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.

