Coroutines and Generators   Chapter Three

3.1   Chapter Overview

This chapter discusses two special types of program units known as coroutines and generators. A coroutine is similar to a procedure and a generator is similar to a function. The principle difference between these program units and procedures/functions is that the call/return mechanism is different. Coroutines are especially useful for multiplayer games and other program flow where sections of code "take turns" executing. Generators, as their name implies, are useful for generating a sequence of values; in many respects generators are quite similar to HLA's iterators without all the restrictions of the iterators (i.e., you can only use iterators within a FOREACH loop).
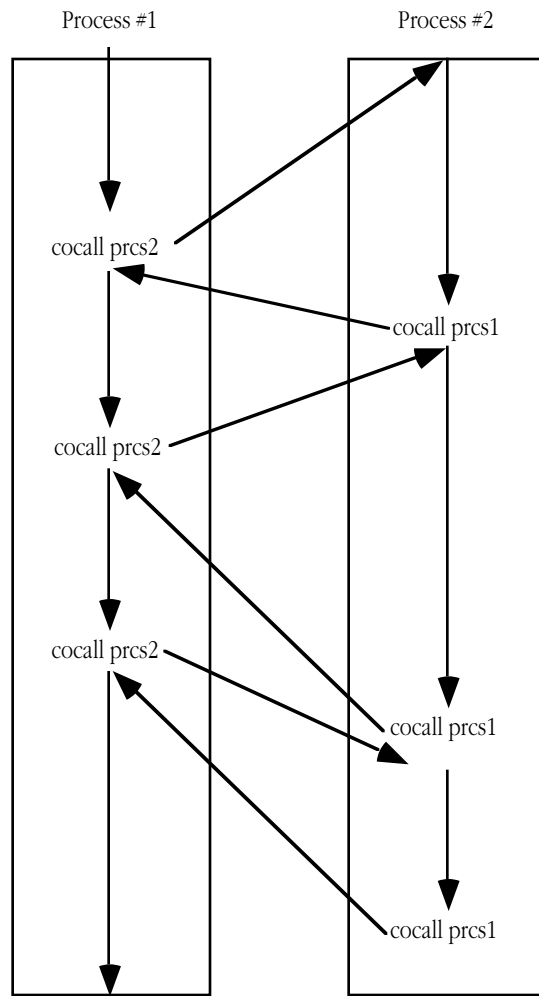
3.2   Coroutines

A common programming paradigm is for two sections of code to swap control of the CPU back and forth while executing. There are two ways to achieve this: preemptive and cooperative. In a preemptive system, two or more *processes* or *threads* take turns executing with the *task switch* occurring independently of the executing code. A later volume in this text will consider preemptive multitasking, where the Operating System takes responsibility for interrupting one task and transferring control to some other task. In this chapter we'll take a look at coroutines that explicitly transfer control to another section of the code[1]

When discussing coroutines, it is instructive to review how HLA's iterators work, since there is a strong correspondence between iterators and coroutines. Like iterators, there are four types of entries and returns associated with a coroutine:

- Initial entry. During the initial entry, the coroutine's caller sets up the stack and otherwise initializes the coroutine.
- Cocall to another coroutine / coreturn to the previous coroutine.
- Coreturn from another coroutine / cocall to the current coroutine.
- Final return from the coroutine (future calls require reinitialization).

A *cocall* operation transfers control between two coroutines. A cocall is effectively a call and a return instruction all rolled into one operation. From the point of view of the process executing the cocall, the cocall operation is equivalent to a procedure call; from the point of view of the processing being called, the cocall operation is equivalent to a return operation. When the second process cocalls the first, control resumes *not at the beginning of the first process*, but immediately after the last cocall operation from that coroutine (this is similar to returning from a FOREACH loop after a *yield* operation). If two processes execute a sequence of mutual cocalls, control will transfer between the two processes in the following fashion:

---

1. The term "cooperative" in this chapter doesn't not imply the use of that oxymoronic term "cooperative multitasking" that Microsoft and Apple used before they got their operating system acts together. Cooperative in this chapter means that two blocks of code explicitly pass control between one another. In a multiprogramming system (the proper technical term for "cooperative multitasking" the operating system still decides which program unit executes after some other thread of execution voluntarily gives up the CPU.
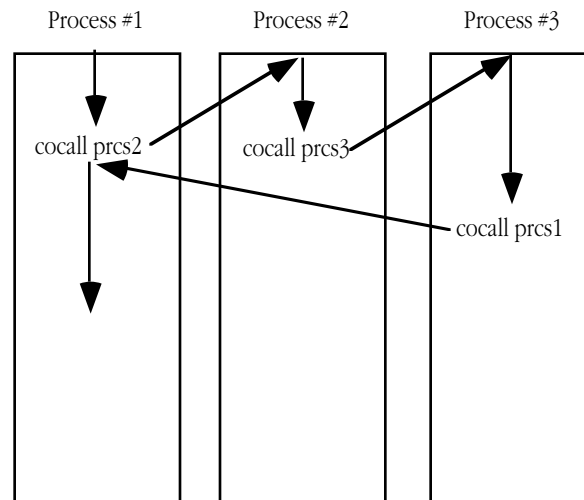
Process #1

Process #2

cocall prcs2

cocall prcs1

cocall prcs2

cocall prcs1

cocall prcs2

cocall prcs1

Cocall Sequence Between Two Processes

Figure 3.1   Cocall Sequence

Cocalls are quite useful for games where the "players" take turns, following different strategies. The first player executes some code to make its first move, then cocalls the second player and allows it to make a move. After the second player makes its move, it cocalls the first process and gives the first player its second move, picking up immediately after its cocall. This transfer of control bounces back and forth until one player wins.

Note, by the way, that a program may contain more than two coroutines. If coroutine one cocalls coroutine two, and coroutine two cocalls coroutine three, and then coroutine three cocalls coroutine one, coroutine one picks up immediately in coroutine one after the cocall it made to coroutine two.

Cocalls Between Three Processes

Figure 3.2   Cocalls Between Three Processes

Since a cocall effectively *returns* to the target coroutine, you might wonder what happens on the *first* cocall to any process. After all, if that process has not executed any code, there is no "return address" where you can resume execution. This is an easy problem to solve, we need only initialize the return address of such a process to the address of the first instruction to execute in that process.

A similar problem exists for the stack. When a program begins execution, the main program (coroutine one) takes control and uses the stack associated with the entire program. Since each process must have its own stack, where do the other coroutines get their stacks? There is also the question of "how much space should one reserve for each stack?" This, of course, varies with the application. If you have a simple application that doesn't use recursion or allocate any local variables on the stack, you could get by with as little as 256 bytes of stack space for a coroutine. On the other hand, if you have recursive routines or allocate storage on the stack, you will need considerably more space. But this is getting a little ahead of ourselves, how do we create and call coroutines in the first place?

HLA does not provide a special syntax for coroutines. Instead, the HLA Standard Library provides a class with set of procedures and methods (in the coroutines library module) that lets you turn any procedure (or set of procedures) into a coroutine. The name of this class is *coroutine* and whenever you want to create a coroutine object, you need to declare a variable of type *coroutine* to maintain important state information about that coroutine's thread of execution. Here are a couple of typical declarations that might appear within the VAR section of your main program:

```
var
    FirstPlayer: pointer to coroutine;
    OtherPlayer: coroutine;
```

Note that a coroutine variable is not the coroutine itself. Instead, the coroutine variable keeps track of the machine state when you switch between the declared coroutine and some other coroutine in the program (including the main program, which is a special case of a coroutine). The coroutine's "body" is a procedure that you write independently of the coroutine variable and associate with that coroutine object.

The coroutine class contains a constructor that uses the conventional name *coroutine.create*. This constructor requires two parameters and has the following prototype:

```
procedure coroutine.create( stacksize:dword;  body:procedure );
```

The first parameter specifies the size (in bytes) of the stack to allocate for this coroutine. The construction will allocate storage for the new coroutine in the system's heap using dynamic allocation (i.e., *malloc*). As a general rule, you should allocate at least 256 bytes of storage for the stack, more if your coroutine requires it (for local variables and return addressees). Remember, the system allocates all local variables in the coroutine (and in the procedures that the coroutine calls) on this stack; so you need to reserve sufficient space to accommodate these needs. Also note that if there are any recursive procedures in the coroutine's thread of execution, you will need some additional stack space to handle the recursive calls.

The second parameter is the address of the procedure where execution begins on the first cocall to this coroutine. Execution begins with the first executable statement of this procedure. If the procedure has some local variables, the procedure must build a stack frame (i.e., you shouldn't specify the NOFRAME procedure option). Procedures you execute via a cocall should never have any parameters (the calling code will not properly set up those parameters and references to the parameter list may crash the machine).

This constructor is a conventional HLA class constructor. On entry, if ESI contains a non-null value, the constructor assumes that it points at a coroutine class object and the constructor initializes that object. On the other hand, if ESI contains NULL upon entry into the constructor, then the constructor allocates new storage for a coroutine object on the heap, initializes that object, and returns a pointer to the object in the ESI register. The examples in this chapter will always assume dynamic allocation of the coroutine object (i.e., we'll use pointers).

To transfer control from one coroutine (including the main program) to another, you use the *coroutine.cocall* method. This method, which has no parameters, switches the thread of execution from the current coroutine to the coroutine object associated with the method. For example, if you're in the main program, the following two cocalls transfer control to the *FirstPlayer* and then the *OtherPlayer* coroutines:

```
FirstPlayer.cocall();
OtherPlayer.cocall();
```

There are two important things to note here. First, the syntax is not quite the same as a procedure call. You don't use cocall along with some operand that specifies which coroutine to transfer to (as you would if this were a CALL instruction); instead, you specify the coroutine object and invoke the cocall method for that object. The second thing to keep in mind is that these are coroutine transfers, not subroutine calls; therefore, the *FirstPlayer* coroutine doesn't necessarily return back to this code sequence. *FirstPlayer* could transfer control directly to *OtherPlayer* or some other coroutine once it finishes whatever tasks it's working on. Some coroutine has to explicitly transfer control to the thread of execution above in order for it to (directly) transfer control to *OtherPlayer*.

Although coroutines are more general than procedures and don't have to use the call/return semantics, it is perfectly possible to simulate and call and return exchange between two coroutines. If one coroutine calls another and that second coroutine cocalls the first, you get semantics that are very similar to a call/return (of course, on the next call to the second coroutine control resumes after the cocall, not at the start of the coroutine, but we will ignore that difference here). The only problem with this approach is that it is not general: both coroutines have to be aware of the other. Consider a cooperating pair of coroutines, *master* and *slave*. The *master* coroutine corresponds to the main program and the *slave* coroutine corresponds to a procedure that the main program calls. Unfortunately, *slave* is not general purpose like a standard procedure because it explicitly calls the *master* coroutine (at least, using the techniques we've seen thus far). Therefore, you cannot call it from an arbitrary coroutine and expect control to transfer back to that coroutine. If *slave* contains a cocall to the *master* coroutine it will transfer control there rather than back to the "calling" coroutine. Although these are the semantics we expect of coroutines, it would sometimes be nice if a coroutine could return to whomever invoked it without explicitly knowing who invoked it. While it is possible to set up some coroutine variables and pass this information between coroutines, the HLA Standard Library Coroutines Module provides a better solution: the *coret* procedure.

On each call to a coroutine, the coroutine run-time support code remembers the last coroutine that made a cocall. The *coret* procedure uses this information to transfer control back to the last coroutine that made a cocall. Therefore, the *slave* coroutine above can execute the coret procedure to transfer control back to whomever called it without knowing who that was.

Note that the *coret* procedure is not a member of the *coroutine* class. Therefore, you do not preface the call with "coroutine." You invoke it directly:

```
coret();
```

Another important issue to keep in mind with *coret* is that it only keeps track of the last cocall. It does not maintain a stack of coroutine "return addresses" (there are several different stacks in use by coroutines, on which one does it keep this information?). Therefore, you cannot make two cocalls in a row and then execute two corets to return control back to the original coroutine. If you need this facility, you're going to need to create and maintain your own stack of coroutine calls. Fortunately, the need for something like this is fairly rare. You generally don't use coroutines as though they were procedures and in the few instances where this is convenient, a single level of return address is usually sufficient.

By default, every HLA main program is a coroutine. Whenever you compile an HLA program (as opposed to a UNIT), the HLA compiler automatically inserts two pieces of extra code at the beginning of the main program. The first piece of extra code initializes the HLA exception handling system, the second sets up a coroutine variable for the main program. The purpose of this variable is to all other coroutines to transfer control back to the main program; after all, if you transfer control from one coroutine to another using a statement like *VarName.cocall*, you're going to need a coroutine variable associated with the main program in order to cocall the main program. HLA automatically creates and initializes this variable when execution of the main program begins. So the only question is, "how do you gain access to this variable?"

The answer is simply, really. Whenever you include the "coroutines.hhf" header file (or "stdlib.hhf" which automatically includes "coroutines.hhf") HLA declares a static coroutine variable for you that is associated with the main program's coroutine object. That declaration looks something like the following[2]:

```
static MainPgm:coroutine; external( "<<external name for MainPgm>>" );
```

Therefore, to transfer control from one coroutine to the main program's coroutine, you'd use a cocall like the following:

```
MainPgm.cocall();
```

The last method of interest to us in the *coroutine* class is the *coroutine.cofree* method. This is the destructor for the *coroutine* class. Calling this method frees up the stack storage associated with the coroutine and cleans up other state information associated with that coroutine. A typical call might look like the following:

```
OtherPlayer.cofree();
```

Warning: do not call the *cofree* method from within the coroutine you are freeing up. There is no guarantee that the stack and coroutine state variables remain valid for a given coroutine after you call *cofree*. Generally, it is a good idea to call the *cofree* method in the same code that originally created the coroutine via the *coroutine.create* call. It goes without saying that you must not call a coroutine after you've destroyed it via the *cofree* method call.

Remember that the *coret* procedure call is really a special form of *cocall*. Therefore, you need to be careful about executing *coret* after calling the *cofree* method as you may wind up "returning" to the coroutine you just destroyed.

After you call the *cofree* method, it is perfectly reasonable create a new coroutine using that same coroutine variable by once again calling the *coroutine.create* procedure. However, you should always ensure that you call the *cofree* method prior to calling *coroutine.create* or the stack space allocated in the original call will be lost in the system (i.e., you'll create a memory leak).

This is very important: you never "return" from a coroutine using a RET instruction (e.g., by "falling off the end of the procedure) or via the HLA EXIT/EXITIF statement. The only legal ways to "return" from a coroutine are via the *cocall* and *coret* operations. If you RETurn or EXIT from a coroutine, that coroutine enters a special mode that rejects any future cocalls and immediately returns control back to whomever

_____

2. The external name doesn't appear here because it is subject to change. See the coroutines.hhf header file if you need to know the actual external name for some reason.

cocalled it in the first place. Most coroutines contain an infinite loop that transfers control back to the start of the coroutine to repeat whatever function they perform once they complete all the code in the coroutine. You will probably want to implement this functionality in your coroutines as well.

3.3   Parameters and Register Values in Coroutine Calls

As you've probably noticed, coroutine calls via cocall don't support generic parameters for transferring data between two coroutines. There are a couple of reasons for this. First of all, passing parameters to a coroutine is difficult because we typically use the stack to pass parameters and coroutines all use different stacks. Therefore, the parameters one coroutine passes to another won't be on the correct stack (and, therefore, inaccessible) when the second coroutine continues execution. Another problem with passing parameters between coroutines is that a typical coroutine has several entry points (immediately after each cocall in that coroutine). Nevertheless, it is often important to communicate information between coroutines. We will explore ways to do that in this section.

If we can pass parameters on the stack, that basically leaves registers and global memory locations[3]. Registers are the easy and obvious solution. However, keep in mind that you have a limited set of registers available so you can't pass much data between coroutines in the registers. On the other hand, you can pass a pointer to a block of data in a register (see "Passing Parameters via a Parameter Block" on page 1353 for details).

Another place to pass parameters between coroutines is in global memory locations. Keep in mind that coroutines each have their own stack. Therefore, one coroutine does not have access to another coroutine's automatic variables appearing in a VAR section (this is true even if those VAR objects appear in the main program). Always use STATIC, STORAGE, or READONLY objects when communicating data between coroutines using global variables. If you must communicate an automatic or dynamic object, pass the address of that object in a register or some static global variable.

A bigger problem than where we pass parameters to a coroutine is "How do we deal with parameters we pass to a coroutine?" Parameters work well in procedures because we always enter the procedure at the same point and the state of the procedure is usually the same upon entry (with the possible exception of static variable values). This is not true for coroutines. Consider the following code that executes as a coroutine:

```
procedure IsACoroutine; nodisplay; noframe;
begin IsACoroutine;

    //*

    << Do something upon initial entry >>

    coret();    // Invoke previous coroutine
    //*

    << Do some more stuff upon return >>

    forever

        OtherCoroutine.cocall();  // Invoke a third coroutine.

        << Do some more stuff here >>

        MainPgm.cocall();         // Transfer control back to the main program.
        //*

        << do some stuff >>
```

---

3. The chapter on low-level parameter implementation in this volume discusses different places you can pass parameters between procedures. Check out that chapter for more details on this subject.

```
        coret();                        // Return to whomever cocalled us.
        //*

        << do some more stuff >>

    endfor;

end IsACoroutine;
```

In this code you'll find several comments of the form "//*". These comments mark the point at which some other coroutine can reenter this code. Note that, in general, the "calling" coroutine has no idea which entry point it will invoke and, likewise, this coroutine has no idea who invoked it at any given point. Passing in meaningful parameters and properly processing them under these conditions is difficult, at best. The only reasonable solution is to make every invocation pass exactly the same type of data in the same location and then write your coroutines to handle this data appropriately upon each entry. Even though this solution is more reasonable than the other possibilities, maintaining code like this is very difficult.

If you're really dead set on passing parameters to a coroutine, the best solution is to have a single entry point into the code so you've only got the handle the parameter data in one spot. Consider the following procedure that other threads invoke as a coroutine:

```
procedure HasAParm; nodisplay;
begin HasAParm;

    << Initialization code goes here, assume no parameter >>
    forever

        coret();  // Or cocall some other coroutine.

        << deal with parameter data passed into this coroutine >>

    endfor;

end HasAParm;
```

Note that there are two entry points into this code: the first occurs on the initial entry. The other occurs whenever the *coret*() procedure returns via a *cocall* to this coroutine. Immediately after the *coret* statement, the code above can process whatever parameter data the calling code has set up. After processing that data, this code returns to the invoking coroutine and that coroutine (directly or indirectly) can invoke this code again with more data.

3.4   Recursion, Reentrancy, and Variables

The fact that each coroutine has its own stack impacts access to variables from within coroutines. In particular, the only automatic (VAR) objects you can normally access are those declared within the coroutine itself and any procedures it calls. In a later chapter of this volume we'll take a look at nested procedures and how one could access local variables outside of the current procedure. For the most part, that discussion does not apply to procedures that are coroutines.

If you wish to share information between two or more coroutines, the best place to put such information is in a static object (STATIC, READONLY, and STORAGE variables). Such data does not appear on a stack and is accessible to multiple coroutines (and other procedures) simultaneously.

Whenever you execute a cocall instruction, the system suspends the current thread of execution and switches to a different coroutine, effectively by returning to that other coroutine and picking up where it left off (via a *coret* or *cocall* operation). This means that it isn't really possible to recursively *cocall* some coroutine. Consider the following (vain) attempt to achieve this:

```
procedure IsACoroutine; nodisplay;
```

```
begin IsACoroutine;

    << Do some initial stuff >>

    IAC.cocall();  // Note: IAC is initialized with the address of IsACoroutine.

    << Do some more stuff >>

end IsACoroutine;
```

This code assumes that *IAC* is a coroutine variable initialized with the address of the *IsACoroutine* proce-dure. The *IAC.cocall* statement, therefore, is an attempt to recursively call this coroutine. However, all that happens is that this call leaves the *IsACoroutine* code and then the coroutine system passes control to where *IAC* last left off. That just happens to be the *IAC.cocall* statement that just left the *IsACoroutine* procedure. Therefore this code immediately returns back to itself.

Although the idea of a recursive coroutine doesn't make sense, it is certainly possible to call procedures from within a coroutine and those procedures can be recursive. You can even make cocalls from those (recursive) procedures and the coroutine system will automatically return back into the recursive calls when control transfers back to the coroutine.

Although coroutines are not recursive, it is quite possible for the coroutine run-time system to reenter a procedure that is a coroutine. This situation can occur when you have two coroutine variables, initialize them both with the address of the same procedure, and then execute a cocall to each of the coroutine objects. consider the following simple example:

```
procedure Reentered; nodisplay;
begin Reentered;

    << do some initialization or other work >>

    coret();

    << do some other stuff >>

end Reentered;
        .
        .
        .
    CV1.create( 256, &Reentered );  // Initialize two coroutine variables with
    CV2.create( 256, &Reentered );  // the address of the same procedure.
    CV1.cocall();                    // Start the first coroutine.
    CV2.cocall();                    // Start the second coroutine.

    << At this point, both CV1 and CV2 are suspended within Reentered >>
```

Notice at the end of this code sequence, both the CV1 and CV2 coroutines are executing inside the Reen-tered procedure. That is, if you cocall either one of them, the cocalled routine will continue execution after the *coret* statement. This is not an example of a recursive coroutine call, but it certainly demonstrates that you can reenter some procedure while some other coroutine is currently executing the code within that pro-cedure.

Reentering some code (whether you do this via a coroutine call or some other mechanism) is a perfectly reasonable thing to do. Indeed, recursion is one example of reentrancy. However, there are some special considerations you must be aware of when it is possible to reenter some code.

The principal issue is the use of variables in reentrant code. Suppose the *Reentered* procedure above had the following declarations:

```
var
    i: int32;
    j: uns32;
```

One concern you might have is that the two different coroutines executing in the same procedure would share these variables. However, keep in mind that HLA allocates automatic variables on the stack. Since each coroutine has its own stack, they're going to get their own private copies of the variables. Therefore, if *CV1* stores a value into *i* and *j*, *CV2* will not see these values. While this may seem to be a problem, this is actually what you want. You generally don't want one coroutine's thread of execution affecting the calculation in a different thread of execution.

The discussion above applies only to automatic variables. HLA does not allocate static objects (those you declare in the STATIC, READONLY, and STORAGE sections) on the stack. Therefore, such variables are not associated with a specific coroutine; instead, all coroutines that are executing in the same procedure share the same variables. Therefore, you shouldn't use static variables within a procedure that serves as a coroutine (especially reentrant coroutines) unless you explicitly want to share that data among other coroutines or other procedures.

Coroutines have their own stack and maintain that stack between cocalls to other coroutines. Therefore, like iterators, coroutines maintain their state (including the value of automatic variables) across cocalls. This is true even if you leave a coroutine via some other procedure than the main procedure for the coroutine. For example, suppose coroutine *A* calls some procedure *B* and then within procedure *B* there is a cocall to some other coroutine *C*. Whenever coroutine *C* executes *coret* or some coroutine (including *C*) cocalls *A*, control transfers back into procedure *B* and procedure *B's* state is maintained (including the values of all local variables *B* initialize prior to the cocall). When *B* executes a return instruction, it will return back to procedure *A* who originally called *B*.

In theory it's even possible to call a procedure as well as cocall that procedure (it's hard to imagine why you would want to do this and it's probably quite difficult to pull it off correctly, but it's certainly possible). This is such a bizarre situation that we won't consider it any farther here.

3.5   Generators

A generator is to a function what a coroutine is to a procedure. That is, the whole purpose of a generator is to return a value like a function result. As far as HLA and the Coroutines Library Module is concerned, there is absolutely no difference between a generator and a coroutine (anymore than there is a syntactical difference between a function and a procedure to HLA). Clearly, there are some semantic differences; this section will describe the semantics of a generator and propose a convention for generator implementation.

The best way to describe a generator is to begin with the discussion of a special-purpose generator object that HLA does support – the iterator. An iterator is a special form of a generator that does not require its own stack. Iterators share the same stack as the calling code (i.e., the code containing the FOREACH loop that invokes the iterator). Because of the semantics of the FOREACH loop, iterators can leave their activation records on the stack and, therefore, maintain their local state between exits to the FOREACH loop body. The disadvantage to this scheme is that the calling semantics of an iterator are very rigidly defined; you cannot call an iterator from an arbitrary point in a program and the iterator's state is preserved only for the execution of the FOREACH loop.

By using its own stack, a generator removes these restrictions. You can call a generator from any point in the program (except, of course, within the generator itself – remember, recursive coroutines are not possible). Also, the state (i.e., the activation record) of a generator is not tied to the execution of some syntactical item like a FOREACH loop. The generator maintains its local state from the point of its first call to the point you call *cofree* on that generator.

One major difference between iterators and generators is the fact that generators don't use the *yield* statement (thunk) to return results back to the calling code. To send a value back to whomever invokes the generator, the generator must cocall the original coroutine. Since one can call a generator from different points in the code, and in particular, from different coroutines, the typical way to "return" a value back to the "caller" is to use the *coret* procedure call after loading the return result into a register.

A typical generator does not use the *cocall* operation. The *cocall* method transfers control to some other (explicitly defined) coroutine. As a general rule, generators (like functions) return control to whomever

called them. They do not explicitly pass control through to some other coroutine. Keep in mind that the *coret* procedure only returns control to the last coroutine. Therefore, if some generator passes control to another coroutine, there is no way to anonymously return back to whomever called the generator in the first place. That information is lost at the point of the second *cocall* operator. Since the main purpose of a generator is to return a value to whomever cocalled it, finding cocalls in a generator would be unusual indeed.

One problem with generators is that, like the coroutines upon which they are based, you cannot pass parameters to a generator via the stack. In most cases this is perfectly acceptable. As their name implies, generators typically generate an independent stream of data once they begin execution. After initialization, a generator generally doesn't require any additional information in order to generate its data sequence. Although this is the typical case, it is not the only case; sometimes you may need to write generators that need parameter data on each call. So what's the best way to handle this?

In a previous section (see "Parameters and Register Values in Coroutine Calls" on page 1334) we discussed a couple of ways to pass parameters to coroutines. While those techniques apply here as well, they are not particularly convenient (certainly not as convenient as passing parameters to a standard HLA procedure). Because of their function-like nature, it is more common to have to pass parameters to a generator (versus a generic coroutine) and you'll probably make more calls to generators that require parameters (versus similar calls to coroutines). Therefore, it would be nice if there were a "high-level" way of passing parameters to generators. Well, with a couple of tricks we can easily accomplish this[4].

Remember that we cannot pass parameters to a coroutine (or generator) on the stack. The most convenient place to pass coroutine parameters is in registers or in static, global, memory locations. Unfortunately, writing a sequence of instructions to load up registers with parameter values (or, worse yet, copy the parameter data to global variables) prior to invoking a generator is a real pain. Fortunately, HLA's macros come to the rescue here; we can easily write a macro that lets us invoke a generator using a high level syntax and the macro can take care of the dirty work of loading registers (or global memory locations) with the necessary values. As an example, consider a generator, *_MyGen*, that expects two parameters in the EAX and EBX registers. Here's a macro, *MyGen*, that sets up the registers and invokes this generator:

```
#macro MyGen( ParmForEAX, ParmForEBX );

    mov( ParmForEAX, eax );
    mov( ParmForEBX, ebx );
    _MyGen.cocall();

#endmacro;
    .
    .
    .
MyGen( 5, i );
```

You could, with just a tiny bit more effort, pass the parameters in global memory locations using this same technique.

In a few situations, you'll really need to pass parameters to a generator on the stack. We'll not go into the reasons or details here, but there are some rare circumstances where this is necessary. In many other circumstances, it may not be necessary but it's certainly more convenient to pass the parameters on the stack because you get to take advantage of HLA's high level parameter passing syntax when you use this scheme (i.e., you get to choose the parameter passing mechanism and HLA will automatically handle a lot of the gory details behind parameter passing for you when you use the stack). The best solution in this situation is to write a wrapper procedure. A wrapper procedure is a short procedure that reorganizes a parameter list before calling some other procedure. The macro above is a simple example of a wrapper – it takes two the (text) parameters and moves their run-time data into EAX and EBX prior to cocalling *_MyGen*. We could have just as easily written a procedure to accomplish this same task:

```
procedure MyGen( ParmForEAX:dword; ParmForEBX:dword ); nodisplay;
begin MyGen;
```

---

4. By the way, generators are coroutines, so these tricks apply to generic coroutines as well.

```
    mov( ParmForEAX, eax );
    mov( ParmForEBX, ebx );
    _MyGen.cocall();

end MyGen;
```

Beyond the obvious time/space trade-offs between macros and procedures, there is one other big difference between these two schemes: the procedure variation allows you to specify a parameter passing mechanism like pass by reference, pass by value/result, pass by name, etc[5]. Once HLA knows the parameter passing mechanism, it can automatically emit code to process the actual parameters for you. Suppose, for example, you needed pass by value/result semantics. Using the macro invocation, you'd have to explicitly write a lot of code to pull this off. In the procedure above, about the only change you'd need is to add some code to store any returned results back into *ParmForEAX* or *ParmForEBX* (whichever uses the pass by value/result mechanism).

Since coroutines and generators share the same memory address space as the calling coroutine, it is not correct to say that a coroutine or generator does not have access to the stack of the calling code. The stack is in the same address space as the coroutine/generator; the only problem is that the coroutine doesn't know exactly where any parameters may be sitting in that memory space. This is because procedures use the value in ESP[6] to indirectly reference the parameters passed on the stack and, unfortunately, the *cocall* method changes the value of the ESP register upon entry into the coroutine/generator. However, were we to pass the original value of ESP (or some other pointer into an activation record) through to a generator, then it would have direct access to those values on the stack. Consider the following modification to the *MyGen* procedure above:

```
procedure MyGen( ParmForEAX:dword; ParmForEBX:dword ); nodisplay;
begin MyGen;

    mov( ebp, ebx );
    _MyGen.cocall();

end MyGen;
```

Notice that this code does not directly copy the two parameters into some locations that are directly accessible in the generator. Instead, this procedure simply copies the base address of *MyGen's* activation record (the value in EBP) into the EBX register for use in the generator. To gain access to those parameters, the generator need only index off of EBX using appropriate offsets for the two parameters in the activation record. Perhaps the easiest way to do this is by declaring an explicit record declaration that corresponds to *MyGen's* activation record:

```
type
    MyGenAR:
        record
            OldEBP:     dword;
            RtnAdrs:    dword;
            ParmForEAX: dword;
            ParmForEBX: dword;
        endrecord;
```

Now, within the *_MyGen* generator code, you can access the parameters on the stack using code like the following:

```
    mov( (type MyGenAR [ebx]).ParmForEAX, eax );
    mov( (type MyGenAR [ebx]).ParmForEBX, edx );
```

---

5. For a discussion of pass by value/result and pass by name parameter passing mechanisms, see the chapter on low-level parameter implementation in this volume.

6. Okay, a procedure typically uses the value in EBP, but that same procedure also loads EBP with the value of ESP in the standard entry sequence.

This scheme works great for pass by value and pass by reference parameters (those we've seen up to this point). There are other parameter passing mechanisms, this scheme doesn't work as well for those other parameter passing mechanisms. Fortunately, you won't use those other parameter passing methods anywhere near as often as pass by value and pass by name.

3.6   Exceptions and Coroutines

Exceptions represent a special problem in coroutines. The HLA TRY..ENDTRY statement typically surrounds a block of statements one wishes to protect from errant code. The TRY.. ENDTRY statement is a dynamic control structure insofar as this statement also protects any procedures you call from within the TRY..ENDTRY block. So the obvious question is "does the TRY..ENDTRY statement protect a coroutine you call from within such a block as well?" The short answer is "no, it does not."

Actually, in the first implementation of the HLA coroutines module, the exception handling system did pass control from one coroutine to another whenever an exception occurred. However, it became immediately obvious that this behavior causes non-intuitive results. Coroutines tend to be independent entities and to have one coroutine transfer control to another without an explicit cocall creates some problems. Therefore, the HLA compiler and the coroutines module now treat each coroutine as a standalone entity as far as exceptions are concerned. If an exception occurs within some coroutine and there isn't an outstanding TRY..ENDTRY block active for that coroutine, then the system behaves as though there were no active TRY..ENDTRY at all, *even if there is an active TRY..ENDTRY block in another coroutine*; in other words, the program aborts execution. Keep this in mind when using exception handling in your coroutines. For more details on exception handling, see the chapter on Exception Handling in this volume.

3.7   Putting It All Together

This chapter discusses a novel program unit – the coroutine. Coroutines have many special properties that make them especially valuable in certain situations. Coroutines are not built into the HLA language. Rather, HLA implements them via the HLA Coroutines Module in the HLA Standard Library. This chapter began by discussing the methods found in that library module. Next, this chapter discusses the use of variables, recursion, reentrancy and machine state in a coroutine. This chapter also discusses how to create generators using coroutines and pass parameters to a generator in a function-like fashion. Finally, this chapter briefly discussed the use of the TRY..ENDTRY statement in coroutines.

Coroutines and generators are like iterators insofar as they are control structures that few high level languages implement. Therefore, most programmers are unfamiliar with the concept of a coroutine. This, unfortunately, leads to the lack of consideration of coroutines in a program, even where a coroutine is the most suitable control structure to use. You should avoid this trap and learn how to use coroutines and generators properly so that you'll know when to use them when the need arises.