



COMPUTER SOLUTIONS, INC.

## Multitasking with Multi-Engines in OpenInsight

### Introduction

At the time this paper is being written OpenInsight 7.0 has been officially released to the developer community for a few months. Its code name, "Leap Frog", is a dual reference to the jump in version numbers (from 4.1.3a to 7.0) as well as the IDE and technological advances that were introduced.

However, this title might have just as well applied to version 4.1. Many significant improvements were introduced which largely seem to have gone unnoticed, perhaps because the Revelation community is not ready to exploit them yet. One of these features, OLE/OCX support, has been documented in our [Using OLE in OpenInsight](#) white paper.

This document will be the first in a possible series related to the new engine (aka OpenEngine.) It could be argued that the benefits from this enhancement are the most difficult to immediately visualize compared to the others. On the other hand, it could equally be said that the new OpenEngine can provide the greatest potential for enhancing the way OpenInsight applications are developed.

### OpenEngine v1.0: An Idea Ahead of its Time

Much of the following discussion will be familiar to veteran OpenInsight programmers. This is especially true for any who have already reviewed the [OI 4.1 New Features.pdf](#) document (see the **References** section below) that came with the first 4.1 upgrade. If you are generally comfortable with the distinctions between the old and new OpenEngine then you may want to go directly to the **Getting Started** section below.

From its humble v1.0 beginnings, OpenEngine (i.e. OENGINE.exe) was designed so that it could stand alone and communicate with a programmer's IDE of choice. This essentially gave the developer a "client/server" approach to designing OpenInsight applications. As a result, less effort was put into the OpenInsight toolset in lieu of focusing on how OpenEngine can be used by Microsoft C, Visual Basic, Turbo Pascal, and other Windows application tools.

While this approach provided a base for a very powerful marriage of the best GUI *client* tools and the (in our opinion) best database engine *server*, it did not help bring the Revelation community on board. Most developers who were interested in OpenInsight were experienced AREV users hoping that this would be their easy pathway for migrating into the Windows platform. Consequently, they were looking for a similar self-contained, easy to use, and robust toolset that AREV offered (or, as some have verbalized, "AREV for Windows".) Another setback for OpenInsight was the absence of third-party tools that Revelation Software had intended to be released with v1.0. Thus, without any bundled tools and a requirement to learn another programming language, the first release of OpenInsight was virtually unusable to most veteran AREV programmers.

Fortunately, Revelation Software made the decision to develop these critical tools and release OpenInsight 2.0 as a complete (well, almost) IDE. Now that virtually everything necessary to design an application (i.e. Table Builder, Database Manager, System Editor, Debugger, Form Designer, and Report Builder) was built-in, the clock began for many AREV programmers to start making the OpenInsight trek.



COMPUTER SOLUTIONS, INC.

Despite this improvement to the native toolset, the ability for OpenEngine to talk to other IDEs still existed. Some developers have used alternative modern IDEs like Visual Basic, Delphi, and Visual C++ to work with OpenEngine to create applications. This ability to keep the "open" in OpenEngine paved the way for the OECGI, Linear Hash ODBC driver, and the XREV com object.

## OpenEngine v4.1 and Beyond

As many programmers of OpenInsight v4.0.x and below already know, there are really two primary executables: OINSIGHT.exe (OpenInsight) and OENGINE.exe (OpenEngine.) Normally we only concern ourselves with OINSIGHT.exe since that is what our shortcuts launch. We are spared the trouble of having to launch OENGINE.exe because it is automatically executed by OINSIGHT.exe via DDE.

With OpenInsight 4.1, the primary functions of OpenEngine have been transferred to OENGINE.dll. This includes the serial number and user count. As a DLL, OpenInsight can now load OpenEngine "in-process" which eliminates the need to use DDE as a communication layer between the two. This also has the benefit of improved speed in most situations.

OENGINE.exe still exists, though its purpose is to load OENGINE.dll so it can still run as a "stand-alone" or "out of process" engine (i.e. sans OINSIGHT.exe) waiting for a connection. This is partly for the benefit of allowing other IDEs the ability to communicate with OpenEngine. Perhaps the most significant purpose, however, is so OpenInsight applications can connect *remotely* to these "out of process" engines. Of course, since the advantage of an "in-process" OpenEngine was just cited, one might ask why an "out of process" engine would be used.

As it turns out, OpenInsight can communicate with "out of process" engines (subject to certain requirements listed in the Requirements section below) whether running on the same machine, the office file server, a web server, or any machine with a public IP address. This opens up several possibilities which were difficult, if not impossible, with previous versions of OpenInsight. Here are a few examples of what applications can now be designed to do:

- Connect to a dedicated engine running on an application server that is designed to access data or monitor activity in a folder that is hidden from the client workstation. This information can then be sent back to the locally running OpenInsight.
- Launch another engine on the local machine to perform lengthy database selects. OpenInsight and the "in-process" engine are then allowed to perform other tasks while the selection is completing.
- Use a web server (or any machine with a public URL/IP address) to store application updates. Vertical applications can connect to the engine running on this machine for available system upgrades. These can then be transferred through the connection directly to the local copy of OpenInsight.
- Synchronize satellite offices (who are running their own internal OpenInsight application) with the parent company's centralized system via a dedicated engine through VPN. This can be real-time using an MFS or at scheduled periods using a batch process.

At first glance each of the above scenarios seems very different. Some of the examples work with already running engines while others dynamically launch engines. Some work with local engines while others



COMPUTER SOLUTIONS, INC.

connect to engines running on a different machine. Despite these differences, they all use the same basic logic to accomplish their tasks. Therefore, it will be well worth the effort to build a generic or "black box" system for working with multiple engines. This white paper will help you to design this foundation.

## Getting Started

Our intent is to provide you everything you need in this document to begin designing applications capable of using multiple engines. Nevertheless, you might be interested in getting a copy of the previously mentioned [OI\\_4.1\\_New\\_Features.pdf](#) document. In some areas it has more detail regarding the new OpenEngine which might be helpful to programmers who desire to dig a little deeper.

When planning the use of multiple or remote engines in an application, the following questions should be considered:

1. Where is the engine located? For instance, is it on the same machine, a UNC path, or on an IP/URL resource?
2. What is the type and name of the connection for the engine?
3. What does the engine require for login purposes?
4. What routine will need to be executed?
5. Should the local OpenInsight wait until the remote engine is done processing?
6. When the remote process is done is there a need for a callback in the local OpenInsight?

Once our "black box" has been designed, each of the above items will become an input (or parameter) that will direct how the remote engine is accessed and used. However, before any programming can take place, the following items need to be added or updated to your system:

- OENGINE\_DLL prototype record in the SYSPROCS table.
- REVCAPI\_EQUATES insert record in the SYSPROCS table.

Complete versions of the above records are available below in the **References** section below.

## Putting It All Together

With our functions and inserts in place we can now begin to have some fun. One task that has long plagued OpenInsight applications is the long database select (of course, all the optimization experts out there are thinking, "If they had designed their data structures and indexes correctly this wouldn't be a problem..." Well, perhaps so. But we'll work on the assumption that this is still a minor annoyance in many applications.

In a typical LAN setup, we have two options for using the new OpenEngine to improve our situation.

### Option #1

1. OpenInsight makes a connection to a dedicated engine running on the application server.
2. A request to make a select against a table is made.
3. OpenInsight then waits for a response from the engine that the selection is done.



Option #2

1. OpenInsight launches its own "out of process" engine which runs on the local workstation.
2. A request to make a select against a table is made.
3. OpenInsight then waits for a response from the engine that the selection is done.

It would appear that the only difference between these two options is in the first step. In one sense this is correct. However, the differences are actually very minimal. This is because the logic needed to launch a new local engine and the logic needed to connect to a remotely running engine is essentially the same. Because of this similarity, and for the sake of convenience, we will be using the phrase "remote engine" for all forms of multiple engine connections (i.e. whether it is a local engine or an engine on another machine.) A "remote engine" is also synonymous with an "out of process" engine.

**Getting the Big Picture**

To help visualize how everything will fit and work together, we should outline the routines that will be used and in what order they will be executed:

Step	Description	Routine Path #1	Routine Path #2	Routine Path #3
1	Make a connection to an "out of process" engine.	CreateEngine	Same	Same
2	Login to the database.	CreateQueue	Same	Same
3	Execute a stored procedure.	CreateRequest	CallFunction	CallSubroutine
4	Get feedback on the progress from the remote engine.	PollForReply		
5	Get the description of the progress or any errors that occurred.	GetReply/ GetStatusText		
6	Close the connection that executed our stored procedure.	CloseRequest		
7	Close the connection that logged us into the database.	CloseQueue	Same	Same
8	Close the connection to the engine.	CloseEngine	Same	Same
9	Execute a callback routine if needed.	Call @SubName	Same	Same

In the chart above are three "paths" that we can take. Path #1 requires more programming to implement, but it allows the developer to return control immediately to the local application. Thus, the end user can continue to work within OpenInsight (i.e. multitask) while the remote engine is processing its request. Path #2 and Path #3 require relatively little programming. However, the local OpenInsight application is suspended until the remote engine has completed its task. Hence, the only functional benefit of following these paths is the ability to request information from an engine that the local "in-process" engine is incapable of doing.

Since our desire is to maximize the functionality of using remote engines, we will use Path #1. Again, since we are building a "black box", this should only need to be done once.

## Making a Connection

Our first task is to connect to a remote engine. Here are the three possibilities that can exist:

1. Connect to a remote engine is already running and waiting on another machine.
2. Dynamically launch a remote engine on the local machine and then connect to it.
3. Connect to a remote engine is already running and waiting on the local machine.

Note that there is no mention to launch dynamically launching a remote engine on another machine. For obvious security reasons this is not possible. While item #3 has been listed as an option, it is probably unlikely that this configuration will ever be used except for testing and R&D.

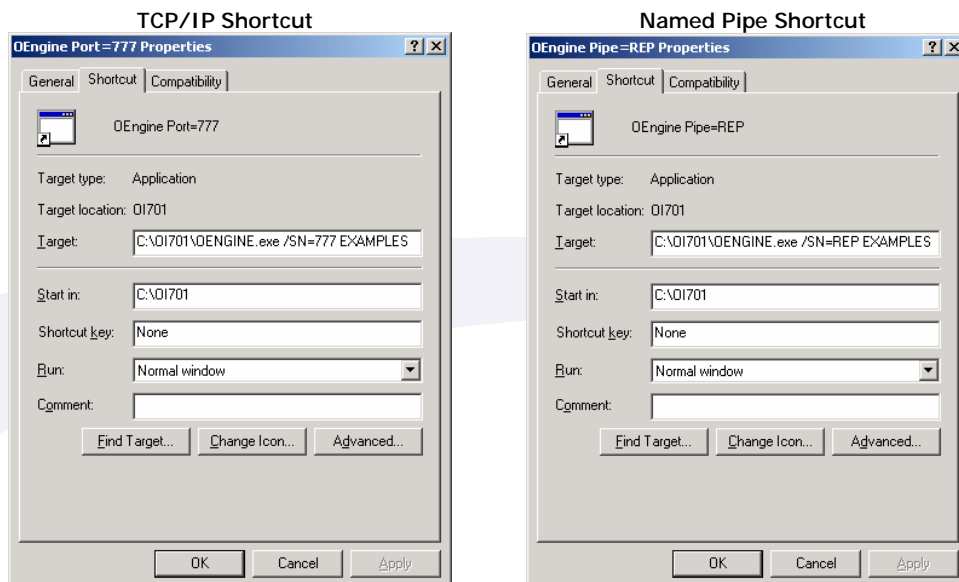
In order to connect to a remote engine (whether it is already running or launched dynamically) we will need to know two things: 1.) The directory, IP address, or UNC path of the machine running the remote engine (Note: Local engines use a special path name, as explained later) and 2.) the TCP/IP port or Named Pipe that it is connected to.

To create a remote "out of process" engine that is waiting for a connection, OENGINE.exe must be executed from a command line or a shortcut using the following syntax:

*EnginePath*: \OENGINE.exe /SN= *ServerName* *DatabaseName*

- EnginePath* - The path where OENGINE.exe exists (if launched from a shortcut.)  
*ServerName* - The TCP/IP port or Named Pipe the engine is waiting on.  
*DatabaseName* - The database (.DBT file) the engine should be connected to.

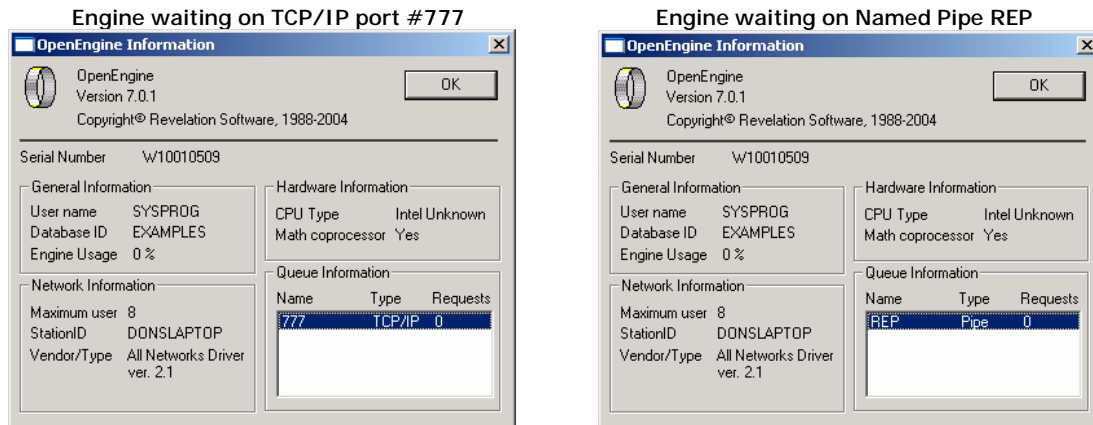
Here are two screenshots of shortcuts that both launch a remote engine connected to the EXAMPLES. One engine will wait for a connection on TCP/IP port #777 while the other will wait for a connection on the REP named pipe. (Note: OENGINE.exe is smart enough to determine whether a TCP/IP or Named Pipe connection is being requested based on the setting used for the /SN switch.)





COMPUTER SOLUTIONS, INC.

If the port or named pipe being requested is not available (for instance, an engine might already be running on that port or named pipe) a message box will appear saying "RevCreateEngine() Failed." Otherwise, an instance of the remote engine will appear on the desktop. If you have multiple user licenses, you can launch multiple engines on different ports, named pipes, as well as a mixture of the two (although there would not be a practical purpose for this other than testing.) Once the engine is running, you can verify that the connection is working correctly by clicking on the EngineInfo button:



Although both TCP/IP and Named Pipes can be used, the rule of thumb is to use TCP/IP when communicating over a network (and obviously the internet) and to use Named Pipes when communicating on the same machine (i.e. using a local remote engine.) In general this provides optimum performance.

Because dynamically launched engines are created automatically, the TCP/IP port or Named Pipe can be hard-coded or determined on-the-fly by the program making the connection. This now brings us to the command that connects to the remote engine:

*Error* = CreateEngine(*hEngine*, *Server*, *Database*, *CreateEngine*, *ShutDown*)

*Error* - Returns error number or 0 if no error (see the **References** section below.)

*hEngine* - Returns the handle to the engine connection if successful. Variable should be initialized to null beforehand.

*Server* - Location and name of the engine. Format is should be as follows:

*\\ServerLocation\ServerName*

Examples of connecting to an engine using TCP/IP port #777:

\\208.252.203.252:777 (Connect to an engine on an IP address)  
\\SUPPORT.SRPCS.COM:777 (Connect to an engine on a URL)  
\\.:777 (Connect to an engine on the local machine)



COMPUTER SOLUTIONS, INC.

Examples of connecting to an engine using the REP named pipe:

\\208.252.203.252\REP (Connect to an engine on an IP address)  
\\ISUPPORT.SRPCS.COM\REP (Connect to an engine on a URL)  
\\. \REP (Connect to an engine on the local machine)

*Database* - The database (.DBT file) the engine should be connected to, like SYSPROG or EXAMPLES.

*CreateEngine* - Flag(s) to determine how the remote engine should be connected to. These are included in the REVCAPAPI\_EQUATES insert (see the **References** section below) and are described as follows:

```
* Connect to an existing engine if possible, else create a new one.
* Use this to create an engine connected to another database.
equ CREATE_ENGINE_OPEN_EXISTING$    to 0x000

* Always create a new engine.
* Engine will be created on the same machine as the calling engine.
equ CREATE_ENGINE_CREATE_NEW$       to 0x001

* Only connect to engines that already exist.
* This is the best way to connect to a remote engine.
equ CREATE_ENGINE_OPEN_ALWAYS$      to 0x002

* Create an engine in indexing mode.
equ CREATE_ENGINE_INDEXER$          to 0x010

* Create an engine that will not shut down when the connections end.
equ CREATE_ENGINE_WAIT_ON_CLOSE$    to 0x020
```

Another flag, which is undocumented, allows a dynamically launched OpenEngine to be invisible while it is running. This is helpful when the client needs to run local remote engines and the developer does not want the end user to see them on the desktop:

```
* Keep the engine invisible
equ REV_CREATE_ENGINE_NO_UI$        to 0x040
```

Example of how to use this flag to hide the remote engine:

```
BitOr(CREATE_ENGINE_OPEN_ALWAYS$, REV_CREATE_ENGINE_NO_UI$)
```

*ShutDown* - Boolean flag to determine if the engine should close itself automatically after the connection is done. This only affects dynamically launched remote engines. We recommend always setting this to 1. There appears to be no benefit keeping the engine open because subsequent requests will not be accepted until it has been closed and created again.



COMPUTER SOLUTIONS, INC.

## Mind Your Ps and Queues

After we have successfully connected to our remote engine we now need to prepare a queue that will be a channel to process our remote request. Currently OpenEngine can only process one queue at a time. Therefore, only one request can be performed by an engine at a time. Multiple requests are "queued" up and processed in sequence. (Note: A particular client can only connect to the same remote engine once. Attempting to make multiple simultaneous connections will hang the remote engine and require a manual close before it will respond to future requests.)

*Error* = CreateQueue(*hQueue*, *hEngine*, *QueueName*, *Database*, *UserName*, *Password*)

- Error* - Returns error number or 0 if no error (see the **References** section below.)
- hQueue* - Returns the handle to the queue if successful. Variable should be initialized to null beforehand.
- hEngine* - The handle returned by the *CreateEngine* function.
- QueueName* - Set to null.
- Database* - The database (.DBT file) the engine should be connected to, like SYSPROG or EXAMPLES. Use the same variable/value as in the *CreateEngine* function.
- UserName* - Any valid username for the database being connected to.
- Password* - The password for the username being used.

## Any Last Requests?

Once a queue has been established, we are clear to submit a request to our remote engine to execute a stored procedure. While most *system* stored procedures can be executed through a remote engine, one will normally call custom subroutines and functions in production.

There are a few caveats when working with remote engines. Please note the following:

1. "Out of process" engines do not run in event context. This means commands like Set\_Property, Utility, and OIPI calls will not work. Anything GUI and obviously event logic will not run.
2. Active selects will not be available once an engine has completed its request and closed the connection. However, one remote request can save a select using the Save\_Select command while another remote request activates it using the Activate\_Save\_Select command. Developers will most likely have a remote request perform the long database query and Save\_Select command while allowing the local "in-process" engine to handle the Activate\_Save\_Select when event context processes (like an OIPI report) can be executed.
3. Remote engines only have access to resources that are local to the machine running the engine. For instance, using OS-based commands like Drive and DirList will work within the



COMPUTER SOLUTIONS, INC.

local settings of that machine. In some cases this is desirable. For instance, the remote engine might have access to a drive that is not shared over the network. Hence, a remote request function could provide extremely controlled access to any information (database or OS files) on that drive. Local clients will be prevented from direct access from both the application and network.

Remote requests are made with the *CreateRequest* function:

```
Error = CreateRequest(hRequest, hQueue, ExecLine, Arg1, Arg2, Argn, ...)
```

- Error* - Returns error number or 0 if no error (see the **References** section below.)
- hRequest* - Returns the handle to the request if successful. Variable should be initialized to null beforehand.
- hQueue* - The handle returned by the *CreateQueue* function.
- ExecLine* - Remote process to execute. Use the same syntax as when entering commands in the System Monitor and the System Editor's Exec line:

```
RUN RoutineName 'Param1', 'Param2', 'Paramn', . . .
```

Note: Any command line that is passed through the ExecLine parameter will be a literal string. Here is an example of how this might look:

```
ExecLine = "RUN CHECKLEGALUSER 'ADMIN', 'MYSECRET'"  
Error = CreateRequest(hRequest, hQueue, ExecLine)
```

As this example demonstrates, parameters are always literals. Additionally, *CreateRequest* does not wait for the remote request to finish processing. Therefore we do not have the ability to detect changes that are made to these parameters as we normally would when passing parameters by reference. If this is a requirement then there are two options: 1.) Design the remote request to send back changes through the normal return data process (see the **Twiddling Our Thumbs** section below), or 2.) use the previously mentioned *CallFunction* or *CallSubroutine* commands instead (see the **Requests for Dummies** section below.) Using these commands, however, means that our local engine will be waiting for the remote engine to finish processing. While this provides us the ability to connect to remote engines, it defeats our purpose of "multitasking".



COMPUTER SOLUTIONS, INC.

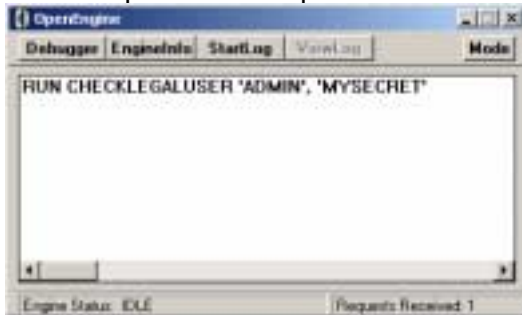
*Arg1...Argn* - Actual arguments to pass into the ExecLine if replaceable parameters are used. In the example used above we embedded the RoutineName parameters in the ExecLine string. *CreateRequest* can also work with replaceable parameters. Using the *CHECKLEGALUSER* command this alternative syntax would look like this:

```
ExecLine = "RUN CHECKLEGALUSER #1, #2"  
Arg1 = "EXAMPLES"  
Arg2 = "MYSECRET"  
Error = CreateRequest(hRequest, hQueue, ExecLine, Arg1, Arg2)
```

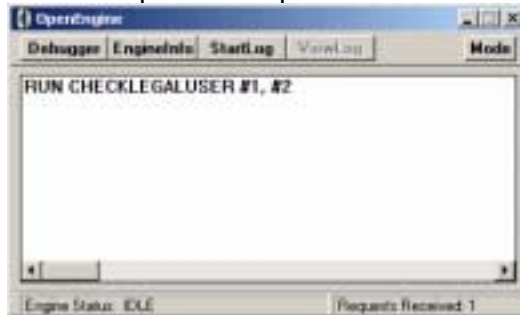
Note that the replaceable parameters were not quoted. Otherwise, *RemoteRequest* will treat them as the literal values to be passed.

We recommend using the replaceable parameters for remote requests rather than including them directly in the ExecLine string. In addition to making it easier to code, it adds a layer of security. This can be clearly seen by comparing the following two screen shots:

RemoteRequest without Replaceable Parameters



RemoteRequest with Replaceable Parameters



If the machine running the remote engine is in public view then it is possible for sensitive information, like passwords, will be seen in the OpenEngine command log. Using replaceable parameters eliminates this problem. For even more security, remote engines can be launched invisibly using the */HE* command line flag:

```
C:\OI701\OENGINE.exe /SN=REP /HE EXAMPLES
```

Invisible engines, however, must be shutdown using the Task Manager's Process tab.

## Requests for Dummies

For a simpler and quicker way to make a request to a remote engine we can use two alternative commands: *CallFunction* and *CallSubroutine*. In the **Getting the Big Picture** section above, our chart indicated that these commands replace four separate steps (and a lot of code) that is required when using *CreateRequest*. This convenience, however, comes at a price: our local engine is suspended until the remote request is completed. Therefore, if you don't have the time to put together a generic remote request utility that uses *CreateRequest*, then these commands might be the solution you need:



COMPUTER SOLUTIONS, INC.

*Error* = CallFunction(*hQueue*, *ReturnValue*, *Function*, *Arg1*, *Arg2*, *Argn*, ...)

*Error* - Returns error number or 0 if no error (see the **References** section below.)

*hQueue* - The handle returned by the *CreateQueue* function.

*ReturnValue* - The value returned by the remote function.

*Function* - Remote function to execute. Unlike *CreateRequest*, the RUN command is not needed nor is it supported. If the function is local to the database that was specified in the *CreateQueue* function then only the function name is required. Otherwise, append an asterisk and the application name to the function name:

```
Function = "MyDivFunc" ; * Or "MyDivFunc*APPNAME" if necessary
Arg1 = 70
Arg2 = 5
Error = CallFunction(hQueue, ReturnValue, Function, Arg1, Arg2)
```

*Arg1...Argn* - Arguments (parameters) for the remote function. Unlike *RemoteRequest*, these arguments can be passed by reference. That is, any changes made to the original values in the remote subroutine will be returned to the variables used in our calling routine.

*Error* = CallSubroutine(*hQueue*, *Subroutine*, *Arg1*, *Arg2*, *Argn*, ...)

*Error* - Returns error number or 0 if no error (see the **References** section below.)

*hQueue* - The handle returned by the *CreateQueue* function.

*Subroutine* - Remote subroutine to execute. Unlike *CreateRequest*, the RUN command is not needed nor is it supported. If the function is local to the database that was specified in the *CreateQueue* function then only the subroutine name is required. Otherwise, append an asterisk and the application name to the subroutine name:

```
Subroutine = "MyDivSub" ; * Or "MyDivSub*APPNAME" if necessary
Arg1 = 70
Arg2 = 5
ReturnValue = "" ; * Initialize the variable
Error = CallSubroutine(hQueue, Function, Arg1, Arg2, ReturnValue)
```

*Arg1...Argn* - Arguments (parameters) for the remote subroutine. Unlike *RemoteRequest*, these arguments can be passed by reference. That is, any changes made to the original values in the remote subroutine will be returned to the variables used in our calling routine.



COMPUTER SOLUTIONS, INC.

## Twiddling Our Thumbs

Once *RemoteRequest* has successfully executed our process, it is our responsibility to monitor the status of the remote engine. Two functions are available for this: *WaitForReply* and *PollForReply*. In the **Getting the Big Picture** chart above only *PollForReply* is mentioned. This is because *WaitForReply* does exactly what its name implies; that is, it *waits* for a reply from the remote engine before proceeding. Therefore, if our remote process is busy performing a long process then our local engine will be forced to wait until it is completed. Actually, to be *technically* correct, if the remote process uses the *Send\_Info* or *Send\_Dyn* commands to send information back then *WaitForReply* will allow the local engine to receive it. *WaitForReply* must then be called again to wait for the next information update or until the remote process has completed. Because many processes do not use *Send\_Dyn* or *Send\_Info* (like *RList* selects), *WaitForReply* is only a little better than using *CallFunction* or *CallSubroutine*.

*PollForReply*, on the other hand, provides true asynchronous processing which gives us the ability to allow our local engine to perform other tasks while the remote engine is performing its duties. Like *WaitForReply*, *PollForReply* will be notified if the remote process uses the *Send\_Info* or *Send\_Dyn* commands.

*Error* = `PollForReply(hRequest, Status, Reply)`

*Error* - Returns error number or 0 if no error (see the **References** section below.)

*hRequest* - The handle returned by the *CreateRequest* function.

*Status* - Flag to determine the status of the remote process. These are included in the REVCAP1\_EQUATES insert (see the **References** section below) and are described as follows:

```
equ UNPROCESSED$    to 0    ;* Server has not begun request.
equ PROCESSING$     to 1    ;* Server is processing request.
equ DATA_AVAILABLE$ to 2    ;* Server has data available.
equ COMPLETED$     to 3    ;* Server has completed request, status
                        ;* information is available.
equ PROC_ERROR$     to 4    ;* Server process failed, status
                        ;* information is available.
equ INFO_AVAILABLE$ to 5    ;* Server has intermediate status
                        ;* information available.
equ INFO_REQUEST$   to 10   ;* Server is requesting information from
                        ;* client.
```

*Reply* - This is supposed to contain information that has been returned from the remote engine. However, there is a known bug where this information is actually returned to the *Status* parameter instead. Fortunately this parameter is optional and we can use two other functions, *GetReply* and *GetResponseText*, to get this information.

A looping mechanism must now be considered that makes the call to *PollForReply*. We will use the *Status* flag to determine if the remote process is done (either because our remote process successfully ended or there was a procedural error) and end our loop. Otherwise, the remote process is still executing. Here is a simple loop to demonstrate how this might look:



COMPUTER SOLUTIONS, INC.

Loop

```
Error = PollForReply(hRequest, Status)

Begin Case

  Case Error
    * An error was returned with the PollForReply function. Retrieve
    * error text and shutdown connection to remote engine.
    GoSub Process_Error
    GoSub Request_Done
    GoSub Return_Results

  Case Status EQ PROCESSING$
    * Only used with PollForReply. Indicates that the remote engine is
    * still processing our request.

  Case Status EQ DATA_AVAILABLE$
    * Information is available when the remote engine uses the Send_Dyn
    * routine in the request.
    GoSub Process_Status

  Case Status EQ INFO_AVAILABLE$
    * Information is available when the remote engine uses the
    * Send_Info routine in the request.
    GoSub Process_Status

  Case Status EQ INFO_REQUEST$
    * The remote engine is requesting information from the client using
    * the Req_Info routine. Use the SendResponse function to reply.

  Case Status EQ COMPLETED$
    * The remote engine has completed our request (i.e. the end of the
    * program has been reached.)
    GoSub Process_Status
    GoSub Request_Done
    GoSub Return_Results

  Case Status EQ PROC_ERROR$
    * An error occurred in the remote engine request. Any Set_Status
    * error codes will be returned.
    GoSub Process_Status
    GoSub Request_Done
    GoSub Return_Results

End Case

Yield()

Until Error OR Status EQ COMPLETED$ OR Status EQ PROC_ERROR$
Repeat
```

Note the use of the Yield command within each iteration. This is necessary to give the local engine the ability to process other local requests and, consequently, give us the ability to simulate "multitasking".

Alternatively one could design a window (running invisibly) to use the TIMER property and event to facilitate the looping mechanism. While this makes our coding more complex, it offers two advantages:

- 1.) The ability to control the pace of each iteration with the delay parameter of the TIMER property while
- 2.) putting the burden of the wait mechanism on the operating system rather than OpenInsight.



COMPUTER SOLUTIONS, INC.

There are now four more areas that need to be covered. These are represented by our `Process_Status`, `Process_Error`, `Request_Done`, and `Return_Results` GoSub references above.

## What's Our Status?

Each call to the *PollForReply* function will set the Status flag. If there was a procedural error in the remote request then we will use the *GetStatusText* function to retrieve the full error text. Otherwise, we will use the *GetReply* function to retrieve any data that is being returned. Here is an example of how this section of code might look:

```
Process_Status:
    If Status EQ PROC_ERROR$ then
        * A procedural error occured in the remote request. Use GetStatusText to
        * retrieve the entire error message.
        Error = GetStatusText(hRequest, @RM, StatusText)
    end else
        * Data/Information available. Use GetReply to retrieve the entire text.
        Error = GetReply(hRequest, StatusText)
    end

    If Error then
        GoSub Process_Error
    end else
        If StatusText[-1, 1] EQ \00\ then StatusText[-1, 1] = ""
        *
        * StatusText now contains data which has been returned using Send_Info,
        * Send_Dyn, or the remote function's Return command. Put whatever logic is
        * desired to manage this information here. Keep in mind that if data
        * received before the Return is needed in the Return_Results then we need
        * to find a way of storing this away until then.
        *
    end

return
```

## Error Messages

Whenever any of our functions returns a value other than 0 we need to compare this with the list of errors from the Error Definitions (REVCERRS.H) file (see the **References** section below for a complete list.) Here is a portion of code that can be used:

```
Process_Error:
    Begin Case
        Case Error EQ 0
            ErrorText = "Successfully Completed"
        Case Error EQ 1
            ErrorText = "Error 1: Invalid name"
        .
        . Put the other error definitions here.
        .
        Case Error EQ 2001
            ErrorText = "Error 2001: REV_ERR_GET_PROC_ADDRESS32"
        Case Error EQ 2002
            ErrorText = "Error 2002: REV_ERR_NO_REVCAP32"
    End Case

return
```



COMPUTER SOLUTIONS, INC.

Once we have the error description we then need to make the end user aware of the problem. Use whatever method works best for your application (i.e. message box, statusline, error log report, etc.)

## Closing Shop

After our remote request is done processing (or in the event of a premature error) we need to properly close our connections. This is perhaps the easiest portion of our code (Note: these must be the same handles that were returned by their respective *CreateEngine*, *CreateQueue*, and *CreateRequest* functions):

```
Request_Done:
    CloseRequest(hRequest)
    CloseQueue(hQueue)
    CloseEngine(hEngine)

return
```

## So Now What?

Phew, we finally made it to the end. However, we still might need to do something with the returned data. For instance, if our remote request performed a large table select it might have saved the results using the *Save\_Select* routine and returned the name it was saved under. Our local engine, having received the select name back, can then activate it with the *Activate\_Save\_Select* routine for a report.

If we had put all of our above code into a nice tidy routine (i.e. our "black box") then we have a couple of options available to us. Let's assume we create a function called *Remote\_Request*:

```
ReturnValue = Remote_Request(Server, CreateEngine, HideEngine, Database, UserName,
-> Password, ExecLine, Arguments, Callback)
```

One approach would be to call our function within another routine that needs the information being returned (e.g. a report as suggested above.) Thus, our calling routine will wait until the remote request is done processing yet our application will be free to run other processes (due to the embedded *Yield* command in our status loop.) When the remote request is done processing, our calling routine can then automatically move on or prompt the end user if it is okay to continue.

Alternatively we can have *Remote\_Request* call another routine for us. This is why the *Callback* parameter has been included in the *Remote\_Request* syntax. Ultimately, then, the *Call @SubName* command will be needed within *Remote\_Request* to make this work.

With this final piece tied together you now have a "black box" approach toward managing remote engines. While this does not exhaust all that is possible with remote engines, we believe this will give many applications a significant enhancement boost in functionality. We hope that you will agree.



COMPUTER SOLUTIONS, INC.

## References

Further information we think will be helpful are listed here for your convenience:

1. [OI 4.1 New Features.pdf](#). This document is still available from the Works Download page on Revelation Software's website. This can be found under the [Updated!! OpenInsight Version 4.1 Upgrade](#) link. There are, in fact, two links with the same name. Both will work. Most of the information in this document can also be found in [The OpenInsight OpenEngine](#) help file ([OpenEngine.chm](#)), which installs in the same directory as OpenInsight.exe.
2. OENGINE\_DLL prototype record. This allows the developer to use Basic+ to connect with remote engines. Note: If you are running OpenInsight v7.0 or higher, this record should already be current:

```
OENGINE.EXE
*
* internal OEngine functions as defined in rev_call.c module
* (see functions CallDLL and CalloEFunc)
*
VOID INTERNAL POINTER_02(VOID) AS RTP90
VOID INTERNAL POINTER_03(VOID) AS C_PACK
VOID INTERNAL POINTER_04(VOID) AS BLD_TEMPLATE
VOID INTERNAL POINTER_05(VOID) AS OEPUTDATA
VOID INTERNAL POINTER_06(VOID) AS OEREQINFO
VOID INTERNAL POINTER_07(VOID) AS OEPUTSTAT
VOID INTERNAL POINTER_08(VOID) AS OEGETCLIENTSTAT
VOID INTERNAL POINTER_09(VOID) AS RETSTACK
VOID INTERNAL POINTER_22(VOID) AS SET_BGND_IX_TIME
VOID INTERNAL POINTER_23(VOID) AS GETIOFLAG
VOID INTERNAL POINTER_24(VOID) AS SETIOFLAG
VOID INTERNAL POINTER_25(VOID) AS GETENGINEWINDOW
VOID INTERNAL POINTER_26(VOID) AS GETHANDLECOUNT
VOID INTERNAL POINTER_27(VOID) AS SETINITDIROPTIONS
VOID INTERNAL POINTER_28(VOID) AS ISEVENTCONTEXT

* OI 7 - added Remote Engine functions

VOID INTERNAL POINTER_30(VOID) AS CREATEENGINE
VOID INTERNAL POINTER_31(VOID) AS CREATEQUEUE
VOID INTERNAL POINTER_32(VOID) AS CREATEREQUEST
VOID INTERNAL POINTER_33(VOID) AS POLLFORREPLY
VOID INTERNAL POINTER_34(VOID) AS WAITFORREPLY
VOID INTERNAL POINTER_35(VOID) AS GETREPLY
VOID INTERNAL POINTER_36(VOID) AS SENDRESPONSE
VOID INTERNAL POINTER_37(VOID) AS GETSTATUSTEXT
VOID INTERNAL POINTER_38(VOID) AS CLOSEREQUEST
VOID INTERNAL POINTER_39(VOID) AS CALLSUBROUTINE
VOID INTERNAL POINTER_40(VOID) AS CALLFUNCTION
VOID INTERNAL POINTER_41(VOID) AS CLOSEQUEUE
VOID INTERNAL POINTER_42(VOID) AS CLOSEENGINE

* end
```

If you had to update OENGINE\_DLL on your system then log into the SYSPROG application and run the following command from the Exec line in the System Editor or from the command line in the System Monitor:

```
RUN DECLARE_FCNS "OENGINE_DLL"
```



COMPUTER SOLUTIONS, INC.

If this is successful, several messages like, "\$CLOSEENGINE written to file SYSOBJ." will appear.

3. REVCAP1\_EQUATES insert record. This insert contains the necessary declarations for those functions defined in the OENGINE\_DLL prototype record. Additionally common equates have also been defined. If you are running OpenInsight 7.0 or higher, this record should be current with the exception of one equate:

```
equ REV_CREATE_ENGINE_NO_UI$ to 0x040
```

However, the version below includes this and all other necessary equates:

```
compile Insert REVCAP1_EQUATES

// CreateEngine Constants
* Connect to an existing engine if possible, else create a new one.
* Use this to create an engine connected to another database.
equ CREATE_ENGINE_OPEN_EXISTING$ to 0x000

* Always create a new engine.
* Engine will be created on the same machine as the calling engine.
equ CREATE_ENGINE_CREATE_NEW$ to 0x001

* Only connect to engines that already exist.
* This is the best way to connect to a remote engine.
equ CREATE_ENGINE_OPEN_ALWAYS$ to 0x002

* Create an engine in indexing mode.
equ CREATE_ENGINE_INDEXER$ to 0x010

* Create an engine that will not shut down when the connections end.
equ CREATE_ENGINE_WAIT_ON_CLOSE$ to 0x020

* Keep the engine invisible (e.g. BitOr(CREATE_ENGINE_OPEN_ALWAYS$,
REV_CREATE_ENGINE_NO_UI$)
equ REV_CREATE_ENGINE_NO_UI$ to 0x040

equ UNPROCESSED$ to 0 ;* Server has not begun request.
equ PROCESSING$ to 1 ;* Server is processing request.
equ DATA_AVAILABLE$ to 2 ;* Server has data available.
equ COMPLETED$ to 3 ;* Server has completed request, status
;* information is available.
equ PROC_ERROR$ to 4 ;* Server process failed, status information is
;* available.
equ INFO_AVAILABLE$ to 5 ;* Server has intermediate status information
;* available.
equ INFO_REQUEST$ to 10 ;* Server is requesting information from client.

declare function CreateEngine
declare function CreateQueue
declare function CreateRequest
declare function PollForReply
declare function WaitForReply
declare function GetReply
declare function SendResponse
declare function GetStatusText
declare function CloseRequest
declare subroutine CloseRequest
declare function CallSubroutine
declare function CallFunction
```



COMPUTER SOLUTIONS, INC.

```
declare function CloseQueue
declare subroutine CloseQueue
declare function CloseEngine
declare subroutine CloseEngine
```

4. Error Definitions. Our thanks to Pat McNerthney for making available these undocumented definitions for the numerical errors that can occur. They have been presented in a Begin/End Case syntax for convenience:

```
Begin Case
Case Error EQ 0
    ErrorText = "Successfully Completed"
Case Error EQ 1
    ErrorText = "Error 1: Invalid name"
Case Error EQ 2
    ErrorText = "Error 2: Out of memory"
Case Error EQ 3
    ErrorText = "Error 3: Unable to locate server"
Case Error EQ 5
    ErrorText = "Error 5: Functionality not implemented"
Case Error EQ 6
    ErrorText = "Error 6: GlobalLock failed"
Case Error EQ 7
    ErrorText = "Error 7: Null pointer passed to function"
Case Error EQ 8
    ErrorText = "Error 8: Unknown request status"
Case Error EQ 9
    ErrorText = "Error 9: Unable to retrieve argument"
Case Error EQ 10
    ErrorText = "Error 10: Too many requests active"
Case Error EQ 11
    ErrorText = "Error 11: Invalid request handle"
Case Error EQ 12
    ErrorText = "Error 12: Invalid DLL procedure name"
Case Error EQ 13
    ErrorText = "Error 13: File not found"
Case Error EQ 14
    ErrorText = "Error 14: Undefined error"
Case Error EQ 15
    ErrorText = "Error 15: Queue does not exist"
Case Error EQ 16
    ErrorText = "Error 16: The server is not responding"
Case Error EQ 17
    ErrorText = "Error 17: Max # of clients exceeded for Queue"
Case Error EQ 18
    ErrorText = "Error 18: Connection to queue failed"
Case Error EQ 19
    ErrorText = "Error 19: Failed to send message"
Case Error EQ 20
    ErrorText = "Error 20: Invalid queue handle"
Case Error EQ 21
    ErrorText = "Error 21: Invalid parameter passed"
Case Error EQ 22
    ErrorText = "Error 22: Client-server handshaking error"
Case Error EQ 23
    ErrorText = "Error 23: Script is incomplete"
Case Error EQ 24
    ErrorText = "Error 24: Data is not available"
Case Error EQ 25
    ErrorText = "Error 25: Error accessing template"
Case Error EQ 26
    ErrorText = "Error 26: Invalid column; column not avail"
Case Error EQ 27
```



COMPUTER SOLUTIONS, INC.

```
        ErrorText = "Error 27: Invalid type"
Case Error EQ 28
        ErrorText = "Error 28: Maximum queues exceeded"
Case Error EQ 29
        ErrorText = "Error 29: The buffer is too small for data"
Case Error EQ 30
        ErrorText = "Error 30: Invalid Send or Receive sequence"
Case Error EQ 31
        ErrorText = "Error 31: Invalid user name"
Case Error EQ 32
        ErrorText = "Error 32: Invalid password"
Case Error EQ 33
        ErrorText = "Error 33: Invalid pathname"
Case Error EQ 34
        ErrorText = "Error 34: Invalid database name"
Case Error EQ 35
        ErrorText = "Error 35: Max # of engines already running"
Case Error EQ 36
        ErrorText = "Error 36: Unable to start engine"
Case Error EQ 37
        ErrorText = "Error 37: Unable to shutdown engine"
Case Error EQ 38
        ErrorText = "Error 38: Unable to create new engine"
Case Error EQ 100
        ErrorText = "Error 100: REV_ERR_RCLINITIALIZE"
Case Error EQ 101
        ErrorText = "Error 101: REV_ERR_RCLVERSION"
Case Error EQ 102
        ErrorText = "Error 102: REV_ERR_RCLOPENCLIENT"
Case Error EQ 103
        ErrorText = "Error 103: REV_ERR_RCLCONNECT"
Case Error EQ 104
        ErrorText = "Error 104: REV_ERR_RCLREQUESTREPLY"
Case Error EQ 105
        ErrorText = "Error 105: REV_ERR_RCLDISCONNECT"
Case Error EQ 106
        ErrorText = "Error 106: REV_ERR_RCLCLOSECLIENT"
Case Error EQ 107
        ErrorText = "Error 107: REV_ERR_RCLTERMINATE"
Case Error EQ 108
        ErrorText = "Error 108: REV_ERR_RCLCHECKSERVER"
Case Error EQ 200
        ErrorText = "Error 200: REV_ERR_HEAPCREATE"
Case Error EQ 201
        ErrorText = "Error 201: REV_ERR_HEAPFREE"
Case Error EQ 202
        ErrorText = "Error 202: REV_ERR_HEAPDESTROY"
Case Error EQ 203
        ErrorText = "Error 203: REV_ERR_CREATEEVENT"
Case Error EQ 1000
        ErrorText = "Error 1000: REV_ERR_INVALID_ENGINE_HANDLE"
Case Error EQ 1001
        ErrorText = "Error 1001: REV_ERR_ENGINE_BUSY"
Case Error EQ 1002
        ErrorText = "Error 1002: REV_ERR_DATABASE_INIT"
Case Error EQ 1003
        ErrorText = "Error 1003: REV_ERR_LOGON"
Case Error EQ 1004
        ErrorText = "Error 1004: REV_ERR_NO_SERVER_MEMORY"
Case Error EQ 1005
        ErrorText = "Error 1005: REV_ERR_REQUEST_ACTIVE"
Case Error EQ 1006
        ErrorText = "Error 1006: REV_ERR_REQUEST_NOT_ACTIVE"
Case Error EQ 1007
        ErrorText = "Error 1007: REV_ERR_UNEXPECTED_PARAMETER"
```



COMPUTER SOLUTIONS, INC.

```
Case Error EQ 1008
    ErrorText = "Error 1008: REV_ERR_MISSING_OBJECT"
Case Error EQ 2000
    ErrorText = "Error 2000: REV_ERR_LOAD_REVCAP32"
Case Error EQ 2001
    ErrorText = "Error 2001: REV_ERR_GET_PROC_ADDRESS32"
Case Error EQ 2002
    ErrorText = "Error 2002: REV_ERR_NO_REVCAP32"
Case 1
    ErrorText = "Unknown Error: ":Error
End Case
```

5. OpenEngine Reference Manual. This is more for developers who are comfortable with programming in other IDEs, especially C++, to communicate through the OpenEngine API. A free copy of this can be downloaded from Revelation Software's Knowledge Base at this link:

<http://www.revelation.com/knowledge.nsf/5f13dfa1e5319ec6852566f50065bc74/607983febe650bc985256957005921b6?OpenDocument>