# 9 Coroutines Module (coroutines.hhf)

HLA provides a powerful coroutines class that lets you easily use coroutines in your programs. The coroutine class provides three procedures and methods you can use to initialize a coroutine, transfer control between coroutines, and free up the storage associated with a coroutine when it completes execution. The coroutine class also has several data fields, but you should treat these as private fields and never disturb their values.

In addition to these class procedures and methods, the coroutine package provides a coret procedure that is useful for returning from a coroutine to whomever "cocalled" the coroutine. This makes it very easy to implement *Generators* using coroutines.

Finally, the coroutine module provides a special coroutine variable, mainPgm, that you can use to cocall the "coroutine" corresponding to the main HLA program.

## 9.1 The Coroutine Module

To use the coroutine functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

#include( "coroutines.hhf" )

or

#include( "stdlib.hhf" )

## 9.2 The Coroutine Class Definition

Here's the definition of the coroutine class data type:

```
// Note: the original declaration was "coroutine"
// but this has been deprecated. The following text
// equate is for legacy code. Someday, this declaration
// will go away.

  const
     coroutine:text := "coroutine_t";

type
    coroutine_t:
        class

            var
                CurrentSP:          dword;
                Stack:              dword;
                ExceptionContext:   dword;
                LastCaller:         dword;

            procedure cocall;
                @external( "COR_COCALL" );

            procedure create( size:uns32; theProc:procedure );
                @external( "COR_CREATE" );

            method cofree;
                @external( "COR_COFREE" );

        endclass;
```

The data fields are all private fields to this class, your applications should not modify these fields. In addition to the two procedures and the method in this class, the coroutines.hhf header file also defines a single external procedure and an external coroutine variable:

```
procedure coret; @external( "COR_CORET" );

    static
      mainPgm_coroutine:coroutine_t; @external( "MainPgmCoroutine__hla_" );
```

## 9.3 Coroutine Functions

**procedure coroutine_t.create( size:uns32; theProc:procedure );**

*coroutine_t.create* is the typical HLA class constructor for the coroutine class. Since this is a class procedure, you can call create one of two different ways:

(1) You can call it via the statement "coroutine_t.create( size, proc);" This form assumes that you wish to create a dynamic coroutine object on the heap. When called this way, the *coroutine_t.create* procedure allocates storage for a coroutine object on the heap and returns a pointer to this new coroutine object in the ESI register. Otherwise it behaves identically to the second form of the *coroutine_t.create* procedure.

(2) You can call *coroutine_t.create* using an invocation of the form "objectName.create( size, proc);" where "objectName" is the name of a coroutine_t variable or a pointer to a coroutine_t object (that, presumably, has been initialized with a valid pointer to a coroutine_t object). Do be aware that this form of the call loads ESI with the address of the coroutine_t object. On return, ESI will contain this new value.

Either form of the call to create will initialize the coroutine_t object, allowing subsequent cocalls to the coroutine_t object.

Coroutines execute using their own stack (independent of other coroutine stacks and independent of the stack the main program uses). The *size* parameter specifies the number of bytes of stack space to reserve for the coroutine. A good minimum value for a coroutine stack is between 256 and 1,024 bytes. If the coroutine allocates lots of local/automatic variables, or calls other procedures that allocate lots of local/automatic storage, you will need to allocate a larger stack as appropriate. Likewise, if your coroutine calls procedures that are recursive, additional stack space may be necessary.

The *theProc* parameter is a pointer to a procedure. This procedure is the code that will execute when you cocall this coroutine. The only thing special about the procedure is that it should never be possible to return to the procedure's caller by executing a RET instruction. You exit coroutine using the *coroutine_t.cocall* procedure or the *coroutine_t.coret* procedure. If your code accidentally "falls off the end of the procedure" or otherwise attempts to return to the caller via a RET instruction, the coroutine will go into a special state in which any attempt to cocall it forces an immediate return by the coroutine to the cocaller.

Object declarations for examples:

```
static
  ptrToCoroutine:pointer to coroutine;
  staticCoroutine:coroutine_t;

HLA high-level calling sequence examples:

coroutine_t.create( 1024, &myCoroutineProc );
mov( esi, ptrToCoroutine );
staticCoroutine.create( 1024, &anotherProc );

HLA low-level calling sequence examples:

  pushd( 1024 );
  pushd( &myCoroutineProc );
  mov( NULL, esi );// Tells create to allocate storage
  call coroutine_t.create;
```

```
    mov( esi, ptrToCoroutine );

    pushd( 1024 );
    pushd( &anotherProc );
    lea( esi, staticCoroutine );
    call coroutine.create;
```

**procedure coroutine_t.cocall();**

*coroutine_t.cocall* is the mechanism you use to invoke a coroutine. Note that this is a procedure for performance reasons. You should never invoke the static procedure *coroutine_t.cocall* as this will raise a run-time exception. Instead, you should always invoke this procedure using an object invocation of the form "objectName.cocall();" This will switch the thread of execution from the current coroutine (or the main program) to the coroutine code associated with "objectName". Note that coroutines rarely begin execution at the first statement of the procedure associated with the coroutine (in fact, this happens exactly once, when you invoke the coroutine for the very first time).

The cocall mechanism provides the standard way of leaving a coroutine. Cocalling some other coroutine switches the execution context from the current coroutine to that other coroutine. The next time some code cocalls a coroutine that leaves via cocall, execution continues with the first statement following the cocall (it's almost as though you had called that other coroutine using a CALL instruction).

```
HLA high-level calling sequence examples:

ptrToCoroutine.cocall();
staticCoroutine.cocall();

HLA low-level calling sequence examples:

  mov( ptrToCoroutine, esi );
  call coroutine_t.cocall;

  lea( esi, staticCoroutine );
  call coroutine_t.cocall;
```

**method coroutine_t.cofree();**

When you are done with a coroutine, you should call the *coroutine_t.cofree* method to free up the stack space associated with that coroutine. You must not call *coroutine_t.cofree* from inside the coroutine you're cleaning up since it still needs its stack to transfer control to some other coroutine.

```
HLA high-level calling sequence examples:

ptrToCoroutine.cofree();
staticCoroutine.cofree();

HLA low-level calling sequence examples:

  mov( ptrToCoroutine, esi );
  mov( [esi], edi );
  call( [edi+@offset(coroutine_t.cofree)] );

  lea( esi, staticCoroutine );
  mov( [esi], edi );
  call( [edi+@offset(coroutine_t.cofree)] );
```

**method coret();**

*coret* is nearly identical to *coroutine_t.cocall* with two major exceptions. First, note that this procedure is not a member of the *coroutine_t* class. Therefore, you do not specify an object name in front of the call to the *coret* procedure. Second, *coret* returns control to whomever cocalled the current coroutine. The current coroutine does not have to know who called it; coret figures this out and cocalls the appropriate coroutine.

Note that *coret* is not a "return" in the usual sense that the coroutine completes execution upon calling *coret*. *coret* is identical to a *coroutine_t.cocall* to the coroutine that called the current coroutine. In particular, after a coroutine returns to another, any future cocalls to this coroutine will continue execution with the first statement following the *coret* call.

```
HLA high-level calling sequence examples:

coret();

HLA low-level calling sequence examples:

  call coret;
```

**static mainPgm:coroutine_t;**

This is a special coroutine_t variable that contains the control information for the main program. If, inside a coroutine, you wish to cocall the main program, just use a cocall of the form "MainPgm.cocall();" and control in the main program will continue at the point of the last cocall executed in the main program. (Note: the term "main program" here does not imply that the cocall has to be in the actual main program of an HLA program, it simply refers to the thread of execution that starts in the main program. Your main program can call a procedure that transfers control to some coroutine via cocall. *MainPgm.cocall* will transfer control back into that procedure.)