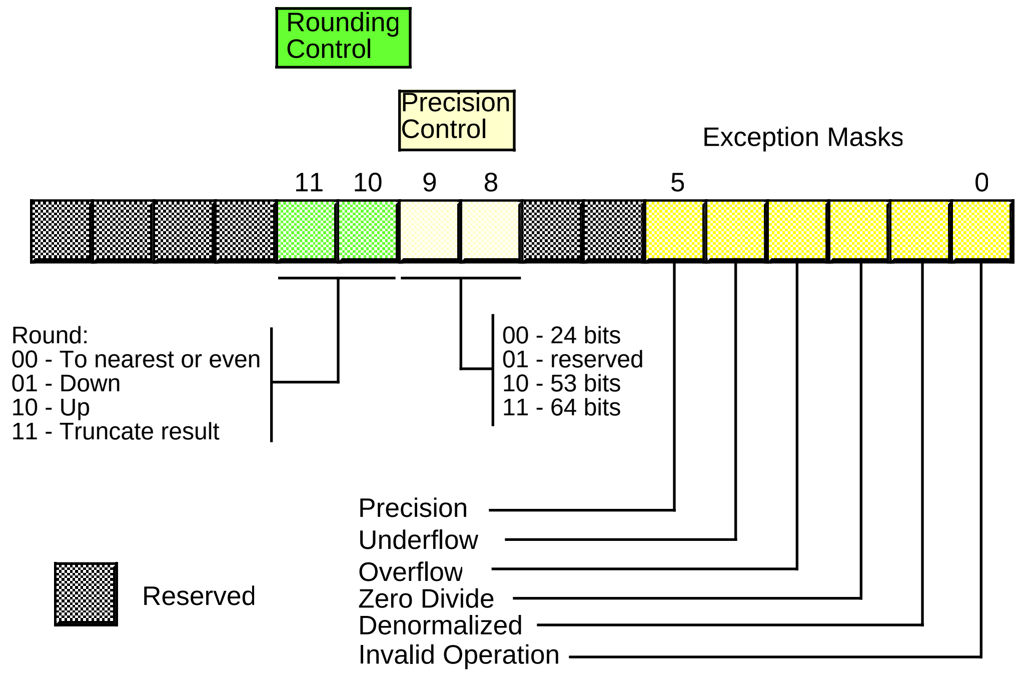

Figure 11.2 FPU Control Register

Bits 10 and 11 provide rounding control according to the following values:

Table 1: Rounding Control

Bits 10 & 11	Function
00	To nearest or even
01	Round down
10	Round up
11	Truncate

The “00” setting is the default. The FPU rounds values above one-half of the least significant bit up. It rounds values below one-half of the least significant bit down. If the value below the least significant bit is exactly one-half of the least significant bit, the FPU rounds the value towards the value whose least significant bit is zero. For long strings of computations, this provides a reasonable, automatic, way to maintain maximum precision.

The round up and round down options are present for those computations where it is important to keep track of the accuracy during a computation. By setting the rounding control to round down and performing the operation, then repeating the operation with the rounding control set to round up, you can determine the minimum and maximum ranges between which the true result will fall.

The truncate option forces all computations to truncate any excess bits during the computation. You will rarely use this option if accuracy is important to you. However, if you are porting older software to the FPU, you might use this option to help when porting the software. One place where this option is extremely useful is when converting a floating point value to an integer. Since most software expects floating point to integer conversions to truncate the result, you will need to use the truncation rounding mode to achieve this.

Bits eight and nine of the control register specify the precision during computation. This capability is provided to allow compatibility with older software as required by the IEEE 754 standard. The precision control bits use the following values:

Table 2: Mantissa Precision Control Bits

Bits 8 & 9	Precision Control
00	24 bits
01	Reserved
10	53 bits
11	64 bits

Some CPUs may operate faster with floating point values whose precision is 53 bits (i.e., 64-bit floating point format) rather than 64 bits (i.e., 80-bit floating point format). Please see the documentation for your specific processor for details. Generally, the CPU defaults these bits to %11 to select the 64-bit mantissa precision.

Bits zero through five are the *exception masks*. These are similar to the interrupt enable bit in the 80x86's flags register. If these bits contain a one, the corresponding condition is ignored by the FPU. However, if any bit contains zero, and the corresponding condition occurs, then the FPU immediately generates an interrupt so the program can handle the degenerate condition.

Bit zero corresponds to an invalid operation error. This generally occurs as the result of a programming error. Problems which raise the invalid operation exception include pushing more than eight items onto the stack or attempting to pop an item off an empty stack, taking the square root of a negative number, or loading a non-empty register.

Bit one masks the *denormalized* interrupt that occurs whenever you try to manipulate denormalized values. Denormalized exceptions occur when you load arbitrary extended precision values into the FPU or work with very small numbers just beyond the range of the FPU's capabilities. Normally, you would probably *not* enable this exception. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fDenormal* exception.

Bit two masks the *zero divide* exception. If this bit contains zero, the FPU will generate an interrupt if you attempt to divide a nonzero value by zero. If you do not enable the zero division exception, the FPU will produce NaN (not a number) whenever you perform a zero division. It's probably a good idea to enable this exception by programming a zero into this bit. Note that if your program generates this interrupt, the HLA run-time system will raise the *ex.fDivByZero* exception.

Bit three masks the *overflow* exception. The FPU will raise the overflow exception if a calculation overflows or if you attempt to store a value which is too large to fit into a destination operand (e.g., storing a large extended precision value into a single precision variable). If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fOverflow* exception.

Bit four, if set, masks the *underflow* exception. Underflow occurs when the result is too *small* to fit in the destination operand. Like overflow, this exception can occur whenever you store a small extended precision value into a smaller variable (single or double precision) or when the result of a computation is too small for extended precision. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fUnderflow* exception.

Bit five controls whether the *precision* exception can occur. A precision exception occurs whenever the FPU produces an imprecise result, generally the result of an internal rounding operation. Although many operations will produce an exact result, many more will not. For example, dividing one by ten will produce an inexact result. Therefore, this bit is usually one since inexact results are very common. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.InexactResult* exception.

Bits six and thirteen through fifteen in the control register are currently undefined and reserved for future use. Bit seven is the interrupt enable mask, but it is only active on the 8087 FPU; a zero in this bit enables 8087 interrupts and a one disables FPU interrupts.

The FPU provides two instructions, FLDCW (load control word) and FSTCW (store control word), that let you load and store the contents of the control register. The single operand to these instructions must be a 16 bit memory location. The FLDCW instruction loads the control register from the specified memory location, FSTCW stores the control register into the specified memory location. The syntax for these instructions is

```
fldcw( mem16 );
fstcw( mem16 );
```

Here's some example code that sets the rounding control to "truncate result" and sets the rounding precision to 24 bits:

```
static
    fcw16: word;
        .
        .
        .
        fstcw( fcw16 );
        mov( fcw16, ax );
        and( $f0ff, ax );      // Clears bits 8-11.
        or( $0c00, ax );       // Rounding control=%11, Precision = %00.
        mov( ax, fcw16 );
        fldcw( fcw16 );
```

11.2.1.3 The FPU Status Register

The FPU status register provides the status of the coprocessor at the instant you read it. The FSTSW instruction stores the16 bit floating point status register into a word variable. The status register is a 16 bit register, its layout appears in Figure 11.3.

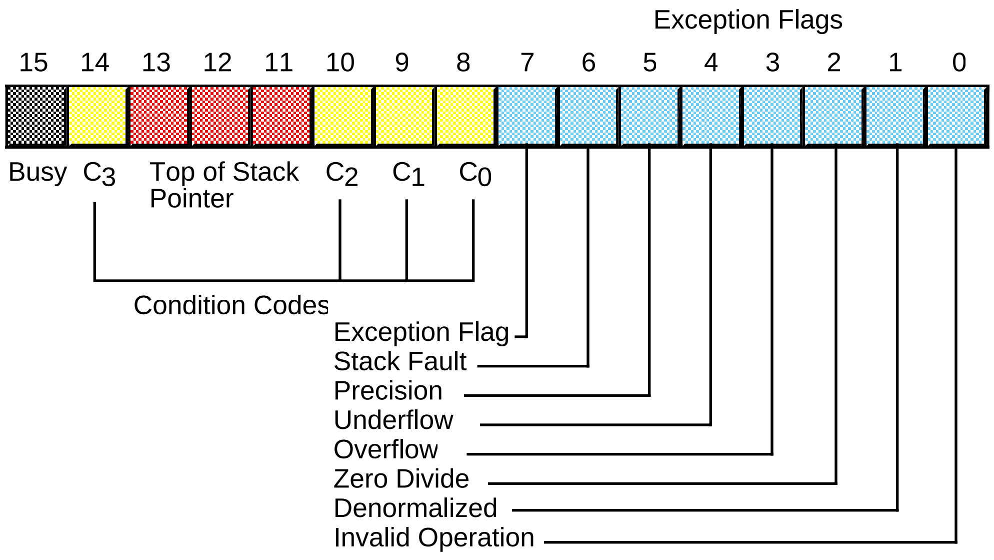

Figure 11.3 The FPU Status Register

Bits zero through five are the exception flags. These bits are appear in the same order as the exception masks in the control register. If the corresponding condition exists, then the bit is set. These bits are independent of the exception masks in the control register. The FPU sets and clears these bits regardless of the corresponding mask setting.

Bit six indicates a *stack fault*. A stack fault occurs whenever there is a stack overflow or underflow. When this bit is set, the C_1 condition code bit determines whether there was a stack overflow (C_1=1) or stack underflow (C_1=0) condition.

Bit seven of the status register is set if *any* error condition bit is set. It is the logical OR of bits zero through five. A program can test this bit to quickly determine if an error condition exists.

Bits eight, nine, ten, and fourteen are the coprocessor condition code bits. Various instructions set the condition code bits as shown in the following table:

Table 3: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
fcom, fcomp, fcompp, ficom, ficomp	0 0 1 1	0 0 0 1	X X X X	0 1 0 1	ST > source ST < source ST = source ST or source undefined
	X = Don’t care				

Table 3: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
ftst	0	0	X	0	ST is positive
	0	0	X	1	ST is negative
	1	0	X	0	ST is zero (+ or -)
	1	1	X	1	ST is uncomparable
fxam	0	0	0		+ Unnormalized
	0	0	1	0	-Unnormalized
	0	1	0	0	+Normalized
	0	1	1	0	-Normalized
	1	0	0	0	+0
	1	0	1	0	-0
	1	1	0	0	+Denormalized
	1	1	1	0	-Denormalized
	0	0	0	1	+NaN
	0	0	1	1	-NaN
	0	1	0	1	+Infinity
	0	1	1	1	-Infinity
	1	X	X	1	Empty register
fucom,	0	0	X	0	ST > source
fucomp,	0	0	X	1	ST < source
fucompp	1	0	X	0	ST = source
	1	1	X	1	Unordered
	X = Don't care				

Table 4: Condition Code Interpretations

Instruction(s)	Condition Code Bits			
	C_0	C_3	C_2	C_1
fcom, fcomp, fcmpp, ftst, fucom, fucomp, fucompp, ficom, ficomp	Result of comparison. See previous table.	Result of comparison. See previous table.	Operands are not comparable	Result of comparison. See previous table. Also denotes stack overflow/underflow if stack exception bit is set.

Table 4: Condition Code Interpretations

Instruction(s)	Condition Code Bits			
	C_0	C_3	C_2	C_1
fxam	See previous table.	See previous table.	See previous table.	Sign of result, or stack overflow/underflow (if stack exception bit is set).
fprem, fprem1	Bit 2 of remainder	Bit 0 of remainder	0- reduction done. 1- reduction incomplete.	Bit 1 of remainder or stack overflow/underflow (if stack exception bit is set).
fist, fbstp, frndint, fst, fstp, fadd, fmul, fdiv, fdivr, fsub, fsubr, fscale, fsqrt, fpatan, f2xm1, fyl2x, fyl2xp1	Undefined	Undefined	Undefined	Round up occurred or stack overflow/underflow (if stack exception bit is set).
fptan, fsin, fcos, fsincos	Undefined	Undefined	0- reduction done. 1- reduction incomplete.	Round up occurred or stack overflow/underflow (if stack exception bit is set).
fchs, fabs, fxch, fincstp, fdecstp, *constant loads*, fxtract, fld, fild, fbld, fstp (80 bit)	Undefined	Undefined	Undefined	Zero result or stack overflow/underflow (if stack exception bit is set).
fldenv, fstor	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.
fldcw, fstenv, fstcw, fstsw, fclex	Undefined	Undefined	Undefined	Undefined
finit, fsave	Cleared to zero.	Cleared to zero.	Cleared to zero.	Cleared to zero.

Bits 11-13 of the FPU status register provide the register number of the top of stack. During computations, the FPU adds (modulo eight) the *logical* register numbers supplied by the programmer to these three bits to determine the *physical* register number at run time.

Bit 15 of the status register is the *busy* bit. It is set whenever the FPU is busy. Most programs will have little reason to access this bit.

11.2.2 FPU Data Types

The FPU supports seven different data types: three integer types, a packed decimal type, and three floating point types. The integer type provides for 64-bit integers, although it is often faster to do the 64-bit arithmetic using the integer unit of the CPU (see the chapter on Advanced Arithmetic). Certainly it is often faster to do 16-bit and 32-bit integer arithmetic using the standard integer registers. The packed decimal type provides a 17 digit signed decimal (BCD) integer. The primary purpose of the BCD format is to convert between strings and floating point values. The remaining three data types are the 32 bit, 64 bit, and 80 bit floating point data types we've looked at so far. The 80x87 data types appear in Figure 11.4, Figure 11.5, and Figure 11.6.

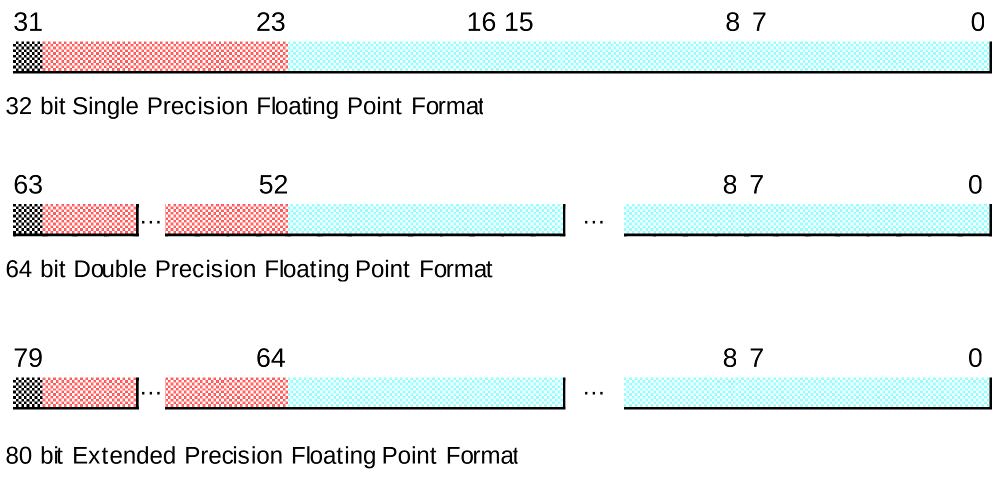
31
23
16 15
8 7
0
32 bit Single Precision Floating Point Format
63
52
8 7
0
64 bit Double Precision Floating Point Format
79
64
8 7
0
80 bit Extended Precision Floating Point Format

Figure 11.4 FPU Floating Point Formats

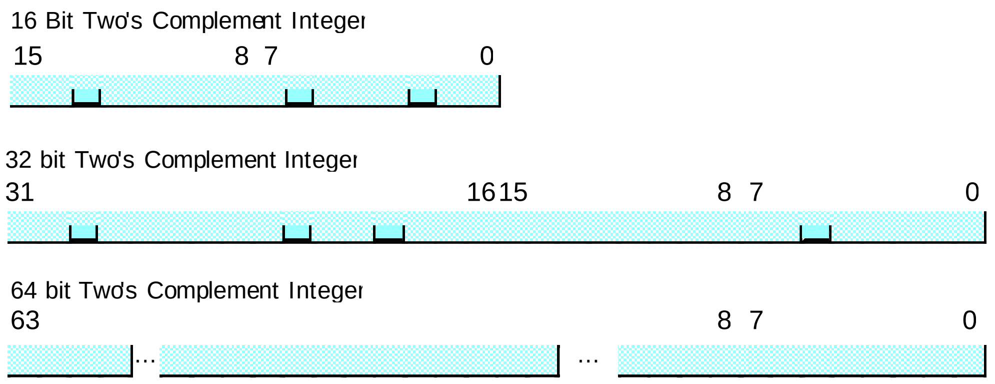
16 Bit Two's Complement Integer
15
8 7
0
32 bit Two's Complement Integer
31
16 15
8 7
0
64 bit Two's Complement Integer
63
8 7
0

Figure 11.5 FPU Integer Formats

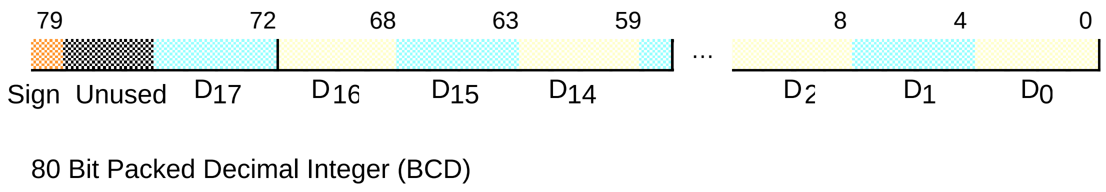

Figure 11.6 FPU Packed Decimal Format

The FPU generally stores values in a *normalized* format. When a floating point number is normalized, the H.O. bit of the mantissa is always one. In the 32 and 64 bit floating point formats, the FPU does not actually store this bit, the FPU always assumes that it is one. Therefore, 32 and 64 bit floating point numbers are always normalized. In the extended precision 80 bit floating point format, the FPU does *not* assume that the H.O. bit of the mantissa is one, the H.O. bit of the mantissa appears as part of the string of bits.

Normalized values provide the greatest precision for a given number of bits. However, there are a large number of non-normalized values which we *cannot* represent with the 80-bit format. These values are very close to zero and represent the set of values whose mantissa H.O. bit is not zero. The FPUs support a special 80-bit form known as *denormalized* values. Denormalized values allow the FPU to encode very small values it cannot encode using normalized values, but at a price. Denormalized values offer fewer bits of precision than normalized values. Therefore, using denormalized values in a computation may introduce some slight inaccuracy into a computation. Of course, this is always better than underflowing the denormalized value to zero (which could make the computation even less accurate), but you must keep in mind that if you work with very small values you may lose some accuracy in your computations. Note that the FPU status register contains a bit you can use to detect when the FPU uses a denormalized value in a computation.

11.2.3 The FPU Instruction Set

The FPU adds over 80 new instructions to the 80x86 instruction set. We can classify these instructions as *data movement instructions*, *conversions*, *arithmetic instructions*, *comparisons*, *constant instructions*, *transcendental instructions*, and *miscellaneous instructions*. The following sections describe each of the instructions in these categories.

11.2.4 FPU Data Movement Instructions

The data movement instructions transfer data between the internal FPU registers and memory. The instructions in this category are FLD, FST, FSTP, and FXCH. The FLD instruction always pushes its operand onto the floating point stack. The FSTP instruction always pops the top of stack after storing the top of stack (tos). The remaining instructions do not affect the number of items on the stack.

11.2.4.1 The FLD Instruction

The FLD instruction loads a 32 bit, 64 bit, or 80 bit floating point value onto the stack. This instruction converts 32 and 64 bit operands to an 80 bit extended precision value before pushing the value onto the floating point stack.

The FLD instruction first decrements the top of stack (TOS) pointer (bits 11-13 of the status register) and then stores the 80 bit value in the physical register specified by the new TOS pointer. If the source operand of the FLD instruction is a floating point data register, ST*i*, then the actual register the FPU uses for the load operation is the register number *before* decrementing the tos pointer. Therefore, "fld(st0);" duplicates the value on the top of the stack.

The FLD instruction sets the stack fault bit if stack overflow occurs. It sets the denormalized exception bit if you load an 80-bit denormalized value. It sets the invalid operation bit if you attempt to load an empty floating point register onto the stop of stack (or perform some other invalid operation).

Examples:

```
        fld( st1 );
        fld( real32_variable );
        fld( real64_variable );
        fld( real80_variable );
        fld( real_constant );
```

Note that there is no way to directly load a 32-bit integer register onto the floating point stack, even if that register contains a REAL32 value. To accomplish this, you must first store the integer register into a memory location then you can push that memory location onto the FPU stack using the FLD instruction. E.g.,

```
        mov( eax, tempReal32 );      // Save REAL32 value in EAX to memory.
        fld( tempReal32 );           // Push that real value onto the FPU stack.
```

Note: loading a constant via FLD is actually an HLA extension. The FPU doesn't support this instruction type. HLA creates a REAL80 object in the "constants" segment and uses the address of this memory object as the true operand for FLD.

11.2.4.2 The FST and FSTP Instructions

The FST and FSTP instructions copy the value on the top of the floating point register stack to another floating point register or to a 32, 64, or 80 bit memory variable. When copying data to a 32 or 64 bit memory variable, the 80 bit extended precision value on the top of stack is rounded to the smaller format as specified by the rounding control bits in the FPU control register.

The FSTP instruction pops the value off the top of stack when moving it to the destination location. It does this by incrementing the top of stack pointer in the status register after accessing the data in ST0. If the destination operand is a floating point register, the FPU stores the value at the specified register number *before* popping the data off the top of the stack.

Executing an "fstp(st0);" instruction effectively pops the data off the top of stack with no data transfer. Examples:

```
        fst( real32_variable );
        fst( real64_variable );
        fst( realArray[ ebx*8 ] );
        fst( real80_variable );
        fst( st2 );
        fstp( st1 );
```

The last example above effectively pops ST1 while leaving ST0 on the top of the stack.

The FST and FSTP instructions will set the stack exception bit if a stack underflow occurs (attempting to store a value from an empty register stack). They will set the precision bit if there is a loss of precision during the store operation (this will occur, for example, when storing an 80 bit extended precision value into a 32 or 64 bit memory variable and there are some bits lost during conversion). They will set the underflow exception bit when storing an 80 bit value into a 32 or 64 bit memory variable, but the value is too small to fit into the destination operand. Likewise, these instructions will set the overflow exception bit if the value on the top of stack is too big to fit into a 32 or 64 bit memory variable. The FST and FSTP instructions set the denormalized flag when you try to store a denormalized value into an 80 bit register or variable[2]. They set the invalid operation flag if an invalid operation (such as storing into an empty register) occurs. Finally, these instructions set the C_1 condition bit if rounding occurs during the store operation (this only occurs when storing into a 32 or 64 bit memory variable and you have to round the mantissa to fit into the destination).

Note: Because of an idiosyncrasy in the FPU instruction set related to the encoding of the instructions, you cannot use the FST instruction to store data into a real80 memory variable. You may, however, store 80-bit data using the FSTP instruction.

11.2.4.3 The FXCH Instruction

The FXCH instruction exchanges the value on the top of stack with one of the other FPU registers. This instruction takes two forms: one with a single FPU register as an operand, the second without any operands. The first form exchanges the top of stack (tos) with the specified register. The second form of FXCH swaps the top of stack with ST1.

Many FPU instructions, e.g., FSQRT, operate only on the top of the register stack. If you want to perform such an operation on a value that is not on the top of stack, you can use the FXCH instruction to swap

2. Storing a denormalized value into a 32 or 64 bit memory variable will always set the underflow exception bit.

that register with tos, perform the desired operation, and then use the FXCH to swap the tos with the original register. The following example takes the square root of ST2:

```
fxch( st2 );
fsqrt();
fxch( st2 );
```

The FXCH instruction sets the stack exception bit if the stack is empty. It sets the invalid operation bit if you specify an empty register as the operand. This instruction always clears the C_1 condition code bit.

11.2.5 Conversions

The FPU performs all arithmetic operations on 80 bit real quantities. In a sense, the FLD and FST/FSTP instructions are conversion instructions as well as data movement instructions because they automatically convert between the internal 80 bit real format and the 32 and 64 bit memory formats. Nonetheless, we'll simply classify them as data movement operations, rather than conversions, because they are moving real values to and from memory. The FPU provides five other instructions that convert to or from integer or binary coded decimal (BCD) format when moving data. These instructions are FILD, FIST, FISTP, FBLD, and FBSTP.

11.2.5.1 The FILD Instruction

The FILD (integer load) instruction converts a 16, 32, or 64 bit two's complement integer to the 80 bit extended precision format and pushes the result onto the stack. This instruction always expects a single operand. This operand must be the address of a word, double word, or quad word integer variable. You cannot specify one of the 80x86's 16 or 32 bit general purpose registers. If you want to push an 80x86 general purpose register onto the FPU stack, you must first store it into a memory variable and then use FILD to push that value of that memory variable.

The FILD instruction sets the stack exception bit and C_1 (accordingly) if stack overflow occurs while pushing the converted value. Examples:

```
fild( word_variable );
fild( dword_val[ ecx*4 ] );
fild( qword_variable );
```

11.2.5.2 The FIST and FISTP Instructions

The FIST and FISTP instructions convert the 80 bit extended precision variable on the top of stack to a 16, 32, or 64 bit integer and store the result away into the memory variable specified by the single operand. These instructions convert the value on tos to an integer according to the rounding setting in the FPU control register (bits 10 and 11). As for the FILD instruction, the FIST and FISTP instructions will not let you specify one of the 80x86's general purpose 16 or 32 bit registers as the destination operand.

The FIST instruction converts the value on the top of stack to an integer and then stores the result; it does not otherwise affect the floating point register stack. The FISTP instruction pops the value off the floating point register stack after storing the converted value.

These instructions set the stack exception bit if the floating point register stack is empty (this will also clear C_1). They set the precision (imprecise operation) and C_1 bits if rounding occurs (that is, if there is any fractional component to the value in ST0). These instructions set the underflow exception bit if the result is too small (i.e., less than one but greater than zero or less than zero but greater than -1). Examples:

```
fist( word_var[ ebx*2 ] );
fist( qword_var );
fistp( dword_var );
```

Don’t forget that these instructions use the rounding control settings to determine how they will convert the floating point data to an integer during the store operation. Be default, the rounding control is usually set to “round” mode; yet most programmers expect FIST/FISTP to truncate the decimal portion during conversion. If you want FIST/FISTP to truncate floating point values when converting them to an integer, you will need to set the rounding control bits appropriately in the floating point control register, e.g.,

```
static
    fcw16:          word;
    fcw16_2:        word;
    IntResult:      int32;
        .
        .
        .
        fstcw( fcw16 );
        mov( fcw16, ax );
        or( $0c00, ax );        // Rounding control=%11 (truncate).
        mov( ax, fcw16_2 );     // Store into memory and reload the ctrl word.
        fldcw( fcw16_2 );

        fistp( IntResult );       // Truncate ST0 and store as int32 object.

        fldcw( fcw16 );           // Restore original rounding control
```

11.2.5.3 The FBLD and FBSTP Instructions

The FBLD and FBSTP instructions load and store 80 bit BCD values. The FBLD instruction converts a BCD value to its 80 bit extended precision equivalent and pushes the result onto the stack. The FBSTP instruction pops the extended precision real value on TOS, converts it to an 80 bit BCD value (rounding according to the bits in the floating point control register), and stores the converted result at the address specified by the destination memory operand. Note that there is no FBST instruction which stores the value on tos without popping it.

The FBLD instruction sets the stack exception bit and C_1 if stack overflow occurs. It sets the invalid operation bit if you attempt to load an invalid BCD value. The FBSTP instruction sets the stack exception bit and clears C_1 if stack underflow occurs (the stack is empty). It sets the underflow flag under the same conditions as FIST and FISTP. Examples:

```
// Assuming fewer than eight items on the stack, the following
// code sequence is equivalent to an fbst instruction:

        fld( st0 );
        fbstp( tbyte_var );

// The following example easily converts an 80 bit BCD value to
// a 64 bit integer:

        fbld( tbyte_var );
        fist( qword_var );
```

11.2.6 Arithmetic Instructions

The arithmetic instructions make up a small, but important, subset of the FPU’s instruction set. These instructions fall into two general categories – those which operate on real values and those which operate on a real and an integer value.

11.2.6.1 The FADD and FADDP Instructions

These two instructions take the following forms:

```
fadd()
faddp()
fadd( st0, sti );
fadd( sti, st0 );
faddp( st0, sti );
fadd( mem_32_64 );
fadd( real_constant );
```

The first two forms are equivalent. They pop the two values on the top of stack, add them, and push their sum back onto the stack.

The next two forms of the FADD instruction, those with two FPU register operands, behave like the 80x86's ADD instruction. They add the value in the source register operand to the value in the destination register operand. Note that one of the register operands must be ST0.

The FADDP instruction with two operands adds ST0 (which must always be the source operand) to the destination operand and then pops ST0. The destination operand must be one of the other FPU registers.

The last form above, FADD with a memory operand, adds a 32 or 64 bit floating point variable to the value in ST0. This instruction will convert the 32 or 64 bit operands to an 80 bit extended precision value before performing the addition. Note that this instruction does *not* allow an 80 bit memory operand.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Like FLD(real_constant), the FADD(real_constant) instruction is an HLA extension. Note that it creates a 64-bit variable holding the constant value and emits the FADD(*mem64*) instruction, specifying the read-only object it creates in the constants segment.

11.2.6.2 The FSUB, FSUBP, FSUBR, and FSUBRP Instructions

These four instructions take the following forms:

```
fsub()
fsubp()
fsubr()
fsubrp()

fsub( st0, sti )
fsub( sti, st0 );
fsubp( st0, sti );
fsub( mem_32_64 );
fsub( real_constant );

fsubr( st0, sti )
fsubr( sti, st0 );
fsubrp( st0, sti );
fsubr( mem_32_64 );
fsubr( real_constant );
```

With no operands, the FSUB and FSUBP instructions operate identically. They pop ST0 and ST1 from the register stack, compute ST1-ST0, and the push the difference back onto the stack. The FSUBR and FSUBRP instructions (reverse subtraction) operate in an almost identical fashion except they compute ST0-ST1 and push that difference.

With two register operands (*source, destination*) the FSUB instruction computes *destination := destination - source*. One of the two registers must be ST0. With two registers as operands, the FSUBP also computes *destination := destination - source* and then it pops ST0 off the stack after computing the difference. For the FSUBP instruction, the source operand must be ST0.

With two register operands, the FSUBR and FSUBRP instruction work in a similar fashion to FSUB and FSUBP, except they compute *destination := source - destination.*

The FSUB(mem) and FSUBR(mem) instructions accept a 32 or 64 bit memory operand. They convert the memory operand to an 80 bit extended precision value and subtract this from ST0 (FSUB) or subtract ST0 from this value (FSUBR) and store the result back into ST0.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. HLA generates a constant segment memory object initialized with the constant's value.

11.2.6.3 The FMUL and FMULP Instructions

The FMUL and FMULP instructions multiply two floating point values. These instructions allow the following forms:

```
fmul()
fmulp()

fmul( sti, st0 );
fmul( st0, sti );
fmul( mem_32_64 );
fmul( real_constant );

fmulp( st0, sti );
```

With no operands, FMUL and FMULP both do the same thing – they pop ST0 and ST1, multiply these values, and push their product back onto the stack. The FMUL instructions with two register operands compute *destination := destination * source*. One of the registers (source or destination) must be ST0.

The FMULP(ST0, ST*i*) instruction computes ST*i* := ST*i* * ST0 and then pops ST0. This instruction uses the value for *i* before popping ST0. The FMUL(mem) instruction requires a 32 or 64 bit memory operand. It converts the specified memory variable to an 80 bit extended precision value and the multiplies ST0 by this value.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Note: the instruction that has a real constant as its operand isn't a true FPU instruction. It is an extension provided by HLA (see the note at the end of the previous section for details).

11.2.6.4 The FDIV, FDIVP, FDIVR, and FDIVRP Instructions

These four instructions allow the following forms:

```
fdiv()
fdivp()
fdivr()
fdivrp()

fdiv( sti, st0 );
fdiv( st0, sti );
fdivp( st0, sti );
```

```
        fdivr( sti, st0 );
        fdivr( st0, sti );
        fdivrp( st0, sti );

        fdiv( mem_32_64 );
        fdivr( mem_32_64 );
        fdiv( real_constant );
        fdivr( real_constant );
```

With no operands, the FDIV and FDIVP instructions pop ST0 and ST1, compute ST1/ST0, and push the result back onto the stack. The FDIVR and FDIVRP instructions also pop ST0 and ST1 but compute ST0/ST1 before pushing the quotient onto the stack.

With two register operands, these instructions compute the following quotients:

```
        fdiv( sti, st0 );     // ST0 := ST0/STi
        fdiv( st0, sti );     // STi := STi/ST0
        fdivp( st0, sti );    // STi := STi/ST0  then pop ST0
        fdivr( st0, sti );    // ST0 := ST0/STi
        fdivrp( st0, sti );   // STi := ST0/STi then pop ST0
```

The FDIVP and FDIVRP instructions also pop ST0 after performing the division operation. The value for *i* in these two instructions is computed before popping ST0.

These instructions can raise the stack, precision, underflow, overflow, denormalized, zero divide, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA.

11.2.6.5 The FSQRT Instruction

The FSQRT routine does not allow any operands. It computes the square root of the value on top of stack (TOS) and replaces ST0 with this result. The value on TOS must be zero or positive, otherwise FSQRT will generate an invalid operation exception.

This instruction can raise the stack, precision, denormalized, and invalid operation exceptions, as appropriate. If rounding occurs during the computation, FSQRT sets the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Example:

```
// Compute Z := sqrt(x**2 + y**2);

        fld( x );      // Load X.
        fld( st0 );    // Duplicate X on TOS.
        fmul();        // Compute X**2.

        fld( y );      // Load Y
        fld( st0 );    // Duplicate Y.
        fmul();        // Compute Y**2.

        fadd();        // Compute X**2 + Y**2.
        fsqrt();       // Compute sqrt( X**2 + Y**2 ).
        fstp( z );     // Store result away into Z.
```

11.2.6.6 The FPREM and FPREM1 Instructions

The FPREM and FPREM1 instructions compute a *partial remainder*. Intel designed the FPREM instruction before the IEEE finalized their floating point standard. In the final draft of the IEEE floating point standard, the definition of FPREM was a little different than Intel's original design. Unfortunately, Intel needed to maintain compatibility with the existing software that used the FPREM instruction, so they designed a new version to handle the IEEE partial remainder operation, FPREM1. You should always use FPREM1 in new software you write, therefore we will only discuss FPREM1 here, although you use FPREM in an identical fashion.

FPREM1 computes the *partial* remainder of ST0/ST1. If the difference between the exponents of ST0 and ST1 is less than 64, FPREM1 can compute the exact remainder in one operation. Otherwise you will have to execute the FPREM1 two or more times to get the correct remainder value. The C_2 condition code bit determines when the computation is complete. Note that FPREM1 does *not* pop the two operands off the stack; it leaves the partial remainder in ST0 and the original divisor in ST1 in case you need to compute another partial product to complete the result.

The FPREM1 instruction sets the stack exception flag if there aren't two values on the top of stack. It sets the underflow and denormal exception bits if the result is too small. It sets the invalid operation bit if the values on tos are inappropriate for this operation. It sets the C_2 condition code bit if the partial remainder operation is not complete. Finally, it loads C_3, C_1, and C_0 with bits zero, one, and two of the quotient, respectively.

Example:

```
// Compute Z := X mod Y

        fld( y );
        fld( x );
        repeat

            fprem1();
            fstsw( ax );     // Get condition code bits into AX.
            and( 1, ah );    // See if C2 is set.

        until( @z );        // Repeat until C2 is clear.
        fstp( z );          // Store away the remainder.
        fstp( st0 );        // Pop old Y value.
```

11.2.6.7 The FRNDINT Instruction

The FRNDINT instruction rounds the value on the top of stack (TOS) to the nearest integer using the rounding algorithm specified in the control register.

This instruction sets the stack exception flag if there is no value on the TOS (it will also clear C_1 in this case). It sets the precision and denormal exception bits if there was a loss of precision. It sets the invalid operation flag if the value on the tos is not a valid number. Note that the result on tos is still a floating point value, it simply does not have a fractional component.

11.2.6.8 The FABS Instruction

FABS computes the absolute value of ST0 by clearing the mantissa sign bit of ST0. It sets the stack exception bit and invalid operation bits if the stack is empty.

Example:

```
// Compute X := sqrt(abs(x));
```

```
        fld( x );
        fabs();
        fsqrt();
        fstp( x );
```

11.2.6.9 The FCHS Instruction

FCHS changes the sign of ST0's value by inverting the mantissa sign bit (that is, this is the floating point negation instruction). It sets the stack exception bit and invalid operation bits if the stack is empty. Example:

```
// Compute X := -X if X is positive, X := X if X is negative.

        fld( x );
        fabs();
        fchs();
        fstp( x );
```

11.2.7 Comparison Instructions

The FPU provides several instructions for comparing real values. The FCOM, FCOMP, and FCOMPP instructions compare the two values on the top of stack and set the condition codes appropriately. The FTST instruction compares the value on the top of stack with zero.

Generally, most programs test the condition code bits immediately after a comparison. Unfortunately, there are no conditional jump instructions that branch based on the FPU condition codes. Instead, you can use the FSTSW instruction to copy the floating point status register (see "The FPU Status Register" on page 615) into the AX register; then you can use the SAHF instruction to copy the AH register into the 80x86's condition code bits. After doing this, you can use the conditional jump instructions to test some condition. This technique copies C_0 into the carry flag, C_2 into the parity flag, and C_3 into the zero flag. The SAHF instruction does not copy C_1 into any of the 80x86's flag bits.

Since the SAHF instruction does not copy any FPU status bits into the sign or overflow flags, you cannot use signed comparison instructions. Instead, use unsigned operations (e.g., SETA, SETB) when testing the results of a floating point comparison. *Yes, these instructions normally test unsigned values and floating point numbers are signed values*. However, use the unsigned operations anyway; the FSTSW and SAHF instructions set the 80x86 flags register as though you had compared unsigned values with the CMP instruction.

The Pentium II and (upwards) compatible processors provide an extra set of floating point comparison instructions that directly affect the 80x86 condition code flags. These instructions circumvent having to use FSTSW and SAHF to copy the FPU status into the 80x86 condition codes. These instructions include FCOMI and FCOMIP. You use them just like the FCOM and FCOMP instructions except, of course, you do not have to manually copy the status bits to the FLAGS register. Do be aware that these instructions are not available on many processors in common use today (as of 1/1/2000). However, as time passes it may be safe to begin assuming that everyone's CPU supports these instructions. Since this text assumes a minimum Pentium CPU, it will not discuss these two instructions any further.

11.2.7.1 The FCOM, FCOMP, and FCOMPP Instructions

The FCOM, FCOMP, and FCOMPP instructions compare ST0 to the specified operand and set the corresponding FPU condition code bits based on the result of the comparison. The legal forms for these instructions are

```
fcom()
fcomp()
fcompp()

fcom( sti )
fcomp( sti )

fcom( mem_32_64 )
fcomp( mem_32_64 )
fcom( real_constant )
fcomp( real_constant )
```

With no operands, FCOM, FCOMP, and FCOMPP compare ST0 against ST1 and set the processor flags accordingly. In addition, FCOMP pops ST0 off the stack and FCOMPP pops both ST0 and ST1 off the stack.

With a single register operand, FCOM and FCOMP compare ST0 against the specified register. FCOMP also pops ST0 after the comparison.

With a 32 or 64 bit memory operand, the FCOM and FCOMP instructions convert the memory variable to an 80 bit extended precision value and then compare ST0 against this value, setting the condition code bits accordingly. FCOMP also pops ST0 after the comparison.

These instructions set C_2 (which winds up in the parity flag) if the two operands are not comparable (e.g., NaN). If it is possible for an illegal floating point value to wind up in a comparison, you should check the parity flag for an error before checking the desired condition.

These instructions set the stack fault bit if there aren't two items on the top of the register stack. They set the denormalized exception bit if either or both operands are denormalized. They set the invalid operation flag if either or both operands are quite NaNs. These instructions always clear the C_1 condition code.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. When HLA encounters such an instruction, it creates a real64 read-only variable in the constants segment and initializes this variable with the specified constant. Then HLA translates the instruction to one that specifies a real64 memory operand. *Note that because of the precision differences (64 bits vs. 80 bits), if you use a constant operand in a floating point instruction you may not get results that are as precise as you would expect.*

Example of a floating point comparison:

```
fcompp();
fstsw( ax );
sahf();
setb( al );   // AL = true if ST1 < ST0.
    .
    .
    .
```

Note that you cannot compare floating point values in an HLA run-time boolean expression (e.g., within an IF statement).

11.2.7.2 The FTST Instruction

The FTST instruction compares the value in ST0 against 0.0. It behaves just like the FCOM instruction would if ST1 contained 0.0. Note that this instruction does not differentiate -0.0 from +0.0. If the value in ST0 is either of these values, ftst will set C_3 to denote equality. Note that this instruction does *not* pop st(0) off the stack. Example:

```
ftst();
fstsw( ax );
sahf();
sete( al );       // Set AL to 1 if TOS = 0.0
```

11.2.8 Constant Instructions

The FPU provides several instructions that let you load commonly used constants onto the FPU's register stack. These instructions set the stack fault, invalid operation, and C_1 flags if a stack overflow occurs; they do not otherwise affect the FPU flags. The specific instructions in this category include:

```
fldz()     ;Pushes +0.0.
fld1()     ;Pushes +1.0.
fldpi()    ;Pushes π.
fldl2t()   ;Pushes log2(10).
fldl2e()   ;Pushes log2(e).
fldlg2()   ;Pushes log10(2).
fldln2()   ;Pushes ln(2).
```

11.2.9 Transcendental Instructions

The FPU provides eight transcendental (log and trigonometric) instructions to compute sin, cos, partial tangent, partial arctangent, 2^x-1, y * $\log_2(x)$, and y * $\log_2(x+1)$. Using various algebraic identities, it is easy to compute most of the other common transcendental functions using these instructions.

11.2.9.1 The F2XM1 Instruction

F2XM1 computes 2^{st0}-1. The value in ST0 must be in the range $-1.0 \leq ST0 \leq +1.0$. If ST0 is out of range F2XM1 generates an undefined result but raises no exceptions. The computed value replaces the value in ST0. Example:

```
; Compute 10^x using the identity: 10^x = 2^(x*lg(10)) (lg = log2).

        fld( x );
        fldl2t();
        fmul();
        f2xm1();
        fld1();
        fadd();
```

Note that F2XM1 computes 2^x-1, which is why the code above adds 1.0 to the result at the end of the computation.

11.2.9.2 The FSIN, FCOS, and FSINCOS Instructions

These instructions pop the value off the top of the register stack and compute the sine, cosine, or both, and push the result(s) back onto the stack. The FSINCOS pushes the sine followed by the cosine of the original operand, hence it leaves cos(ST0) in ST0 and sin(ST0) in ST1.

These instructions assume ST0 specifies an angle in radians and this angle must be in the range $-2^{63} < ST0 < +2^{63}$. If the original operand is out of range, these instructions set the C_2 flag and leave ST0 unchanged. You can use the FPREM1 instruction, with a divisor of 2π, to reduce the operand to a reasonable range.

These instructions set the stack fault/C_1, precision, underflow, denormalized, and invalid operation flags according to the result of the computation.

11.2.9.3 The FPTAN Instruction

FPTAN computes the tangent of ST0 and pushes this value and then it pushes 1.0 onto the stack. Like the FSIN and FCOS instructions, the value of ST0 is assumed to be in radians and must be in the range $-2^{63}<\text{ST0}<+2^{63}$. If the value is outside this range, FPTAN sets C_2 to indicate that the conversion did not take place. As with the FSIN, FCOS, and FSINCOS instructions, you can use the FPREM1 instruction to reduce this operand to a reasonable range using a divisor of 2π.

If the argument is invalid (i.e., zero or π radians, which causes a division by zero) the result is undefined and this instruction raises no exceptions. FPTAN will set the stack fault, precision, underflow, denormal, invalid operation, C_2, and C_1 bits as required by the operation.

11.2.9.4 The FPATAN Instruction

This instruction expects two values on the top of stack. It pops them and computes the following:

$\text{ST0} = \tan^{-1}(\text{ ST1} / \text{ST0 })$

The resulting value is the arctangent of the ratio on the stack expressed in radians. If you have a value you wish to compute the tangent of, use FLD1 to create the appropriate ratio and then execute the FPATAN instruction.

This instruction affects the stack fault/C_1, precision, underflow, denormal, and invalid operation bits if an problem occurs during the computation. It sets the C_1 condition code bit if it has to round the result.

11.2.9.5 The FYL2X Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes:

$\text{ST0} = \text{ST1} * \log_2(\text{ ST0 })$

This instruction has no operands (to the instruction itself). The instruction uses the following syntax:

```
fyl2x();
```

Note that this instruction computes the base two logarithm. Of course, it is a trivial matter to compute the log of any other base by multiplying by the appropriate constant.

11.2.9.6 The FYL2XP1 Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes:

$\text{ST0} = \text{ST1} * \log_2(\text{ ST0} + 1.0 \text{ })$

The syntax for this instruction is

```
fyl2xp1();
```

Otherwise, the instruction is identical to FYL2X.

11.2.10Miscellaneous instructions

The FPU includes several additional instructions which control the FPU, synchronize operations, and let you test or set various status bits. These instructions include FINIT/FNINIT, FLDCW, FSTCW, FCLEX/FNCLEX, and FSTSW.

11.2.10.1 The FINIT and FNINIT Instructions

The FINIT instruction initializes the FPU for proper operation. Your applications should execute this instruction before executing any other FPU instructions. This instruction initializes the control register to 37Fh (see "The FPU Control Register" on page 612), the status register to zero (see "The FPU Status Register" on page 615) and the tag word to 0FFFFh. The other registers are unaffected.   Examples:

```
FINIT();
FNINIT();
```

The difference between FINIT and FNINIT is that FINIT first checks for any pending floating point exceptions before initializing the FPU; FNINIT does not.

11.2.10.2 The FLDCW and FSTCW Instructions

The FLDCW and FSTCW instructions require a single 16 bit memory operand:

```
    fldcw( mem_16 );
    fstcw( mem_16 );
```

These two instructions load the control register (see "The FPU Control Register" on page 612) from a memory location (FLDCW) or store the control word to a 16 bit memory location (FSTCW).

When using the FLDCW instruction to turn on one of the exceptions, if the corresponding exception flag is set when you enable that exception, the FPU will generate an immediate interrupt before the CPU executes the next instruction. Therefore, you should use the FCLEX instruction to clear any pending interrupts before changing the FPU exception enable bits.

11.2.10.3 The FCLEX and FNCLEX Instructions

The FCLEX and FNCLEX instructions clear all exception bits the stack fault bit, and the busy flag in the FPU status register (see "The FPU Status Register" on page 615). Examples:

```
fclex();
fnclex();
```

The difference between these instructions is the same as FINIT and FNINIT.

11.2.10.4 The FSTSW and FNSTSW Instructions

```
    fstsw( ax )
    fnstsw( ax )
    fstsw( mem_16 )
    fnstsw( mem_16 )
```

These instructions store the FPU status register (see "The FPU Status Register" on page 615) into a 16 bit memory location or the AX register. These instructions are unusual in the sense that they can copy an FPU value into one of the 80x86 general purpose registers (specifically, AX). Of course, the whole purpose behind allowing the transfer of the status register into AX is to allow the CPU to easily test the condition code register with the SAHF instruction. The difference between FSTSW and FNSTSW is the same as for FCLEX and FNCLEX.

11.2.11 Integer Operations

The 80x87 FPUs provide special instructions that combine integer to extended precision conversion along with various arithmetic and comparison operations. These instructions are the following:

```
fiadd( int_16_32 )
fisub( int_16_32 )
fisubr( int_16_32 )
fimul( int_16_32 )
fidiv( int_16_32 )
fidivr( int_16_32 )

ficom( int_16_32 )
ficomp( int_16_32 )
```

These instructions convert their 16 or 32 bit integer operands to an 80 bit extended precision floating point value and then use this value as the source operand for the specified operation. These instructions use ST0 as the destination operand.

11.3 Converting Floating Point Expressions to Assembly Language

Because the FPU register organization is different than the 80x86 integer register set, translating arithmetic expressions involving floating point operands is a little different than the techniques for translating integer expressions. Therefore, it makes sense to spend some time discussing how to manually translate floating point expressions into assembly language.

In one respect, it's actually easier to translate floating point expressions into assembly language. The stack architecture of the Intel FPU eases the translation of arithmetic expressions into assembly language. If you've ever used a Hewlett-Packard calculator, you'll be right at home on the FPU because, like the HP calculator, the FPU uses *reverse polish notation*, or *RPN*, for arithmetic calculations. Once you get used to using RPN, it's actually a bit more convenient for translating expressions because you don't have to worry about allocating temporary variables - they always wind up on the FPU stack.

RPN, as opposed to standard *infix notation*, places the operands before the operator. The following examples give some simple examples of infix notation and the corresponding RPN notation:

```
infix notation       RPN notation
   5 + 6                5  6  +
   7 - 2                7  2  -
   x * y                x  y  *
   a / b                a  b  /
```

An RPN expression like "5 6 +" says "push five onto the stack, push six onto the stack, then pop the value off the top of stack (six) and add it to the new top of stack." Sound familiar? This is exactly what the FLD and FADD instructions do. In fact, you can calculate this using the following code:

```
fld( 5.0 );
fld( 6.0 );
fadd();        // 11.0 is now on the top of the FPU stack.
```

As you can see, RPN is a convenient notation because it's very easy to translate this code into FPU instructions.

One advantage to RPN (or *postfix notation*) is that it doesn't require any parentheses. The following examples demonstrate some slightly more complex infix to postfix conversions:

```
infix notation              postfix notation
(x + y) * 2                 x  y + 2 *
x * 2 - (a + b)             x 2 * a b + -
(a + b) * (c + d)           a b + c d + *
```

The postfix expression "x y + 2 *" says "push x, then push y; next, add those values on the stack (producing X+Y on the stack). Next, push 2 and then multiply the two values (two and X+Y) on the stack to produce two times the quantity X+Y." Once again, we can translate these postfix expressions directly into assembly language. The following code demonstrates the conversion for each of the above expressions:

```
//      x y + 2 *

        fld( x );
        fld( y );
        fadd();
        fld( 2.0 );
        fmul();

//      x 2 * a b + -

        fld( x );
        fld( 2.0 );
        fmul();
        fld( a );
        fld( b );
        fadd();
        fsub();

//      a b + c d + *

        fld( a );
        fld( b );
        fadd();
        fld( c );
        fld( d );
        fadd();
        fmul();
```

11.3.1 Converting Arithmetic Expressions to Postfix Notation

Since the process of translating arithmetic expressions into assembly language involves postfix (RPN) notation, converting arithmetic expressions into postfix notation seems like the right place to start. This section will concentrate on that conversion.

For simple expressions, those involving two operands and a single expression, the translation is trivial. Simply move the operator from the infix position to the postfix position (that is, move the operator from inbetween the operands to after the second operand). For example, "5 + 6" becomes "5 6 +". Other than separating your operands so you don't confuse them (i.e., is it "5" and "6" or "56"?) there isn't much to converting simple infix expressions into postfix notation.

For complex expressions, the idea is to convert the simple sub-expressions into postfix notation and then treat each converted subexpression as a single operand in the remaining expression. The following discussion will surround completed conversions in square brackets so it is easy to see which text needs to be treated as a single operand in the conversion.

As for integer expression conversion, the best place to start is in the inner-most parenthetical sub-expression and then work your way outward considering precedence, associativity, and other parenthetical sub-expressions. As a concrete working example, consider the following expression:

```
x = ((y-z)*a) - ( a + b * c )/3.14159
```

A possible first translation is to convert the subexpression "(y-z)" into postfix notation. This is accomplished as follows:

```
x = ([y z -] * a) - ( a + b * c )/3.14159
```

Square brackets surround the converted postfix code just to separate it from the infix code. These exist only to make the partial translations more readable. Remember, for the purposes of conversion we will treat the text inside the square brackets as a single operand. Therefore, you would treat "[y z -]" as though it were a single variable name or constant.

The next step is to translate the subexpression "([y z -] * a)" into postfix form. This yields the following:

```
x = [y z - a *] - ( a + b * c )/3.14159
```

Next, we work on the parenthetical expression "(a + b * c)." Since multiplication has higher precedence than addition, we convert "b*c" first:

```
x = [y z - a *] - ( a + [b c *])/3.14159
```

After converting "b*c" we finish the parenthetical expression:

```
x = [y z - a *] - [a b c * +]/3.14159
```

This leaves only two infix operators: subtraction and division. Since division has the higher precedence, we'll convert that first:

```
x = [y z - a *] - [a b c * + 3.14159 /]
```

Finally, we convert the entire expression into postfix notation by dealing with the last infix operation, subtraction:

```
x = [y z - a *] [a b c * + 3.14159 /] -
```

Removing the square brackets to give us true postfix notation yields the following RPN expression:

```
x = y z - a * a b c * + 3.14159 / -
```

Here is another example of an infix to postfix conversion:

```
a = (x * y - z + t)/2.0
```

Step 1: Work inside the parentheses. Since multiplication has the highest precedence, convert that first:

```
a = ( [x y *] - z + t)/2.0
```

Step 2: Still working inside the parentheses, we note that addition and subtraction have the same precedence, so we rely upon associativity to determine what to do next. These operators are left associative, so we must translate the expressions in a left to right order. This means translate the subtraction operator first:

```
a = ( [x y * z -] + t)/2.0
```

Step 3: Now translate the addition operator inside the parentheses. Since this finishes the parenthetical operators, we can drop the parentheses:

```
a = [x y * z - t +]/2.0
```

Step 4: Translate the final infix operator (division). This yields the following:

```
a = [x y * z - t + 2.0 / ]
```

Step 5: Drop the square brackets and we're done:

```
a = x y * z - t + 2.0 /
```

11.3.2 Converting Postfix Notation to Assembly Language

Once you've translated an arithmetic expression into postfix notation, finishing the conversion to assembly language is especially easy. All you have to do is issue an FLD instruction whenever you encounter an operand and issue an appropriate arithmetic instruction when you encounter an operator. This section will use the completed examples from the previous section to demonstrate how little there is to this process.

```
x = y z - a * a b c * + 3.14159 / -
```

- Step 1: Convert *y* to FLD(y);
- Step 2: Convert *z* to FLD(z);
- Step 3: Convert "-" to FSUB();
- Step 4: Convert *a* to FLD(a);
- Step 5: Convert "*" to FMUL();
- Steps 6-n: Continuing in a left-to-right fashion, generate the following code for the expression:

```
fld( y );
fld( z );
fsub();
fld( a );
fmul();
fld( a );
fld( b );
fld( c );
fmul();
fadd();
fldpi();      // Loads pi (3.14159)
fdiv();
fsub();

fstp( x );    // Store result away into x.
```

Here's the translation for the second example in the previous section:

```
a = x y * z - t + 2.0 /
```

```
fld( x );
fld( y );
fmul();
fld( z );
fsub();
fld( t );
fadd();
fld( 2.0 );
fdiv();

fstp( a );    // Store result away into a.
```

As you can see, the translation is fairly trivial once you've converted the infix notation to postfix notation. Also note that, unlike integer expression conversion, you don't need any explicit temporaries. It turns out that the FPU stack provides the temporaries for you[3]. For these reasons, conversion of floating point expressions into assembly language is actually easier than converting integer expressions.

11.3.3 Mixed Integer and Floating Point Arithmetic

Throughout the previous sections on floating point arithmetic an unstated assumption was made: all operands in the expressions were floating point variables or constants. In the real world, you'll often need to mix integer and floating point operands in the same expression. Thanks to the FILD instruction, this is a trivial exercise.

Of course, the FPU cannot operate on integer operands. That is, you cannot push an integer operand (in integer format) onto the FPU stack and add this integer to a floating point value that is also on the stack. Instead, you use the FILD instruction to load and translate the integer value; this winds up pushing the floating point equivalent of the integer onto the FPU stack. Once the value is converted to a floating point number, you continue the calculation using the standard real arithmetic operations.

Embedding a floating point value in an integer expression is a little more problematic. In this case you must convert the floating point value to an integer value for use in the integer expression. To do this, you must use the FIST instruction. FIST converts the floating point value on the top of stack into an integer value according to the setting of the rounding bits in the floating point control register (See "The FPU Control Register" on page 612). By default, FIST will round the floating point value to the nearest integer before storing the value into memory; if you want to use the more common fraction truncation mode, you will need to change the value in the FPU control register. You compute the integer expression using the techniques from the previous chapter (see "Complex Expressions" on page 600). The FPU participates only to the point of converting the floating point value to an integer.

```
static
    intVal1 : uns32 := 1;
    intVal2 : uns32 := 2;
    realVal : real64;
        .
        .
        .
        fild( intVal1 );
        fild( intVal2 );
        fadd();
        fstp( realVal );
        stdout.put( "realVal = ", realVal, nl );
```

11.4 HLA Standard Library Support for Floating Point Arithmetic

The HLA Standard Library provides several routines that support the use of real number on the FPU. In Volume One you saw, with one exception, how the standard input and output routines operate. This section will not repeat that discussion, see "HLA Support for Floating Point Values" on page 93 for more details. One input function that Volume One only mentioned briefly was the *stdin.getf* function. This section will elaborate on that function. The HLA Standard Library also includes the "math.hhf" module that provides several mathematical functions that the FPU doesn't directly support. This section will discuss those functions, as well.

3. Assuming, of course, that your calculations aren't so complex that you exceed the eight-element limitation of the FPU stack.

11.4.1 The stdin.getf and fileio.getf Functions

The *stdin.getf* function reads a floating point value from the standard input device. It leaves the converted value in ST0 (i.e., on the top of the floating point stack). The only reason Chapter Two did not discuss this function thoroughly was because you hadn't seen the FPU and FPU registers at that point.

The *stdin.getf* function accepts the same inputs that "stdin.get(fp_variable);" would except. The only difference between the two is where these functions store the floating point value.

As you'd probably surmise, there is a corresponding *fileio.getf* function as well. This function reads the floating point value from the file whose file handle is the single parameter in this function call. It, too, leaves the converted result on the top of the FPU stack.

11.4.2 Trigonometric Functions in the HLA Math Library

The FPU provides a small handful of trigonometric functions. It does not, however, support the full range of trig functions. The HLA MATH.HHF module fills in most of the missing functions. The trigonometric functions that HLA provides include

- ACOS(arc cosine)
- ACOT (arc cotangent)
- ACSC(arc cosecant)
- ASEC (arc secant)
- ASIN (arc sin)
- COT (cotangent)
- CSC (cosecant)
- SEC (secant)

The HLA Standard Library actually provides five different routines you can call for each of these functions. For example, the prototypes for the first four COT (cotangent) routines are:

```
procedure cot32( r32: real32 );
procedure cot64( r64: real64 );
procedure cot80( r80: real80 );
procedure _cot();
```

The first three routines push their parameter onto the FPU stack and compute the cotangent of the result. The fourth routine above (*_cot*) computes the cotangent of the value in ST0.

The fifth routine is actually an overloaded procedure that calls one of the four routines above depending on the parameter. This call uses the following syntax:

```
cot();         // Calls _cot() to compute cot(ST0).
cot( r32 );    // Calls cot32 to compute the cotangent of r32.
cot( r64 );    // Calls cot64 to compute the cotangent of r64.
cot( r80 );    // Calls cot80 to compute the cotangent of r80.
```

Using this fifth form is probably preferable since it is much more convenient. Note that there is no efficiency loss when you used *cot* rather than one of the other cotangent routines. HLA actually translates this statement directly into one of the other calls.

The HLA trigonometric functions that require an angle as a parameter expect that angle to be expressed in radians, not degrees. Keep in mind that some of these functions produce undefined results for certain input values. If you've enabled exceptions on the FPU, these functions will raise the appropriate FPU exception if an error occurs.

11.4.3 Exponential and Logarithmic Functions in the HLA Math Library

The HLA MATH.HHF module provides several exponential and logarithmic functions in addition to the trigonometric functions. Like the trig functions, the exponential and logarithmic functions provide five different interfaces to each function depending on the size and location of the parameter. The functions that MATH.HHF supports are

• TwoToX	(raise 2.0 to the specified power).
• TenToX	(raise 10.0 to the specified power).
• exp	(raises e [2.718281828...] to the specified power).
• YtoX	(raises first parameter to the power specified by the second parameter).
• log	(computes base 10 logarithm).
• ln	(computes base e logarithm).

Except for the *YtoX* function, all these functions provide the same sort of interface as the *cot* function mentioned in the previous section. For example, the *exp* function provides the following prototypes:

```
procedure exp32( r32: real32 );
procedure exp64( r64: real64 );
procedure exp80( r80: real80 );
procedure _exp();
```

The exp function, by itself, automatically calls one of the above functions depending on the parameter type (and presence of a parameter):

```
exp();          // Calls _exp() to compute exp(ST0).
exp( r32 );     // Calls exp32 to compute the e**r32.
exp( r64 );     // Calls exp64 to compute the e**r64.
exp( r80 );     // Calls exp80 to compute the e**r80.
```

The lone exception to the above is the *YtoX* function. *YtoX* has its own rules because it has two parameters rather than one (Y and X). *YtoX* provides the following function prototypes:

```
procedure YtoX32( y: real32; x: real32 );
procedure YtoX64( y: real64; x: real64 );
procedure YtoX80( y: real80; x: real80 );
procedure _YtoX();
```

The *_YtoX* function computes ST1**ST0 (i.e., ST1 raised to the ST0 power).

The YtoX function provides the following interface:

```
YtoX();                // Calls _YtoX() to compute exp(ST0).
YtoX( y32, x32);       // Calls YtoX32 to compute y32**x32.
YtoX( y64, x64 );      // Calls YtoX64 to compute y64**x64.
YtoX( y80, x80 );      // Calls YtoX80 to compute y80**x80.
```

11.5 Sample Program

This chapter presents a simple “Reverse Polish Notation” calculator that demonstrates the use of the 80x86 FPU. In addition to demonstrating the use of the FPU, this sample program also introduces a few new routines from the HLA Standard Library, specifically the *arg.c*, *arg.v*, and *conv.strToFlt* routines.

The HLA Standard Library conversions module (“conv.hhf”) contains dozens of procedures that translate data between various formats. A large percentage of these routines convert data between some internal numeric form and a corresponding string format. The *conv.strToFlt* routine, as its name suggests, converts string data to a floating point value. The prototype for this function is the following:

```
procedure conv.strToFlt( s:string; index:dword );
```

The first parameter is the string containing the textual representation of the floating point value. The second parameter contains an index into the string where the floating point text actually begins (usually the *index* parameter is zero if the string contains nothing but the floating point text). The *conv.strToFlt* procedure will attempt to convert the specified string to a floating point number. If there is a problem, this function will raise an appropriate exception (e.g., *ex.ConversionError*). In fact, the HLA *stdin* routines that read floating point values from the user actually read string data and call this same procedure to convert that data to a floating point value; hence, you should protect this procedure call with a TRY..ENDTRY statement exactly the same way you protect a call to *stdin.get* or *stdin.getf*. If this routine is successful, it leaves the converted floating point value on the top of the FPU stack.

The HLA Standard Library contains numerous other procedures that convert textual data to the corresponding internal format. Examples include *conv.strToi8*, *conv.strToi16*, *conv.strToi32*, *conv.strToi64*, and more. See the HLA Standard Library documentation for more details.

This sample program also uses the *arg.c* and *arg.v* routines from the HLA Standard Library's command line arguments module ("args.hhf"). These functions provide access to the text following the program name when you run a program from the command line prompt. This calculator program, for example, expects the user to supply the desired calculation on the command line immediately after the program name. For example to add the two values 18 and 22 together, you'd specify the following command line:

```
rpncalc 18  22  +
```

The text "18 22 +" is an example of three *command line parameters*. Programs often use command line parameters to communicate filenames and other data to the application. For example, the HLA compiler uses command line parameters to pass the names of the source files to the compiler. The rpncalc program uses the command line to pass the RPN expression to the calculator.

The *arg.c* function ("argument count") returns the number of parameters on the command line. This function returns the count in the EAX register. It does not have any parameters. In general, you can probably assume that the maximum possible number of command line arguments is between 64 and 128. Note that the operating system counts the program's name on the command line in this argument count. Therefore, this value will always be one or greater. If *arg.c* returns one, then there are no extra command line parameters; the single item is the program's name.

A program that expects at least one command line parameter should always call *arg.c* and verify that it returns the value two or greater. Programs that process command line parameters typically execute a loop of some sort that executes the number of times specified by *arg.c*'s return value. Of course, when you use *arg.c*'s return value for this purpose, don't forget to subtract one from the return result to account for the program's name (unless you are treating the program name as one more parameter).

The *arg.v* function returns a string containing one of the program's command line arguments. This function has the following prototype:

```
procedure arg.v( index:uns32 );
```

The *index* parameter specifies which command line parameter you wish to retrieve. The value zero returns a string containing the program's name. The value one returns the first command line parameter following the program's name. The value two returns a string containing the second command line parameter following the program's name. Etc. The value you provide as a parameter to this function must fall in the range 0..*arg.c()*-1 or *arg.v* will raise an exception.

The *arg.v* procedure allocates storage for the string it returns on the heap by calling stralloc. It returns a pointer to the allocated string in the EAX register. Don't forget to call *strfree* to return the storage to the system after you are done processing the command line parameter.

Well, without further ado, here is the RPN calculator program that uses the aforementioned functions.

```
// This sample program demonstrates how to use
// the FPU to create a simple RPN calculator.
// This program reads a string from the user
// and “parses” that string to figure out the
// calculation the user is requesting.  This
// program assumes that any item beginning
// with a numeric digit is a numeric operand
// to push onto the FPU stack and all other
// items are operators.
//
//  Example of typical user input:
//
//      calc 123.45 67.89 +
//
//  The program responds by printing
//
//      Result = 1.91340000000000000e+2
//
//
//  Current operators supported:
//
//      + - * /
//
//  Current functions supported:
//
//      sin sqrt


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


static
    argc:       uns32;
    curOperand: string;
    ItemsOnStk: uns32;
    realRslt:   real80;


    // The following function converts an
    // angle (in ST0) from degrees to radians.
    // It leaves the result in ST0.

    procedure DegreesToRadians; @nodisplay;
    begin DegreesToRadians;

        fld( 2.0 ); // Radians = degrees*2*pi/360.0
        fmul();
        fldpi();
        fmul();
        fld( 360.0 );
        fdiv();

    end DegreesToRadians;





begin RPNcalculator;

    // Initialize the FPU.
```

```
finit();

// Okay, extract the items from the Windows
// CMD.EXE command line and process them.

arg.c();
if( eax <= 1 ) then

    stdout.put( “Usage: ‘rpnCalc <rpn expression>’” nl );
    exit RPNcalculator;

endif;

// ECX holds the index of the current operand.
// ItemsOnStk keeps track of the number of numeric operands
//  pushed onto the FPU stack so we can ensure that each
//  operation has the appropriate number of operands.

mov( eax, argc );
mov( 1, ecx );
mov( 0, ItemsOnStk );

// The following loop repeats once for each item on the
// command line:

while( ecx < argc ) do

    // Get the string associated with the current item:

    arg.v( ecx );   // Note that this malloc’s storage!
    mov( eax, curOperand );


    // If the operand begins with a numeric digit, assume
    // that it’s a floating point number.

    if( (type char [eax]) in ‘0’..’9’ ) then

        try

            // Convert this string representation of a numeric
            // value to the equivalent real value.  Leave the
            // result on the top of the FPU stack.  Also, bump
            // ItemsOnStk up by one since we’re pushing a new
            // item onto the FPU stack.

            conv.strToFlt( curOperand, 0 );
            inc( ItemsOnStk );

          exception( ex.ConversionError )

            stdout.put(“Illegal floating point constant” nl );
            exit RPNcalculator;

          anyexception

            stdout.put
            (
                “Exception “,
                (type uns32 eax ),
                " while converting real constant"
                nl
            );
            exit RPNcalculator;

        endtry;



    // Handle the addition operation here.

    elseif( str.eq( curOperand, "+" )) then

        // The addition operation requires two
        // operands on the stack.  Ensure we have
        // two operands before proceeding.

        if( ItemsOnStk >= 2 ) then

            fadd();
            dec( ItemsOnStk );  // fadd() removes one operand.

        else

            stdout.put( "'+' operation requires two operands." nl );
            exit RPNcalculator;

        endif;


    // Handle the subtraction operation here.  See the comments
    // for FADD for more details.

    elseif( str.eq( curOperand, "-" )) then

        if( ItemsOnStk >= 2 ) then

            fsub();
            dec( ItemsOnStk );

        else

            stdout.put( "'-' operation requires two operands." nl );
            exit RPNcalculator;

        endif;


    // Handle the multiplication operation here.  See the comments
    // for FADD for more details.

    elseif( str.eq( curOperand, "*" )) then

        if( ItemsOnStk >= 2 ) then

            fmul();
            dec( ItemsOnStk );

        else

            stdout.put( "'*' operation requires two operands." nl );
```

```
                exit RPNcalculator;

            endif;


        // Handle the division operation here.  See the comments
        // for FADD for more details.

        elseif( str.eq( curOperand, "/" )) then

            if( ItemsOnStk >= 2 ) then

                fdiv();
                dec( ItemsOnStk );

            else

                stdout.put( "'/' operation requires two operands." nl );
                exit RPNcalculator;

            endif;



        // Provide a square root operation here.

        elseif( str.eq( curOperand, "sqrt" )) then

            // Sqrt is a monadic (unary) function.  Therefore
            // we only require a single item on the stack.

            if( ItemsOnStk >= 1 ) then

                fsqrt();

            else

                stdout.put
                (
                    "SQRT function requires at least one operand."
                    nl
                );
                exit RPNcalculator;

            endif;


        // Provide the SINE function here.  See SQRT comments for details.

        elseif( str.eq( curOperand, "sin" )) then

            if( ItemsOnStk >= 1 ) then

                DegreesToRadians();
                fsin();

            else

                stdout.put( "SIN function requires at least one operand." nl );
                exit RPNcalculator;
            endif;

        else

            stdout.put( "'", curOperand, "' is an unknown operation." nl );
            exit RPNcalculator;

        endif;

        // Free the storage associated with the current item.

        strfree( curOperand );

        // Move on to the next item on the command line:

        inc( ecx );

    endwhile;
    if( ItemsOnStk = 1 ) then

        fstp( realRslt );
        stdout.put( "Result = ", realRslt, nl );

    else

        stdout.put( "Syntax error in expression. ItemsOnStk=", ItemsOnStk, nl );

    endif;


end RPNcalculator;
```

Program 11.1 A Floating Point Calculator Program

11.6 Putting It All Together

Between the FPU and the HLA Standard Library, floating point arithmetic is actually quite simple. In this chapter you learned about the floating point instruction set and you learned how to convert arithmetic expressions involving real arithmetic into a sequence of floating point instructions. This chapter also presented several transcendental functions that the HLA Standard Library provides. Armed with the information from this chapter, you should be able to deal with floating point expressions just as easily as integer expressions.

Calculation Via Table Lookups Chapter Twelve

12.1 Chapter Overview

This chapter discusses arithmetic computation via table lookup. By the conclusion of this chapter you should be able to use table lookups to quickly compute complex functions. You will also learn how to construct these tables programmatically.

12.2 Tables

The term “table” has different meanings to different programmers. To most assembly language programmers, a table is nothing more than an array that is initialized with some data. The assembly language programmer often uses tables to compute complex or otherwise slow functions. Many very high level languages (e.g., SNOBOL4 and Icon) directly support a table data type. Tables in these languages are essentially arrays whose elements you can access with a non-integer index (e.g., floating point, string, or any other data type). HLA provides a table module that lets you index an array using a string. However, in this chapter we will adopt the assembly language programmer’s view of tables.

A table is an array containing preinitialized values that do not change during the execution of the program. A table can be compared to an array in the same way an integer constant can be compared to an integer variable. In assembly language, you can use tables for a variety of purposes: computing functions, controlling program flow, or simply “looking things up”. In general, tables provide a fast mechanism for performing some operation at the expense of some space in your program (the extra space holds the tabular data). In the following sections we’ll explore some of the many possible uses of tables in an assembly language program.

Note: since tables typically contain preinitialized data that does not change during program execution, the READONLY section is a good place to declare your table objects.

12.2.1 Function Computation via Table Look-up

Tables can do all kinds of things in assembly language. In HLLs, like Pascal, it’s real easy to create a formula which computes some value. A simple looking arithmetic expression can be equivalent to a considerable amount of 80x86 assembly language code. Assembly language programmers tend to compute many values via table look up rather than through the execution of some function. This has the advantage of being easier, and often more efficient as well. Consider the following Pascal statement:

if (character >= ‘a’) and (character <= ‘z’) then character := chr(ord(character) - 32);

This Pascal if statement converts the character variable character from lower case to upper case if character is in the range ‘a’..’z’. The HLA code that does the same thing is

```
mov( character, al );
if( al in ‘a’..’z’ ) then

   and( $5f, al );      // Same as SUB( 32, al ) in this code.

endif;
mov( al, character );
```

Note that HLA’s high level IF statement translates into four machine instructions in this particular example. Hence, this code requires a total of seven machine instructions.

Had you buried this code in a nested loop, you'd be hard pressed to improve the speed of this code without using a table look up. Using a table look up, however, allows you to reduce this sequence of instructions to just four instructions:

```
mov( character, al );
lea( ebx, CnvrtLower );
xlat
mov( al, character );
```

You're probably wondering how this code works and what is this new instruction, XLAT? The XLAT, or *translate*, instruction does the following:

```
mov( [ebx+al*1], al );
```

That is, it uses the current value of the AL register as an index into the array whose base address is contained in EBX. It fetches the byte at that index in the array and copies that byte into the AL register. Intel calls this the translate instruction because programmers typically use it to translate characters from one form to another using a lookup table. That's exactly how we are using it here.

In the previous example, *CnvrtLower* is a 256-byte table which contains the values 0..$60 at indices 0..$60, $41..$5A at indices $61..$7A, and $7B..$FF at indices $7Bh..0FF. Therefore, if AL contains a value in the range $0..$60, the XLAT instruction returns the value $0..$60, effectively leaving AL unchanged. However, if AL contains a value in the range $61..$7A (the ASCII codes for 'a'..'z') then the XLAT instruction replaces the value in AL with a value in the range $41..$5A. $41..$5A just happen to be the ASCII codes for 'A'..'Z'. Therefore, if AL originally contains an lower case character ($61..$7A), the XLAT instruction replaces the value in AL with a corresponding value in the range $61..$7A, effectively converting the original lower case character ($61..$7A) to an upper case character ($41..$5A). The remaining entries in the table, like entries $0..$60, simply contain the index into the table of their particular element. Therefore, if AL originally contains a value in the range $7A..$FF, the XLAT instruction will return the corresponding table entry that also contains $7A..$FF.

As the complexity of the function increases, the performance benefits of the table look up method increase dramatically. While you would almost never use a look up table to convert lower case to upper case, consider what happens if you want to swap cases:

```
Via computation:

        mov( character, al );
        if( al in 'a'..'z' ) then

            and( $5f, al );

        elseif( al in 'A'..'Z' ) then

            or( $20, al );

        endif;
        mov( al, character ):
```

The IF and ELSEIF statements generate four and five actual machine instructions, respectively, so this code is equivalent to 13 actual machine instructions.

The table look up code to compute this same function is:

```
mov( character, al );
lea( ebx, SwapUL );
xlat();
mov( al, character );
```

As you can see, when using a table look up to compute a function only the table changes, the code remains the same.

Table look ups suffer from one major problem – functions computed via table look up have a limited domain. The domain of a function is the set of possible input values (parameters) it will accept. For example, the upper/lower case conversion functions above have the 256-character ASCII character set as their domain.

A function such as SIN or COS accepts the set of real numbers as possible input values. Clearly the domain for SIN and COS is much larger than for the upper/lower case conversion function. If you are going to do computations via table look up, you must limit the domain of a function to a small set. This is because each element in the domain of a function requires an entry in the look up table. You won't find it very practical to implement a function via table look up whose domain the set of real numbers.

Most look up tables are quite small, usually 10 to 128 entries. Rarely do look up tables grow beyond 1,000 entries. Most programmers don't have the patience to create (and verify the correctness) of a 1,000 entry table.

Another limitation of functions based on look up tables is that the elements in the domain of the function must be fairly contiguous. Table look ups take the input value for a function, use this input value as an index into the table, and return the value at that entry in the table. If you do not pass a function any values other than 0, 100, 1,000, and 10,000 it would seem an ideal candidate for implementation via table look up, its domain consists of only four items. However, the table would actually require 10,001 different elements due to the range of the input values. Therefore, you cannot efficiently create such a function via a table look up. Throughout this section on tables, we'll assume that the domain of the function is a fairly contiguous set of values.

The best functions you can implement via table look ups are those whose domain and range is always 0..255 (or some subset of this range). You can efficiently implement such functions on the 80x86 via the XLAT instruction. The upper/lower case conversion routines presented earlier are good examples of such a function. Any function in this class (those whose domain and range take on the values 0..255) can be computed using the same two instructions: "lea(table, ebx);" and "xlat();" The only thing that ever changes is the look up table.

You cannot (conveniently) use the XLAT instruction to compute a function value once the range or domain of the function takes on values outside 0..255. There are three situations to consider:

- The domain is outside 0..255 but the range is within 0..255,
- The domain is inside 0..255 but the range is outside 0..255, and
- Both the domain and range of the function take on values outside 0..255.

We will consider each of these cases separately.

If the domain of a function is outside 0..255 but the range of the function falls within this set of values, our look up table will require more than 256 entries but we can represent each entry with a single byte. Therefore, the look up table can be an array of bytes. Next to look ups involving the XLAT instruction, functions falling into this class are the most efficient. The following Pascal function invocation,

```
B := Func(X);
```

where *Func* is

```
function Func(X:dword):byte;
```

consists of the following HLA code:

```
    mov( X, ebx );
    mov( FuncTable[ ebx ], al );
    mov( al, B );
```

This code loads the function parameter into EBX, uses this value (in the range 0..??) as an index into the *FuncTable* table, fetches the byte at that location, and stores the result into *B*. Obviously, the table must contain a valid entry for each possible value of *X*. For example, suppose you wanted to map a cursor position on the video screen in the range 0..1999 (there are 2,000 character positions on an 80x25 video display) to its *X* or *Y* coordinate on the screen. You could easily compute the *X* coordinate via the function:

X:=Posn mod 80

and the *Y* coordinate with the formula

Y:=Posn div 80

(where *Posn* is the cursor position on the screen). This can be easily computed using the 80x86 code:

```
        mov( Posn, ax );
        div( 80, ax );

// X is now in AH, Y is now in AL
```

However, the DIV instruction on the 80x86 is very slow. If you need to do this computation for every character you write to the screen, you will seriously degrade the speed of your video display code. The following code, which realizes these two functions via table look up, would improve the performance of your code considerably:

```
        movzx( Posn, ebx );          // Use a plain MOV instr if Posn is uns32
        mov( YCoord[ebx], al );      // rather than an uns16 value.
        mov( XCoord[ebx], ah );
```

If the domain of a function is within 0..255 but the range is outside this set, the look up table will contain 256 or fewer entries but each entry will require two or more bytes. If both the range and domains of the function are outside 0..255, each entry will require two or more bytes and the table will contain more than 256 entries.

Recall from the chapter on arrays that the formula for indexing into a single dimensional array (of which a table is a special case) is

```
        Address := Base + index * size
```

If elements in the range of the function require two bytes, then the index must be multiplied by two before indexing into the table. Likewise, if each entry requires three, four, or more bytes, the index must be multiplied by the size of each table entry before being used as an index into the table. For example, suppose you have a function, F(x), defined by the following (pseudo) Pascal declaration:

```
function F(x:dword):word;
```

You can easily create this function using the following 80x86 code (and, of course, the appropriate table named *F*):

```
        mov( X, ebx );
        mov( F[ebx*2], ax );
```

Any function whose domain is small and mostly contiguous is a good candidate for computation via table look up. In some cases, non-contiguous domains are acceptable as well, as long as the domain can be coerced into an appropriate set of values. Such operations are called conditioning and are the subject of the next section.

12.2.2 Domain Conditioning

Domain conditioning is taking a set of values in the domain of a function and massaging them so that they are more acceptable as inputs to that function. Consider the following function:

$$\sin x = \langle \sin x | x \in [-2\pi, 2\pi] \rangle$$

This says that the (computer) function SIN(x) is equivalent to the (mathematical) function *sin x* where

$-2\pi \le x \le 2\pi$

As we all know, sine is a circular function which will accept any real valued input. The formula used to compute sine, however, only accept a small set of these values.

This range limitation doesn't present any real problems, by simply computing SIN(X mod (2*pi)) we can compute the sine of any input value. Modifying an input value so that we can easily compute a function is called conditioning the input. In the example above we computed "X mod 2*pi" and used the result as the input to the *sin* function. This truncates *X* to the domain *sin* needs without affecting the result. We can apply input conditioning to table look ups as well. In fact, scaling the index to handle word entries is a form of input conditioning. Consider the following Pascal function:

```
function val(x:word):word; begin
    case x of
        0: val := 1;
        1: val := 1;
        2: val := 4;
        3: val := 27;
        4: val := 256;
        otherwise val := 0;
    end;
end;
```

This function computes some value for x in the range 0..4 and it returns zero if *x* is outside this range. Since *x* can take on 65,536 different values (being a 16 bit word), creating a table containing 65,536 words where only the first five entries are non-zero seems to be quite wasteful. However, we can still compute this function using a table look up if we use input conditioning. The following assembly language code presents this principle:

```
        mov( 0, ax );         // AX = 0, assume X > 4.
        movzx( x, ebx );      // Note that H.O. bits of EBX must be zero!
        if( bx <= 4 ) then

            mov( val[ ebx*2 ], ax );

        endif;
```

This code checks to see if *x* is outside the range 0..4. If so, it manually sets AX to zero, otherwise it looks up the function value through the *val* table. With input conditioning, you can implement several functions that would otherwise be impractical to do via table look up.

12.2.3 Generating Tables

One big problem with using table look ups is creating the table in the first place. This is particularly true if there are a large number of entries in the table. Figuring out the data to place in the table, then laboriously entering the data, and, finally, checking that data to make sure it is valid, is a very time-staking and boring process. For many tables, there is no way around this process. For other tables there is a better way – use the computer to generate the table for you. An example is probably the best way to describe this. Consider the following modification to the sine function:

$$(\sin x) \times r = \langle \frac{(r \times (1000 \times \sin x))}{1000} | x \in [0, 359] \rangle$$

This states that *x* is an integer in the range 0..359 and *r* must be an integer. The computer can easily compute this with the following code:

```
        movzx( x, ebx );
        mov( Sines[ ebx*2], eax );   // Get SIN(X) * 1000
        imul( r, eax );              // Note that this extends EAX into EDX.
        idiv( 1000, edx:eax );       // Compute (R*(SIN(X)*1000)) / 1000
```

Note that integer multiplication and division are not associative. You cannot remove the multiplication by 1000 and the division by 1000 because they seem to cancel one another out. Furthermore, this code must compute this function in exactly this order. All that we need to complete this function is a table containing 360 different values corresponding to the sine of the angle (in degrees) times 1,000. Entering such a table into an assembly language program containing such values is extremely boring and you'd probably make several mistakes entering and verifying this data. However, you can have the program generate this table for you. Consider the following HLA program:

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

var
    outFile: dword;
    angle:   int32;
    r:       int32;

readonly
    RoundMode: uns16 := $23f;



begin GenerateSines;

    // Open the file:

    mov( fileio.openNew( "sines.hla" ), outFile );

    // Emit the initial part of the declaration to the output file:

    fileio.put
    (
        outFile,
        stdio.tab,
        "sines: int32[360] := " nl,
        stdio.tab, stdio.tab, stdio.tab, "[" nl );

    // Enable rounding control (round to the nearest integer).

    fldcw( RoundMode );

    // Emit the sines table:

    for( mov( 0, angle); angle < 359; inc( angle )) do

        // Convert angle in degrees to an angle in radians
        // using "radians := angle * 2.0 * pi / 360.0;"

        fild( angle );
        fld( 2.0 );
        fmul();
        fldpi();
        fmul();
        fld( 360.0 );
        fdiv();

        // Okay, compute the sine of ST0

        fsin();

        // Multiply by 1000 and store the rounded result into
        // the integer variable r.
```

```
            fld( 1000.0 );
            fmul();
            fistp( r );

            // Write out the integers eight per line to the source file:
            // Note: if (angle AND %111) is zero, then angle is evenly
            // divisible by eight and we should output a newline first.

            test( %111, angle );
            if( @z ) then

                fileio.put
                (
                    outFile,
                    nl,
                    stdio.tab,
                    stdio.tab,
                    stdio.tab,
                    stdio.tab,
                    r:5,
                    ‘,’
                );

            else

                fileio.put( outFile, r:5, ‘,’ );

            endif;

        endfor;

        // Output sine(359) as a special case (no comma following it).
        // Note: this value was computed manually with a calculator.

        fileio.put
        (
            outFile,
            “  -17”,
            nl,
            stdio.tab,
            stdio.tab,
            stdio.tab,
            “];”,
            nl
        );
        fileio.close( outFile );

    end GenerateSines;
```

Program 12.1 An HLA Program that Generates a Table of Sines

The program above produces the following output:

```
sines: int32[360] :=
    [

            0,   17,   35,   52,   70,   87,  105,  122,
             139,  156,  174,  191,  208,  225,  242,  259,
             276,  292,  309,  326,  342,  358,  375,  391,
             407,  423,  438,  454,  469,  485,  500,  515,
             530,  545,  559,  574,  588,  602,  616,  629,
             643,  656,  669,  682,  695,  707,  719,  731,
             743,  755,  766,  777,  788,  799,  809,  819,
             829,  839,  848,  857,  866,  875,  883,  891,
             899,  906,  914,  921,  927,  934,  940,  946,
             951,  956,  961,  966,  970,  974,  978,  982,
             985,  988,  990,  993,  995,  996,  998,  999,
             999, 1000, 1000, 1000,  999,  999,  998,  996,
             995,  993,  990,  988,  985,  982,  978,  974,
             970,  966,  961,  956,  951,  946,  940,  934,
             927,  921,  914,  906,  899,  891,  883,  875,
             866,  857,  848,  839,  829,  819,  809,  799,
             788,  777,  766,  755,  743,  731,  719,  707,
             695,  682,  669,  656,  643,  629,  616,  602,
             588,  574,  559,  545,  530,  515,  500,  485,
             469,  454,  438,  423,  407,  391,  375,  358,
             342,  326,  309,  292,  276,  259,  242,  225,
             208,  191,  174,  156,  139,  122,  105,   87,
              70,   52,   35,   17,    0,  -17,  -35,  -52,
             -70,  -87, -105, -122, -139, -156, -174, -191,
            -208, -225, -242, -259, -276, -292, -309, -326,
            -342, -358, -375, -391, -407, -423, -438, -454,
            -469, -485, -500, -515, -530, -545, -559, -574,
            -588, -602, -616, -629, -643, -656, -669, -682,
            -695, -707, -719, -731, -743, -755, -766, -777,
            -788, -799, -809, -819, -829, -839, -848, -857,
            -866, -875, -883, -891, -899, -906, -914, -921,
            -927, -934, -940, -946, -951, -956, -961, -966,
            -970, -974, -978, -982, -985, -988, -990, -993,
            -995, -996, -998, -999, -999,-1000,-1000,-1000,
            -999, -999, -998, -996, -995, -993, -990, -988,
            -985, -982, -978, -974, -970, -966, -961, -956,
            -951, -946, -940, -934, -927, -921, -914, -906,
            -899, -891, -883, -875, -866, -857, -848, -839,
            -829, -819, -809, -799, -788, -777, -766, -755,
            -743, -731, -719, -707, -695, -682, -669, -656,
            -643, -629, -616, -602, -588, -574, -559, -545,
            -530, -515, -500, -485, -469, -454, -438, -423,
            -407, -391, -375, -358, -342, -326, -309, -292,
            -276, -259, -242, -225, -208, -191, -174, -156,
            -139, -122, -105,  -87,  -70,  -52,  -35,  -17
      ];
```

Obviously it's much easier to write the HLA program that generated this data than to enter (and verify) this data by hand. Of course, you don't even have to write the table generation program in HLA. If you prefer, you might find it easier to write the program in Pascal/Delphi, C/C++, or some other high level language. Obviously, the program will only execute once, so the performance of the table generation program is not an issue. If it's easier to write the table generation program in a high level language, by all means do so. Note, also, that HLA has a built-in interpreter that allows you to easily create tables without having to use an external program. For more details, see the chapter on macros and the HLA compile-time language.

Once you run your table generation program, all that remains to be done is to cut and paste the table from the file (sines.hla in this example) into the program that will actually use the table.

12.3 High Performance Implementation of cs.rangeChar

Way back in Chapter Three this volume made the comment that the implementation of the *cs.rangeChar* was not very efficient when generating large character sets (see "Character Set Functions That Build Sets" on page 449). That chapter also mentioned that a table lookup would be a better solution for this function if you generate large character sets. That chapter also promised an table lookup implementation of *cs.rangeChar.* This section fulfills that promise.

Program 12.2 provides a table lookup implementation of this function. To understand how this function works, consider the two tables (*StartRange* and *EndRange*) appearing in this program.

Each element in the *StartRange* table is a character set whose binary representation contains all one bits from bit position zero through the index into the table. That is, element zero contains a single '1' bit in bit position zero; element one contains one bits in bit positions zero and one; element two contains one bits in bit positions zero, one, and two; etc.

Each element of the *EndRange* table contains one bits from the bit position specified by the index into the table through to bit position 127. Therefore, element zero of this array contains all one bits from positions zero through 127; element one of this array contains a zero in bit position zero and ones in bit positions one through 127; element two of this array contains zeros in bit positions zero and one and it contains ones in bit positions two through 127; etc.

The *fastRangeChar* function builds a character set containing all the characters between two characters specified as parameters. The calling sequence for this function is

```
fastRangeChar( LowBoundChar, HighBoundChar, CsetVariable );
```

This function constructs the character set "{ LowBoundChar..HighBoundChar }" and stores this character set into *CsetVariable*.

As you may recall from the discussion of *cs.rangeChar's* low-level implementation, it constructed the character set by running a FOR loop from the *LowBoundChar* through to the *HighBoundChar* and set the corresponding bit in the character set on each iteration of the loop. So to build the character set {'a'..'z'} the loop would have to execute 26 times. The *fastRangeChar* function avoids this iteration by construction a set containing all elements from #0 to *HighBoundChar* and intersecting this set with a second character set containing all the characters from *LowBoundChar* to #127. The *fastRangeChar* function doesn't actually build these two sets, of course, it uses *HighBoundChar* and *LowBoundChar* as indices into the *StartRange* and *EndRange* tables, respectively. The intersection of these two table elements computes the desired result set. As you'll see by looking at *fastRangeChar*, this function computes the intersection of these two sets on the fly by using the AND instruction. Without further ado, here's the program:

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

static

    // Note: the following tables were generated
    // by the genRangeChar program:

    StartRange: cset[128] :=
            [
                {#0},
                {#0..#1},
                {#0..#2},
                {#0..#3},
                {#0..#4},
                {#0..#5},
                    {#0..#6},
                    {#0..#7},
                    {#0..#8},
                    {#0..#9},
                    {#0..#10},
                    {#0..#11},
                    {#0..#12},
                    {#0..#13},
                    {#0..#14},
                    {#0..#15},
                    {#0..#16},
                    {#0..#17},
                    {#0..#18},
                    {#0..#19},
                    {#0..#20},
                    {#0..#21},
                    {#0..#22},
                    {#0..#23},
                    {#0..#24},
                    {#0..#25},
                    {#0..#26},
                    {#0..#27},
                    {#0..#28},
                    {#0..#29},
                    {#0..#30},
                    {#0..#31},
                    {#0..#32},
                    {#0..#33},
                    {#0..#34},
                    {#0..#35},
                    {#0..#36},
                    {#0..#37},
                    {#0..#38},
                    {#0..#39},
                    {#0..#40},
                    {#0..#41},
                    {#0..#42},
                    {#0..#43},
                    {#0..#44},
                    {#0..#45},
                    {#0..#46},
                    {#0..#47},
                    {#0..#48},
                    {#0..#49},
                    {#0..#50},
                    {#0..#51},
                    {#0..#52},
                    {#0..#53},
                    {#0..#54},
                    {#0..#55},
                    {#0..#56},
                    {#0..#57},
                    {#0..#58},
                    {#0..#59},
                    {#0..#60},
                    {#0..#61},
                    {#0..#62},
                    {#0..#63},
                    {#0..#64},
                    {#0..#65},
                    {#0..#66},
{#0..#67},
{#0..#68},
{#0..#69},
{#0..#70},
{#0..#71},
{#0..#72},
{#0..#73},
{#0..#74},
{#0..#75},
{#0..#76},
{#0..#77},
{#0..#78},
{#0..#79},
{#0..#80},
{#0..#81},
{#0..#82},
{#0..#83},
{#0..#84},
{#0..#85},
{#0..#86},
{#0..#87},
{#0..#88},
{#0..#89},
{#0..#90},
{#0..#91},
{#0..#92},
{#0..#93},
{#0..#94},
{#0..#95},
{#0..#96},
{#0..#97},
{#0..#98},
{#0..#99},
{#0..#100},
{#0..#101},
{#0..#102},
{#0..#103},
{#0..#104},
{#0..#105},
{#0..#106},
{#0..#107},
{#0..#108},
{#0..#109},
{#0..#110},
{#0..#111},
{#0..#112},
{#0..#113},
{#0..#114},
{#0..#115},
{#0..#116},
{#0..#117},
{#0..#118},
{#0..#119},
{#0..#120},
{#0..#121},
{#0..#122},
{#0..#123},
{#0..#124},
{#0..#125},
{#0..#126},
{#0..#127}
```

```
                ];

        EndRange: cset[128] :=
                [
                    {#0..#127},
                    {#1..#127},
                    {#2..#127},
                    {#3..#127},
                    {#4..#127},
                    {#5..#127},
                    {#6..#127},
                    {#7..#127},
                    {#8..#127},
                    {#9..#127},
                    {#10..#127},
                    {#11..#127},
                    {#12..#127},
                    {#13..#127},
                    {#14..#127},
                    {#15..#127},
                    {#16..#127},
                    {#17..#127},
                    {#18..#127},
                    {#19..#127},
                    {#20..#127},
                    {#21..#127},
                    {#22..#127},
                    {#23..#127},
                    {#24..#127},
                    {#25..#127},
                    {#26..#127},
                    {#27..#127},
                    {#28..#127},
                    {#29..#127},
                    {#30..#127},
                    {#31..#127},
                    {#32..#127},
                    {#33..#127},
                    {#34..#127},
                    {#35..#127},
                    {#36..#127},
                    {#37..#127},
                    {#38..#127},
                    {#39..#127},
                    {#40..#127},
                    {#41..#127},
                    {#42..#127},
                    {#43..#127},
                    {#44..#127},
                    {#45..#127},
                    {#46..#127},
                    {#47..#127},
                    {#48..#127},
                    {#49..#127},
                    {#50..#127},
                    {#51..#127},
                    {#52..#127},
                    {#53..#127},
                    {#54..#127},
                    {#55..#127},
                    {#56..#127},
                    {#57..#127},
                    {#58..#127},
                    {#59..#127},
                    {#60..#127},
                    {#61..#127},
                    {#62..#127},
                    {#63..#127},
                    {#64..#127},
                    {#65..#127},
                    {#66..#127},
                    {#67..#127},
                    {#68..#127},
                    {#69..#127},
                    {#70..#127},
                    {#71..#127},
                    {#72..#127},
                    {#73..#127},
                    {#74..#127},
                    {#75..#127},
                    {#76..#127},
                    {#77..#127},
                    {#78..#127},
                    {#79..#127},
                    {#80..#127},
                    {#81..#127},
                    {#82..#127},
                    {#83..#127},
                    {#84..#127},
                    {#85..#127},
                    {#86..#127},
                    {#87..#127},
                    {#88..#127},
                    {#89..#127},
                    {#90..#127},
                    {#91..#127},
                    {#92..#127},
                    {#93..#127},
                    {#94..#127},
                    {#95..#127},
                    {#96..#127},
                    {#97..#127},
                    {#98..#127},
                    {#99..#127},
                    {#100..#127},
                    {#101..#127},
                    {#102..#127},
                    {#103..#127},
                    {#104..#127},
                    {#105..#127},
                    {#106..#127},
                    {#107..#127},
                    {#108..#127},
                    {#109..#127},
                    {#110..#127},
                    {#111..#127},
                    {#112..#127},
                    {#113..#127},
                    {#114..#127},
                    {#115..#127},
                    {#116..#127},
                    {#117..#127},
                    {#118..#127},
                    {#119..#127},
                    {#120..#127},
                    {#121..#127},
                    {#122..#127},
                    {#123..#127},
                    {#124..#127},
                    {#125..#127},
                    {#126..#127},
                    {#127}
                ];


    /*********************************************************************/
    /*                                                                   */
    /* fastRangeChar-                                                    */
    /*                                                                   */
    /* A fast implementation of cs.rangeChar that uses a table lookup    */
    /* to speed up the generation of the character set for really large  */
    /* sets (note: because of memory latencies, this function is probably */
    /* slower than cs.rangeChar for small character sets).               */
    /*                                                                   */
    /*********************************************************************/

    procedure fastRangeChar( LowBound:char; HighBound:char; var csDest:cset );
    begin fastRangeChar;

        push( eax );
        push( ebx );
        push( esi );
        push( edi );
        mov( csDest, ebx );     // Get pointer to destination character set.

        // Copy EndRange[ LowBound ] into csDest.  This adds all the
        // characters from LowBound to #127 into csDest.  Then intersect
        // this set with StartRange[ HighBound ] to trim off all the
        // characters after HighBound.

        movzx( LowBound, esi );
        shl( 4, esi );               // *16 ‘cause each element is 16 bytes.
        movzx( HighBound, edi );
        shl( 4, edi );

        mov( (type dword EndRange[ esi + 0 ]), eax );
        and( (type dword StartRange[ edi + 0 ]), eax );  // Does the intersection.
        mov( eax, (type dword [ ebx+0 ]));

        mov( (type dword EndRange[ esi + 4 ]), eax );
        and( (type dword StartRange[ edi + 4 ]), eax );
        mov( eax, (type dword [ ebx+4 ]));

        mov( (type dword EndRange[ esi + 8 ]), eax );
        and( (type dword StartRange[ edi + 8 ]), eax );
        mov( eax, (type dword [ ebx+8 ]));

        mov( (type dword EndRange[ esi + 12 ]), eax );
        and( (type dword StartRange[ edi + 12 ]), eax );
        mov( eax, (type dword [ ebx+12 ]));

        pop( edi );
        pop( esi );
        pop( ebx );
        pop( eax );

    end fastRangeChar;



    static
        TestCset: cset := {};

    begin csRangeChar;

        fastRangeChar( ‘a’, ‘z’, TestCset );
        stdout.put( “Result from fastRangeChar: {“, TestCset, “}” nl );

    end csRangeChar;
```

Program 12.2 Table Lookup Implementation of cs.rangeChar

Naturally, the *StartRange* and *EndRange* tables were not hand-generated. An HLA program generated these two tables (with a combined 512 elements). Program 12.3 is the program that generated these tables.

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

var
    outFile: dword;



begin GenerateRangeChar;

    // Open the file:

    mov( fileio.openNew( “rangeCharCset.hla” ), outFile );

    // Emit the initial part of the declaration to the output file:

    fileio.put
    (
        outFile,
        stdio.tab,
        “StartRange: cset[128] := “ nl,
        stdio.tab, stdio.tab, stdio.tab, “[“ nl,
        stdio.tab, stdio.tab, stdio.tab, stdio.tab, “{#0},” nl  // element zero

    );
    for( mov( 1, ecx ); ecx < 127; inc( ecx )) do

        fileio.put
        (
            outFile,
            stdio.tab, stdio.tab, stdio.tab, stdio.tab,
            "{#0..#",
            (type uns32 ecx),
            "},” nl
        );

    endfor;
    fileio.put
    (
        outFile,
        stdio.tab, stdio.tab, stdio.tab, stdio.tab,
        "{#0..#127}” nl,
        stdio.tab, stdio.tab, stdio.tab, "];” nl
    );


    // Now emit the second table to the file:

    fileio.put
    (
        outFile,
        nl,
        stdio.tab,
        "EndRange: cset[128] := “ nl,
        stdio.tab, stdio.tab, stdio.tab, "[“ nl
    );
    for( mov( 0, ecx ); ecx < 127; inc( ecx )) do

        fileio.put
        (
            outFile,
            stdio.tab, stdio.tab, stdio.tab, stdio.tab,
            "{#",
            (type uns32 ecx),
            "..#127},” nl
        );

    endfor;
    fileio.put
    (
        outFile,
        stdio.tab, stdio.tab, stdio.tab, stdio.tab, "{#127}” nl,
        stdio.tab, stdio.tab, stdio.tab, "];” nl nl
    );
    fileio.close( outFile );

end GenerateRangeChar;
```

Program 12.3 Table Generation Program for the csRangeChar Program

Questions, Projects, and Labs Chapter Thirteen

13.1 Questions

1) What is the purpose of the INTO instruction?
2) What was the main purpose for the INTMUL instruction in this volume? What was it introduced before the chapter on integer arithmetic?
3) Describe how to declare byte variables. Give several examples. What would you normally use byte variables for in a program?
4) Describe how to declare word variables. Give several examples. Describe what you would use them for in a program.
5) Repeat question 4 for double word variables.
6) What are qword and tbyte objects?
7) How does HLA differentiate string and character constants?
8) Explain the purpose of the TYPE section. Give several examples of its use.
9) What is a pointer variable?
10) How do you declare pointer variables in a program?
11) How do you access the object pointed at by a pointer. Give an example using 80x86 instructions.
12) What is the difference between a CONST object and a VAL object?
13) What is the "?" statement used for?
14) What is an enumerated data type?
15) Given the two literal string constants "Hello " and "World", provide two fundamentally different ways to concatenate these string literal constants into a single string within an HLA source file.
16) What is the difference between a STRING constant and a TEXT constant?
17) What is a pointer constant? What is the limitation on a pointer constant?
18) What is a pointer expression? What are the limitations on pointer expressions?
19) What is a dangling pointer? How do you avoid the problem of dangling pointers in your programs?
20) What is a memory leak? What causes it? How can you avoid memory leaks in y our programs?
21) What is likely to happen if you use an uninitialized pointer?
22) What is a composite data type?
23) If you have a variable of type STRING, how many bytes in memory does this variable consume? What value(s) appear in this variable?
24) What is the data format for HLA string data?
25) What is the difference between a length-prefixed and a zero terminated string? What are the advantages and disadvantages of each?
26) Are HLA string zero terminated or length prefixed?
27) How do you directly obtain the length of an HLA string (i.e., w/o using the str.length function)?
28) What are the two pieces of length data maintained for each HLA string? What is the difference between them?
29) Why would you want to use the *str.cpy* routine (which is slow) rather than simply copying the pointer data from one string to another (which is fast)?

30) Many string functions contain the prefix “a_” in their name (e.g., str.cpy vs. str.a_cpy). What is the difference between the functions containing the “a_” prefix and those that do not contain this prefix?

31) Explain how to access the individual characters in an HLA string variable.

32) What is the purpose of string concatenation?

33) What is the difference between the str.cat and str.insert? Which is the more general of the two routines? How could you use the more general routine to simulate the other?

34) How can you perform a case-insensitive string comparison using the HLA Standard Library Strings module?

35) Explain how to convert an integer to its string representation using functions from the HLA Standard Library’s String module.

36) How does HLA implement character sets? How many bytes does a character set variable consume?

37) If *CST* is a character set constant, what does “-CST” produce?

38) Explain the purpose of the character classification routines in the HLA Standard Library chars.hhf module.

39) How do you compute the intersection, union, and difference of two character set constants in an HLA constant expression?

40) How do you compute the interesection, union, and difference of two character set variables in an HLA program at run-time?

41) What is the difference between a subset and a proper subset?

42) Explain how you can use a character set to validate user input to a program.

43) How do you declare arrays in assembly language? Give the code for the following arrays:

a) A two dimensional 4x4 array of bytes   b) An array containing 128 double words

c) An array containing 16 words   d) A 4x5x6 three dimensional array of words

44) Describe how you would access a single element of each of the above arrays. Provide the necessary formulae and 80x86 code to access said element (assume variable I is the index into single dimension arrays, I & J provide the index into two dimension arrays, and I, J, & K provide the index into the three dimensional array). Assume row major ordering, where appropriate.

45) Repeat question (44) using column major ordering.

46) Explain the difference between row major and column major array ordering.

47) Suppose you have a two-dimensional array of bytes whose values you want to initialize as follows:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Provide the variable declaration to accomplish this. Note: Do not use 80x86 machine instructions to initialize the array. Initialize the array in your STATIC section.

Questions (48)-(52) use these HLA declarations:

```
type

    Date:   Record
                    Month:int8;
                    Day:int8;
                    Year:int8;
            endrecord;
    Time=   record
                Hours:int8;
                Minutes:int8;
                Seconds:int8;
            endrecord;

    VideoTape:      record
                        Title:string;
                        ReleaseDate:Date;
                        Price:Real32;
                        Length: Time;
                        Rating:char;
                    endrecord;

var

    TapeLibrary : VideoTape[128];
```

48) Suppose EBX points at an object of type VideoTape. What is the instruction that properly loads the *Rating* field into AL?

49) Provide an example of a STATIC *VideoTape* variable initialized with an appropriate constant (should be legal in the HLA STATIC section).

50) This data structure (Date) suffers from the good old “Y2K” bug. Provide a correction for this problem (See Chapter Six in this volume if you have any questions about “Y2K” or the solution to this problem).

51) A one-character rating is probably not sufficient since there are ratings like “PG” and “PG-13” on movies. On the other hand, the longest rating string is five characters, so a string variable (minimum of 16 characters between the pointer and the resident string data) is probably overkill. Describe a better way to handle this using a one-byte variable/data type.

52) Provide a variable declaration for a pointer to a *VideoTape* object.

53) Provide an example of an HLA array constant containing eight *uns8* values.

54) How many bytes of memory does a 4x5 array of *dwords* require?

55) What is the difference between an array of characters and a string?

56) What is the difference between the indexes you would use to access an HLA array constant and the indexes you would use to access an array element in memory at run-time?

57) Why is the bubble sort generally a poor choice of algorithm when sorting data? Under what circumstances is the bubble sort a good choice?

58) What is a dynamically allocated array? How do you create one?

59) Explain the use of the @size compile-time function when indexing into an array at run-time.

60)

61) How does HLA store the fields of a record in memory?

62) Provide an example of a *student* record constant (see “Records” on page 483 for a description of the *student* record).

63) If you have an array of records and you want to compute an index into the array, describe how HLA can compute the size of each element of the array (i.e., the record size) for you.

64) Given the following declaration, provide the code to access element a[i].b[j] of this array:

```
type
    bType: uns32[16];
    rType:
        record
            i:int32;
            u:uns32;
            b:bType;
        endrecord;

    pType:pointer to rType

static
    a:rType[32];
```

65) Given the definitions in question (64), explain how you would access the object the *j*th element of the *b* field of the record where *p* points in assembly language ("p->b[j]" using C syntax, "p^.b[j]" for Pascal programmers).

66) Explain how to set the offset of the first field in a record. Give an example of why you would want to do this.

67) Explain how to align a particular field in a record to a given boundary.

68) Describe how to pad a record so that its size is an even multiple of some number of bytes.

69) What is the difference between a record and a union?

70) What is an anonymous union? What would you use it for?

71) What is a variant type? What data structure(s) would you use to create a variant object?

72) What is a namespace? What is its purpose?

73) What is the difference between a namespace and a record?

74) What type of declaration may not appear in a namespace?

75) What is namespace pollution? How do namespaces solve this problem?

76) What is the data structure for an HLA Standard Library DATE data type?

77) What is the data structure for an HLA Standard Library TIME data type?

78) What is the rule for computing whether a given year is a leap year?

79) What Standard Library routines would you call to get the current date and current time?

80) What is a Julian Day Number? Why are such dates interesting to us?

81) What routines does the HLA Standard Library provide to perform date arithmetic?

82) What is the difference between a random access file and a sequential file?

83) What is the difference between a binary and a text file?

84) What is the difference between a variable-length record and a fixed length record? Which would you use in a random access file? Why?

85) What is an ISAM file? What is the purpose of an ISAM list?

86) Explain how to copy one file to another using the HLA *fileio* routines.

87) How does computing the file size help you determine the number of records in a random access file?

88) What is the syntax for an HLA procedure declaration?

89) What is the syntax for an HLA procedure invocation (call)?

90) Why is it important to save the machine state in a procedure?

91) What is lexical scope?

92) What is the lifetime of a variable?

93) What types of variables use automatic storage allocation?

94) What types of variables maintain their values across a procedure call (i.e., still have the value from the last call during the current call)?

95) What is the lifetime of a global variable?

96) What is the lifetime of a local variable?

97) What are the six different ways HLA lets you pass parameters?

98) What are two different ways to declare pass by value parameters in an HLA procedure?

99) How do you declare pass by reference parameters in an HLA procedure?

100) What are the advantages and disadvantages of callee-preservation and caller-preservation when saving registers inside a procedure?

101) How do pass by value parameters work?

102) How do pass by reference parameters work?

103) How would you load the value associated with a pass by value parameter, V, into the EAX register?

104) How would you load the value associated with a pass by reference parameter, R, into the EAX register?

105) What is the difference between a function and a procedure?

106) What is the syntactical difference between a function and a procedure in HLA?

107) Where do you typically return function results?

108) What is instruction composition?

109) What is the purpose of the RETURNS option in a procedure declaration?

110) What is a side effect?

111) What is wrong with side effects in your programs?

112) What is recursion?

113) What is the FORWARD procedure option used for?

114) What is the one time that a forward procedure declaration is absolutely necessary?

115) Give an example of when it would be convenient to use a forward declaration even if it was specifically required.

116) Explain how to return from a procedure in the middle of that procedure (i.e., without “running off the end” of the procedure).

117) What does the #INCLUDE directive do?

118) What do you normally use the #INCLUDE directive for?

119) What is the difference between the #INCLUDE and the #INCLUDEONCE directives?

120) What is the syntax for an HLA unit?

121) What is the purpose of an HLA unit?

122) What is the purpose of separate compilation?

123) How do you declare EXTERNAL procedures? Variables?

124) What variable types cannot be EXTERNAL?

125) How do you declare a public symbol in HLA?

126) What is a header file?

127) What is a file dependency?

128) What is the basic syntax for a makefile?

129) What is a recursive file inclusion? How can you prevent this?

130) What are the benefits of code reuse?

131) What is a library?

132) Why would you not want to compile all your library routines into a single OBJ file?

133) What types of files are merged together to form a .LIB file?

134) What program would you use to create a library file?

135) How does a library contribute to name space pollution? How can you solve this problem?

136) What are the differences between the INTMUL and the IMUL instruction?

137) What must you do before executing the DIV and IDIV instructions?

138) How do you compute the remainder after the division of two operands?

139) Assume that VAR1 and VAR2 are 32 bit variables declared with the DWORD type. Write code sequences that will compute the boolean result (true/false) of each of these, leaving the result in BL:

a) VAR1 = VAR2

b) VAR1 <> VAR2

c) VAR1 < VAR2 (Unsigned and signed versions

d) VAR1 <= VAR2 for each of these)

e) VAR1 > VAR2

f) VAR1 >= VAR2

140) Provide the code sequence that will compute the result of the following expressions. Assume all variables are UNS32 values and you are computing unsigned results:

a) A = (X-2) * (Y + 3);

b) B = (A+2)/(A - B);

c) C = A/X * B - C - D;

d) D = (A + B)/12 * D;

e) E = (X * Y) / (X-Y) / C+D;

f) F = (A <= B) && (C != D) || (E > F);

(note to Pascal users, the above is “F := (A <= B) and (C <> D) or (E > F);”)

141) Repeat question (140) assuming all variables are INT32 objects and you are using signed integer arithmetic.

142) Repeat question (140) assuming all variables are REAL64 objects and you are using floating point arithmetic. Also assume that an acceptable error value for floating point comparisons is 1.0e-100.

143) Convert the following expressions into assembly language code employing shifts, additions, and subtractions in place of the multiplication:

a) EAX*15

b) EAX*129

c) EAX*1024

d) EAX*20000

144) How could you use the TEST instruction (or a sequence of TEST instructions) to see if bits zero and four in the AL register are both set to one? How would the TEST instruction be used to see if either bit is set? How could the TEST instruction be used to see if neither bit is set?

145) Why are commutative operators easier to work with when converting expressions to assembly language (as opposed to non-commutative operators)?

146) Provide a single LEA instruction that will multiply the value in EAX by five.

147) What happens if INTMUL produces a result that is too large for the destination register?

148) What happens if IMUL or MUL produces a result that is too large for EAX?

149) Besides divide by zero, what is the other division error that can occur when executing IDIV or DIV?

150) What is the difference between the MOD instruction and the DIV instruction?

151) After a CMP instruction, how do the condition code bits indicate that one signed operand is less than another signed operand?

152) What instruction is CMP most similar to?

153) What instruction is TEST most similar to?

154) How can you compute a pseudo-random number between 1 and 10 using the HLA Standard Library rand.hhf module?

155) Explain how to copy the floating point status register bits into the 80x86 FLAGs register so you can use the SETcc instructions after a floating point comparison.

156) Why is it not a good idea to compare two floating point values for equality?

157) Explain how to activate floating point exceptions on the FPU.

158) What are the purpose of the rounding control bits in the FPU control register?

159) Besides where they leave their results, what is the difference between the FIST instruction and the FRNDINT instruction?

160) Where does *stdin.getf()* leave the input value?

161) What is a mantissa?

162) Why does the IEEE floating point format mantissa represent values between 1.0 and 2.0?

163) How can you control the precision of floating point calculations on the FPU? Provide an example that shows how to set the precision to real32.

164) When performing floating point comparisons, you use unsigned comparisons rather than signed comparisons, even though floating point values are signed. Explain.

165) What is postfix notation?

166) Convert the expression in question (140) to postfix notation.

167) Why is postfix notation convenient when working with the FPU?

168) Explain how the XLAT instruction operates.

169) Suppose you had a *cipher* (code) that swaps various characters in the alphabet. Explain how you could use the XLAT instruction to decode a message encoded with this cipher. Explain how you could encode the message.

170) What is the purpose of *domain conditioning*?

171) What is the maximum set of values for the domain and range when using the XLAT instruction?

172) Explain the benefits of using one program to generate the lookup tables for another program.

13.2 Programming Projects

1) 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) Write a procedure that reads an unsigned integer value from the user and reprompts for the input if there is any type of error. The procedure should return the result in the EAX register. It shouldn't require any parameters. Compile this procedure in a UNIT. Provide an appropriate header file and a main program that will test the function. Include a makefile that will compile and run the program.

3) Extend the previous project (2) by writing a second routine for the UNIT that has two parameters: a minimum value and a maximum value. Not only should this routine handle all input problems, but it should also require that the input value fall between the two values passed as parameters.

4) Extend the unit in (3) by writing two addition procedures that provide the same facilities for signed integer values.

5) Write a program that reads a filename from the user and then counts the number of lines in the text file specified by the filename.

6) Write a program that reads a filename from the user and then counts the number of words in the specified text file. The program should display an accurate count of words. For our purposes, a "word" is defined as any sequence of non-whitespace characters. A whitespace character is the space (#$20), tab (#$9), carriage return (#$d), and the line feed (#$a). Read the text file a line at a time using fileio.gets or fileio.a_gets. Also remember that a zero byte (#0) marks the end of a string, so this clearly terminates a word as well.

7) Modify the CD database program in Laboratory Exercise 13.3.10 so that it reads the CD database from a text file. Your program should make two passes through the text file. On the first pass it should count the number of lines in the text file and divide this value by four (four lines per record) to determine how many records are in the database. Then it should dynamically allocate an array of CDs of the specified size. On the second pass, it should read the data from the text file into dynamically allocated array. Obviously, you must also modify the program so that it works with a variable number of CDs in the list rather than the fixed number currently specified by the NumCDs constant.

8) Modify program (7) to give the user the option of displaying all the records in the database sorted by title, artist, publisher, or date. Use the QuickSort algorithm appearing in this volume to do the sorting (note: the solution requiring the least amount of thought will require four different sort routines; a more intelligent version can get by with two versions of quicksort).

9) Write a "CD Database Input" program that reads the textfile database for program (7) into a fixed array in memory. Allow enough space for up to 500 records in the database (display an error and stop if the input exceeds this amount). Of course, the value 500 should be a symbolic constant that can easily be changed if you need a larger or smaller database. After reading the records into memory, the program should prompt the user to enter additional CDs for the database. When the user is through entering data, the program should write all the records back to the database file (including the new records).

10) Modify program (9) so that the user is given the option of deleting existing records (perhaps by specifying the index into the array) in addition to entering new entries into the database.

11) Write a MsgBoxInput procedure that has the following parameters:

```
procedure MsgBoxInput( prompt:string; row:word; col:word; result:string );
```

The procedure should do the following:

1) Save the rectangular region of the screen specified by (row, col) as the upper left hand corner and (row+4, col+length(prompt) +2) as the lower right hand corner. Stop the program with an error if this rectangle is outside the bounds (0,0) <-> (24,79). Use the console.a_getRect function to save this portion of the screen.

2) Fill the rectangular region saved above with spaces and a dark blue background. Set the foreground attribute to yellow. (console.fillRect).

3) Print the prompt message starting at position (row+1, col+1).

4) Position the cursor at (row+2, col+1).

5) Read a string from the user with no more than length(prompt) characters (see below).

7) Restore the attributes in the rectangular region to black and white. (console.fillRectAttr)

8) Redraw the text saved in step one above.

Note that you cannot use *stdin.gets* or *stdin.a_gets* to read the string from the user since these functions won't limit the number of input characters. Instead, you will have to write your own input routine using *stdin.getc* to read the data a character at a time. Don't forget to properly handle the backspace character. Also, *stdin.getc* does not echo the character, so you will have to handle this yourself.

Put these procedures in a unit and write a companion main program that you can use to test your dialog box input routine.

12) The Windows console device is a *memory mapped I/O device*. That is, the display adapter maps each character on the text display to a character in memory. The display is an 80x25 array of characters declared as follows:

```
display:char[25,80];
```

Display[0,0] corresponds to the upper left hand corner of the screen, display[0,79] is the upper right hand corner, display[24,0] is the lower left hand corner, and display[24,79] is the lower right hand corner of the display. Each array element contains the ASCII code of the character to appear on the screen.

The lab4_10_7.hla program demonstrates how to copy a matrix to a portion of the screen (or to the entire console, if you so desire). Currently, this program draws and erases a single asterisk over and over again on the screen to give the illusion of animation. Modify this program so that it draws a rectangular or triangular shape, based at the row/column address specified by the for-loops. Be sure you don't "draw" outside the bounds of the character array (*playingField*).

13) Create a program with a single dimension array of records. Place at least four fields (your choice) in the record. Write a code segment to access element *i* (*i* being a DWORD variable) in the array.

14) Modify the program above so that it contains two identical arrays of records. Have the program copy the data from the first array to the second array and then print the values in the second array to verify the copy operation.

15) Modify the program in (14) above to use a dynamically allocated array, rather than a statically allocated array, as the destination array.

16) Write a program which copies the data from a 4x4 array and stores the data into a second 4x4 array. For the first 4x4 array, store the data in row major order. For the second 4x4 array, store the data in column major order. Write the for loops to do this "transposition" yourself. Do not use the *array.transpose* routine in the HLA Standard library.

17) Modify program (16) so that a single constant controls the size of the matrices (they will always be square, having the same number of rows as columns, so a single constant will suffice).

18) Write a program that reads two lines of text from the user and displays all characters that appear on both lines, but does not display any characters that appear on only one of the two lines of text. (hint: use the character set functions.)

19) Modify program (18) so that it prints the characters common to both lines, the characters appearing only in the first string, and the characters appearing only in the second string. Print these three sets of characters on different lines with an appropriate message explaining their content.

20) Write a program that reads a line of text from the user and counts the number of occurrences of each character on the line. The program should display the character and count for each unique character found in the input line whose count is not zero (hint: create an array of 256 integers and use the ASCII code as an index into this array).

21) Modify program (20) so that it also displays the number of numeric characters, the number of alphabetic characters (upper and lower case), the number of uppercase alphabetic characters, the number of lower case alphabetic characters, the number of control characters (ASCII code <= $1f and $7f), and the number of punctuation characters (this is all characters other than the previously mentioned sets). Display these counts one per line before the output of the count for each character from problem (20).

22) Write a program that reads a line of text from the user and counts the number of words on that line. For our purposes, a “word” is defined as any sequence of non-whitespace characters. A whitespace character is the space (#$20), tab (#$9), carriage return (#$d), and the line feed (#$a). Do not use the *str.tokenize* or *str.tokenize2* routines to implement this program.

23) Look up the *str.tokenize2* routine in the HLA Standard Library strings module. Implement problem (22) using this procedure.

24) Write a program that reads a string from the user and checks the string to see if it takes the form “mm/dd/yy”, or “mm-dd-yy” where *mm*, *dd*, and *yy* are all exactly two decimal digits. If the input string matches either of these patterns, extract these substrings into separate string variables and print them on separate lines with an appropriate description (i.e., month, day, and year).

25) Modify program (24) to verify that *mm* is in the range 01-12, *dd* is in the range 01-31, and *yy* is in the range 00-99. Report match/no match on the input depending upon whether the input string matches this format. Note: you will probably find this problem to be a whole lot less work if you look up the *conv.strTou8* procedure in the conversions module of the HLA Standard Library and do numeric comparisons rather than string comparisons).

26) Modify program (25) to allow single digit values for the month and day components when they fall in the range 1-10 (the year component, *yy*, still requires exactly two digits). Also allow leading or trailing whitespace in the string (hint: use the char.ispace function or the str.trim function). Display match/no match as before depending upon the validity of the string.

27) Modify program (26) so that it checks the validity of the date subject to the Gregorian Calendar. That is, the day value should not exceed the maximum number of days for each month. Note: you may use the date.IsLeapYear function but you may not use date.IsValid or date.validate for this project. Assume that the centuries portion of the date is in the range 2000..2099 (that is, add 2000 to the year value to obtain the correct year).

28) The Y2K problem): Modify program (27) as follows: (1) it reads a string from the user. (2) it searches for the pattern “mm/dd/yy” or “mm-dd-yy” anywhere in the string (there may be other text in the string in this program). If the program finds a string of the form “mm/dd/yy” or “mm-dd-yy” then it will replace that string with a string of the form “mm/dd/19yy” or “mm-dd-19yy” to make the string Y2K compatible (of course, a good programmer would use a string constant for “19” so it could be easily changed to “20” or any other two digits). If there is more than one date in the string that matches a valid Y2K date, you will need to find and convert that date as well (i.e., convert all possible dates found in the string). After the conversion process is complete, display the string which should be the original text typed by the user with these Y2K modifications. Do not translate any dates that don’t exactly match one of these two patterns:

Examples of strings containing good dates (ones you should translate):

```
01-01-01
02-29-96
End of the 1900s: 12/31/99
Today is '3/17/90'
Start of the 20th century: 1/1/01, end of the 20th century: 12/31/2000
(note: program must not convert 12/31/2000 above to 12/31/192000)
Yesterday was 7/4/76, tomorrow is 7/6/96.
(must convert both dates above.)
```

Examples of bad dates that your program must ignore:

```
01-01/01
02/29-96
End of the 1900s: 12/31/1999
Today is '123/17/90'
01/32/00
02/29/99
```

29) Write a program that reads a string of characters from the user and verifies that these characters are all decimal digits in the range '0'..'7'. Treat this string as an octal (base eight) number. Convert the string to an integer value. Note: values in the range 0..7 consume exactly three bits. You can easily construct the octal number using the SHL instruction; you will not need to use the INTMUL instruction for this assignment. Write a loop that reads the octal string from the user and, if the string is valid, converts the number to an integer and displays that integer value in decimal.

30) Modify the student database program in the laboratory exercises (see "Records, Arrays, and Pointers Laboratory Exercise" on page 698) to sort the students by their name prior to printing the result. Since the name field is a string, you will want to use the string comparison routines to compare them. You should also use the case insensitive string comparisons (e.g., *str.ile*) since upper and lower case are irrelevant to most people when viewing sorted lists of names.

31) Modify program (30) so that it uses two fields for the name (last name and first name). The user should still enter the full name. Under program control, extract the first and last names from the single input string and store these two name components into their respective fields in the record.

32) Write a program that computes the two solutions to the quadratic equation.

$$x_1 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

$$x_2 = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

The program should verify that a is non-zero and that the value $b^2\text{-}4ac$ is positive or zero before attempting to compute these roots.

33) Suppose you are given a line and two points on that line. Compute the point that is midway between the two points via the equations $X_{mid} = (x_1 + x_2)/2$ and $Y_{mid} = (y_1 + y_2)/2$. Read the X and Y coordinates for the two points from the user. Use a RECORD containing two fields, X and Y, to denote a single point in your program. Note: X and Y are real64 variables.

34) Write a program that computes the amount of interest earned on a savings account. The formula for computing interest is

$$DollarsEarned = InitialDeposit \times \left(1.0 + \frac{InterestRate}{100.0}\right)^{Years}$$

Hint: you will need to use routines from the “math.hhf” library module for this assignment.

35) Write a program that inputs a temperature in Celsius and converts it to degrees Fahrenheit. The translation is given by the following formula:

$$f = (9/5) \times c + 32$$

36) Solve the equation above (35) for c and write a program that also solves for Celsius given a temperature in Fahrenheit.

37) 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

Display the GPA using the format X.XX.

38) Modify program (37) to handle courses where each course may have a different (integral) number of units. Multiply the grade points by the number of units for a given grade and then divide by the total number of units.

39) Write a program that accepts a dollars and cents amount as a floating point value and converts this to an integer representing the number of pennies in the entered amount (round the input to the nearest penny if

it is off a little bit). Translate this value into the minimum number of pennies, nickels, dimes, quarters, one dollar bills, five dollar bills, ten dollar bills, twenty dollar bills, and one hundred dollar bills that would be necessary to represent this result. Display your answer as follows:

```
The amount $xxx.xx requires no more than
       p pennies,
       n nickels,
       d dimes,
       q quarters,
       d $1 bills,
       f $5 bills,
       t $10 bills,
and    h $100 bills.
```

The italicized letters represent small integer constants (except *h* which can be a large integer constant). Do not display an entry if the count is zero (e.g., if there are zero $100 bills required, do not display the line "0 $100 bills.")

40) Modify program (39) so that it displays a singular noun if there is only one of a given item (e.g., display "1 nickel," rather than "1 nickels,")

41) Write a program that plots a sine curve on the console (see "Bonus Section: The HLA Standard Library CONSOLE Module" on page 192 for a discussion of the console support routines). Begin by clearing the screen with console.cls(). Then draw a row of dashes along line 12 on the screen to form the X-axis. Draw a column of vertical bars in column 0 to denote the Y-axis. Finally, plot the sine curve by using console.gotoxy to position the cursor prior to printing an asterisk ("*") to represent one point on the screen. Plot the curve between 0 and 4π radians. Each column on the screen should correspond to 4π/80 radians. Since the FSIN instruction only returns values between -1 and +1, you will need to scale the result by adding one to the value and multiplying it by 12. This will give the Y-coordinate of the point to plot. Increment through each of the X-coordinate positions on the screen when plotting the points (remember, each X-coordinate is 4π/80 radians greater than the previous X-coordinate).

42) Modify program (41) to simultaneously plot the COS curve at the same time you plot the SIN curve. Use at signs ("@") to plot the cosine points.

43) Write a program that accepts an integer value as a decimal (base 10) number and outputs the value in a user specified base between two and ten.

44) Modify program (43) so that it accepts an integer in one user-specified base and outputs the number in a second user-specified base (bases 2-10).

45) The factorial of a non-negative integer, *n!*, is the product of all integers between one and *n*. For example, 3! is 3*2*1 = 6. Write a function that computes n! using integer arithmetic. What is the largest value of *n* for which you can compute n! without overflow using 32-bit integers?

46) Since *n!* overflows so rapidly when using 32-bit integers, use the 64-bit integer capabilities of the FPU to compute *n!* (see problem 45). What is the maximum value of *n* for which you can compute n! when using 64-bit numbers?

47) You can estimate the value of the constant e using the following equation:

$$e = \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

Obviously, you cannot carry this out to an infinite number of terms, but the more terms you use the better the approximation. Write a program that will calculate *e* using this approximation that is accurate to at least 12 decimal places (e=2.71828182846).

48) You can compute the value of e^x using the following mathematical series:

$$e^x = \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Using the same number of terms required by problem (47), compute e^x with 12 digits of accuracy.

49) Write a program that reads the data from a text file and swaps the case of the alphabetic letters and writes the result to the console. Use a lookup table to perform the case swap (hint: it's probably easiest to have the program fill in the table using the first few instructions of the program rather than by typing the table by hand or write a separate program to generate the table). The program should prompt the user for the file name to translate.

50) Genokey Technologies manufactures portable keypads for laptop computers. They have boxes that will hold one, five, ten, twenty-five, and one hundred keypads. Write a program for the Genokey shipping clerk that will automatically choose the minimum number of boxes (and tell how many of each is required) that are necessary to ship *n* keypads to a distributor whenever that distributor orders *n* keypads.

51) Write a program that reads two eight-bit unsigned integer values from the user, zero-extends these integers to 16-bits, and then computes and prints their product as follows:

```
   123
x  241
------
   123
  492
 246
------
 29643
```

52) Extend the RPNcalculator program (found in the sample program section of the chapter on Real Arithmetic, see "Sample Program" on page 640) to add all the function which there are FPU instructions (e.g., COS, CHS, ABS, FPREM1, etc.).

53) Extend the calculator program above (52) to support the functions provided in the HLA "math.hhf" module.

54) Modify the RPN calculator program above (53) to support infix notation rather than postfix notation. For this assignment, remove the unary functions and operators.

13.3 Laboratory Exercises

Note: since this volume is rather long you should allow about twice as much time as normal to complete the following 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 *Volume2* and *Volume3*, with the names obviously corresponding to the volumes in this textbook. All the source code to the example programs in this volume can be found in the subdirectories found in the Volume3 subdirectory. The Volume3\Ch13 subdirectory also contains the programs for this chapter's laboratory exercises. Please see this directory for more details.

13.3.1 Using the BOUND Instruction to Check Array Indices

The following program (Program 13.1) demonstrates how to use the BOUND instruction to check array indices at run time. This simple program reads an unsigned integer from the user and uses that value as an index into an array containing 10 characters then displays the character at the specified index. However, if the array index is out of bounds, this triggers an *ex.BoundInstr* exception and prints an error message (this program also handles a couple of other exceptions that *stdin.get* can raise).

```
// Program to demonstrate BOUND instruction.

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

static

    index:          uns32;
    arrayBounds:    dword[2] := [ 0, 9 ];
    arrayOfChars:   char[ 10 ] :=
                        ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' ];

    answer:         char;

begin BoundLab;

    repeat

        try

            // Read an integer index into "arrayOfChars" from the user:

            stdout.put( "Enter an integer index between 0 and 9: " );
            stdin.flushInput();
            stdin.get( index );

            // Verify that the input is in the range 0..9:

            mov( index, eax );
            bound( eax, arrayBounds );

            // If it was a good index, display the character at that
            // index:

            mov( arrayOfChars[ eax ], al );
            stdout.put
```

```
                (
                    “The character at that index is ‘”,
                    (type char al),
                    “‘”
                    nl
                );


            // Handle the exceptions possible in the stdin.get call as well
            // as the BOUND instruction exception.

            exception( ex.BoundInstr )

                stdout.put( “The index you entered is not in the range 0..9” nl );

            exception( ex.ConversionError )

                stdout.put( “You did not enter a valid unsigned integer” nl );

            exception( ex.ValueOutOfRange )

                stdout.put( “That integer value is way out of range!” nl );

            endtry;


            // Ask the user if they want to try again.  Force the user to
            // enter only a “Y” or an “N”:
            repeat

                stdout.put( “Do you wish to try another index (Y/N)? “ );
                stdin.flushInput();
                stdin.getc();

                // If the user enters a lower case character, convert it
                // to upper case.

                if( al in ‘a’..’z’ ) then

                    and( $5f, al );

                endif;
                mov( al, bl );  // Because cs.member wipes out AL.

            until( cs.member( al, {‘Y’, ‘N’} ));

        until( bl = ‘N’ );


    end BoundLab;
```

Program 13.1 Using the BOUND Instruction to Check Array Indicies

Exercise A: Run this program and verify that it works in an acceptable fashion when the input is a legal integer in the range 0..9. Next, try entering a value greater than 10. Report the results of these experiments in your lab report.

Exercise B: At the end of the program, it asks you whether you want to rerun the program. The program uses the cs.member function to allow only a yes or no (“Y” or “N”) response. Verify that this program will not accept any data other than “Y” or “N” when asking the user if they want to retry the operation.

Exercise C: As you can see, there is code that converts the character in the AL register to an upper case character if it is a lower case character. After the conversion, the character set membership test only checks for upper case characters. Verify that you can enter upper or lower case “Y” or “N” characters and the program still works.

Exercise D: The HLA Standard Library includes the CHARS module that supplies various character classification routines including functions like *chars.isLower* and *chars.isUpper.* These functions take a single character parameter and return true or false in the EAX register if the parameter is a lower case alphabetic or upper case alphabetic character (respectively). Modify the code in this sample program to test for lower case by making a call to the *chars.isLower* routine rather than using the boolean expression “(al in ‘a’..’z’)”. Rerun the program and verify that it is still working correctly.[1] Include a copy of this converted program with your lab report.

Exercise E: In addition to character classification routines, the CHARS module also includes a couple of character conversion routines. The *chars.toUpper* routine will convert its single character parameter to upper case if it is lower case (it returns all other characters unchanged). Likewise, the *chars.toLower* routine converts the input parameter to a lower case character (if it was upper case to begin with). Both routines return the character in the AL register. They both return the original character if it was not alphabetic. Modify this program to use the *chars.toUpper* routine to do the upper case conversion rather than the IF statement it currently uses. Test the resulting program. Include a copy of this converted program with your lab report.

Exercise F: HLA supports a special, extended, syntax for the BOUND instruction that has three parameters. The second and third parameters of BOUND are integer constants specifying the lower and upper bounds to check. This form is often more convenient than the BOUND instruction appearing in Program 13.1 because you don’t have to declare a two-element initialized array for use as the BOUND parameter. Modify your current program by eliminating the *arrayBounds* variable and using this form of the BOUND instruction. Verify that your program still allows indices in the range 0..9. Include the program listing of your modified program in your lab report.

Exercise G: Program 13.1 uses the literal constants nine and ten to denote the upper bounds on the array as well as the number of elements in the array. Using literal constants like this makes programs much more difficult to read and maintain. A better solution is to use a symbolic constant that you define once at the top of the program and reference throughout the code. By doing so, you can change the size of the array by changing only one or two statements in your program[2]. Modify this program so that it contains a single constant specifying the number of array elements. Use this constant wherever the number of array elements or the array’s upper bound appears in the program (hint: use constant expressions to compute the value nine from this constant). Verify that the program still works correctly and include the source code with your lab report.

Exercise H: HLA predefines a special boolean VAL constant, @bound, that controls the compilation of the BOUND instruction. If @bound is true (the default) then HLA will compile BOUND instructions found in your program. If @bound is false, then HLA ignores all BOUND instructions it encounters. You can change the value of the @bound constant by using a statement of the form:

```
?@bound := false;
```

Modify Program 13.1 (the original version) and set the *arrayBounds* value so that it only allows array indices in the range 0..5. Run the program and verify this new operation. Next, insert “?@bound:=false;” immediately after the “begin pgm4_17;” statement in the program. Recompile and run the program and ver-

1. Note that using the IN operator is much more efficient than calling the *chars.isLower* function. Don’t get the impression from this exercise that *chars.isLower* is the best way to check for a lower case character. The *chars.isLower* routine is appropriate in many cases, but this probably isn’t a good example of where you would want to use this function. The only reason this exercise uses this function is to introduce it to you.
2. Don’t forget, if you change the size of the character array in Program 13.1 then you will need to change the number of array elements in the initializer for that array as well.

ify that the BOUND instruction is no longer active by entering an index in the range 6..9. Describe the results in your laboratory report.

13.3.2 Using TEXT Constants in Your Programs

HLA's TEXT constants let you replace long repetitive sequences in your program with a single identifier, similar to the #define statement in C/C++[3]. This laboratory exercise demonstrates how you can use TEXT constants to save considerable typing in your programs.

Consider Program 13.2. This program uses TEXT constants to compress often-used text in three main areas: first, it reduces the strings "stdout.put" and "stdin.get" to the single identifiers *put* and *get* saving a bit of typing (since these calls occur frequently in HLA programs). Second, it replaces the string "(type uns32 i)" with the single identifier *ui*. Although this program only uses *ui* once, in a real program you might wind up using a single string like "(type uns32 i)" on several occasions. Finally, this program uses TEXT constants to combine several common exceptions into the *endOfTry* constant, saving considerable typing in each TRY..ENDTRY block.

```
// Demonstration of TEXT constants in a program.

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

const
    put: text := “stdout.put”;
    get: text := “stdin.get”;
    ui:  text := “(type uns32 i)”;

    VOR: string := “Value out of range”;
    CE:  string := “Conversion error”;

    exRange: text :=
        “exception( ex.ValueOutOfRange ); “
            “put( VOR, nl );”;

    exConv:  text :=
        “exception( ex.ConversionError ); “
            “put( CE, nl );”;

    endOfTry:text :=
        “exRange; exConv; endtry”;



static

    i:      int32;
    range:  dword[2] := [1,10];

begin TextLab;

    try

        put( “Enter an integer value: “ );
```

3. HLA's TEXT constants don't support parameters making them weaker than C/C++'s #define macro capability. Fear not, however, HLA has it's own macro facility that is much more powerful than C/C++'s. You will learn about HLA's macro facilities in the chapter on macros and the compile-time language.

```
            get( i );
            put( “The value you entered was “, i, nl );

        endOfTry;


        try

            repeat

                put( “Now enter a negative integer: “ );
                get( i );

            until( i < 0 );
            put( “As an unsigned integer, ‘i’ is “, ui, nl );

        endOfTry;


        try

            repeat

                put( “Now enter an integer between one and ten: “ );
                get( i );
                mov( i, eax );
                bound( eax, range );

            until( i < 0 );
            put( “The value you entered was “, i, nl );

          exception( ex.BoundInstr );

            put( “The value was not in the range 1..10”, nl );

        endOfTry;


    end TextLab;
```

Program 13.2   TEXT Constant Laboratory Exercise Code

Exercise A: Compile and run this program. For each TRY..ENDTRY block, enter values that raise the conversion error exception and the value out of range exception[4]. For the last TRY..ENDTRY block, enter a value outside the range 1..10 to raise an *ex.BoundsInstr* exception. Describe what happens in your lab report.

Exercise B: Add a TEXT constant declaration for the *exBound* symbol. Set the value of the TEXT object equal to the string associated with the BOUND instruction exception (see the examples for *ex.ConversionError* and *ex.ValueOutOfRange* to figure out how to do this). Modify the last TRY..ENDTRY statement to use this TEXT constant in place of the associated exception handling section. Run the program to verify its operation. Include the modified program with your laboratory report.

4. Entering a 12-digit integer value will raise the *ex.ValueOutOfRange* exception. Typing non-numeric characters will raise the *ex.ConversionError* exception.

Exercise C: Modify the first TRY..ENDTRY block above to read and display an unsigned integer value using the *ui* TEXT constant in place of the *i* variable. Verify, by running the program, that it will no longer let you enter negative values. Include the modified program with your lab report.

Exercise D: In the *exRange* and *exConv* constants, the *put* statement uses the symbolic string constants *CE* and *VOR* to display the error message. Replace these symbolic identifiers with their corresponding literal string constants (Hint: you need to remember how to embed quotation marks inside a string). Run and test the program. Include the source code to this modification in your lab report and comment on the wisdom of using string constants rather than string literals in your programs.

13.3.3 Constant Expressions Lab Exercise

This laboratory exercise demonstrates the use of constant expressions in an HLA program. As you are reading through the following program, keep in mind that many of the constant expressions were added to this program simply to demonstrate various operators; you wouldn't normally use these operators in a trivial program such as this one. However, to write a complex program that fully uses constant expressions is beyond the scope of this laboratory exercise. Part of this laboratory exercise is to undo some of the excessive use of constant expressions in the program. So bear with this example.

```
// Demonstration of constant expressions in a program.

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

const
    ElementsInArray := 10;

    sin:    string  := “stdin.”;
    sout:   string  := “stdout.”;

    put:    text    := sout + “put”;
    get:    text    := sin  + “get”;
    flush:  text    := sin  + “flushInput”;


var
    input:          uns32;
    GoodInput:      boolean;
    InputValues:    int32[ ElementsInArray ];

readonly

    IV_bounds:  dword[2] := [0, ElementsInArray-1 ];


begin ConstExprLab;

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

        repeat

            mov( false, GoodInput );
            try

                put( “Enter integer #”, (type uns32 ebx), “: “ );
                flush();
                get( input );
                bound( ebx, IV_bounds );
                mov( input, InputValues[ ebx*4 ] );
                mov( true, GoodInput );

              exception( ex.ConversionError )

                put( “Number contains illegal characters”, nl );

              exception( ex.ValueOutOfRange )

                put( “The number is way too large”, nl );

              exception( ex.BoundInstr )

                put( “Array index is out of bounds”, nl );

            endtry;

        until( GoodInput );

    endfor;

    // Now print the values entered by the user in the reverse order
    // they were entered. Note: this code doesn’t use the scaled (*4)
    // indexed addressing mode.  So it initializes EBX with four times
    // the index of the last element in the array.

    mov( ( ElementsInArray - 1 ) * 4, ebx );
    while( (type int32 ebx) >= 0 ) do

        put
        (
            “Value at offset “,
            (type uns32 ebx):2,
            “ was “,
            InputValues[ebx],
            nl
        );

        // Since we’re not using the scaled (*4) index addressing mode,
        // drop the count by four for each array element since it’s an
        // array of dwords.

        sub( 4, ebx );

    endwhile;

end ConstExprLab;
```

Program 13.3 Constant Expressions Laboratory Exercise

Exercise A: Compile this program and run it. Describe the operation of the program in your laboratory report.

Exercise B: One extreme use of constant expressions in this program appears in the following code:

```
sin:    string  := "stdin.";
sout:   string  := "stdout.";
```

```
    put:    text    := sout + "put";
    get:    text    := sin  + "get";
    flush:  text    := sin  + "FlushInput";
```

Modify the program to eliminate the *sin* and *sout* constants from the code. The program should be easier to read once you have completed this task. Compile and run the program and verify that it produces the same results as before the change. Include the source code to the modified program with your lab report. Briefly explain why you think that the use of the string concatenation operator wasn't worth much in this code.

Exercise C: This program provides the *ElementsInArray* constant that lets you easily change the size of the array that this program works with. Change the value of this constant to five and recompile and run the program. Verify that it works correctly for five array elements (versus ten in the previous version). Identify all locations in the program that modifying the *ElementsInArray* constant affects (use a hi-liter or other such tool to make these points visible in the listing appearing in your lab report). Describe in the lab report write-up the effort that would be needed to change the size of the array had this program not used the *ElementsInArray* constant.

Exercise D: The *get* and *put* constants save considerable typing in this program. Can you think of a reason why it might not be such a good idea to use TEXT constants like these to save typing in your program? Justify your answer in your laboratory report.

13.3.4 Pointers and Pointer Constants Exercises

Pointers are especially important in assembly language programs. This laboratory exercise demonstrates that HLA programs often use pointers to access elements of an array. This program also demonstrates how to use the BOUND instruction to check to see if a pointer contains an address between two extremes.

Conceptually, this program is very simple. It asks the user for an integer and then prints that many characters out of an array (checking, of course, to verify that the program doesn't exceed array bounds). To print those characters, this program loads the base address of an array into a register and then increments that register to step through the array. Contrast this with examples appearing later in these lab exercises where the program leaves the register containing the base address unmodified and increments a separate index register.

```
// Demonstration of pointers and pointer constants in a program.

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

const
    NumberOfChars := 8;

type
    CharPtr: pointer to char;

static

    CharArray: char[ NumberOfChars ] :=
                    [‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’ ];

    CA_bounds: CharPtr[2] := [&CharArray, &CharArray + (NumberOfChars-1)];


begin PointerLab;

    // Get the number of characters to print into EAX:
```

```
    stdout.put( “How many characters would you like to print? “ );
    stdin.getu32();

    // Get address of CharArray into EBX:

    lea( ebx, CharArray );

    // Print the specified number of characters:

    while( eax > 0 ) do

        bound( ebx, CA_bounds );
        stdout.putc( [ebx] );
        inc( ebx );
        dec( eax );

    endwhile;
    stdout.newln();

end PointerLab;
```

Program 13.4 Pointers and Pointer Constants Laboratory Exercise Code

Exercise A: Compile and run this program. Describe its operation in your laboratory report.

Exercise B: This program contains a constant *NumberOfChars*. Change this constant from eight to ten and recompile the program. Explain why this one change is not sufficient to change the size of the array from eight to ten elements. Make an appropriate change to this program so that it will compile and run correctly. Compare the number of changes needed in the program to get it to compile and run with the number of changes that would be necessary had this program not used the *NumberOfChars* constant. Describe the benefits to this program of using the *NumberOfChars* constant.

Exercise C: This program does not contain a TRY..ENDTRY block to handle the BOUND instruction exception. Add such a statement to the program that will print a nice error message. If the exception occurs, the program should quit.

13.3.5 String Exercises

In this laboratory exercise you will work with statically and dynamically allocated strings. The first new feature found in this program is the *str.strvar* declaration. The *str.strvar* data type lets you statically allocate storage for a string variable in the STATIC section of your program[5]. This data type takes a single parameter that specifies the maximum number of legal characters in the string variable. This data type sets aside the appropriate amount of storage for the string, initializes the maximum length field to the value passed as a parameter, and then initializes the string variable (*s1* in the example below) with the address of this newly allocated string. The *str.strvar* data type is very useful for declaring small strings in your STATIC section because it provides both declaration and allocation in a single statement.

This program uses *str.strvar* to allocate storage for a ten character string. When the program starts running, it requests that the user enter a string longer than ten characters to demonstrate the *ex.StringOverflow* exception. Finally, this program concludes by demonstrating how to access the individual characters in a string.

5. Note that *str.strvar* is legal *only* in the STATIC section. You can not use it in any of the other HLA variable declaration sections.

```
// Demonstration of strings in a program.

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

static

    s1: str.strvar( 10 );
    s2: string;

begin StringLab;

    try

        stdout.put( “Enter a string longer than 10 characters: “ );
        stdin.gets( s1 );
        str.length( s1 );
        stdout.put
        (
            “The string you entered was only “,
            (type uns32 eax ),
            “ characters long, “
            nl
            “try a longer string next time”
            nl
        );

      exception( ex.StringOverflow )

        stdout.put( “stdin.gets raised the ex.StringOverflow exception” nl );

    endtry;


    // Use stdin.a_gets to read a string and allocate storage for it.
    // Print the characters in the string one at a time after reading
    // the string.
    //
    // There is something wrong with this code, can you fix it?

    stdout.put( nl “Enter a string of any length: “ );
    stdin.a_gets();
    mov( eax, s2 );
    mov( eax, ebx );
    str.length( eax );
    stdout.put
    (
        “The string length was “,
        (type uns32 eax),
        nl,
        “The string you entered was”,
        nl
    );

    while( eax > 0 ) do

        stdout.put( (type char [ebx]), nl );
        dec( eax );
        inc( ebx );
    endwhile;
    stdout.newln();

end StringLab;
```

Program 13.5 String Exercises Code

Exercise A: Compile this program and run it. Be sure to follow all directions provided in the program. Describe what the program does in your lab report.

Exercise B: There is a defect in the way this program handles dynamically allocated strings. Locate the defect and correct it. Include a copy of the corrected program with your lab report. Note that this program runs correctly; you will have to analyze the source code to find the defect.

13.3.6 String and Character Set Exercises

In this laboratory exercise you'll be working with a program that disassembles an input string. The program begins by reading the user's full name (first and last name). It then extracts the first and last names into separate strings. Although this is a relatively simple program, it demonstrates a very important concept in string manipulation. Most complex programs that manipulate strings need to break those strings down into their constituent parts. The techniques this program uses are the basis for much larger and more complex programs.

This program also demonstrates the use of character sets and character set expressions. In particular, note the following constant declaration in the program:

```
NonAlphaChars: cset := -AlphabeticChars - {#0};
```

The "-" operator, prefacing a character set, takes the *complement* of that set. The complement of a set contains all the characters that are *not* in that set. So "-AlphabeticChars" returns the set of all characters that are not alphabetic. The "- {#0}" component of the expression above removes the NUL character from the set of nonalphabetic characters. (Note that in this case the "-" operator is a binary operator denoting set difference.) This operation removes the NUL character from the set, not because it's an alphabetic character, but because the program uses the *NonAlphaChars* set to skip over characters while looking for the last name. If a last name is not present, this code needs to stop scanning once it encounters the end of the string. The #0 character always marks the end of an HLA string. By removing #0 from the *NonAlphaChars* set, the code that skips non-alphabetic characters will automatically stop if it encounters the end of the string.

```
// Demonstration of strings and character setsin a program.

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

const
    AlphabeticChars:    cset := {'a'..'z', 'A'..'Z' };
    NonAlphaChars:      cset := -AlphabeticChars - {#0};

var
    StartOfFirstName:   uns32;
    LengthOfFirstName:  uns32;
    StartOfLastName:    uns32;
    LengthOfLastName:   uns32;
```

```
        FullName:           string;
        FirstName:          string;
        LastName:           string;

    begin strcsetLab;

        // Read the full name from the user:

        stralloc( 256 );        // Allocate storage for the string.
        mov( eax, FullName );

        stdout.put( “Enter your full name (first, then last): “ );
        stdin.get( FullName );

        // Skip over any leading spaces in the name:

        mov( FullName, ebx );
        mov( 0, esi );
        while( (type char [ebx+esi]) = ‘ ‘ ) do

            inc( esi );

        endwhile;

        // Okay, presumably we’re at the beginning of the first
        // name.  Find the end of it here:

        mov( esi, StartOfFirstName );
        mov( esi, LengthOfFirstName );
        while( cs.member( (type char [ebx+esi]), AlphabeticChars )) do

            inc( esi );

        endwhile;
        sub( esi, LengthOfFirstName );
        neg( LengthOfFirstName );

        // Skip over any non-alphabetic characters

        while( cs.member( (type char [ebx+esi]), NonAlphaChars )) do

            inc( esi );

        endwhile;

        // Okay, find the end of the last name:

        mov( esi, StartOfLastName );
        while( cs.member( (type char [ebx+esi]), AlphabeticChars )) do

            inc( esi );

        endwhile;
        sub( StartOfLastName, esi );
        mov( esi, LengthOfLastName );


        // Extract the first and last names from the string:

        str.a_substr( FullName, StartOfFirstName, LengthOfFirstName );
        mov( eax, FirstName );

        str.a_substr( FullName, StartOfLastName, LengthOfLastName );
        mov( eax, LastName );

        // Display the results:

        stdout.put( “First: “, FirstName, nl );
        stdout.put( “Last:  “, LastName, nl );

        // Clean up the allocated storage:

        strfree( FullName );
        strfree( FirstName );
        strfree( LastName );

    end strcsetLab;
```

Program 13.6 String and Character Set Exercises Code

Exercise A: Compile and run this program. Describe the operation of the program and its output in your laboratory report.

Exercise B: This program uses statements like “while(cs.member((type char [ebx+esi]), AlphabeticChars)) do” to loop while the current character in the string is a member of some character set. Explain why you cannot use the “IN” operator described in this chapter in place of the *cs.member* function. (Hint: if the reason isn’t obvious, trying changing the program and compiling it.)

The following sample program also demonstrates the synergy between strings and character sets in a program. This program reads a string from the user and converts that string into a character set (effectively removing duplicate characters from the string). It then displays the character set, the vowels, the consonants, and non-alphabetic characters found in the string. This program demonstrates some run-time character set functions (like intersection) as well as compile-time constant expressions. This program also demonstrates the new form of the IN operator in boolean expressions that this chapter introduces.

```
// Demonstration of character sets and strings in a program.

program strcsetLab2;
#include( “stdlib.hhf” );

const

    AlphabeticChars:    cset := {‘a’..’z’, ‘A’..’Z’ };

    Vowels:             cset := {
                                    ‘a’, ‘e’, ‘i’, ‘o’, ‘u’, ‘y’, ‘w’,
                                    ‘A’, ‘E’, ‘I’, ‘O’, ‘U’, ‘Y’, ‘W’
                                };

    Consonants:         cset := AlphabeticChars - Vowels;


var

    InputLine:      string;
    InputSet:       cset;
    VowelSet:       cset;
    ConsonantSet:   cset;
    NonAlphaSet:    cset;

begin strcsetLab2;

    repeat

        stdout.put( “Enter a line of text: “ );
        stdin.flushInput();
        stdin.a_gets();
        mov( eax, InputLine );
        stdout.put( nl, “The input line: ‘”, InputLine, “‘” nl nl );

        cs.strToCset( InputLine, InputSet );
        stdout.put( “Characters on this line: ‘”, InputSet, “‘” nl );

        cs.cpy( InputSet, VowelSet );
        cs.intersection( Vowels, VowelSet );
        stdout.put( “Vowels on this line: ‘”, VowelSet, “‘” nl );

        cs.cpy( InputSet, ConsonantSet );
        cs.intersection( Consonants, ConsonantSet );
        stdout.put( “Consonants on this line: ‘”, ConsonantSet, “‘” nl );

        cs.cpy( InputSet, NonAlphaSet );
        cs.intersection( -AlphabeticChars, NonAlphaSet );
        stdout.put
        (
            “Nonalphabetic characters on this line: ‘”,
            NonAlphaSet,
            “‘”
            nl
        );

        repeat

            stdout.put
            (
                “Would you like to try another line of text? (Y/N): “
            );
            stdin.flushInput();
            stdin.getc();

        until( al in {‘Y’, ‘y’, ‘N’, ‘n’} );

    until( al in {‘n’, ‘N’} );


end strcsetLab2;
```

Program 13.7   Character Classification Program

Exercise A: Compile and run this program. Describe its output and operation in your laboratory report.

Exercise B: Modify this program to display any decimal digit characters appearing in the input string. Compile and run the program to test its operation. Include the program listing in your laboratory report.

13.3.7 Console Array Exercise

Perhaps the most commonly encountered two-dimensional object a programmer encounters is the video display. The Windows console window, for example, is typically an 80x25 array of characters[6]. The HLA Standard Library console module provides various routines that let you manipulate the matrix of characters on the screen. In the previous chapter, for example, you saw how to use the *console.fillRect* routine to fill a rectangular region of the screen with a given character and attribute. This laboratory exercise demonstrates how to extract a rectangular region from the screen and how to redraw that text taken from the screen. This sample program also demonstrates how to write an arbitrary matrix of text to the screen.

This program calls the HLA Standard Library *console.getRect* routine to read the data from a rectangular portion of the screen into a local array of characters. This program uses *console.getRect* to save the screen display upon entry into the program. In addition to *console.getRect*, this program uses *console.putRect* to rapidly write the contents of a two-dimensional array of characters to the console display. For example, this program calls *console.putRect* to restore the original screen display when this program quits.

Between saving and restoring the original screen contents, this program moves a sequence of asterisks around in a local 80x25 array of characters and (after each movement) copies the characters to the video display. This is an extremely slow way to redraw the screen; however, today's machines are so fast that this inefficiency works to this program's advantage - it's a built-in governor that slows down the program so you can see it work.

```
// Demonstration of arrays in a program.
//
//  This code demonstrates how to access elements of
//  a two-dimensional array.  It also demonstrates how to
//  use the console.getRect and console.putRect routines.

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

const
    NumCols := 80;
    NumRows := 25;

type
    screen: char[ NumRows, NumCols ];

var

    column:     uns32;
    row:        uns32;
    SnapShot:   screen;


static

    playingField: screen := [ NumRows dup [ NumCols dup [' ']]];

begin consoleArrayLab;

    console.getRect(  0, 0, NumRows-1, NumCols-1, (type char SnapShot));
```

6. Actually, under Windows the console window can be made various different sizes. 80x25, however, is the default size.

```
    console.cls();

    // Fill an 80x24 region of the screen with a blue background.
    // Set the foreground attribute to yellow:

    console.fillRect
    (
        0,
        0,
        NumRows-1,
        NumCols-1,
        ' ',
        win.bgnd_Blue | win.fgnd_Yellow
    );

    // Play the "game".  Repeat the nested loops once for each
    // row on the screen.

    for( mov( 0, row ); row < NumRows; inc( row )) do

        // For each row, draw a row of asterisks across the screen
        // one character at a time by storing the asterisk into
        // the "playingField" array and then copying this array to
        // the screen.

        for( mov( 0, column ); column < NumCols; inc( column )) do

            intmul( NumCols, row, ebx );
            add( column, ebx );
            mov( '*', playingField[ ebx ] );
            console.putRect
            (
                0,
                0,
                NumRows-1,
                NumCols-1,
                (type byte playingField)
            );

        endfor;

        // Now erase the row of asterisks one character at a time
        // by overwriting the asterisks in "playingField" and drawing
        // the matrix to the screen after blotting out each character.

        for( mov( 0, column ); column < NumCols; inc( column )) do

            intmul( NumCols, row, ebx );
            add( column, ebx );
            mov( ' ', playingField[ ebx ] );
            console.putRect
            (
                0,
                0,
                NumRows-1,
                NumCols-1,
                (type byte playingField)
            );

        endfor;
    endfor;

    stdout.put( “Press Enter to continue: “ );
    stdin.readLn();
    console.cls();
    console.putRect
    (
        0,
        0,
        NumRows-1,
        NumCols-1, (type byte SnapShot)
    );
    console.gotoxy( NumRows-1, 0 );


end consoleArrayLab;
```

Program 13.8 Console Matrix Program

Exercise A: Compile and run this program. Describe it’s operation and output in your laboratory report.

Exercise B: Change the NumCols and NumRows constants so that the program operates in a 40x20 window rather than an 80x25 window. Compile and rerun the program and verify that it still works properly. Include the source code for the modified program in your lab report.

Exercise C: Modify the program so that it draws the string of asterisks from top to bottom, then left to right rather than from left to right, then top to bottom. Include the source code to the modified program in your lab report.

13.3.8 Multidimensional Array Exercises

The program for this exercise demonstrates an *outer product* computation. An outer product is an operation that takes an N-element array and an M-element array and produces an NxM matrix with cell [i,j] containing the value specified by some function f(Narray[i], Marray[j]) for all values 0 <= i < N and 0 <= j < M. The classic example of an outer product is the multiplication table you learned in elementary school (where the function is the multiplication operation). The program in this example computes an addition table by summing the elements of the two arrays together.

```
// Demonstration of multidimensional arrays in a program.

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

const
    NumRows := 10;
    NumCols := 10;

static

    RowStart:   uns32;
    ColStart:   uns32;

    SumArray:   uns32[ NumRows, NumCols];
    ColVals:    uns32[ NumCols ];
```

```
        RowVals:    uns32[ NumRows ];

        GoodInput:  boolean;

    begin multidimArrayLab;

        stdout.put
        (
            “Addition table generator” nl
            “-------------------------” nl
            nl
        );

        // Get the starting value for the rows from the user:

        repeat

            try

                mov( false, GoodInput );
                stdout.put( “Enter a starting value for the rows: “ );
                stdin.get( RowStart );
                mov( true, GoodInput );

              exception( ex.ConversionError )

                stdout.put( “Bad characters in number, please re-enter”, nl );

              exception( ex.ValueOutOfRange )

                stdout.put( “The value is out of range, please re-enter”, nl );

            endtry;

        until( GoodInput );

        // Fill in the RowVals array:

        mov( RowStart, eax );
        for( mov( 0, ebx ); ebx < NumRows; inc( ebx )) do

            mov( eax, RowVals[ ebx*4 ] );
            inc( eax );

        endfor;




        // Get the starting value for the columns from the user:

        repeat

            try

                mov( false, GoodInput );
                stdout.put( “Enter a starting value for the columns: “ );
                stdin.get( ColStart );
                mov( true, GoodInput );
```

```
      exception( ex.ConversionError )

        stdout.put( “Bad characters in number, please re-enter”, nl );

      exception( ex.ValueOutOfRange )

        stdout.put( “The value is out of range, please re-enter”, nl );

    endtry;

until( GoodInput );

// Fill in the ColVals array:

mov( ColStart, eax );
for( mov( 0, ebx ); ebx < NumCols; inc( ebx )) do

    mov( eax, ColVals[ ebx*4 ] );
    inc( eax );

endfor;

// Now generate the addition table:

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

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

        intmul( NumCols, ebx, edx );    // Compute index into
        add( ecx, edx );                // array as ebx*NumCols+ecx.
        mov( ColVals[ ecx*4 ], eax );   // Compute sum for this cell.
        add( RowVals[ ebx*4 ], eax );
        mov( eax, SumArray[ edx*4 ]  );

    endfor;

endfor;


// Now print the addition table.
//
// Begin by printing the Column values above the matrix

stdout.put( nl “ add |” );
for( mov( 0, ebx ); ebx < NumCols; inc( ebx )) do

    stdout.put( ColVals[ ebx*4 ]:4, “ | “ );

endfor;

// Print a line below the column values.

stdout.put( nl “-----+” );
for( mov( 0, ebx ); ebx < NumCols-1; inc( ebx )) do

    stdout.put( “-----+-” );

endfor;
stdout.put( “-----|” nl );
```

```
    // Now print the body of the matrix.  This also prints Row
    // values in the left hand column of the matrix.

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

        // Print the current row value.

        stdout.put( RowVals[ ebx*4 ]:4, " :" );

        // Now print the values for this row.

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

            intmul( NumCols, ebx, edx );    // Compute index into
            add( ecx, edx );                // array as ebx*NumCols+ecx.
            stdout.put( SumArray[ edx*4 ]:4, " | " );

        endfor;

        // Print a line below each row of numbers

        stdout.put( nl "-----+" );
        for( mov( 0, ecx ); ecx < NumCols-1; inc( ecx )) do

            stdout.put( "-----+-" );

        endfor;
        stdout.put( "-----|" nl );

    endfor;


end multidimArrayLab;
```

Program 13.9 Addition Table Generator

Exercise A: Compile and run this program. Describe its output in your lab report.

Exercise B: Modify the NumRows and NumCols constants in this program to generate a 5x6 array rather than a 10x10 array. Recompile and run the program. Describe the output in your lab report.

Exercise C: (optional) Modify the program to use the line drawing graphic characters (see appendix B) rather than the ASCII dash, vertical bar, and hyphen characters.

13.3.9 Console Attributes Laboratory Exercise

The program in this laboratory exercise demonstrates the *console.info* call that returns various bits of interesting information about the Windows console device. This information comes in handy when writing console applications that work regardless of the size of the console window. For more information about the information that *console.info* returns, please see the definition of the windows console screen buffer info record in the "WIN32.HHF" header file.

```
// Demonstration of records and the console.info routine
// (and the win.CONSOLE_SCREEN_BUFFER_INFO data type).
```

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

static

    csbi:       win.CONSOLE_SCREEN_BUFFER_INFO;

begin ConsoleAttrLab;

    // Grab the screen metric information:

    console.info( csbi );


    // Clear the screen.

    console.cls();

    // Set the whole screen to blue:

    movzx( csbi.dwSize.Y, eax );
    movzx( csbi.dwSize.X, ebx );
    dec( eax );
    dec( ebx );
    console.fillRectAttr( 0, 0, ax, bx, win.bgnd_Blue );
    console.setOutputAttr( win.bgnd_Blue | win.fgnd_Yellow );

    // Read and display the current console window title:

    console.a_getTitle();
    stdout.put
    (
        “Previous console window title was: ‘”,
        (type string eax),
        “’” nl
    );
    strfree( eax );

    // Set the console window title:

    console.setTitle( “Art of Assembly Program 4.26” );

    // Display the screen metric information:

    stdout.put
    (
        “Size X=”, csbi.dwSize.X, nl,
        “Size Y=”, csbi.dwSize.Y, nl,
        “Cursor X=”, csbi.dwCursorPosition.X, nl,
        “Cursor Y=”, csbi.dwCursorPosition.Y, nl,
        “Window Position Left=”, csbi.srWindow.left, nl,
        “Window Position Right=”, csbi.srWindow.right, nl,
        “Window Position Top=”, csbi.srWindow.top, nl,
        “Window Position Bottom=”, csbi.srWindow.bottom, nl,
        “Character display attribute=$”, csbi.wAttributes, nl,
        nl
        “Maximum window height=”, csbi.dwMaximumWindowSize.Y, nl,
        “Maximum window width =”, csbi.dwMaximumWindowSize.X, nl,
        nl
    );
    // Let the user see everything before we clean up and quit.

    stdout.put( “Press ENTER to continue: “ );
    stdin.readLn();

    // Be nice and set the console window back to black & white:

    console.setOutputAttr( csbi.wAttributes );
    console.cls();


end ConsoleAttrLab;
```

Program 13.10 Displaying Console Attributes

Exercise A: Compile and run this program. Describe the console metrics for your particular system in your laboratory report.

Exercise B: Modify this program to save the screen text upon entry into the program and restore the screen when this program exits. Use the cursor position in the *csbi* record to restore the cursor to its original position upon program exit. Include the modified source code with your lab report.

13.3.10 Records, Arrays, and Pointers Laboratory Exercise

The following program demonstrates dynamic allocation of an array of records. It also demonstrates enumerated data types and record constants. This program asks the user to specify some number of students and then it inputs the data for the specified students and stores the data into the dynamic array it creates.

```
// Demonstration of a pointer to an array of records.
// This is a small student database program the lets the
// user input a bunch of student names and it prints a report
// after the data is entered.

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

const

    // The following text string appears frequently in this code.

    CurStudent: text := “(type EngStudent [edi+ecx])”;

type
    EngMajor:   enum{ ElecEng, CompSci, MechEng, BioChemEng };
    Standing:   enum{ Freshman, Sophomore, Junior, Senior, Grad };

    // Here’s the basic data type for our student database:

    EngStudent:
        record

            StudentName:    string;
            ID:             string;
```

```
                major:          EngMajor;
                Year:           Standing;

            endrecord;

        ESPtr:  pointer to EngStudent;


    readonly

        // The following string make it easy to output
        // the EngMajor and Standing enumerated data types.

        MajorStrs:      string[4] :=
                            [
                                "Electrical Engineering ",
                                "Computer Science       ",
                                "Mechanical Engineering ",
                                "Biochemical Engineering"
                            ];

        YearStrs:       string[4] :=
                            [
                                "Freshman ",
                                "Sophomore",
                                "Junior   ",
                                "Senior   "
                            ];

        // An extra student to copy into the list.
        // This exists here mainly to demonstrate record constants.

        JohnSmith:      EngStudent :=
                            EngStudent:
                            [
                                "John Smith",
                                "555-55-5555",
                                CompSci,
                                Junior
                            ];


    static

        NumStudents:    uns32;
        StudentArray:   ESPtr;
        GoodInput:      boolean;




    begin StudentDatabase;

        stdout.put( "Engineering Student Database Program" nl nl );

        // Get the number of students the user wants to enter into
        // the database.  Because this is just a demo, ask them if
        // they really want to enter more than 10 students if they
        // enter a large number.

        repeat
            try

                mov( false, GoodInput );
                stdout.put
                (
                    nl
                    “How many students would you like to enter? “
                );
                stdin.get( NumStudents );
                mov( true, GoodInput );

                // Allow 0..10 students with no questions asked.
                // But if they specify more than 10...

                if( NumStudents > 10 ) then

                    stdout.put
                    (
                        “Are you sure you want to enter “,
                        NumStudents,
                        “ students into the database? (Y/N):”
                    );
                    stdin.flushInput();
                    stdin.getc();

                    // Set GoodInput to true if they answer yes.

                    cs.member( al, {‘y’, ‘Y’} );
                    mov( al, GoodInput );

                endif;

              exception( ex.ConversionError )

                stdout.put( “Illegal characters in number, please re-enter” nl );

              exception( ex.ValueOutOfRange )

                stdout.put( “Value is too big, please re-enter” nl );

            endtry;

        until( GoodInput );


        // Okay, we’ve got the number of students.  Bump in by one so we
        // can sneak John Smith’s record into the data base.  Then allocate
        // storage for a dynamic array of records.

        mov( NumStudents, eax );    // Get the user-specified count.
        inc( eax );                 // Make room for John Smith.

        // Since each student consumes “@size( EngStudent )” bytes, we need
        // to multiply our student count by this value to compute the number
        // of bytes we will need.

        intmul( @size( EngStudent ), eax );

        // Allocate storage for the array of students.
```

```
malloc( eax );
mov( eax, StudentArray );


// Copy the data for John Smith to the first element of
// our array.  Note: This code just copies the pointers
// to the StudentName and ID fields because no modifications
// are ever made to the JohnSmith record in this program.

mov( JohnSmith.StudentName, ecx );
mov( ecx, (type EngStudent [eax]).StudentName );

mov( JohnSmith.ID, ecx );
mov( ecx, (type EngStudent [eax]).ID );

mov( JohnSmith.major, cl );
mov( cl, (type EngStudent [eax]).major );

mov( JohnSmith.Year, cl );
mov( cl, (type EngStudent [eax]).Year );


// Okay, now input each of the remaining students from the user:

mov( StudentArray, edi );   // Get base address of array.

for( mov( 1, ebx ); ebx <= NumStudents; inc( ebx )) do

    intmul( @size( EngStudent ), ebx, ecx );    // Index into array.

    stdout.put( “Enter student’s name: “ );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, CurStudent.StudentName );

    stdout.put( “Enter the student ID: “ );
    stdin.a_gets();
    mov( eax, CurStudent.ID );

    // Get (and verify) F, S, J, or N from the user to select
    // this student’s standing in the college.

    repeat

        stdout.put
        (
            “Is this student a “
            “F)reshman, S)ophomore, J)unior, or seN)ior? “
        );
        stdin.flushInput();
        stdin.getc();
        chars.toUpper( al );

    until( al in {‘F’, ‘S’, ‘J’, ‘N’} );
    if( al = ‘F’ ) then

        mov( Freshman, CurStudent.Year );

    elseif( al = ‘S’ ) then

        mov( Sophomore, CurStudent.Year );
```

```
elseif( al = 'J' ) then

    mov( Junior, CurStudent.Year );

else

    mov( Senior, CurStudent.Year );

endif;


// Get the student's major from the user.  Again,
// verify that the input is correct before proceeding.

repeat

    stdout.put
    (
        "What is this student's major?" nl
        "  B)iochemical Engineering" nl
        "  C)omputer Science" nl
        "  E)lectrical Engineering" nl
        "  M)echanical Engineering" nl
        nl
        "(B,C,E,M)? "
    );
    stdin.flushInput();
    stdin.getc();
    chars.toLower( al );

until( al in {'b', 'c', 'e', 'm'} );
if( al = 'b' ) then

    mov( BioChemEng, CurStudent.major );

elseif( al = 'c' ) then

    mov( CompSci, CurStudent.major );

elseif( al = 'e' ) then

    mov( ElecEng, CurStudent.major );

else

    mov( MechEng, CurStudent.Year );

endif;

// Okay, store away the CurStudent variable data into
// the next available slot in the ESPtr array.

mov( StudentArray, eax );

// Compute index into array:

intmul( @size( EngStudent ), ebx, ecx );

mov( CurStudent.StudentName, edx );
mov( edx, (type EngStudent [eax+ecx]).StudentName );
```

```
            mov( CurStudent.ID, edx );
            mov( edx, (type EngStudent [eax+ecx]).ID );

            mov( CurStudent.major, dl );
            mov( dl, (type EngStudent [eax+ecx]).major );

            mov( CurStudent.Year, dl );
            mov( dl, (type EngStudent [eax+ecx]).Year );



        endfor;

        // Okay, now that we’ve got all the students into the system,
        // print a brief report.

        stdout.put
        (
            “-----Name----------- “
            “-----ID----- “
            “-------Major----------- “
            “--Year---”
            nl
            nl
        );

        mov( StudentArray, edi );   // Not really needed, but just to be sure.

        for( mov( 0, ebx ); ebx <= NumStudents; inc( ebx )) do

            intmul( @size( EngStudent ), ebx, ecx );    // Index into array.
            movzx( CurStudent.major, edx );
            movzx( CurStudent.Year, esi );
            stdout.put
            (
                CurStudent.StudentName:-20, “ “,
                CurStudent.ID:-12, “ “,
                MajorStrs[ edx*4 ]:-23, “ “,
                YearStrs[ esi*4 ],
                nl
            );

        endfor;
        stdout.newln();



    end StudentDatabase;
```

Program 13.11 Student Database Program

Exercise A: Compile and run this program. Describe its output in your lab report.

Exercise B: Modify this program to let the user specify that only those students in a single major be printed (by specifying a single letter to select the major- see the body of the code for details on this). Provide an “A)ll” option that prints the full report as the program currently does.

13.3.11Separate Compilation Exercises

In this laboratory exercise you will compile a multipart program and link it into a single executable file. You will also construct a makefile for the project and test the workings of the makefile by making minor modifications to the program. Finally, you will split the main program into two pieces and move one section into its own module.

The following listings provide the necessary source code for this program. Note that these modules can be found in the CH05 subdirectory of the accompanying CD. The main program is lab5_1.hla; the header file is lab5_1.hhf; the first unit can be found in lab5_1db.hla; the second unit is in the file lab5_1unita.hla.

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

    // GetYorN-
    //
    //  This procedure prints a prompt that requires a yes/no
    //  answer.  It continually prompts the user to input Y or N
    //  until they enter one of these characters.  It returns
    //  true in AL if the user entered “Y” or “y”.  It returns
    //  false if the user entered “n” or “N”.

    procedure GetYorN( Prompt:string ); returns( “AL” );
    begin GetYorN;

        repeat

            stdout.put( Prompt, “ (Y/N):” );
            stdin.flushInput();
            stdin.getc();

        until( al in {‘y’, ‘n’, ‘Y’, ‘N’ });
        if( al in {‘y’, ‘Y’} ) then

            mov( true, al );

        else

            mov( false, al );

        endif;

    end GetYorN;


    // GetCinSet-
    //
    //  This is another procedure, similar to the above, that
    //  repeatedly requests input from the user until the user
    //  enters a valid character.  This procedure returns once
    //  the user enters a character that is a member of the
    //  cset passed as a parameter.

    procedure GetCinSet( legalInput:cset ); returns( “AL” );
    begin GetCinSet;
```

```
        repeat

            stdout.put( " {", legalInput, "}:" );
            stdin.flushInput();
            stdin.getc();

        until( al in legalInput );

    end GetCinSet;


var
    artist: string;
    title:  string;


begin cor_mainPgm;

    stdout.put
    (
        "Welcome to the CD database program" nl
        "----------------------------------" nl
        nl
    );
    repeat

        stdout.put( "Select CD by A)rtist or by T)itle? " );
        GetCinSet( {'a', 'A', 't', 'T'} );
        chars.toUpper( al );
        if( al = 'A' ) then

            stdout.put( "Input artist's name: " );
            stdin.flushInput();
            stdin.a_gets();
            mov( eax, artist );
            SearchByArtist( artist, MyCDCollection );
            strfree( artist );

        else

            stdout.put( "Input CD's title: " );
            stdin.flushInput();
            stdin.a_gets();
            mov( eax, title );
            SearchByTitle( title, MyCDCollection );
            strfree( title );

        endif;

        stdout.newln();
        GetYorN( "Would you like to search for another CD?" );

    until( !al );


end cor_mainPgm;
```

Program 13.12 The MAIN Program for this Laboratory Exercise (mainPgm.hla)

```
const
    NumCDs := 10;

type
    CD: record

            Title:     string;
            Artist:    string;
            Publisher:string;
            CopyrightYear:uns32;

        endrecord;

    CDs: CD[ NumCDs ];


static
    MyCDCollection: CDs; external;



    procedure SearchByTitle( Title:string; var CDList:CDs ); external;
    procedure SearchByArtist( Artist:string; var CDList:CDs ); external;
```

Program 13.13 The sepCompDemo.hhf Header File for this Laboratory Exercise

```
unit Unitdb;
#include( “stdlib.hhf” );
#include( “sepCompDemo.hhf” );


static
    MyCDCollection: CDs :=
        [
            CD:[ “The Cars Greatest Hits”, “The Cars”, “Elektra”, 1985 ],
            CD:[ “Boston”, “Boston”, “Epic”, 1976 ],
            CD:[ “Imaginos”, “Blue Oyster Cult”, “Columbia”, 1988 ],
            CD:[ “Greatest Hits”, “Dan Fogelberg”, “Epic”, 1982 ],
            CD:[ “Walk On”, “Boston”, “MCA”, 1994 ],
            CD:[ “Big Ones”, “Loverboy”, “Columbia”, 1989 ],
            CD:[ “James Bond 30th Anniversary”, “various”, “EMI”, 1992 ],
            CD:[ “Give Thanks”, “Don Moen”, “Hosanna! Music”, 1991 ],
            CD:[ “Star Tracks”, “Erich Kunzel”, “Telarc”, 1984 ],
            CD:[ “Tones”, “Eric Johnson”, “Warner Bros”, 1986 ]
        ];



end Unitdb;
```

Program 13.14 The First UNIT Associated with this Lab (the Database File)

```
unit DBunit;
#include( “stdlib.hhf” );
#include( “sepCompDemo.hhf” );


// PrintInfo
//
//  This is a private procedure (local to this unit)
//  that displays the information about the CD passed
//  as a parameter.

procedure PrintInfo( var theCD:CD );
const
    cdt: text := “(type CDs [ebx])”;

begin PrintInfo;

    push( ebx );
    mov( theCD, ebx );
    stdout.put( “Title:    “, cdt.Title, nl );
    stdout.put( “Artist:   “, cdt.Artist, nl );
    stdout.put( “Publisher:”, cdt.Publisher, nl );
    stdout.put( “Year:     “, cdt.CopyrightYear, nl nl );
    pop( ebx );

end PrintInfo;


// SearchByTitle
//
//  This procedure searches through a CD database for
//  all CDs with a given title.  The title of the CD
//  to search for and the CD database array are the
//  parameters for this procedure.

procedure SearchByTitle( Title:string; var CDList:CDs );
const

    // Because CDList is passed by reference, we access
    // it indirectly using a 32-bit register.  The
    // following text equate lets us use a more meaningful
    // name for this pointer than “[ebx]”.

    cdArray: text := “(type CDs [ebx])”;


var
    FoundCD: boolean;   // Set to true if we find at least
                        // one CD with the given title.

begin SearchByTitle;

    push( ebx );
```

```
        push( esi );

        mov( CDList, ebx ); // Put address of array in EBX so we can
                            // use the “cdArray” symbol to reference
                            // the array passed by reference

        mov( false, FoundCD );      // Assume we didn’t find a CD.
        mov( 0, esi );              // Index into cdArray.
        repeat

            // See if the parameter Title matches the current
            // title in cdArray.  Note that str.ieq does as
            // case insenstive comparison.

            if( str.ieq( Title, cdArray.Title[ esi ] )) then

                // If we found it, print the information
                // associated with this CD.

                PrintInfo( (type CD cdArray[ esi ]) );
                mov( true, FoundCD );

            endif;

            // Move on to the next element of the CDList array.
            // Note that rather than multiplying the index into
            // cdArray by @size( CD ) on each access, this code
            // simply bumps ESI by the size of a cdArray element
            // each time through the loop.  Therefore, ESI will
            // contain the index of each successive array element
            // on each pass through the loop.

            add( @size( CD ), esi );

            // We’re done when ESI moves off the end of the array.
            // The constant “@size(CD)*NumCDs” is the index of
            // the first memory location beyond the end of the array.

        until( esi >= @size( CD ) * NumCDs );

        // Print an appropriate message if the lookup operation did
        // not find a CD with the specified title.

        if( !FoundCD ) then

            stdout.put( “That title does not exist in the database” nl );

        endif;
        pop( esi );
        pop( ebx );

    end SearchByTitle;




    // SearchByArtist-
    //
    //  Searches for all CDs by a given artist name.
    //  This code is nearly identical to SearchByTitle.
    //  See the comments in SearchByTitle for additional
```

```
//  comments on the operation of this code.

procedure SearchByArtist( Artist:string; var CDList:CDs );
const
    cdArray: text := “(type CDs [ebx])”;

var
    FoundCD: boolean;

begin SearchByArtist;

    push( ebx );
    push( esi );

    mov( CDList, ebx ); // Put address of array in EBX so we can
                        // use the “cdArray” symbol to reference
                        // the array passed by reference

    mov( false, FoundCD );      // Assume we didn’t find a CD.
    mov( 0, esi );
    repeat

        if( str.ieq( Artist, cdArray.Artist[ esi ] )) then

            PrintInfo( (type CD cdArray[ esi ]) );
            mov( true, FoundCD );

        endif;

        // Move on to the next element of the CDList array.

        add( @size( CD ), esi );

    until( esi >= @size( CD ) * NumCDs );
    if( !FoundCD ) then

        stdout.put( “That artist does not exist in the database” nl );

    endif;
    pop( esi );
    pop( ebx );

end SearchByArtist;

end DBunit;
```

Program 13.15 The Second Unit Need by this Lab Exercise (DBunit.hla)

Exercise A: Compile this program with the following HLA command.

```
hla mainPgm lnitb DBunit
```

Run the program a few times and learn how to use it. This is a small (trivial) little database program that stores information about a private CD collection (for brevity, it only has 10 CDs in the database). When the user runs the program it will ask if they want to look up a CD by artist or by title. As you can see in the sample database, “Boston” is both a title and an artist. “Imaginos” is an example of a title. Run the program and specify these values for artist and title to familiarize yourself with this code. If possible, include a printout of the program's operation with your lab report, otherwise describe the output in your lab report.

Exercise B: Determine the dependencies between these files. Write a makefile for this code. Include the makefile with your lab report. Verify that the makefile works properly by changing each of the files and then running the make program after each modification.

Exercise C: The *GetYorN* and *GetCinSet* procedures in the main program really belong in their own unit. Move these subroutines to the unit *InputUnit* in the file inputunit.hla. Create a header file named InputUnit.hhf to contain the external declarations. Modify your makefile to accommodate these changes. Recompile the system and verify that it still works. Include the modified source code and makefile in your lab report.

13.3.12The HLA (Pseudo) Random Number Unit

In this laboratory exercise, you will test the quality of the two HLA random number generators. The following program code provides a simple test by plotting asterisks at random positions on the screen. This program works by choosing two random numbers, one between zero and 79, the other between zero and 23. Then the program uses the *console.puts* function to print a single asterisk at the (X,Y) coordinate on the screen specified by these two random numbers. After 10,000 iterations of this process the program stops and lets you observe the result. **Note**: since random number generators generate random numbers, you should not expect this program to fill the entire screen with asterisks in only 10,000 iterations.

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

begin testRandom;

    console.cls();
    mov( 10_000, ecx );
    repeat

        // Generate a random X-coordinate
        // using rand.range.

        rand.range( 0, 79 );
        mov( eax, ebx );            // Save the X-coordinate for now.

        // Generate a random Y-coordinate
        // using rand.urange.

        rand.urange( 0, 23 );

        // Print an asterisk at
        // the specified coordinate on the screen.

        console.puts( ax, bx, "*" );

        // Repeat this 10,000 times to get
        // a good distribution of values.

        dec( ecx );

    until( @z );

    // Position the cursor at the bottom of the
    // screen so we can observe the results.
    console.gotoxy( 24, 0 );

end testRandom;
```

Program 13.16 Screen Plot Test of the HLA Random Number Generators

Exercise A: Run this program and note how many “holes” are left unfilled on the screen. If possible, make a print-out of the screen to include with your lab manual. Compare the result of your output with another student’s output for this exercise. Did they get the same output as you? Explain why or why not in your lab report.

Exercise B: Add the statement “rand.randomize()” as the first statement in this program. Recompile and run the program. Compare this output to the output from Exercise A. Compare this output to that of another student in your class for this exercise. Did they get the same output as you? Explain why or why not in your lab report.

Exercise C: Remove the call to *rand.randomize* that you inserted in Exercise B. Bump up the number of iterations to 25,000. Does the program fill the screen with asterisks? Determine to the nearest 100, the minimum number of iterations it takes to completely fill the screen with asterisks.

13.3.13File I/O in HLA

The HLA Standard Library provides a file input/output module that makes writing data to a file and reading data from a file almost as easy as working with the standard output and standard input devices. In this laboratory exercise you will discover how easy it is to read and write file data in HLA.

The following program code opens a new file, writes some data to it, closes that file, reopens it for reading, reads the data from the file, displays the data to the standard output, and finally closes the file.

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

var
    fileHandle: dword;
    s:          string;

begin fileDemo;

    // Note the use of instruction composition to
    // store the return value from stralloc (which
    // is in EAX) directly into s.

    mov( stralloc( 256 ), s );

    // Open a brand new file and write some
    // data to it.  Note that the filename “x.txt”
    // must not already exist or the fileio.OpenNew
    // call will raise an exception.

    mov( fileio.openNew( “x.txt” ), fileHandle);

    fileio.put
    (
        fileHandle,
        "Data Written to a text file" nl
        "--------------------------" nl
        nl
    );

    for( mov( -5, ebx ); (type int32 ebx) <= 5; inc( ebx )) do

        fileio.put( fileHandle, "ebx=", (type int32 ebx ), nl );

    endfor;

    // Done writing the data to the file, close it.

    fileio.close( fileHandle );


    // Reopen the file, this time for reading:

    mov( fileio.open( "x.txt", fileio.r ), fileHandle);

    // Read the data from the file and write it to the
    // standard output device:

    while( !fileio.eof( fileHandle )) do

        fileio.gets( fileHandle, s );
        stdout.put( s, nl );

    endwhile;
    fileio.close( fileHandle );
    strfree( s );

end fileDemo;
```

Program 13.17 HLA File I/O Demonstration

Exercise A: Compile and run this program. Note that this program creates a text file “x.txt” on your disk. Include a printout of this file with your laboratory report.

Exercise B: Modify this program so that it reads the filename from the user and opens the specified file. Compile and test the modified program. Include a printout of the program with your laboratory report. **Warning: when running this program, be careful not to specify the name of an existing file on your disk.** Remember, this program will delete any pre-existing data in the file you specify.

Exercise C: Split this program into two separate programs. The first program should write the data to the text file, the second program should read and display the data in the text file. Compile and run the programs to verify proper operation. Include the source listings of these two programs with your lab reports.

13.3.14 Timing Various Arithmetic Instructions

In this laboratory exercise you will run several sections of code that execute various arithmetic instructions one billion times. Using a stopwatch, you will measure the time required by each computation and compare the execution times of these various instructions.

```
program TimeArithmetic;
```

```
#include( “stdlib.hhf” );


begin TimeArithmetic;


    // The following code does one billion additions to
    // provide a baseline for further instruction timing
    // comparisons.

    stdout.put
    (
        “Multiplication timing test:” nl
        nl
        “This program will compute EAX+12 1,000,000,000 times” nl
        “using the ADD instruction.” nl
        nl
        “You should time this with a stop watch” nl
        nl
        “Press ENTER to begin the additions:”
    );
    stdin.readLn();
    for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

        mov( 123_456_789, eax );
        add( 12, eax );

    endfor;
    stdout.put( stdio.bell, “Okay, stop timing the code” nl );




    // The following code demonstrates the expense of
    // the INTMUL instruction relative to addition:

    stdout.put
    (
        “Multiplication timing test:” nl
        nl
        “This program will compute EAX*12 1,000,000,000 times” nl
        “using the INTMUL instruction.” nl
        nl
        “You should time this with a stop watch” nl
        nl
        “Press ENTER to begin the multiplications:”
    );
    stdin.readLn();
    for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

        mov( 123_456_789, eax );
        intmul( 12, eax );

    endfor;
    stdout.put( stdio.bell, “Okay, stop timing the code” nl );



    // The following code demonstrates the expense of IMUL
    // relative to addition:
```

```
stdout.put
(
    “Now this program will compute EAX*12 1,000,000,000 times” nl
    “using the IMUL instruction.” nl
    nl
    “You should time this with a stop watch” nl
    nl
    “Press ENTER to begin the multiplications:”
);
stdin.readLn();
for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

    mov( 123_456_789, eax );
    imul( 12, eax );

endfor;
stdout.put( stdio.bell, “Okay, stop timing the code” nl );


// The following code demonstrates the cost of multiplication
// via shifts and adds:

stdout.put
(
    “Now this program will compute EAX*12 1,000,000,000 times” nl
    “using the shifts and adds instruction.” nl
    nl
    “You should time this with a stop watch” nl
    nl
    “Press ENTER to begin the multiplications:”
);
stdin.readLn();
for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

    mov( 123_456_789, eax );
    mov( 123_456_789, ebx );
    shl( 1, ebx );
    shl( 3, eax );
    add( ebx, eax );

endfor;
stdout.put( stdio.bell, “Okay, stop timing the code” nl );



// The following code demonstrates the cost of the IDIV
// instruction relative to addition (warning: IDIV is slow!):

stdout.put
(
    “Now this program will compute EAX/12 1,000,000,000 times” nl
    “using the IDIV instruction.” nl
    nl
    “You should time this with a stop watch” nl
    nl
    “Press ENTER to begin the divisions:”
);
stdin.readLn();
for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do
```

```
            mov( 123_456_789, eax );
            cdq();
            idiv( 12, edx:eax );

        endfor;
        stdout.put( stdio.bell, "Okay, stop timing the code" nl );


end TimeArithmetic;
```

Program 13.18 Instruction Timing Code (TimeArithmetic.hla)

Exercise A: Compile and run this program. Measure the times required by each instruction sequence in the program. Describe the differences in timing in your laboratory report.

Exercise B: Simply dividing the time measured for each section doesn't accurately compute the time for each instruction because you're also measuring the time required by the FOR loop as well as the MOV instruction and any other instructions appearing in the loop (that aren't the ADD, INTMUL, IMUL, IDIV, or SHL instructions). To get a better result, you should time an empty loop and subtract the time for a loop without the specified instruction (i.e., remove the ADD instruction from the first code section in Program 13.18 above and use the resulting code as your base code).

Exercise C: Modify the program so that it computes the time required by an empty loop. Run this program and report the results in your lab report. Subtract this time from your other results to get results that are closer to reality.

13.3.15 Using the RDTSC Instruction to Time a Code Sequence

One problem with timing statements as done in the previous section is that the time you obtain is extremely inaccurate. This is because the operating system periodically interrupts the code, effectively freezing it in its tracks for a few moments, while other activities take place. Since your stopwatch is still running while the program is frozen, your timings will not accurate reflect the time it takes to execute these instructions.

On Pentium and later processors, Intel added the RDTSC (ReaD Time Stamp Counter) instruction to help software engineers better time certain instruction sequences. The Time Stamp Counter is a 64-bit internal register that the CPU increments by one on each clock cycle. Executing the RDTSC instruction copies this 64-bit value into the EDX:EAX register pair. By reading the value of the Time Stamp Counter before and after an instruction sequence, then computing the difference between these two values, you can effectively measure the time of some instructions in clock cycles.

There is one major problem with the RDTSC instruction: not all CPUs support this instruction. Intel CPUs earlier than the Pentium do not support this instruction. Furthermore, certain non-Intel CPUs do not support this instruction either. If you happen to have such a CPU, executing the RDTSC instruction will raise an "invalid instruction" exception. If this is the case for your machine, then simply skip this lab exercise.

There are two additional, but minor, problems with the RDTSC instruction. First, it produces a 64-bit result. Since we've only seen how to do 32-bit arithmetic on the 80x86, computing the difference of the two 64-bit values will prove to be difficult[7]. We can solve this problem by simply ignoring the H.O. DWORD of the 64-bit result that RDTSC returns. Once in a while we will get a nonsensical result (generally negative), but as long as we're willing to ignore such results and re-run the program to get correct results, we'll do fine.

7. Actually, we could use the FILD and FIST instructions on the FPU, but that won't prove to be necessary.

Note that the chapter on Advanced Arithmetic discusses extended precision arithmetic, for those who want to learn how to manipulate 64-bit (or larger) integer values.

The second problem with using RDTSC is that it is still possible for an interrupt to occur in the middle of the instruction sequence we're timing. This will produce inaccurate results. Fortunately, it is very rare for an interrupt to occur in the middle of a short instruction sequence, so it is unlikely that this will affect the results you get. To be sure this problem doesn't occur, all you've got to do is execute your program twice and verify that the results you obtain are similar; if one value is off by an order of magnitude, or more, then you should re-run the program a few more times.

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


begin RDTSCdemo;



    stdout.put
    (
        “Instruction timing test:” nl
        nl
    );

    // Note: repeat the following code twice in order
    // to ensure that all code comes out of the cache.
    // Compute the timings on the second execution of the loop.

    for( mov( 2, edi ); edi > 0; dec( edi )) do

        rdtsc();
        mov( eax, ecx );    // Save L.O. word in ECX

        // Do the instruction over 8 times to eliminate
        // certain problems inherent in RDTSC.

        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );

        rdtsc();            // Read time stamp counter after computations.

    endfor;
    sub( ecx, eax );        // Computer timer difference.
    stdout.put( “Cycles for ADD sequence: “, (type uns32 eax), nl );
```

```
// Time the INTMUL instruction:

for( mov( 2, edi ); edi > 0; dec( edi )) do

    rdtsc();
    mov( eax, ecx );    // Save L.O. word in ECX

    // Do the instruction over 8 times to eliminate
    // certain problems inherent in RDTSC.

    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );

    rdtsc();            // Read time stamp counter after computations.

endfor;
sub( ecx, eax );        // Computer timer difference.
stdout.put( “Cycles for INTMUL sequence: “, (type uns32 eax), nl );



// Time the IMUL instruction:

for( mov( 2, edi ); edi > 0; dec( edi )) do

    rdtsc();
    mov( eax, ecx );    // Save L.O. word in ECX

    // Do the instruction over 8 times to eliminate
    // certain problems inherent in RDTSC.

    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
```

```
            rdtsc();            // Read time stamp counter after computations.

        endfor;
        sub( ecx, eax );        // Computer timer difference.
        stdout.put( “Cycles for IMUL sequence: “, (type uns32 eax), nl );




        // Time the IDIV instruction:

        for( mov( 2, edi ); edi > 0; dec( edi )) do

            rdtsc();
            mov( eax, ecx );    // Save L.O. word in ECX

            // Do the instruction over 8 times to eliminate
            // certain problems inherent in RDTSC.

            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );

            rdtsc();            // Read time stamp counter after computations.

        endfor;
        sub( ecx, eax );        // Computer timer difference.
        stdout.put( “Cycles for IDIV sequence: “, (type uns32 eax), nl );


    end RDTSCdemo;
```

Program 13.19 Timing Code Using RDTSC

Exercise A: Compile and run this program. Compare the times produced by this program against the previous exercise (to compute time from clock cycles, multiply by one billion and divide by the clock frequency of your system). Describe the differences in timing in your laboratory report.

Exercise B: The RDTSC instruction is quite slow compared to many other instructions. Devise a short program to test the execution time of the RDTSC instruction. Report the result of this timing in your lab report.

13.3.16Timing Floating Point Instructions

Floating point instructions tend to execute more slowly than their integer counterparts. In this laboratory exercise you will compute the running time of the FADD, FSUB, FMUL, and FDIV instructions. This code uses the RDTSC instruction. If your CPU does not support this instruction, modify the code to use the timing loops in the first lab exercise in this chapter and time your code that way.

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


begin FPTiming;


    stdout.put
    (
        “Instruction timing test:” nl
        nl
    );

    // Note: repeat the following code twice in order
    // to ensure that all code comes out of the cache.
    // Compute the timings on the second execution of the loop.

    for( mov( 2, edi ); edi > 0; dec( edi )) do

        rdtsc();
        mov( eax, ecx );    // Save L.O. word in ECX

        // Do the instruction over 8 times to eliminate
        // certain problems inherent in RDTSC.

        fld( 123_456_789e0 );
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
```

```
                rdtsc();            // Read time stamp counter after computations.

            endfor;
            sub( ecx, eax );        // Computer timer difference.
            stdout.put( "Cycles for FADD sequence: ", (type uns32 eax), nl );


            // Time the FSUB instruction:

            for( mov( 2, edi ); edi > 0; dec( edi )) do

                rdtsc();
                mov( eax, ecx );    // Save L.O. word in ECX

                // Do the instruction over 8 times to eliminate
                // certain problems inherent in RDTSC.

                fld( 123_456_789e0 );
                fld( 12.0 );
                fsub();
                fld( 12.0 );
                fsub();
                fld( 12.0 );
                fsub();
                fld( 12.0 );
                fsub();
                fld( 12.0 );
                fsub();
                fld( 12.0 );
                fsub();
                fld( 12.0 );
                fsub();
                fld( 12.0 );
                fsub();


                rdtsc();            // Read time stamp counter after computations.

            endfor;
            sub( ecx, eax );        // Computer timer difference.
            stdout.put( "Cycles for FSUB sequence: ", (type uns32 eax), nl );




            // Time the FMUL instruction:

            for( mov( 2, edi ); edi > 0; dec( edi )) do

                rdtsc();
                mov( eax, ecx );    // Save L.O. word in ECX

                // Do the instruction over 8 times to eliminate
                // certain problems inherent in RDTSC.

                fld( 123_456_789e0 );
                fld( 12.0 );
                fmul();
                fld( 12.0 );
```

```
        fmul();
        fld( 12.0 );
        fmul();
        fld( 12.0 );
        fmul();
        fld( 12.0 );
        fmul();
        fld( 12.0 );
        fmul();
        fld( 12.0 );
        fmul();
        fld( 12.0 );
        fmul();


        rdtsc();            // Read time stamp counter after computations.

    endfor;
    sub( ecx, eax );        // Computer timer difference.
    stdout.put( “Cycles for FMUL sequence: “, (type uns32 eax), nl );



    // Time the FDIV instruction:

    for( mov( 2, edi ); edi > 0; dec( edi )) do

        rdtsc();
        mov( eax, ecx );    // Save L.O. word in ECX

        // Do the instruction over 8 times to eliminate
        // certain problems inherent in RDTSC.

        fld( 123_456_789e0 );
        fld( 12.0 );
        fdiv();
        fld( 12.0 );
        fdiv();
        fld( 12.0 );
        fdiv();
        fld( 12.0 );
        fdiv();
        fld( 12.0 );
        fdiv();
        fld( 12.0 );
        fdiv();
        fld( 12.0 );
        fdiv();
        fld( 12.0 );
        fdiv();

        rdtsc();            // Read time stamp counter after computations.

    endfor;
    sub( ecx, eax );        // Computer timer difference.
    stdout.put( “Cycles for FDIV sequence: “, (type uns32 eax), nl );


end FPTiming;
```

Program 13.20 Timing FADD, FSUB, FMUL, and FDIV

Exercise A: Compile and run this program. Compare the cycles counts (or times) produced by this program against the previous exercise. Describe the differences in timing in your laboratory report.

Exercise B: Floating point timings are very data dependent. Modify the program by changing the constants around. Retry the computations with constants that are close to one another and more widely separated than in this example. (Hint: create a CONST definition for the two constants this program uses in order to facilitate the changes; can you see the benefit of using CONST here?)

Exercise C: Modify the program so that the floating point control register specifies the 64-bit rounding mode (vs. 53 bits or 24 bits). Re-run the program and check the timings. Then set the round to 53 bits and 24 bits. Compare the results.

Exercise D: Modify the prorgram to factor out the time required by the RDTSC instruction. Report your results in your lab report.

13.3.17Table Lookup Exercise

The following laboratory exercise computes a sequence of sine values using table lookup and using the FPU FSIN instruction so you can see the difference in timing between the two techniques. Like the previous two exercises, this code uses the RDTSC instruction to time the sequences. If your CPU does not support the RDTSC instruction, rewrite the code to use surrounding loops as used in the first laboratory exercise of this chapter.

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

static
    angle:      uns32;
    theSIN:     uns32;

readonly
    sines: int32[360] :=
            [

                  0,   17,   35,   52,   70,   87,  105,  122,
                139,  156,  174,  191,  208,  225,  242,  259,
                276,  292,  309,  326,  342,  358,  375,  391,
                407,  423,  438,  454,  469,  485,  500,  515,
                530,  545,  559,  574,  588,  602,  616,  629,
                643,  656,  669,  682,  695,  707,  719,  731,
                743,  755,  766,  777,  788,  799,  809,  819,
                829,  839,  848,  857,  866,  875,  883,  891,
                899,  906,  914,  921,  927,  934,  940,  946,
                951,  956,  961,  966,  970,  974,  978,  982,
                985,  988,  990,  993,  995,  996,  998,  999,
                999, 1000, 1000, 1000,  999,  999,  998,  996,
                995,  993,  990,  988,  985,  982,  978,  974,
                970,  966,  961,  956,  951,  946,  940,  934,
                927,  921,  914,  906,  899,  891,  883,  875,
                866,  857,  848,  839,  829,  819,  809,  799,
                788,  777,  766,  755,  743,  731,  719,  707,
                695,  682,  669,  656,  643,  629,  616,  602,
                     588,  574,  559,  545,  530,  515,  500,  485,
                     469,  454,  438,  423,  407,  391,  375,  358,
                     342,  326,  309,  292,  276,  259,  242,  225,
                     208,  191,  174,  156,  139,  122,  105,   87,
                      70,   52,   35,   17,    0,  -17,  -35,  -52,
                     -70,  -87, -105, -122, -139, -156, -174, -191,
                    -208, -225, -242, -259, -276, -292, -309, -326,
                    -342, -358, -375, -391, -407, -423, -438, -454,
                    -469, -485, -500, -515, -530, -545, -559, -574,
                    -588, -602, -616, -629, -643, -656, -669, -682,
                    -695, -707, -719, -731, -743, -755, -766, -777,
                    -788, -799, -809, -819, -829, -839, -848, -857,
                    -866, -875, -883, -891, -899, -906, -914, -921,
                    -927, -934, -940, -946, -951, -956, -961, -966,
                    -970, -974, -978, -982, -985, -988, -990, -993,
                    -995, -996, -998, -999, -999,-1000,-1000,-1000,
                    -999, -999, -998, -996, -995, -993, -990, -988,
                    -985, -982, -978, -974, -970, -966, -961, -956,
                    -951, -946, -940, -934, -927, -921, -914, -906,
                    -899, -891, -883, -875, -866, -857, -848, -839,
                    -829, -819, -809, -799, -788, -777, -766, -755,
                    -743, -731, -719, -707, -695, -682, -669, -656,
                    -643, -629, -616, -602, -588, -574, -559, -545,
                    -530, -515, -500, -485, -469, -454, -438, -423,
                    -407, -391, -375, -358, -342, -326, -309, -292,
                    -276, -259, -242, -225, -208, -191, -174, -156,
                    -139, -122, -105,  -87,  -70,  -52,  -35,  -17
              ];

begin TableLookup;


    // The following touches all the items in
    // the sines array so that it is all in the cache.

    for( mov( 0, angle ); angle < 360; inc( angle )) do

        mov( angle, ebx );
        mov( sines[ ebx*4 ], eax );
        mov( eax, theSIN );

    endfor;

    // Compute the time required to look up the sines of the
    // angles 0..359:

    rdtsc();
    mov( eax, ecx );
    for( mov( 0, angle ); angle < 360; inc( angle )) do

        mov( angle, ebx );
        mov( sines[ ebx*4 ], eax );
        mov( eax, theSIN );

    endfor;
    rdtsc();
    sub( ecx, eax );
    stdout.put
    (
        “Time required to compute 360 sines via lookup:”,
        (type uns32 eax),
        nl
    );

    // Compute the time required to compute 360 sines using FSIN:


    rdtsc();
    mov( eax, ecx );
    for( mov( 0, angle ); angle < 360; inc( angle )) do

        fild( angle );
        fldpi();
        fmul();
        fld( 2.0 );
        fmul();
        fld( 180.0 );
        fdiv();
        fsin();
        fld( 1000.0 );
        fmul();
        fistp( theSIN );

    endfor;
    rdtsc();
    sub( ecx, eax );
    stdout.put
    (
        “Time required to compute 360 sines via FSIN:”,
        (type uns32 eax),
        nl
    );


end TableLookup;
```

Program 13.21 FSIN vs. Table Lookup

Exercise A: Compile and run this program. Compare the cycles counts between the two methods for computing SIN. Describe the differences in timing and comment on these differences in your laboratory report.

Exercise B: In one respect, this program cheats. The first look in the program “touches” all the memory locations in the *sines* table so that they are all loaded into the cache before the code actually times the execution of the lookup table code. Remove this initial loop and rerun the program. Comment on the difference in timing. Comment on the effectiveness of the cache. Is it worthwhile to use a lookup table if you rarely access that table (and it’s data is not in cache)? Justify your answer.