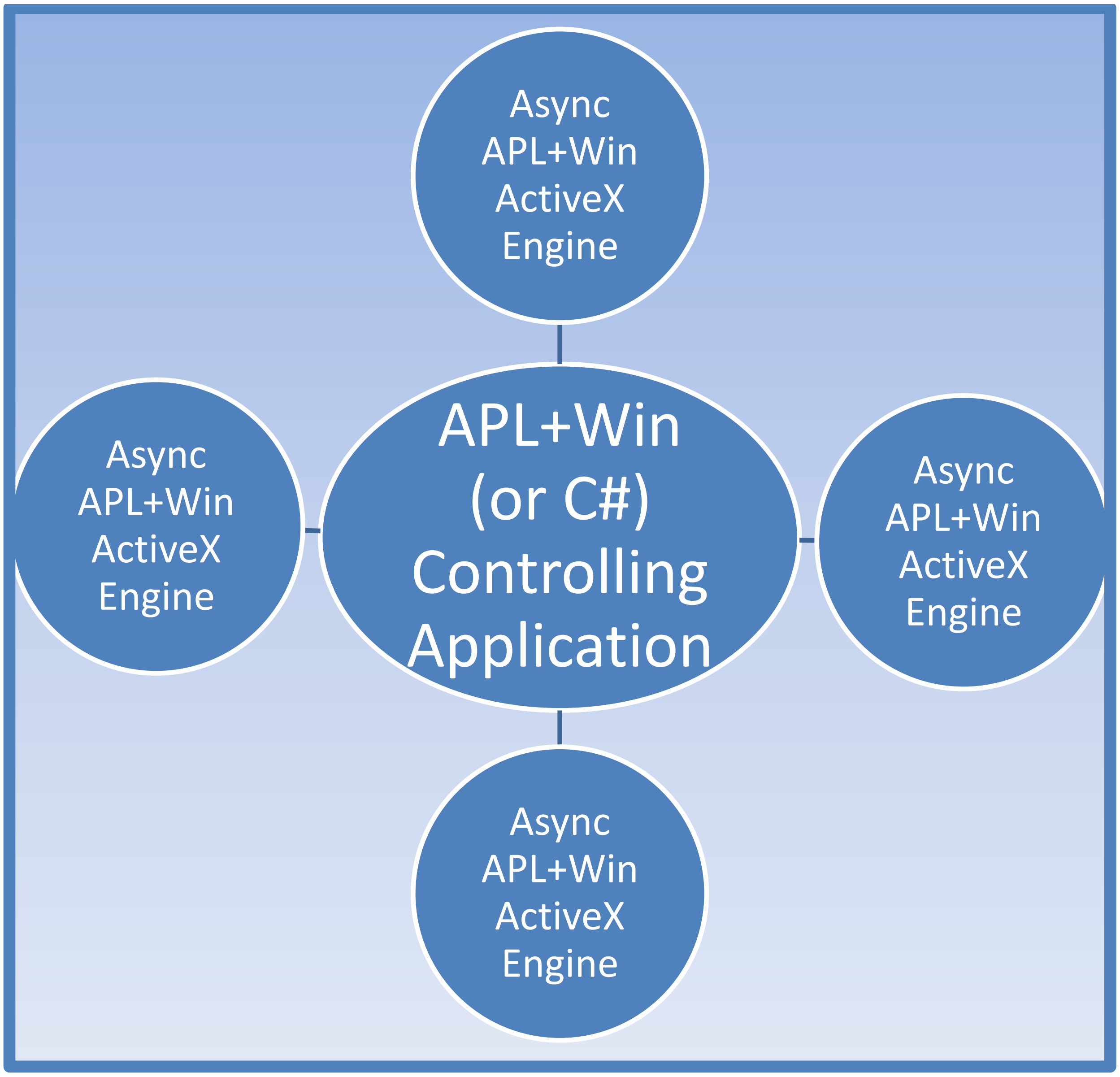
APLNext Supervisor

Programmer-controlled Multi-threading for APL+Win


Async APL+Win ActiveX Engine
Async APL+Win ActiveX Engine
APL+Win (or C#) Controlling Application
Async APL+Win ActiveX Engine
Async APL+Win ActiveX Engine

APLNext Supervisor for APL+Win

Table of Contents

APLNext Supervisor for APL+Win

Overview

An APL+Win application configured to use the APLNext Supervisor can employ ‘coarse-grained’ multi-threading to improve performance. The ‘coarse-grain’ adjective indicates that the application system designer determines where multi-threading is possible within, and would be beneficial to, the application system performance. The ‘graininess’ of the multi-threading indicates the amount of ‘work’, or number of operations, to be performed by each parallel processing operation. Tools in the APL+Win programming language, such as the monitor (⎕MF) system function, can help to identify the potential benefits of ‘coarse-grain’ multi-threading in an application system.

The APLNext Supervisor is a productivity platform for executing APL+Win functions in a multi-threaded manner. When the computer hardware on which APL+Win has been installed has multiple processors or multiple ‘cores’, installing the APLNext Supervisor and configuring the APL+Win application system to use it means that programmer-designated operations associated with APL+Win functions can be executed concurrently.

Concurrent execution of an APL+Win function means that the processing performed by that function occurs asynchronously with respect to other processes of the APL+Win application system. Thus the ‘main thread’ of an APL+Win application system is not ‘blocked’ while another APL+Win function of that application system is running concurrently because it is running in a separate ‘thread’.

Application systems which have sections of the processing algorithm that can be asynchronously executed can usually benefit from multi-threading. Typically such application systems perform repeated, in-memory processing of a collection of similar data elements, such as those associated with time series, populations, stochastic processes, simulations, etc.

Symmetric multi-processing is supported in APL+Win via the APLNext Supervisor. One or more independent threads, each providing a potentially-identical, asynchronously-operating APL+Win instance are spawned by an instance of the APLNext Supervisor created by the ‘controlling application’ designed and implemented by an APL+Win application system programmer. The ‘controlling application’ issues processing requests to an instance of the APLNext Supervisor to execute a ‘Kernel’ function asynchronously. Based on programmer design, the ‘controlling application’ designates the APL+Win ‘Kernel’ function and its arguments that will be run in each of these independent threads. The ‘controlling application’ developed by an APL+Win application system programmer does not explicitly assign processing requests to specific threads, processors or cores, because these considerations are handled by the APLNext Supervisor technology and the Windows operating system.

Programmer re-configuration of a linearly-programmed application system into a multi-threaded, parallel-processing application involves:

- Identifying those processes which are performed multiple times

- Removing 'side effects' from those processes, such as interactions with other potentially parallel processes, to assure independence

- Ideally modifying those processes to be in-memory operations, eliminating database access or GUI-presentations

- Minimizing the data required by the process arguments as well as the result data returned by the process

- Developing a 'controlling application'-based protocol for starting such a process, accumulating the results of successful execution of such a process and handling failed execution attempts for such a process

- Incorporating the APLNext Supervisor methodology into the controlling application

- Preparing the subordinate APL+Win workspace containing the 'kernel' functions (with explicit results) supporting the application system processes to be executed asynchronously

- Installing the re-configured application on a machine with multi-core/processor hardware

- Comparing the performance of the linearly-programmed form of the application to that of the Supervisor-enabled form of the application

Why Multi-threading?
Multi-threading is sometimes called non-linear programming, parallel processing or concurrent processing. Parallel processing is all about increasing the performance of an application system. Businessmen, engineers and scientists need increased application system performance to remain competitive, develop better designs and make new discoveries.

- Previously, significant application system performance increases came about due to increases in digital processor clock speed and component density. These enhancements were often noted as evidence of Moore's law. These type of computer hardware enhancements often resulted in immediate performance benefits for application systems which use linear programming techniques. Little or no re-configuration of a linearly-programmed application system was required.

- As the density of electronic elements and connections on digital processors is increased, the power and associated waste heat removal requirements are substantially increased. In addition component density increases require reduced connection widths resulting in signal 'leakage'. The computer hardware industry's response to these problems has been to make available multiple-processor and multiple-core hardware. Such hardware incorporates supervisory hardware and software that can delegate processing work among several identical digital processors that perform their designated tasks asynchronously. Often the computer operating system, such as Microsoft Windows, provides for the association of application-specific 'threads' or processing requests with available hardware processors.

- For an application system to take advantage of multi-processor and multi-core computer hardware, non-linear techniques utilizing parallel processing is necessary. This often means that re-configuration of the application system is necessary.

How is Parallel Processing Implemented

It would be nice if the mere presence of multiple processors in the hardware was sufficient to improve the performance of an application system, but generally that does not occur. The reason is that most application systems are designed using linear programming techniques so that before the n^{th} processing step is attempted, the $(n-1)^{th}$ processing step must be completed.

Except for special-purpose applications, the operating system (or the APL interpreter) cannot automatically invoke multiple processors when instructions in a linear program are being executed. Here's why:

- The operating system does not know, without explicit indications from the programmer, which instructions can be concurrently executed independently of any other instruction. Often an application system instruction relies on the results of previous instruction, so that the order of execution is important. These hidden complexities are often called 'side-effects' and are related to a program design which assumes that all application system processes have access to a shared, mutable state. This shared, mutable state is initially established by 'setup' instructions and subsequently modified by additional inter-dependent instructions of the application system. Only the program designer can know which application system processes are may be independently executed.

- The performance of a hardware and software ensemble when executing an instruction in a multi-processor environment includes not only the base processing time associated with the operation itself, but also includes the processing time necessary to marshal the input and output data between multiple processors and the supervising processor. The relative contribution of this data marshaling overhead to the total instruction processing time varies greatly depending on hardware differences and data dimensions. For example one might naively assume that since APL is designed to perform mathematical operations on arrays, the APL interpreter should automatically parallelize these operations using multiple processors for element-wise calculations. Academic research, primarily performed by IBM decades ago, indicated that such automatic parallelization would generally reduce performance significantly, except in the case of extremely large arrays (which are rare in most application systems) or very cpu-intensive operations (which are generally only occur as programmer-combinations of many basic operations or in special case processors such as those for gene sequencing, weather analysis or chemical process simulation). Such research as also indicated that the 'cost' of performing tests to determine if parallelization would be beneficial or not would significantly reduce overall performance.

Currently, general purpose parallel processing implemented by multiple processor hardware is explicitly invoked by the programmer for specified sections of the application system algorithm. The programmer must re-purpose an application system that was previously designed using a linear programming style to a new hybrid style which identifies those application system instructions which can be asynchronously executed and which also would benefit from such asynchronous execution. The application programmer then indicates to the supervisory program those application-specific processing requests which should be executed in parallel.

Because only the time the parallel processing request is initiated, and not the time that request is fulfilled is known, the ‘controlling application’ receives information from processing requests completed by a ‘kernel’ function via the execution of an event handler function. The ‘controlling application’ that is initiating parallel processing requests using the APLNext Supervisor subscribes to the ‘ProcessCompleteCallback’ event of the APLNext Supervisor and indicates the application-specific event handler method which will receive the results of a completed processing request.

Similarly although the time that all processing requests have been submitted by the ‘controlling application’ is known, the time that all such processing requests have been fulfilled cannot be known in advance. In order to determine when all such processing requests are complete, the ‘controlling application’ can subscribe to the ‘ProcessAllDoneCallback’ event of the Supervisor.

A processing request may be considered complete if it successfully performed the intended algorithm, but a processing request may also be considered complete if there has been some error condition within the ‘kernel’ function which worked on that processing request. Generally it is up to the application system designer to anticipate such a request which is complete but with an error condition and handle that request properly either in the ‘kernel’ function itself or in the ‘ProcessCompleteCallback’ event handler method.

Configuring an APL+Win Application for Multi-threading

The APLNext Supervisor is a programmer-controlled extension of APL+Win for parallel processing that is designed for the application of high-level functional programming techniques.

The APL+Win-based application system will probably require re-configuration to identify and isolate those elements of the application algorithm that should be processed asynchronously.

Functional programming techniques used in this reconfiguration include:

- Reduced reliance on a shared, mutable state while the parallel processing elements of the application system algorithm are being executed. In APL terms, this suggests avoiding global variables which are modified by and depended upon by those asynchronously-executed, application system operations.

- Association of asynchronously-executed, application system operations with (APL) functions which have explicit results and arguments.

- Where beneficial, the use of composition of functions or Currying (see Moses Schönfinkel for more information) to reduce the number and content of function arguments as much as possible to minimize data marshaling overhead.

- Where beneficial, the reduction in the amount of data returned from a parallel processing element in the application system.

The APLNext Supervisor is designed to use the following components:

- A 'controlling application', e.g. an instance of APL+Win which runs on the 'main thread'. Since the APLNext Supervisor is a .Net assembly, the 'controlling application' can also be a C#, VB.Net or VisualAPL project. If the controlling application is APL+Win-based, it accesses the APLNext Supervisor as an ActiveX object. When the APLNext Supervisor is installed it is exposes as an ActiveX component. The 'controlling application':

    - Initiates the configuration and start-up of the APLNext Supervisor, using the APLNext Supervisor OpenConfigFromFile, OpenConfigFromText and Start methods.

    - Defines the application-specific results container to receive the output of the parallel processing threads.

    - Defines the ProcessCompleteCallback handler function which will be executed when an application-specific, asynchronous processing request ['kernel' function call] has been completed. Within this handler function, the results of a specific processing request are incorporated into the application-specific results container.

    - Subscribes to the APLNext Supervisor's ProcessCompleteCallback event, so that when the application-specific, asynchronous processing request is completed, the APLNext Supervisor can notify the controlling application to execute the ProcessCompleteCallback handler function.

- Defines the ProcessAllDoneCallback handler function which will be executed when all submitted requests have completed processing. This handler function will consolidate the processing results in the application-specific container to complete the application analysis, update the application system's GUI and stop the APLNext Supervisor.

- Subscribes to the APLNext Supervisor's ProcessAllDoneCallback event, so that when all submitted requires have completed processing, the APLNext Supervisor can notify the controlling application to execute the ProcessAllDoneCallback handler function.

- Submits the parallel processing requests to the APLNext Supervisor, using the APLNext Supervisor BeginCall method with an argument based on the argument valence of the 'kernel' functions of the application system that will be executed in parallel.

- An APL+Win workspace containing the 'kernel' functions which constitute the application specific algorithms to be concurrently executed by the APLNext Supervisor upon request by the controlling application.

  - A kernel function signature must have an explicit result and may have an argument structure which is niladic, monadic or dyadic. Arguments may be nested or simple APL objects.

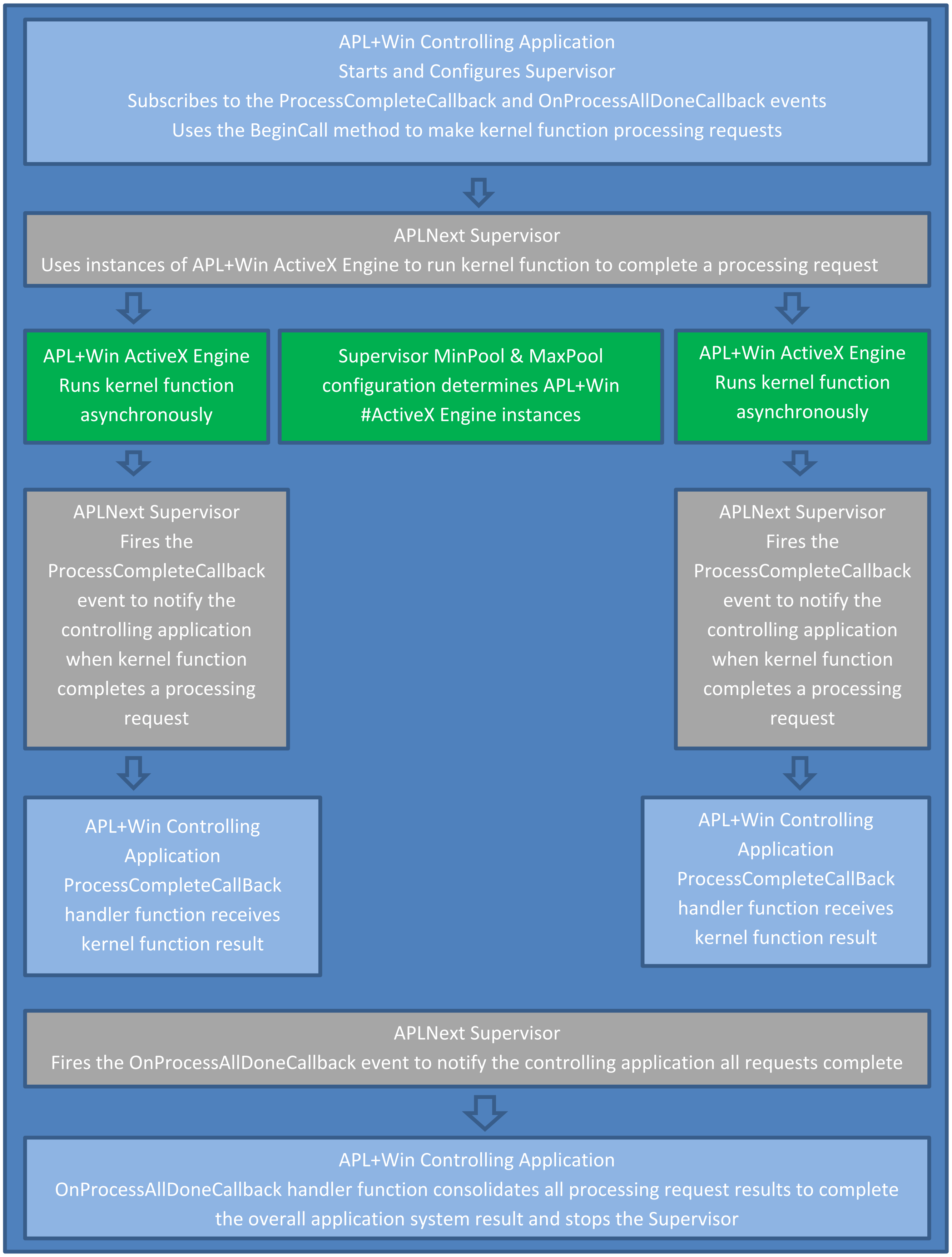
  - The 'graininess' of the algorithms supported by the kernel functions is programmer-controlled. It is generally beneficial to develop a kernel function which performs significant work, i.e. number of operations. If the kernel function graininess is too fine, i.e. too few operations are performed in each kernel function call, the performance will be limited not by those operations, but instead by the time necessary to marshall input data between the 'controlling application' and the 'kernel' function and output data between the 'kernel' function and the 'ProcessCompleteCallback' event handler function.

  - This 'kernel' function will be executed from an instance of APL+Win as an ActiveX server in an independent thread created by the APLNext Supervisor.

  - The application-specific configuration of the APLNext Supervisor can specify the minimum and maximum number of such instances of APL+Win as an ActiveX server. The application programmer should experiment to determine the relationship between the number of available processors or 'cores' on the target machine and the number of instances of APL+Win ActiveX servers.

  - When multiple processors or 'cores' are available on the hardware in use, the Microsoft Windows operating system will automatically associate this independent thread with an available processor.

APLNext Supervisor for APL+Win


APL+Win Controlling Application
Starts and Configures Supervisor
Subscribes to the ProcessCompleteCallback and OnProcessAllDoneCallback events
Uses the BeginCall method to make kernel function processing requests
APLNext Supervisor
Uses instances of APL+Win ActiveX Engine to run kernel function to complete a processing request
APL+Win ActiveX Engine Runs kernel function asynchronously
Supervisor MinPool & MaxPool configuration determines APL+Win #ActiveX Engine instances
APL+Win ActiveX Engine Runs kernel function asynchronously
APLNext Supervisor Fires the ProcessCompleteCallback event to notify the controlling application when kernel function completes a processing request
APLNext Supervisor Fires the ProcessCompleteCallback event to notify the controlling application when kernel function completes a processing request
APL+Win Controlling Application ProcessCompleteCallBack handler function receives kernel function result
APL+Win Controlling Application ProcessCompleteCallBack handler function receives kernel function result
APLNext Supervisor
Fires the OnProcessAllDoneCallback event to notify the controlling application all requests complete
APL+Win Controlling Application
OnProcessAllDoneCallback handler function consolidates all processing request results to complete the overall application system result and stops the Supervisor

Conceptual Example #1 – Stochastic Simulation of an Economic Process
Suppose that a model of an economic process has been prepared which depends on assumptions which will vary in the future according to some random variables. In order to obtain a future estimate of the outcome of this economic process, many 'independent trials' of this model are performed, each of which uses a pseudo-randomly-selected assumption set. If these trials were performed asynchronously, the overall simulation time could be significantly reduced.

In this example, the 'kernel' function would be the function which represents the model and has a specific assumption set as its argument. The 'controlling application' would select the desired number of assumption sets based upon appropriate probability distributions and request the APLNext Supervisor to execute the 'kernel' functions for each of these assumption sets.

As the APLNext Supervisor satisfies these processing requests, the 'ProcessCompleteCallback' event would fire and execute the 'ProcessCompleteCallback' event handler function of the 'controlling application'. This 'ProcessCompleteCallback' event handler function would receive the results of a specific completed 'kernel' function processing request and accumulate these results for subsequent consolidation, analysis and reporting.

After all such processing requests have been completed and their associated 'ProcessCompleteCallBack' events have fired, the 'ProcessAllDoneCallback' event would fire. The 'ProcessAllDoneCallback' event handler would perform consolidation, analysis and reporting on the 'kernel' function results accumulated by the individual 'ProcessCompleteCallback' event handlers. For example the 'ProcessAllDoneCallback' event handler function might average some results, compared the distribution of some results to a known distribution, etc.

Conceptual Example #2 – Employee Benefit Plan Valuation

Suppose that an employee benefit plan sponsored by an employer has a large number of participants and beneficiaries and that a valuation is required to estimate the present value of the employer's liability for benefits anticipated to be paid under the plan to these participants and beneficiaries.

A 'kernel' function could be developed which determined the benefit liability under the plan for a plan participant under a specific assumption set. Rather than executing this function sequentially one employee record at a time, the APLNext Supervisor could be used to execute it in parallel, potentially reducing overall processing time when hardware with multiple processors is available. The application-specific results container might be an APL array which contains the individual participant liability for the census under consideration. The 'ProcessCompleteCallback' event handler function would fill this array as each 'kernel' function completed processing of an employee. When all 'kernel' function executions are complete, the 'ProcessAllDoneCallback' event handler function would consolidate, analyze and report the employer's liability under the plan.

Another alternative implementation might be to perform the employee-level calculations en-masse using memory-resident APL arrays and instead use the APLNext Supervisor to perform a stochastic simulation using varying assumption sets, reflecting varying future investment earnings, expenses and employment levels which might apply during the plan's liability period.

APLNext Supervisor for APL+Win

APLNext Supervisor FAQ

Q1 How do I license the APLNext Supervisor?

A1 The initial version of the APLNext Supervisor was included with the customer subscription to APL+Win version 10. Subsequent versions of the APLNext Supervisor are included with subsequent versions of APL+Win.

There is no additional license cost for the APLNext Supervisor for APL+Win subscribers. APL+Win subscribers may download the APLNext Supervisor installer from the APL2000 Forum.

Contact sales@apl2000.com for additional information.

Q2 How do I obtain technical assistance for the APLNext Supervisor?

A2 Carefully review the APLNext Supervisor pdf-format documentation installed with the product. The most up-to-date version of the APLNext Supervisor documentation is on the APL2000 Forum section for the Supervisor. Customers with a current subscription to APL+Win may submit their questions to support@apl2000.com or post them to the APL2000 Forum.

The APLNext Supervisor section of the APL2000 Forum includes several sample applications illustrating the design and setup of a prototype application incorporating APLNext Supervisor technology.

Customized consulting to configure your application system to use the APLNext Supervisor is available by contacting sales@apl2000.com.

Q3 Will my application system performance improve using parallel processing?

A3 Generally yes, if the application system is calculation-intensive and a significant part of the application-specific algorithm can be executed in parallel. Results depend on the skill of the application programmer and the re-configuration of the application system using parallel programming techniques. If the application system processing performance is limited by file input and output, performance may not be improved. Using parallel processing without multi-processor hardware will not benefit performance. Using parallel processing when the processing to be performed is not calculation-intensive or the data sets are very small will not improve performance because marshaling data to and from the multiple processors requires overhead compared to using traditional in-memory, en-masse, array-based APL calculations.

Q4 Is there a performance difference when APL+Win or C# is the 'controlling application'?

A4 There can be a significant performance advantage when using C# as the 'controlling application'. APL+Win is based on Microsoft Win32 and is therefore a single-threaded Windows application. C# is based on Microsoft .Net and is therefore designed for multi-threading. This is why the APLNext Supervisor is built using C# and Microsoft .Net.

When the ‘controlling application’ submits asynchronous processing requests to execute the APL+Win ‘kernel’ functions, the APLNext Supervisor places these requests into a ‘request’ queue. The order in which these processing requests are completed is not necessarily the order in which they were submitted, because each execution of the ‘kernel’ function may take a different time.

When a processing request is completed, the APLNext Supervisor removes the request from the ‘request’ queue and places the completed request into a ‘completed’ queue. As requests are added to the ‘completed’ queue, the APLNext Supervisor must now signal the ‘controlling application’ by firing the ‘ProcessCompleteCallBack’ event. The ‘controlling application’ has subscribed to this event in order to receive notification of a completed asynchronous processing request.

If APL+Win is the ‘controlling application’, since is single-threaded, the APLNext Supervisor must fire the ‘ProcessCompleteCallBack’ events one-at-a-time, waiting for APL+Win to signal-back that it has received the event notification. In the single-threaded environment of APL+Win, the signal-back from APL+Win occurs only when the execution of the APL+Win, application-specific ‘ProcessCompleteCallBack’ event handler has been completed.

Thus it is vitally important that the APL+Win, application-specific ‘ProcessCompleteCallBack’ event handler of the ‘controlling application’ be a very efficient function. Otherwise the event handler processing of the APL+Win ‘controlling application’ could become an event handler ‘bottleneck’ in the overall application system processing.

If C# is the ‘controlling application’, the APLNext Supervisor can fire the ‘ProcessCompleteCallBack’ events in rapid-succession, virtually immediately, upon the completion of an asynchronous processing request. This is because C# can immediately respond to the ‘ProcessCompleteCallBack’ event upon receiving notification of that event firing, even though the C# application-specific event handler method may not have completed execution. This latter advantage is a feature of .Net languages which are inherently designed for multi-threading. Thus C# as the ‘controlling application’ cannot become an event handler ‘bottleneck’ in the overall application system processing.

Q5 How do I configure the APLNext Supervisor for the number of processors?

A5 When deploying an application which incorporates the APLNext Supervisor each target machine can have a different number of processors. To optimize the APLNext Supervisor configuration some testing may be required. Generally it is appropriate to assume that the operating system will consume one processor and the APLNext Supervisor and associated ‘controlling application’ will consume another processor if the ‘Start’ rather than the ‘StartSync’ mode of APLNext Supervisor operation is selected. The remaining processors on the target machine can each handle one asynchronous instance of the APL+Win ActiveX engine.

Although it is possible to increase the 'maxpool' setting above the number of (virtual) processors in the workstation, starting with #virtual processors – 2 (asynchronous mode) or #virtual processors – 1 (synchronous mode) is suggested, with subsequent variation of this value to observe potential differences in performance.

Write and include a configuration section in the 'controlling application' which will:

- Create an instance of the APLNext Supervisor
- Get the value of the NProcessors property of that instance of the APLNext Supervisor which applies to the machine on which the APLNext Supervisor has been deployed.
- Use the information obtained about the target machine's processors to appropriately set the 'minpool' and 'maxpool' elements of the APLNext Supervisor configuration, e.g. (using an APL+Win-based controlling application):

```
S←'S'⎕wi 'Create' 'APLNext.Supervisor'
config←'<?xml version="1.0"?>' '<config>' '<workspaces>'
 config←config, ⊂'<workspace id="',WSID,'">'
 config←config, ⊂'<minpool>1</minpool>'
 config←config, ⊂'<maxpool>',(⍕1⌈¯2+2⊃ S ⎕wi 'xNProcessors'),'</maxpool>'
 config←config, ⊂'<timeout>',(⍕60×1000),'</timeout>'
 config←config, ⊂'<debug>1</debug>'
 config←config, ⊂'<visible>1</visible>'
 config←config, ⊂'<wslocation>',⎕wsid,'.w3</wslocation>'
 config←config, ⊂'<wssize>16000000</wssize>'
 config←config, ⊂'<evlevel>2</evlevel>'
 config←config, ⊂'<busyid />'
 config←config, ⊂'<user_state>started</user_state>'
 config←config, ⊂'</workspace>'
 config←config, ⊂'</workspaces>'
 config←config, ⊂'</config>'
  config←⎕enlist config
…
 S ⎕wi 'XOpenConfigFromText' config
```

Q6 What is the overall application system architecture when using the APLNext Supervisor?

A6 The 'controlling application' function, which instantiates and configures the APLNext Supervisor instance, can be programmed in APL+Win (or any language which supports an ActiveX interface) or C# (or any .Net language). The 'Kernel Function', which processes the queued tasks, is always written in APL+Win.

In 'async' mode, i.e. using the 'Start' method:

- Create an instance of the APLNext.Supervisor object in the 'controlling application'

- Create the xml-format configuration and use the configuration via the 'OpenConfigurationFromText' or 'OpenConfigurationFromFile' in the 'controlling application'. The configuration includes parameters such as the 'minpool' and 'maxpool' values which determine the number of instances of the APL+Win ActiveX engine which will run the 'Kernel Function' to process the tasks queued by the 'controlling application'.
- Create the handler functions for the 'ProcessCompleteCallback' and 'ProcessAllDoneCallback' events. These functions are written in the same programming language as the 'controlling application'.
- Create the 'Kernel Function' to perform processing of tasks which will be submitted to the APLNext Supervisor queue. Generally this function has an explicit result which will be received by the 'ProcessCompleteCallback' event handler or the 'Kernel Function' can also persist its results to a shared repository such as an APL+Win component file or a database like Microsoft SQL Server. The 'Kernel Function' may be in a separate workspace from that of the 'controlling application'. The 'Kernel Function' will run in an independent instance of the APL+Win ActiveX engine. There may be more than one 'Kernel Function'.
- Use the 'Start' method in the 'controlling application'
- In the 'Controlling App' subscribe to the 'ProcessCompleteCallback' event before queueing any tasks.
- Submit processing tasks to the APLNext Supervisor queue using the 'BeginCall' method in the 'controlling application'. Processing of these tasks by the 'Kernel Function' will begin immediately. This 'Kernel Function' processing is asynchronous with respect to each instance of the APL+Win ActiveX engine, i.e. each ActiveX instance independently processes its assigned queued task. This 'Kernel Function' processing is asynchronous with respect to the 'controlling application', i.e. the execution of the 'controlling application' is not stopped while 'Kernel Function' processing occurs unless the 'controlling application' explicitly checks the 'QueueSize' property to prevent completion of its execution until the 'QueueSize' property value is zero.
- In the 'Controlling App' subscribe to the 'ProcessAllDoneCallback' event after queueing all tasks.
- When processing of all queued tasks is complete, use the 'Stop', 'Close' and 'Delete' methods to dispose of the APLNext Supervisor instance. This can be done in the 'ProcessAllDoneCallback' event handler, deferring it until the execution of that event handler has completed.

In 'sync' mode, i.e. using the 'StartSync' method:

- Create an instance of the APLNext.Supervisor object in the 'controlling application'
- Create the xml-format configuration and use the configuration via the 'OpenConfigurationFromText' or 'OpenConfigurationFromFile' in the 'controlling application'. The configuration includes parameters such as the 'minpool' and 'maxpool' values which determine the number of instances of the APL+Win ActiveX engine which will run the 'Kernel Function' to process the tasks queued by the 'controlling application'.
- Create the 'Kernel Function' to perform processing of tasks which will be submitted to the APLNext Supervisor queue. Since the APLNext Supervisor events do not fire in this mode of APLNext Supervisor operation, the 'Kernel Function' must persist its results to a shared

repository such as an APL+Win component file or a database like Microsoft SQL Server. The 'Kernel Function' may be in a separate workspace from that of the 'controlling application'. The 'Kernel Function' will run in an independent instance of the APL+Win ActiveX engine. There may be more than one 'Kernel Function'.

- Use the 'StartSync' method in the 'controlling application'
- Submit processing tasks to the APLNext Supervisor queue using the 'BeginCall' method in the 'controlling application'. Processing of these tasks by the 'Kernel Function' will not being at this point in the execution of the 'controlling application'.
- After all the tasks have been submitted to the APLNext Supervisor queue by the 'controlling application', use the 'ExecuteAPL' method in the 'controlling application' to commence the processing of the queued tasks by the 'Kernel Function'. This processing is asynchronous with respect to each instance of the APL+Win ActiveX engine, i.e. each ActiveX instance independently processes its assigned queued task. This processing is synchronous with respect to the 'controlling application', i.e. the execution of the 'controlling application' is stopped until the processing by the 'Kernel Function' of all queued tasks is complete.
- When processing of all queued tasks is complete, use the 'Stop', 'Close' and 'Delete' methods in the 'controlling application' to dispose of the APLNext Supervisor instance. This can be done in the 'controlling application' because its execution will recommence when all queued tasks have been processed.

Q7 Are the versions of the APLNext Supervisor and APL+Win linked?

A7 Yes, when the APLNext Supervisor 'Start' or 'StartSyn' method is used by the 'Controlling Application, the Supervisor will determine if a supported version of the APL+Win ActiveX engine registered on the target workstation.

If not, the APLNext Supervisor will throw an error. The error message will depend on the version of the Supervisor installed on the target workstation, e.g.:

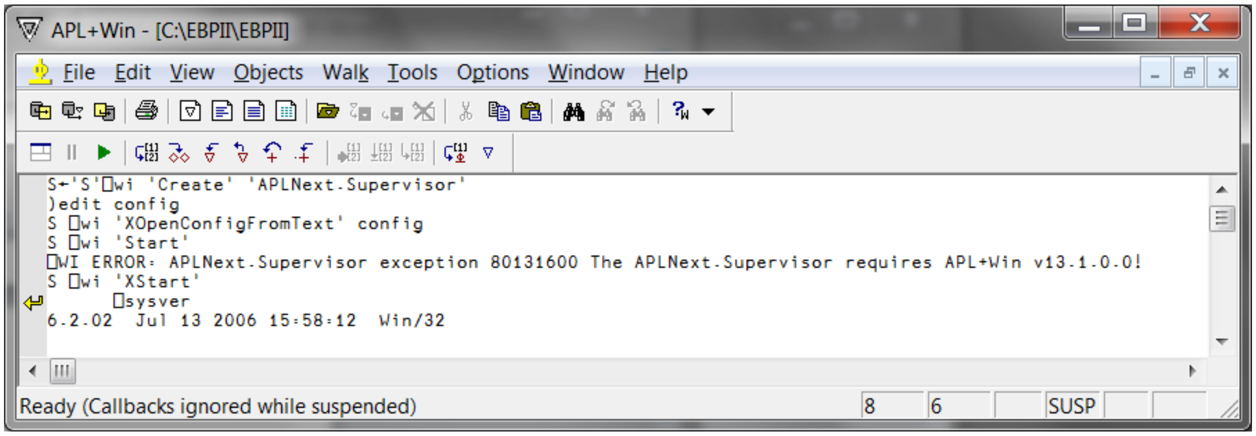
- ⎕WI ERROR: APLNext.Supervisor exception -2146232832 (0x80131600) APL+Win COM instantiation failed
- ⎕WI ERROR: APLNext.Supervisor exception 80131600 APL+Win version not supported: 14.2.01 Aug 18 2014 11:09:09 Win/32

In this case the Supervisor log file, with location specified in the Supervisor configuration provided by the 'controlling application', will contain a related error message, e.g. "APL+Win version 15 is required. Version not supported: 14.2.01 Aug 18 2014 11:09:09 Win/32".

For example the APLNext Supervisor v1.7.1.0 requires APL+Win version 14.0.0.0 or higher and the APLNext Supervisor v1.9.4.0 requires APL+Win version 15.0.0.0 or higher. Subsequent versions of the APLNext Supervisor may require specific minimum versions of APL+Win for proper operation.

The example below illustrates that the 'Create' method can be successfully used to create an instance of the APLNext Supervisor, but the 'Start' (or 'StartSync') method of the APLNext

Supervisor will fail with an explicit error message if an appropriate version of APL+Win is not registered on the target workstation.


APL+Win - [C:\EBPII\EBPII]
File Edit View Objects Walk Tools Options Window Help

```
S←'S'⎕wi 'Create' 'APLNext.Supervisor'
)edit config
S ⎕wi 'XOpenConfigFromText' config
S ⎕wi 'Start'
⎕WI ERROR: APLNext.Supervisor exception 80131600 The APLNext.Supervisor requires APL+Win v13.1.0.0!
S ⎕wi 'XStart'
      ⎕sysver
6.2.02  Jul 13 2006 15:58:12  Win/32
```

Ready (Callbacks ignored while suspended) | 8 | 6 | SUSP

Q8 Do I need to know .Net, Visual Studio or C# to use the APLNext Supervisor?

A8 No, all features of the APLNext Supervisor may be accessed using APL+Win. The 'kernel' function(s) which satisfy the processing requests submitted to the Supervisor by the 'controlling application' are always application-specific APL+Win user-defined functions. The 'controlling application' is generally an application-specific APL+Win function.

Since the APLNext Supervisor is based upon Microsoft .Net, the 'controlling application' may optionally be a C# or other .Net programming language method. In this documentation and in the APLNext Supervisor prototype application samples, 'controlling application' implementations using APL+Win and using C# are provided.

Q9 Is the APLNext Supervisor a 'client/server' application?

A9 Yes, in a conceptual way an application system employing the APLNext Supervisor contains functions which have 'client' and 'server' roles.

The 'controlling application' which submits processing requests to the APLNext Supervisor has the role of 'client'.

The 'kernel' function(s) which satisfy these processing requests have the role of 'servers'.

The APLNext Supervisor has the role of communicating between the 'client' and the 'servers' by:

- Facilitating the configuration of a Supervisor instance by the 'controlling application'
- Queueing processing requests submitted by the 'controlling application'
- Scheduling, commencing and terminating independent processing threads in which the 'kernel' function execution takes place
- Passing arguments from the 'controlling application' to the 'kernel' function(s).
- Reporting 'kernel' function results to the 'controlling application'
- Supporting events which can be subscribed by the 'controlling application'

From a hardware point of view the APLNext Supervisor is designed to support the both conceptual 'client' and 'server' roles on the same workstation on which the APLNext Supervisor has been installed. Thus for an APLNext Supervisor-based application system, there is no 'server' hardware which is separate from the 'client' hardware. An application system configured to use the APLNext Supervisor may be run on 'server' hardware as if that hardware were a 'workstation'.

For scenarios in which the application will be deployed on hardware where the 'client' role is supported on 'workstation' hardware and the 'server' role is supported on separate, possibly-shared 'server' hardware, the APLNext Application Server product provides similar functionality to that of the APLNext Supervisor on a single workstation.

Application systems which are configured to use the APLNext Supervisor may be readily configured to use the APLNext Application Server because configuration for the APLNext Supervisor requires the clear identification and separation of the 'client' and 'server' function roles.

Q10 How many processing requests can be submitted to the Supervisor?

A10 Any application running on a workstation, including an application system based on the APLNext Supervisor, depend on the memory made available to it by the operating system. A specific answer to this question depends on application-specific information, but the major considerations are outlined below.

The memory requirements of the arguments to the 'kernel' function are the main factor in limiting the number of processing requests which may be submitted by the 'controlling application' to an instance of the APLNext Supervisor using the 'BeginCall' method.

An important way to minimize the memory requirements of submitted processing request for a 'kernel' function which would otherwise require a large argument is to store the large arguments in a shared APL+Win component file. Instead of passing the large argument data to the 'kernel' function via the 'BeginCall' method of the APLNext Supervisor, which would otherwise rapidly exhaust the memory of the queue, pass the component number of that argument data to the 'kernel' function bia the 'BeginCall' method of the APLNext Supervisor.

Once the 'kernel' function begins execution to handle the submitted processing request on an independent thread created by the APLNext Supervisor, the memory requirements of the functions and data in the APL+Win workspace containing the 'kernel' function will determine how many simultaneous processing threads can be accommodated on a particular workstation. Since this information is application-specific, some experimentation which varies the 'maxpool' value in the APLNext Supervisor configuration provided by the 'controlling application' will be necessary. Establishing the optimal 'maxpool' setting also requires consideration of the number of processors available on the target workstation.

Processing requests which take significant processing time will consume memory for a longer time and will also limit the number of requests which can be simultaneously processed.

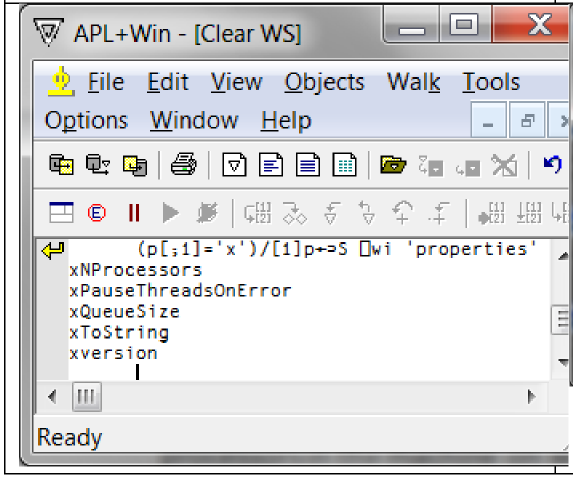
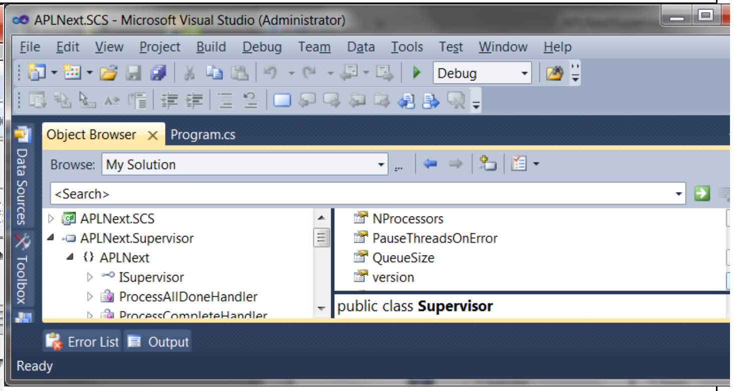
When a 'kernel' function has completed execution to satisfy a queued processing request, consideration of the memory required to contain the 'kernel' function results is necessary.

Generally the ‘kernel’ function results are combined into an overall summary result which substantially reduces the memory requirements. In the situation where a large amount of ‘kernel’ function result must be saved, an option is to store the result in a shared APL+Win component file indexed by the ‘processId’ given to the processing request by the ‘controlling application’ when that request was submitted using the ‘BeginCall’ method of the APLNext Supervisor.

APLNext Supervisor for APL+Win

Supervisor Properties

These properties can be used by the ‘controlling application’ and the event handler functions.

In APL+Win	In Visual Studio (C#)
	

```
      (p[;1]='x')/[1]p←⊃S ⎕wi 'properties'
xNProcessors
xPauseThreadsOnError
xQueueSize
xToString
xversion
```

APL+Win - [Clear WS]
File Edit View Objects Walk Tools
Options Window Help
Ready

APLNext.SCS - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Object Browser
Program.cs
Browse: My Solution
<Search>
Data Sources
Toolbox
Error List
Output
Ready

```
APLNext.SCS
APLNext.Supervisor
  APLNext
    ISupervisor
    ProcessAllDoneHandler
    ProcessCompleteHandler
```

```
NProcessors
PauseThreadsOnError
QueueSize
version
```

public class **Supervisor**

NProcessors

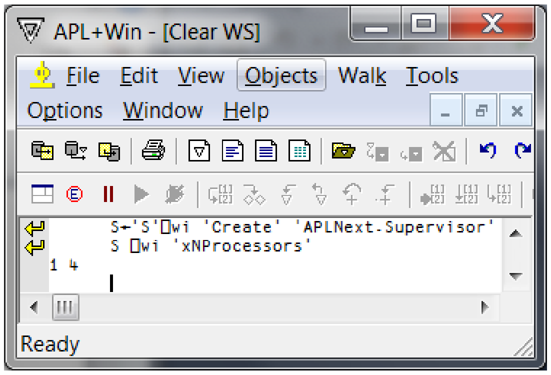
(Read Only) Returns a two element integer vector of containing the number of physical and virtual processors in the machine on which this instance of the APLNext Supervisor has been installed. This information is also available by manual inspection of the Windows Control Panel > System > Device Manager > Processors tree element.

Syntax:

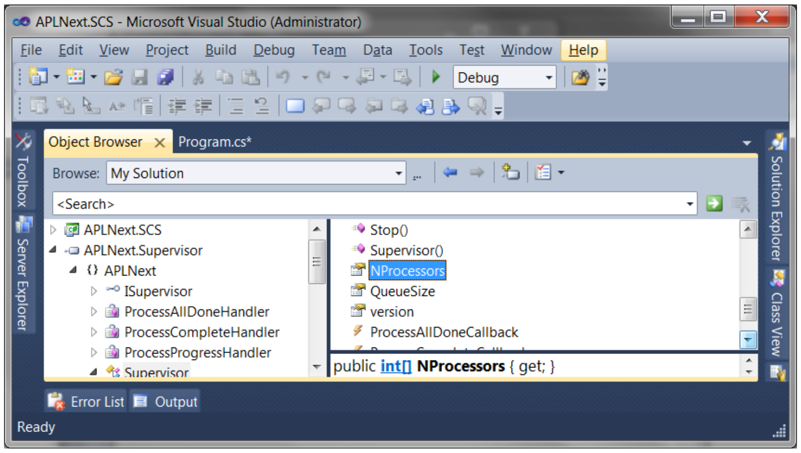
Controlling Application	xNProcessors property syntax
APL+Win	'spvr' ⎕wi 'xNProcessors'
C#	Int32[] np = spvr.NProcessors;

The 'NProcessors' property value can be used to configure the APLNextSupervisor 'minpool' and 'maxpool' configuration elements under program control to customize the configuration for machines with varying capacity.

Example: The result '1 4' below indicates that there is one physical processor and four virtual processors on this particular machine.



```
      S←'S'⎕wi 'Create' 'APLNext.Supervisor'
      S ⎕wi 'xNProcessors'
1 4
```



```
public int[] NProcessors { get; }
```

PauseThreadsOnError

If the value of this property is true(C#) or 1(APL+Win) and if the APLNext Supervisor detects an error condition, the Supervisor will signal all uncompleted processing request threads to pause. Because processing threads are asynchronous there is no guarantee that all threads will be paused when the 'PauseThreadsOnError' property value is true/1. A thread may complete processing before the .Net Framework has been able to issue the pause notification to that thread.

The APLNext Supervisor can detect error conditions which are within its domain including:

- Errors creating a new instance of the APL+Win ActiveX engine, such as :
    - The applicable APL+Win interpreter is not registered as an ActiveX component
    - The Windows operating system will not grant the APLNext Supervisor an additional thread
    - There is no available memory on the workstation for the APLNext Supervisor operation
    - The workstation memory available to the APLNext Supervisor does not permit adding the current BeginCall processing request to the APLNext Supervisor processing request queue
    - APL+Win ActiveX engine instance could not be created, e.g. due to memory or permissions
    - APL+Win ActiveX engine version incorrect
    - APL+Win ActiveX engine could not load the specified 'kernel' function workspace

- Errors using the APL+Win ActiveX engine interface, such as executing an APL+Win ActiveX engine method, e.g. the 'Call', 'SysCall' or 'SysCmd' methods which are used by the APLNext Supervisor to interact with the instances of the APL+Win ActiveX engine and execute the 'kernel' functions.

- Errors involving .Net interoperation with the Microsoft Windows COM interface

Execution of the remaining processing request threads, if any, can be resumed by:

- Set the Supervisor 'PauseThreadsOnError' property value to false/0
- Use the Supervisor 'Resume' method

To clear the Supervisor pending processing requests queue:

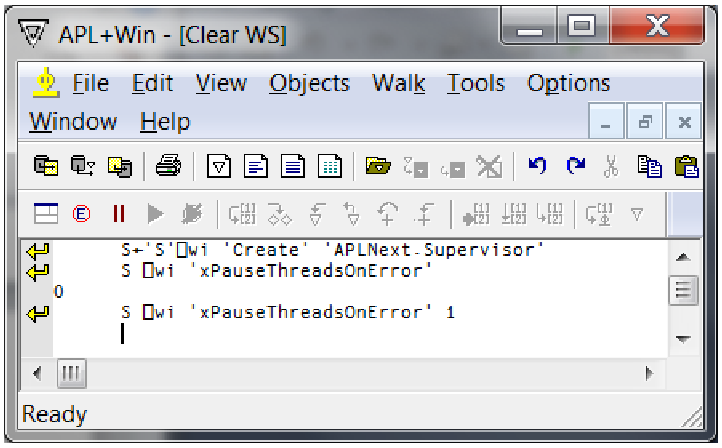
- Use the Supervisor 'Stop' method
- Use the Supervisor 'Start' method

The APLNext Supervisor configuration contains a 'debug' element which controls the use of the Timeout element. This 'debug' element is not affected by the value of the 'PauseThreadsOnError' property.

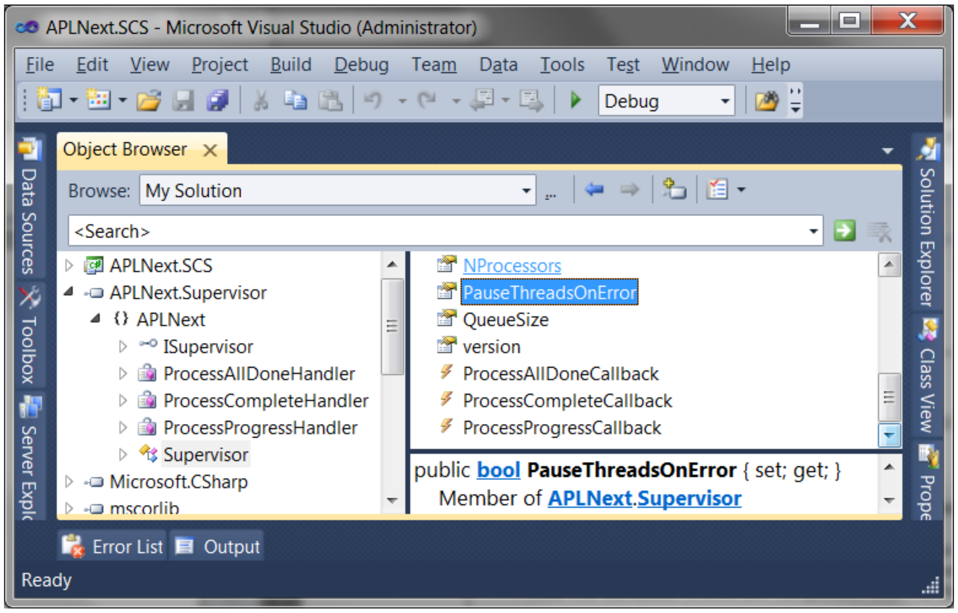
Syntax:

Controlling Application	**xPauseThreadsOnError Syntax**
APL+Win	'spvr' ⎕wi 'xPauseThreadsOnError' 1
C#	Bool b = spvr.PauseThreadsOnError;

Example:



```
      S←'S'⎕wi 'Create' 'APLNext.Supervisor'
      S ⎕wi 'xPauseThreadsOnError'
0
      S ⎕wi 'xPauseThreadsOnError' 1
```



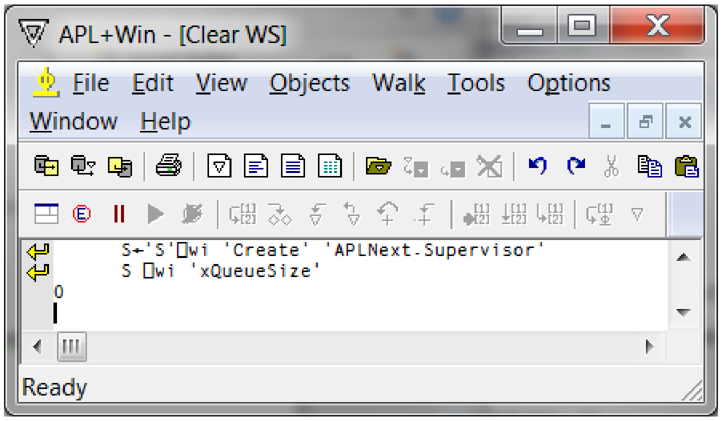
public **bool** **PauseThreadsOnError** { set; get; }
Member of **APLNext.Supervisor**

QueueSize

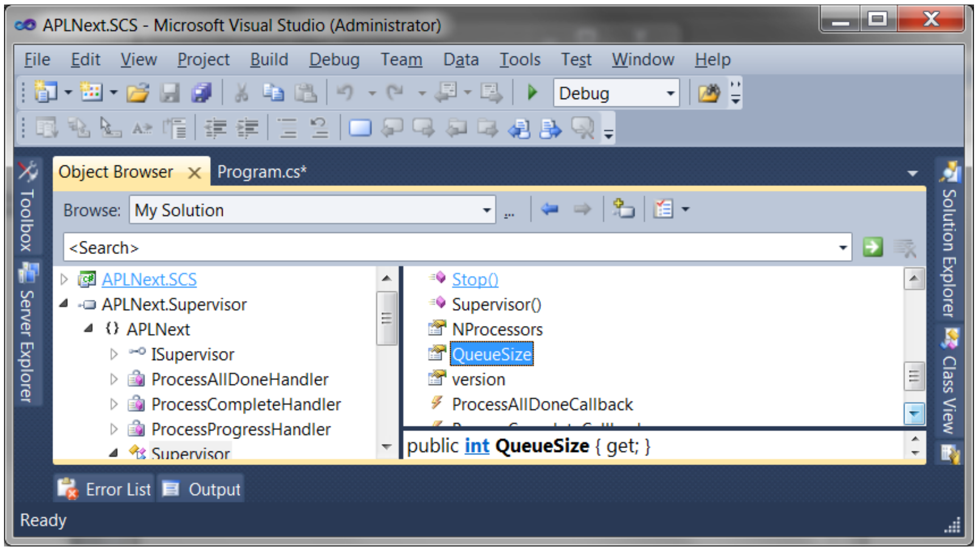
(Read Only) Returns the number of outstanding processing requests [pending BeginCall requests] for this instance of the APLNext Supervisor.

Syntax:

Controlling Application	xQueueSize property syntax
APL+Win	'spvr' ⎕wi 'xQueueSize'
#	Int32 qs = spvr.QueueSize;



```
      S←'S'⎕wi 'Create' 'APLNext.Supervisor'
      S ⎕wi 'xQueueSize'
0
```



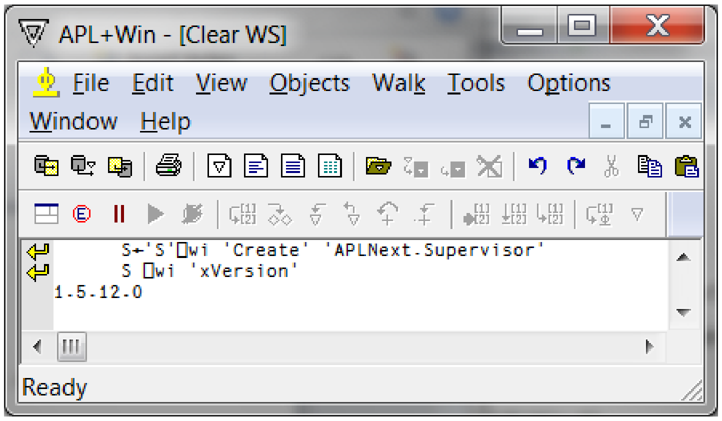
```
public int QueueSize { get; }
```

version

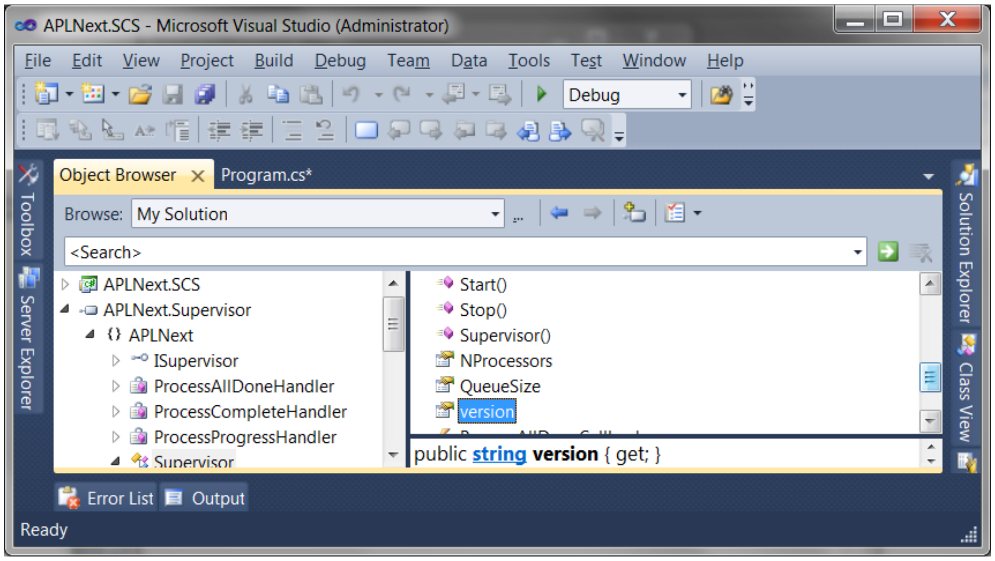
(Read Only) Returns the version number of this instance of the APLNext Supervisor in the format [major].[minor].[revision].[build]

Syntax:

Controlling Application	xversion property syntax
APL+Win	'spvr' ⎕wi 'xversion'
C#	string s = spvr.version;



```
      S←'S'⎕wi 'Create' 'APLNext.Supervisor'
      S ⎕wi 'xVersion'
1.5.12.0
```

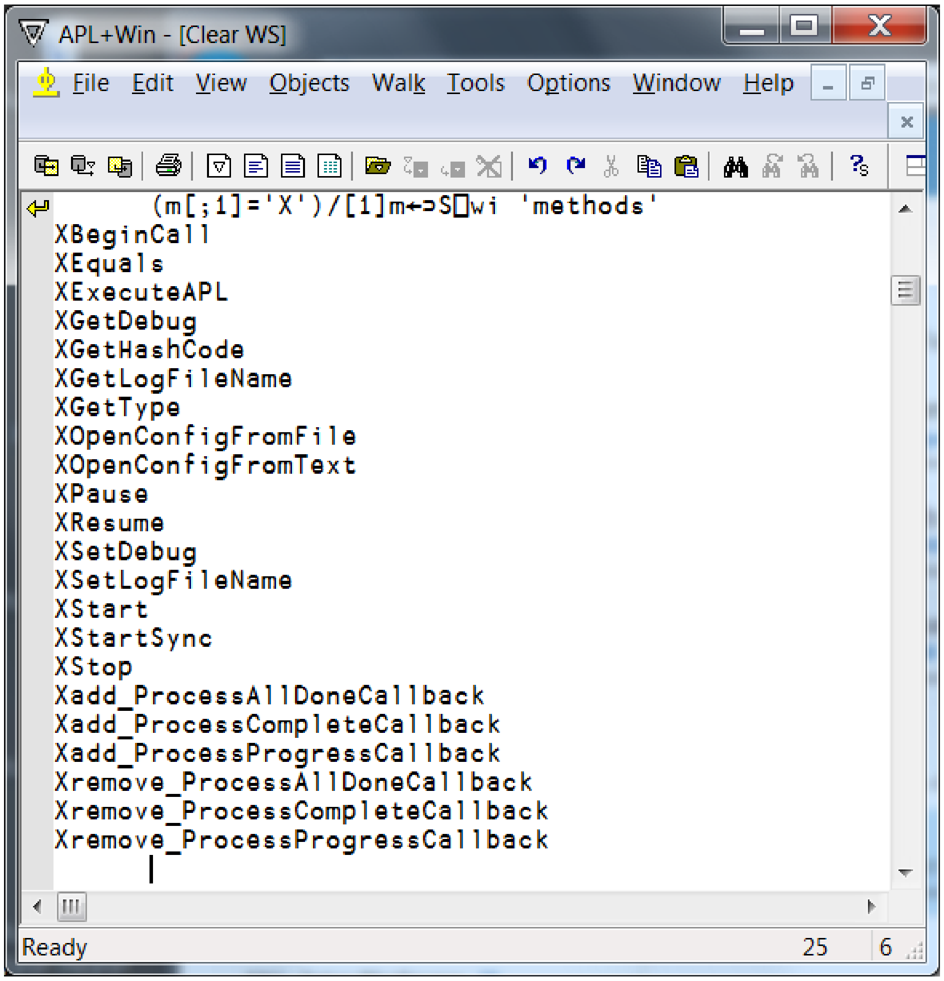


```
public string version { get; }
```

Supervisor Methods

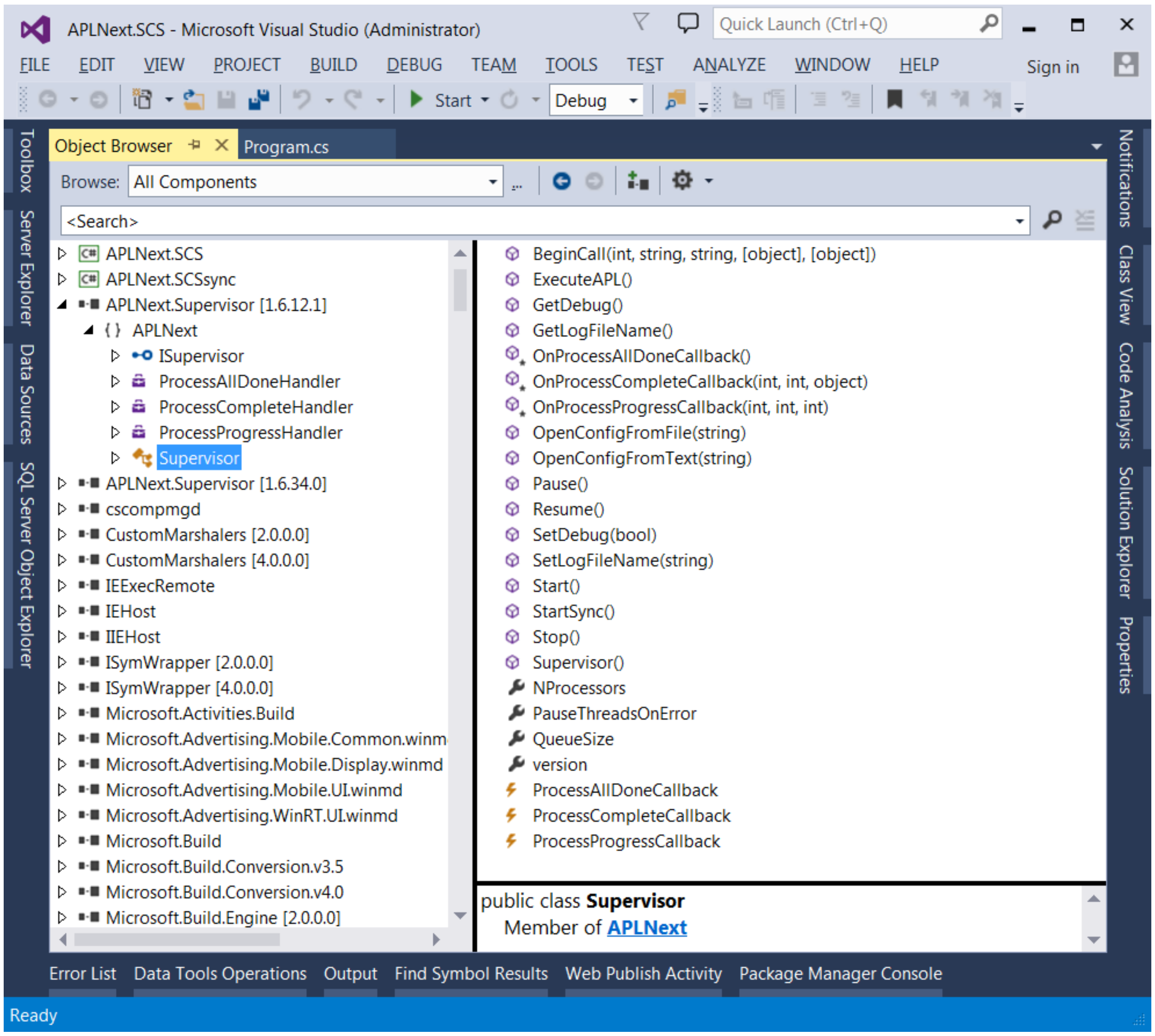
The APLNext Supervisor methods are used by the 'controlling application' or the event handler functions to request the execution of APL+Win 'kernel' functions and configure, start or stop this instance of the APLNext Supervisor.

In APL+Win:



```
      (m[;1]='X')/[1]m←⊃S⎕wi 'methods'
XBeginCall
XEquals
XExecuteAPL
XGetDebug
XGetHashCode
XGetLogFileName
XGetType
XOpenConfigFromFile
XOpenConfigFromText
XPause
XResume
XSetDebug
XSetLogFileName
XStart
XStartSync
XStop
Xadd_ProcessAllDoneCallback
Xadd_ProcessCompleteCallback
Xadd_ProcessProgressCallback
Xremove_ProcessAllDoneCallback
Xremove_ProcessCompleteCallback
Xremove_ProcessProgressCallback
```

In Microsoft Visual Studio (C#):



Browse: All Components

<Search>

```
▷ APLNext.SCS
▷ APLNext.SCSsync
◢ APLNext.Supervisor [1.6.12.1]
  ◢ { } APLNext
      ▷ ISupervisor
      ▷ ProcessAllDoneHandler
      ▷ ProcessCompleteHandler
      ▷ ProcessProgressHandler
      ▷ Supervisor
▷ APLNext.Supervisor [1.6.34.0]
▷ cscompmgd
▷ CustomMarshalers [2.0.0.0]
▷ CustomMarshalers [4.0.0.0]
▷ IEExecRemote
▷ IEHost
▷ IIEHost
▷ ISymWrapper [2.0.0.0]
▷ ISymWrapper [4.0.0.0]
▷ Microsoft.Activities.Build
▷ Microsoft.Advertising.Mobile.Common.winm
▷ Microsoft.Advertising.Mobile.Display.winmd
▷ Microsoft.Advertising.Mobile.UI.winmd
▷ Microsoft.Advertising.WinRT.UI.winmd
▷ Microsoft.Build
▷ Microsoft.Build.Conversion.v3.5
▷ Microsoft.Build.Conversion.v4.0
▷ Microsoft.Build.Engine [2.0.0.0]
```

```
BeginCall(int, string, string, [object], [object])
ExecuteAPL()
GetDebug()
GetLogFileName()
OnProcessAllDoneCallback()
OnProcessCompleteCallback(int, int, object)
OnProcessProgressCallback(int, int, int)
OpenConfigFromFile(string)
OpenConfigFromText(string)
Pause()
Resume()
SetDebug(bool)
SetLogFileName(string)
Start()
StartSync()
Stop()
Supervisor()
NProcessors
PauseThreadsOnError
QueueSize
version
ProcessAllDoneCallback
ProcessCompleteCallback
ProcessProgressCallback
```

public class **Supervisor**
Member of **APLNext**

BeginCall

This method requests the execution of a nyladic, monadic or dyadic APL+Win kernel function in a specified APL+Win workspace. This method will add the processing request associated with the APL+Win function 'fnname' in the APL+Win workspace 'wsId' to the APLNext Supervisor request queue. The 'processid' is an application-specific integer used by the 'controlling application' to identify the specific parallel processing request.

Arguments:

Argument Name	**Description**	**Optional Additional Argument**
processId	Programmer-specified, integer processing request identifier	Not Applicable
wsId	Workspace identifier defined in the APLNext Supervisor configuration	Not Applicable
kernelfnname	Niladic Kernel Function Name	Not Applicable
	Monadic Kernel Function Name	Right argument
	Dyadic Kernel Function Name	Left Argument

Syntax and Result:

Controlling Application	**Kernel Function Syntax**	**BeginCall method syntax**	**Result**
APL+Win	Niladic	'spvr' ⎕wi 'XBeginCall' processId wsId kernelfnname	
	Monadic	'spvr' ⎕wi 'XBeginCall' processId wsId kernelfnname rarg	
	Dyadic	'spvr' ⎕wi 'XBeginCall' processId wsId kernelfnname rarg larg	1/OK 0/Fail
C#	Niladic	bool b = spvr.BeginCall(processId, wsId, kernelfnname);	
	Monadic	bool b = spvr.BeginCall(processId, wsId, kernelfnname, rarg);	
	Dyadic	bool b = spvr.BeginCall(processId, wsId, kernelfnname, rarg, larg);	True/false

There can be more than one APL+Win 'kernel' function each with a different argument structure. Each 'BeginCall' method use can request the execution of one of those 'kernel' functions.

When C# is the controlling application, the optional right and left arguments of the APL+Win kernel function must be C# data types which have analogous APL+Win data types, e.g. string, double, Int32 or object arrays of these data types.

APLNext Supervisor for APL+Win

Create

The method is applicable to all APL+Win ⎕wi-created objects including the APLNext.Supervisor. If APL+Win is the controlling application, the APLNext.Supervisor must be registered as an ActiveX object on the target workstation.

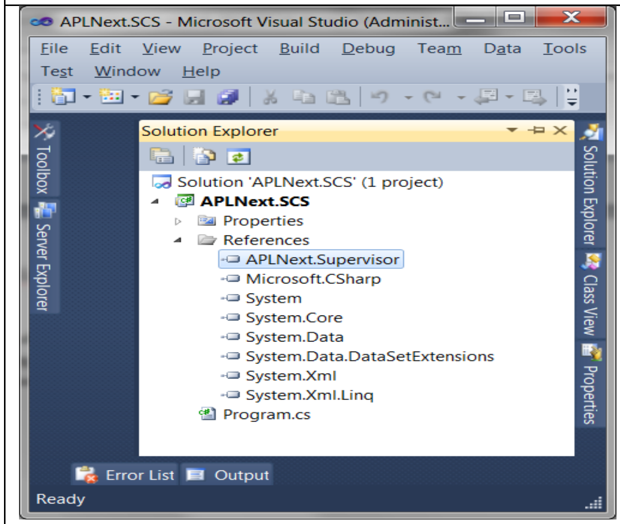
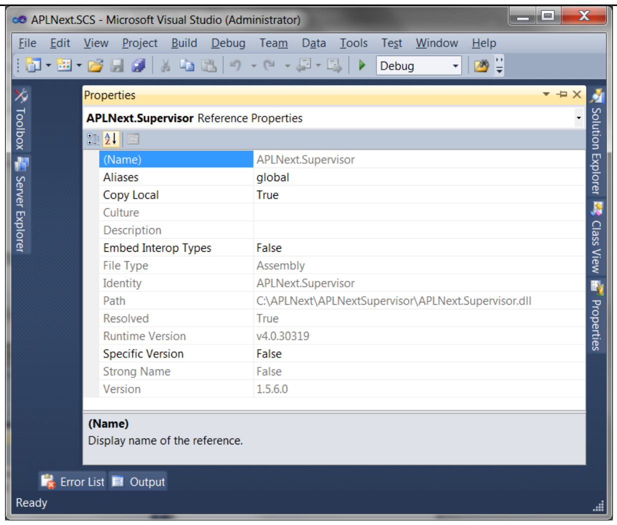
Arguments:

(none)

Syntax and Result:

Controlling Application	Create method syntax	Result
APL+Win	Spvr ← ‘spvr’ ⎕wi ‘Create’ ‘APLNext.Supervisor’	0/Fail 1/OK
C#	APLNext.Supervisor spvr = new APLNext.Supervisor();	Instance of the APLNext.Supervisor class

If C# is the controlling application, the APLNext Supervisor need not be registered as an ActiveX object on the target machine because the APLNext.Supervisor .Net assembly can be directly referenced in a C# project. In C# the analogue of the ‘Create’ method is the ‘new’ key word in association with the constructor for the APLNext.Supervisor class.

C# Controlling Application Reference to the APLNext.Supervisor .Net Assembly	APLNext.Supervisor Reference Properties
	

APLNext.SCS - Microsoft Visual Studio (Administ...
File Edit View Project Build Debug Team Data Tools Test Window Help
Toolbox
Server Explorer
Solution Explorer
Class View
Properties
Error List
Output
Ready

Solution Explorer

- Solution 'APLNext.SCS' (1 project)
  - **APLNext.SCS**
    - Properties
    - References
      - APLNext.Supervisor
      - Microsoft.CSharp
      - System
      - System.Core
      - System.Data
      - System.Data.DataSetExtensions
      - System.Xml
      - System.Xml.Linq
    - Program.cs

APLNext.SCS - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Toolbox
Server Explorer
Solution Explorer
Class View
Properties
Error List
Output
Ready

Properties

APLNext.Supervisor Reference Properties

(Name)	APLNext.Supervisor
Aliases	global
Copy Local	True
Culture	
Description	
Embed Interop Types	False
File Type	Assembly
Identity	APLNext.Supervisor
Path	C:\APLNext\APLNextSupervisor\APLNext.Supervisor.dll
Resolved	True
Runtime Version	v4.0.30319
Specific Version	False
Strong Name	False
Version	1.5.6.0

(Name)
Display name of the reference.

Close

The method is applicable to all APL+Win □wi-created objects including the APLNext.Supervisor. After using the APLNext.Supervisor ‘Stop’ method, the APL+Win-based ‘controlling application’ would use the ‘Close’ method to end the specified instance of the APLNext.Supervisor. This method does not apply to a C# ‘controlling application’.

Arguments:

(none)

Syntax and Result:

Controlling Application	Close method syntax	Result
APL+Win	‘spvr’ □wi ‘Close’	0/Fail 1/OK
C#	Not applicable	Not applicable

Delete

The method is applicable to all APL+Win □wi-created objects including the APLNext.Supervisor. After using the APLNext.Supervisor 'Stop' method and the APLNext.Supervisor 'Close' method, the APL+Win-based 'controlling application' would use the 'Delete' method to dispose of the specified instance of the APLNext.Supervisor. This method does not apply to a C# 'controlling application'.

Arguments:

(none)

Syntax and Result:

Controlling Application	**Delete method syntax**	**Result**
APL+Win	'spvr' □wi 'Delete'	0 0ρ0
C#	Not applicable	Not applicable

ExecuteAPL

This method applies only if the APLNext Supervisor instance is started using the ‘StartSync’ method. The ‘ExecuteAPL’ method is used after the ‘controlling application’ function uses the ‘BeginCall’ method to submit processing requests to the APLNext Supervisor instance. When the ‘ExecuteAPL’ method is called the APLNext Supervisor instance will stop the execution of the ‘controlling application’ function and commence the processing of the queued tasks by the ‘Kernel Function’. Execution of the ‘controlling application’ will recommence after all queued tasks have been processed.

Arguments:

(none)

Syntax and Result:

Controlling Application	**Delete method syntax**	**Result**
APL+Win	‘spvr’ ⎕wi ‘ExecuteAPL’	1/OK 0/Fail
C#	bool b = spvr.ExecuteAPL	true/false

GetDebug

Obtain the current value of the APLNext Supervisor log file entry verbose setting. Do not confuse this method with the ‘Debug’ element in the XML-format APLNext Supervisor configuration.

Arguments:

(none)

Syntax and Result:

Controlling Application	**GetLogFileName method syntax**	**Result**
APL+Win	‘spvr’ ⎕wi ‘XGetDebug’	0/No or 1/Yes
C#	bool verboseLog = spvr.GetDebug();	True or False

GetLogFileName

Obtain the fully-qualified name of the log file in use with this instance of the APLNext Supervisor.

Arguments:

(none)

Syntax and Result:

Controlling Application	GetLogFileName method syntax	Result
APL+Win	‘spvr’ ⎕wi ‘XGetLogFileName’	Log file name
C#	string lfn = spvr.GetLogFileName();	

OpenConfigFromFile
Reads the XML-format APLNext Supervisor configuration file from disk and uses its content to configure this instance of the APLNext Supervisor.

Arguments:
configFileName: The full path of the XML-format APLNext Supervisor configuration file. Refer to the documentation of the schema of the APLNext Supervisor XML-format configuration document for an explanation of the elements of the APLNext Supervisor configuration.

Syntax and Result:

Controlling Application	**OpenConfigFromFile method syntax**	**Result**
APL+Win	'spvr' ⎕wi 'XOpenConfigFromFile' configFileName	1/OK 0/Fail
C#	bool b = spvr.OpenConfigFromFile(configFileName);	true/false

OpenConfigFromText
This method uses the memory-based, xml-format APLNext Supervisor configuration text string to configure this instance of the APLNext Supervisor.

Arguments:
xmlString: The xml configuration text. Refer to the documentation of the schema of the APLNext Supervisor XML-format configuration document for an explanation of the elements of the APLNext Supervisor configuration.

Syntax and Result:

Controlling Application	**OpenConfigFromText method syntax**	**Result**
APL+Win	‘spvr’ ⎕wi ‘XOpenConfigFromText’ xmlTextVector	0/Fail 1/OK
C#	bool b = spvr. OpenConfigFromText(xmlString);	False/true

Pause

This method signals all uncompleted threads to pause, i.e. stop processing. Because all processing request threads are asynchronously operating with respect to the ‘controlling application’ which uses the Pause method, the .Net Framework message to a particular thread issued by the APLNext Supervisor Pause method may be received by that thread after that thread has completed processing.

Arguments:

(none)

Syntax and Result:

Controlling Application	**Pause method syntax**	**Result**
APL+Win	‘spvr’ ⎕wi ‘XPause’	0/Fail 1/OK
C#	bool b = spvr.Pause();	False/true

SetDebug
Set the current value of the APLNext Supervisor log file entry verbose setting. Do not confuse this method with the 'Debug' element in the XML-format APLNext Supervisor configuration.

Arguments:
If called from APL+Win (0 indicates not verbose and 0 indicates verbose). If called from C# (false indicates not verbose and true indicates verbose).

Syntax and Result:

Controlling Application	**GetLogFileName method syntax**	**Argument**	**Result**
APL+Win	'spvr' ⎕wi 'XSetDebug'	0/No or 1/Yes	Prior value
C#	bool verboseLog = spvr.GetDebug();	True or False	Prior value

Resume

This method signals all uncompleted threads to resume processing.

Arguments:
(none)

Syntax and Result:

Controlling Application	Resume method syntax	Result
APL+Win	‘spvr’ ⎕wi ‘XResume’	0/Fail 1/OK
C#	bool b = spvr.Resume();	False/true

SetLogFileName

This method sets the name of the log file for this instance of the APLNext Supervisor.

Arguments:

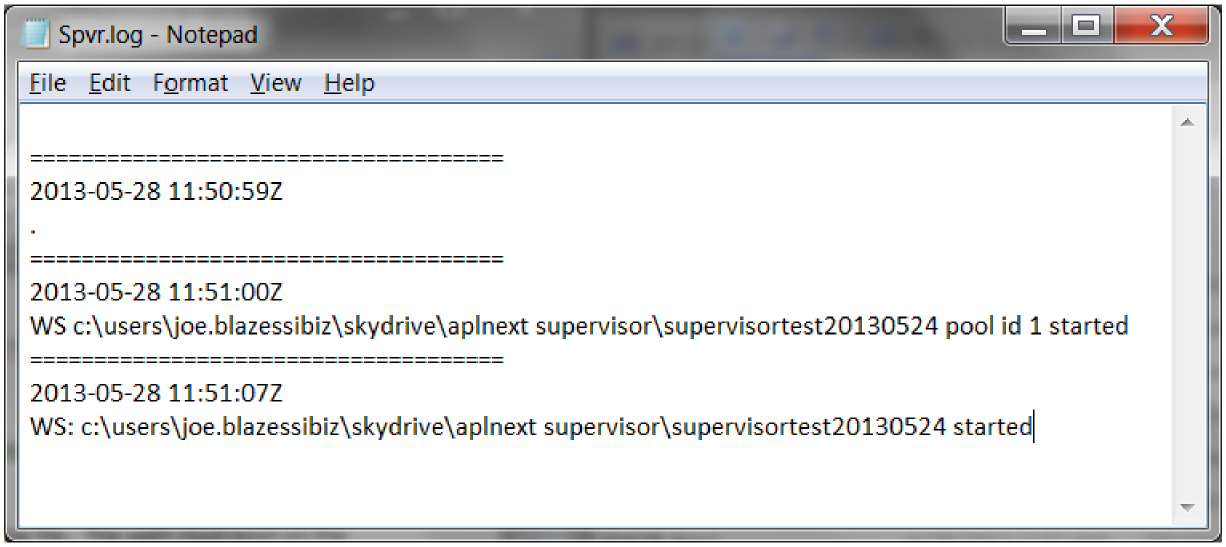
filename: The full path and filename of the file to be used as the log file. The path must exist on the target drive, but the log filename need not exist.

Syntax and Result:

Controlling Application	SetLogFileName method syntax	Result
APL+Win	‘spvr’ □wi ‘SetLogFileName’ filename	0/Fail 1/OK
C#	bool b = spvr. SetLogFileName(filename);	False/true

Typical (non-error condition) entries in the APLNext Supervisor log file:

- The time when the log file was created
- The time that the minpool number of APL+Win ActiveX engines was started
- The time that the APLNext Supervisor Start method was used by the ‘controlling application’



```
=====================================
2013-05-28 11:50:59Z
.
=====================================
2013-05-28 11:51:00Z
WS c:\users\joe.blazessibiz\skydrive\aplnext supervisor\supervisortest20130524 pool id 1 started
=====================================
2013-05-28 11:51:07Z
WS: c:\users\joe.blazessibiz\skydrive\aplnext supervisor\supervisortest20130524 started
```

Start
This method starts this instance of the APLNext Supervisor. Prior to using the 'Start' method, the 'controlling application' must create an instance of the APLNext Supervisor and configure that instance using the APLNext Supervisor method 'OpenConfigFromText' or 'OpenConfigFromFile'.

Arguments:
(none)

Syntax and Result:

Controlling Application	Start method syntax	Result
APL+Win	'spvr' ⎕wi 'Start'	0/Fail 1/OK
C#	Int32 r = spvr.Start();	False/true

When the 'Start' method is used an asynchronous mode of operation occurs among the 'controlling application' and the instances of APL+Win as an ActiveX engine which process the queued tasks using the 'Kernel Function'. This means that:

- The 'StartSync' method cannot be used by this instance of the APLNext Supervisor until all queued tasks submitted while the 'Start' method was used have been processed or the 'Stop' method is used.
- The execution of the 'controlling application' function is not synchronized with the completion of processing of all the queued tasks. Events fired by the APLNext Supervisor, which can be subscribed by the 'controlling application', signal the completion of a specific processing task (ProcessCompleteCallback event) and signal the completion of all queued processing tasks (ProcessAllDoneCallback event).
- When the 'BeginCall' method is used to submit a task to the queue, the APLNext Supervisor will immediately being the processing of that task using the 'Kernel Function' when an instance of the APL+Win ActiveX engine becomes available. Such instances become available as the processing of previously-submitted tasks are completed.
- If the 'controlling application' is written in APL+Win, a '□wgive 0' statement should be included after tasks are submitted to the queue via a 'BeginCall' statement. Without the '□wgive 0' statement, the 'controlling application' execution would continue, preventing the APLNext Supervisor from processing the task, since APL+Win is a single-threaded application.
- The execution of the 'controlling application' will not be suspended while the 'Kernel Function' is processing a queued task.
- After all the processing requests have been submitted to the queue by the 'controlling application', the execution of the 'controlling application' will continue, possibly to completion, while the processing of those tasks is being performed by the 'Kernel Function'. The execution of the 'controlling application' may be complete before the processing of all queued tasks have been completed.
- The completion of the processing of tasks submitted to the APLNext Supervisor queue using the 'BeginCall' method will not necessarily occur in the order the tasks were submitted to the queue.

- Generally a loop, or subscribing to the 'ProcessAllDoneCallback' event, is required in the 'controlling application' to know when to close and delete the APLNext Supervisor instance. If a loop is used, the 'QueueSize' property value can be checked to determine when it is zero.
    - Because the execution of the 'controlling application' function is not suspended while processing of the queued tasks occurs, the APLNext Supervisor events will fire. The event handler function can perform additional processing or accumulation of results using the explicit result of the 'Kernel Function' and the □warg, □wres and □wevent system variables.
    - If the 'ProcessAllDoneCallback' event handler is used to close and delete the APLNext Supervisor instance, these actions should be deferred until that event handler execution has been completed.

StartSync
This method starts this instance of the APLNext Supervisor. Prior to using the 'StartSync' method, the 'controlling application' must create an instance of the APLNext Supervisor and configure that instance using the APLNext Supervisor method 'OpenConfigFromText' or 'OpenConfigFromFile'.

Arguments:
(none)

Syntax and Result:

Controlling Application	Start method syntax	Result
APL+Win	'spvr' ⎕wi 'StartSync'	0/Fail 1/OK
C#	Int32 r = spvr.StartSync();	False/true

When the 'StartSync' method is used a synchronous mode of operation occurs between the 'controlling application' and the instances of APL+Win as an ActiveX engine which process the queued tasks using the 'Kernel Function'. This means that:

- The 'Start' method cannot be used by this instance of the APLNext Supervisor until the 'ExecuteAPL' method has been used or the 'Stop' method is used.
- The execution of the 'controlling application' function is synchronized with the completion of processing of all the queued tasks.
- When the 'BeginCall' method is used to submit a task to the queue, the APLNext Supervisor will queue the task, but will not begin processing of any queued tasks until the APLNext Supervisor 'ExecuteAPL' method is used.
- If the 'controlling application' is written in APL+Win, a '⎕wgive 0' statement is not necessary after tasks are submitted to the queue via a 'BeginCall' statement.
- The execution of the 'controlling application' will be suspended after the 'ExecuteAPL' method is used, i.e. while the 'Kernel Function' is processing any queued task and recommenced after the processing of all queued tasks is completed.
- Since the 'controlling application' execution stops while queued tasks are being processed by the 'Kernel Function', it is generally possible to set the value of the APLNext Supervisor 'maxpool' configuration parameter to #processors – 1.
- The completion of the processing of tasks submitted to the APLNext Supervisor queue using the 'BeginCall' method will not necessarily occur in the order the tasks were submitted to the queue.
- After all queued tasks have been processed by the 'Kernel Function', the execution of the 'controlling application' will recommence so that the 'controlling application' can then use the 'Close' and 'Delete' methods on the APLNext Supervisor instance.
- Because the execution of the 'controlling application' function is suspended while processing of the queued tasks occurs, the APLNext Supervisor events do not fire.
    - The 'StartSync' mode of operation of the APLNext Supervisor has the potential to improve performance and simplify the architecture of the 'controlling application', however it requires that the 'Kernel Function' perform the 'work' that would have been performed by the 'ProcessCompleteCallback' event handler.

- Any processing or results accumulation work that would have been done by the event handler functions when the ‘Start’ method is used must be done by the ‘Kernel Function’ when the ‘StartSync’ method is used.
- An explicit result of the ‘Kernel Function’ cannot be captured.
- Work done by the ‘Kernel Function’ must be recorded in a location on the target workstation accessible to the ‘controlling application’, for example in a shared APL+Win component file or a database such as Microsoft SQL Server.

Stop

The Stop method signals all processing request threads to abort processing if they are not already complete. Each such thread is aborted in order that they were requested so that some threads may complete processing even after the Stop method is used, because it takes time for each thread to be addressed as well as time for the stop message to be sent to and received by a thread.

The 'Stop' method also ends any instances of the APL+Win ActiveX engine which had been created by the APLNext Supervisor.

Arguments:
(none)

Syntax and Result:

Controlling Application	Stop method syntax	Result
APL+Win	'spvr' ⎕wi 'Stop'	0/Fail 1/OK
C#	Int32 r = spvr.Stop();	False/true

Using the 'Stop' method will reset the execution mode of the APLNext Supervisor instance. For example, using the 'Start' method followed by the 'Stop' method means that either the 'Start' or the 'StartSync' method may then be used.

Internal Methods
Certain methods are inherent for a .Net type such as the APLNext.Supervisor and these methods are generally not used by an APL+Win ‘Controlling Application.
XEquals
XGetHasCode
XGetType

In order to expose .Net events to the controlling application, the APLNext Supervisor includes the following internal methods. These methods are not directly used by the controlling application, but are necessary features of the APLNext Supervisor to accommodate the differences in event handling between .Net and APL+Win.
Xadd_ProcessAllDoneCallback
Xadd_ProcessCompleteCallback
Xadd_ProcessProgressCallback
Xremove_ProcessAllDoneCallback
Xremove_ProcessCompleteCallback
Xremove_ProcessProgressCallback

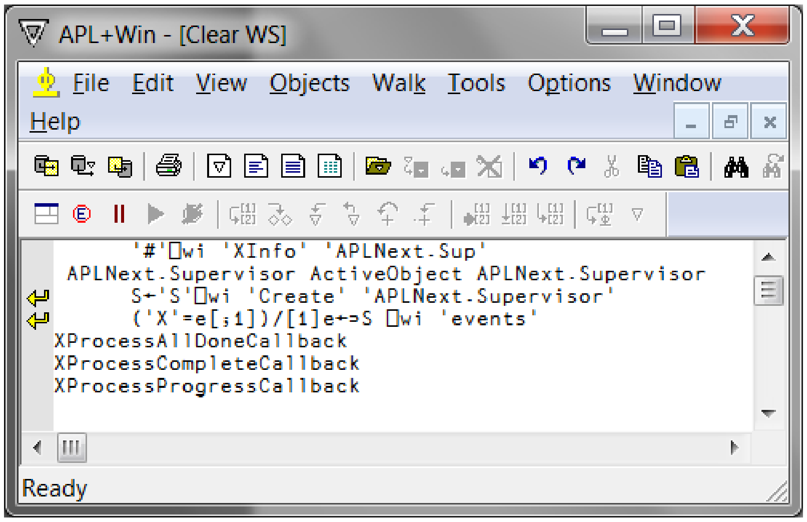
Supervisor Events

The 'controlling application' can subscribe to these events to receive information about the current state of a specified instance of the APLNext Supervisor. The application-specific kernel function is executed via asynchronous processing requests submitted to the APLNext Supervisor queue by the BeginCall method. Therefore processing requests may not be completed in the order in which they were submitted. The controlling application can subscribe to APLNext Supervisor events which will report when a processing request is completed and when all processing requests have been completed.

APLNext Supervisor events fire only if the 'Start' method' (and not the 'StartSync' method) is used by the 'controlling application'. When the 'StartSync' method is used, the APLNext Supervisor events do not fire since the 'Controlling App' could not receive notification of their firing because the execution of the 'Controlling App' is suspended while queued tasks are being processed.

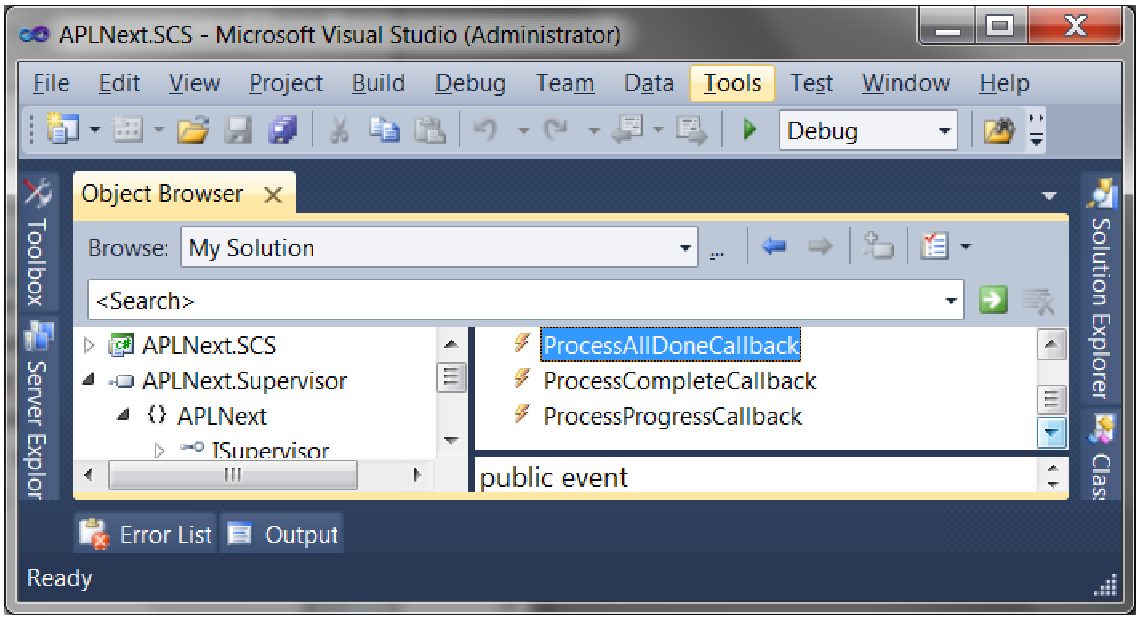
When debugging an APLNext Supervisor event handler function consider the suggestions in the debugging section of this document.

In APL+Win:



```
     '#'⎕wi 'XInfo' 'APLNext.Sup'
 APLNext.Supervisor ActiveObject APLNext.Supervisor
     S←'S'⎕wi 'Create' 'APLNext.Supervisor'
     ('X'=e[;1])/[1]e←⊃S ⎕wi 'events'
XProcessAllDoneCallback
XProcessCompleteCallback
XProcessProgressCallback
```

In Visual Studio (C#):


APLNext.SCS - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Object Browser
Browse: My Solution
<Search>
APLNext.SCS
APLNext.Supervisor
APLNext
ISupervisor
ProcessAllDoneCallback
ProcessCompleteCallback
ProcessProgressCallback
public event
Toolbox
Server Explorer
Solution Explorer
Error List
Output
Ready

ProcessAllDoneCallback
This event occurs when there are no more pending processing requests in the Supervisor queue and there are no more processing requests being executed by a kernel function.

Generally the Controlling Application will subscribe to this event after submitting all processing requests using the Supervisor's BeginCall method.

To subscribe to this event the 'ControllingApplication' can specify the name of the application-specific 'ProcessAllDoneCallback' event handler function. This event handler function is in the domain of the 'controlling application' and is executed when this event occurs. The name of the event handler function is programmer-controlled, although if C# is the 'controlling application' a default event handler name will be created in Visual Studio.

Event Subscription Syntax:

Controlling Application	**ProcessAllDoneCallback event subscription syntax**
APL+Win	'spvr' ⎕wi 'onXProcessAllDoneCallBack' 'ProcessAllDoneCallBackEH'
C#	sprv. ProcessAllDoneCallBack += new APLNext. ProcessAllDoneCallBack (ProcessAllDoneCallbackEH);

Event Data Available to Event Handler:
(none)

Event Handler Information:

Controlling Application	**ProcessAllDoneCallback event handler data**
APL+Win	Implicitly available as ⎕warg, which for this event will be empty
C#	Explicitly available in Event handler function syntax, which for this event is niladic: void ProcessAllDoneCallbackEH(){…}

Typically the 'ProcessAllDoneCallback' event will prepare the application system output based on a consolidation of the results accumulated from the 'ProcessCompleteCallback' events which fired for each 'kernel' function processing thread.

If the 'ProcessAllDoneCallback' event handler is used to 'Close' and 'Delete' the APLNext Supervisor instance, the use of these methods should be deferred until the event handler execution has completed.

ProcessCompleteCallback

This event occurs when:

- A processing request is completed by an APL+Win kernel function, or
- The an APL+Win 'kernel' function processing time exceeds the 'timeout' element value in the Supervisor configuration and the 'debug' element value in that configuration is '0', or
- The Supervisor detected an error when using the APL+Win ActiveX Engine to execute the 'kernel' function

Generally the 'controlling application' will subscribe to the 'ProcessCompleteCallback' event prior to submitting any 'BeginCall' parallel processing requests to the APLNext Supervisor.

To subscribe to this event the 'ControllingApplication' can specify the name of the application-specific 'ProcessCompleteCallback' event handler function. This event handler function is in the domain of the 'controlling application' and is executed when this event occurs. The name of the event handler function is programmer-controlled, although if C# is the 'controlling application' a default event handler name will be created in Visual Studio.

Event Subscription Syntax:

Controlling Application	**ProcessCompleteCallback event subscription syntax**
APL+Win	'spvr' ⎕wi 'onXProcessCompleteCallBack' 'ProcessCompleteEH'
C#	sprv.ProcessCompleteCallback += new APLNext.ProcessCompleteHandler(ProcessCompleteEH);

Event Data Available to Event Handler:

Element	**Element Description**
processId	The application-specific 'processId' which was associated with this processing request when the 'controlling application' made the processing request using the 'BeginCall' method of the APLNext Supervisor.
errorCode	The errorCode returned by the APLNext Supervisor when this processing request is completed. The value of the errorCode is 0 for success (no errors) or a negative integer error code.
resultData	**errorCode** | **resultData** 0 | The 'kernel' function result for this completed processing request. negative | The error description text

Event Handler Information:

Controlling Application	**ProcessAllDoneCallback event handler data**
APL+Win	Implicitly available as ⎕warg
C#	Explicitly available in Event handler function syntax: void ProcessCompleteEH(int processId, int errorCode, object resultData){…}

If an APL+Win 'kernel' function processing time exceeds the 'timeout' element value in the Supervisor configuration and the 'debug' element value in that configuration text is '0', the 'ProcessCompleteCallback' event will fire. In this case the 'ProcessCompleteCallback' event handler in the domain of the 'controlling application' will receive the Supervisor-provided errorCode as '-1' and the resultData as 'Thread was being aborted'.

ProcessProgressCallback
This event occurs when the application-specific 'kernel' function uses the 'Notify' method of the APL+Win ActiveX engine, e.g. '#' ⎕wi 'Notify' code data, where 'code' and 'data' are APL+Win values available within the scope of the 'kernel' function.

Generally the Controlling Application will subscribe to the' ProcessProgressCallback' event prior to submitting any processing requests using the Supervisor's BeginCall method.

To subscribe to this event the 'ControllingApplication' must specify the name of the application-specific 'ProcessProgressCallback' event handler function. This event handler function is in the domain of the 'ControllingApplication' and is executed when this event occurs. The name of the event handler function is programmer-controlled.

The 'ProcessProgressCallback' event is fired asynchronously by the APLNext Supervisor. When this event fires, the processing of the 'kernel' function which used the 'Notify' method to trigger the 'ProcessProgressCallback' event will continue processing and is not suspended while the 'ProcessProgressCallback' event handler in the domain of the 'controlling application' is executing.

Event Subscription Syntax:

Controlling Application	**ProcessAllDoneCallback event subscription syntax**
APL+Win	'spvr' ⎕wi 'onXProcessProgressCallback' 'ProcessProgressbackEH'
C#	sprv. ProcessProgressCallback += S_ProcessAllDoneCallback;

Event Data Available to Controlling Application 'ProcessProgressCallback' Event Handler:

Element #1	processId which was specified when the processing request was made by the application-specific 'ControllingApplication'
Element #2	An object containing the 'code' value provided by the 'kernel' function when it used the 'Notify' method
Element #3	An object containing the 'data' value provided by the 'kernel' function when it used the 'Notify' method

Event Handler Information:

Controlling Application	**ProcessAllDoneCallback event handler data**
APL+Win	Implicitly available in ⎕warg: processId code data
C#	Explicitly available in the arguments to the event handler function, e.g. `void S_ProcessProgressCallback(int id, object code, object data)`

The ‘ProcessProgressCallback’ event will be fired when the ‘kernel’ function uses the ‘Notify’ method to inform the ‘ControllingApplication’ of the status of the processing request which the ‘kernel’ function is currently satisfying.

If the ‘kernel’ function requires significant time to satisfy a processing request, it is reasonable to use the ‘ProcessProgressCallback’ method to provide processing progress updates to the ‘ControllingApplication'.

Using the ‘Notify’ method to fire many ‘ProcessProgressCallback’ events over a short time interval, for example when the ‘kernel’ function processing is very fast, may result in an increase in processing time or concurrency problems such as a race condition or deadlock.

Although ‘ProcessProgressCallback’ event is associated with the APL+Win ‘Notify’ method, the ‘result’ parameter of the APL+Win ‘Notify’ method is not applicable to the ‘ProcessProgressCallback’ event and cannot be set by the ‘ProcessProgressCallback’ event handler.

The ‘kernel’ function using the ‘Notify’ event to trigger the APLNext Supervisor ‘ProcessProgressCallback’ event will not ‘wait’ for a response from the event handler function or for that event handler function to complete execution.

Example:

- Prior to submitting processing requests to the APLNext Supervisor, the APL+Win-based ‘controlling application’:
    - Defines the ‘ProcessProgressCallBackEH’ event handler function in the domain of the ‘controlling application’
    - Subscribes to the ‘ProcessProgressCallback’ event, e.g. ‘spvr’ ⎕wi ‘onProcessProgressCallback’ ‘ProcessProgressCallBackEH’

- To provide processing feedback to the domain of the ‘controlling application’, the APL+Win ‘kernel’ function uses the APL+Win ‘Notify’ method to trigger the ‘ProcessProgressCallback’ event of the Supervisor, e.g.
  ‘#’ ⎕wi ‘Notify’ ‘Code: 99’ ‘Data: Calculation is 20% complete’

- The Supervisor observes the use of the ‘Notify’ method by the ‘kernel’ function and the Supervisor fires the ‘ProcessProgressCallback’ event.

- The ‘ProcessProgressCallback’ event handler in the domain of the ‘controlling application’ is executed.

- During the execution of the ‘ProcessProgressCallback’ event handler:

- The execution of the ‘kernel’ function which used the ‘Notify’ method continues and does not ‘wait’ for the event handler to complete execution.
- The ‘ProcessProgressCallback’ event handler has access to the event data. If the ‘ControllingApplication’ is APL+Win-based:
  - 1⊃⎕warg] contains the processId (integer argument provided by the ‘controlling application’ to the ‘BeginCall’ method)
  - 1⊃2⊃⎕warg contains the value of the ‘code’ provided by the ‘kernel’ function when it used the ‘Notify’ method, e.g. ‘Code: 99’
  - 2⊃2⊃⎕warg contains the value of the ‘data’ provided by the ‘kernel’ function when it used the ‘Notify’ method, e.g. ‘Data: Calculation is 20% complete’
- Since the ‘ProcessProgressCallback’ event handler is in the domain of the ‘Controlling App’ the event handler may display or accumulate information in that domain, e.g. in the APL+Win session in which the ‘controlling application’ and the ‘ProcessProgressCallback’ event handler are running.

Xml Schema of the APLNext Supervisor Configuration

Each instance of the APLNext Supervisor created by the 'controlling application' is configured by an APLNext Supervisor configuration xml document. This xml document uses the following xml schema:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="config">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="workspaces">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="workspace">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="minpool" type="xs:int" />
                    <xs:element name="maxpool" type="xs:int" />
                    <xs:element name="timeout" type="xs:int" />
                    <xs:element name="debug" type="xs:byte" />
                    <xs:element name="visible" type="xs:byte" />
                    <xs:element name="wslocation" type="xs:string" />
                    <xs:element name="wssize" type="xs:int" />
                    <xs:element name="evlevel" type="xs:int" />
                    <xs:element name="busyid" type="xs:string" />
                    <xs:element name="user_state" type="xs:string" />
                  </xs:sequence>
                  <xs:attribute name="id" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

config Element

The config element is the root element of the APLNext Supervisor configuration xml.

workspaces Element

The workspaces element must contain at least one workspace element.

workspace Element
There must be at least one workspace element in the APLNext Supervisor configuration xml. A workspace element identifies the APL+Win workspace which will contain the APL+Win kernel function(s). More than one workspace element may exist in the APLNext Supervisor configuration xml.

id Attribute of the workspace Element
The 'workspace' XML element must have an id attribute which is the name of the workspace that will be used in the APLNext Supervisor BeginCall method.

minpool Element
The 'minpool' element valueindicates the number of instances of APL+Win that will be instantiated as ActiveX servers when the Start method of the APLNext Supervisor is used by the 'controlling application'.

maxpool Element
The 'maxpool' element value indicates the maximum number of instances of APL+Win that will be instantiated as ActiveX servers while the 'controlling application' uses the APLNext Supervisor 'BeginCall' method to request asynchronous execution of 'kernel' functions. The actual number of instances of APL+Win as ActiveX servers that will exist while an instance of the APLNext Supervisor is being used will vary with the number of outstanding requests for execution of 'kernel' functions. If the number of requests exceeds the number of available instances of APL+Win as an ActiveX server, the requests will be queued for processing.

timeout Element
The 'timeout' element value indicates the number of milliseconds which a specific asynchronous processing request being performed by an APL+Win 'kernel' function can consume before the APLNext Supervisor will fire the ProcessCompleteCallback event.

debug Element
If the 'debug' element value is '0' the 'timeout' element value will be considered by the APLNext Supervisor. If the 'debug' element value is '1', the 'timeout' element value will not be considered by the APLNext Supervisor. A 'debug' element value of '1' can result in an infinite processing time for a request, should an APL+Win 'kernel' function incorporate an 'infinite loop' or a stop. Setting this element value to '1' is useful when testing and debugging an application system kernel function.

Do not confuse the 'debug' element with the APLNext Supervisor 'GetDebug' or 'SetDebug' methods.

visible Element
The 'visible' element value indicates if the instances of APL+Win as an ActiveX server created by the APLNext Supervisor will be visible (value = '1') or invisible (value = '0'). This field applies only if the developer version of APL+Win is registered on the target workstation.

wslocation Element
The ‘wslocation’ element value is the physical path to the APL+Win workspace containing the ‘kernel’ function(s) to be executed by the APLNext Supervisor.

wssize Element
The ‘wssize’ element is a reserved field.

evlevel Element
The ‘evlevel’ element is a reserved field. The value of this xml element should be ‘2’. Refer to the documentation of APL+Win for details of the ⎕evlevel system function options.

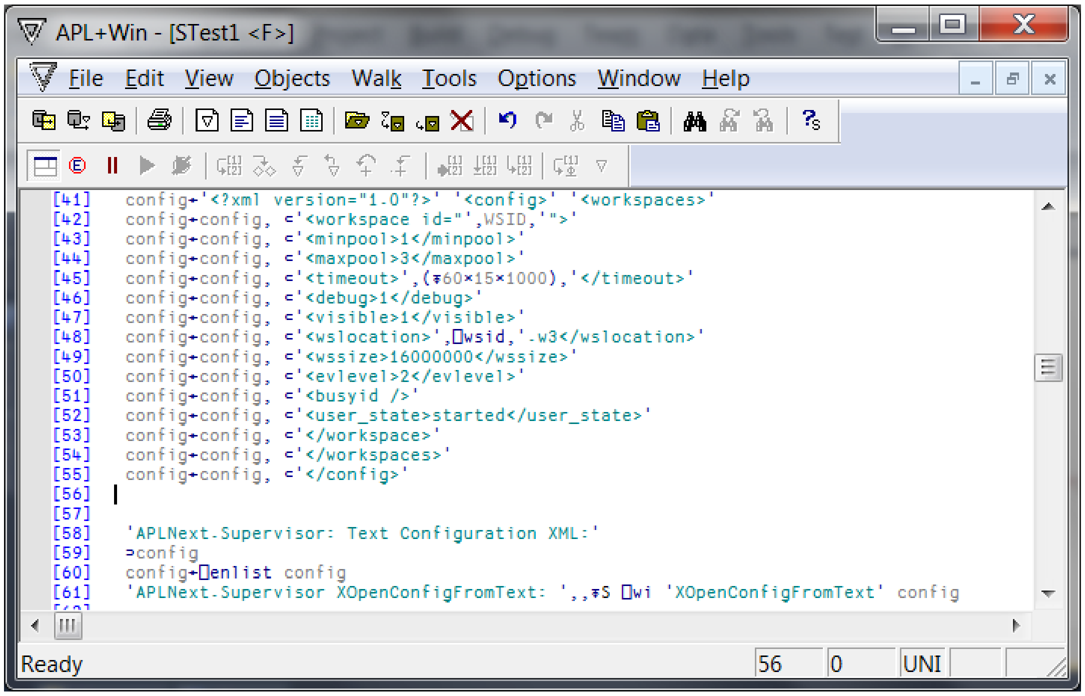
busyid Element
The ‘busyid’ element is a reserved field. The value of this xml element should be empty.

user_state Element
The ‘user_state’ element indicates the ‘started’ or ‘stopped’ stated of the instance of the APLNext Supervisor. Generally the value of this xml element should be ‘started’.

APLNext Supervisor for APL+Win

Sample APLNext Supervisor Configuration xml

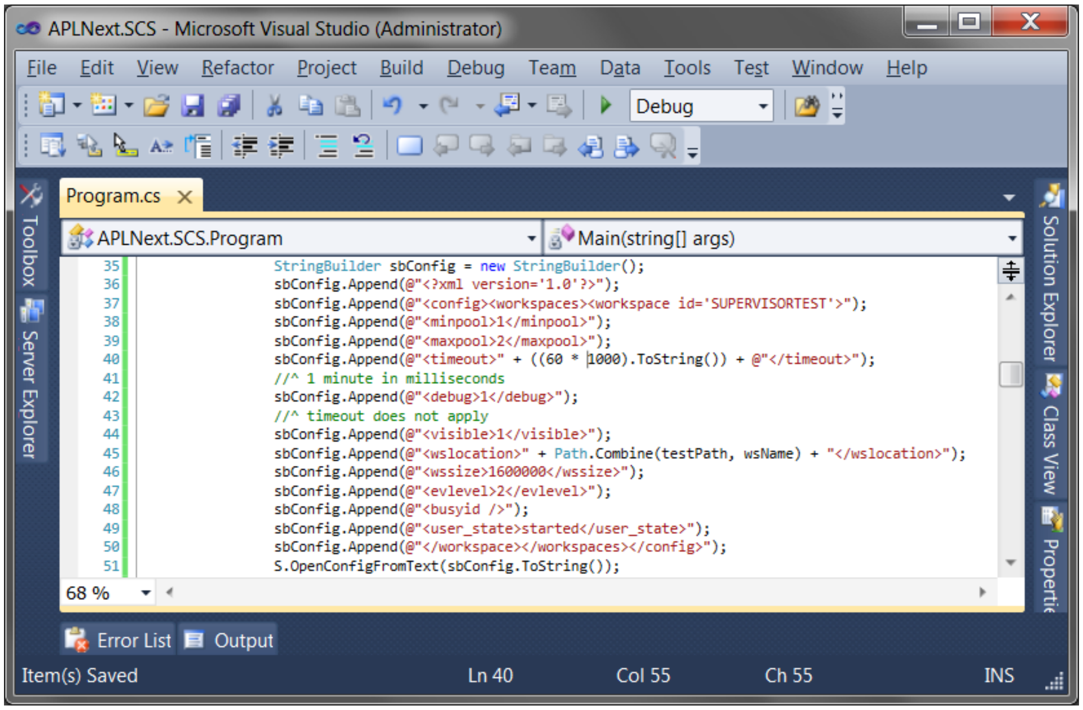
In APL+Win:



```
config←'<?xml version="1.0"?>' '<config>' '<workspaces>'
config←config, ⊂'<workspace id="',WSID,'">'
config←config, ⊂'<minpool>1</minpool>'
config←config, ⊂'<maxpool>3</maxpool>'
config←config, ⊂'<timeout>',(⍕60×15×1000),'</timeout>'
config←config, ⊂'<debug>1</debug>'
config←config, ⊂'<visible>1</visible>'
config←config, ⊂'<wslocation>',⎕wsid,'.w3</wslocation>'
config←config, ⊂'<wssize>16000000</wssize>'
config←config, ⊂'<evlevel>2</evlevel>'
config←config, ⊂'<busyid />'
config←config, ⊂'<user_state>started</user_state>'
config←config, ⊂'</workspace>'
config←config, ⊂'</workspaces>'
config←config, ⊂'</config>'


'APLNext.Supervisor: Text Configuration XML:'
⊃config
config←⎕enlist config
'APLNext.Supervisor XOpenConfigFromText: ',,⍕S ⎕wi 'XOpenConfigFromText' config
```

In C#:

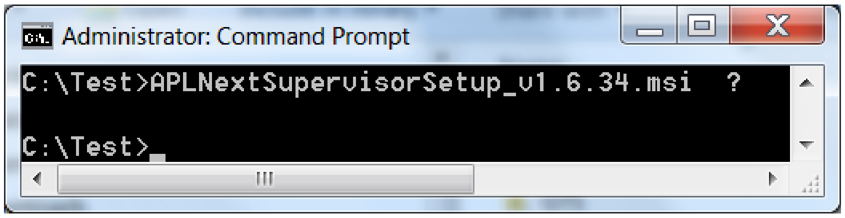


```
StringBuilder sbConfig = new StringBuilder();
sbConfig.Append(@"<?xml version='1.0'?>");
sbConfig.Append(@"<config><workspaces><workspace id='SUPERVISORTEST'>");
sbConfig.Append(@"<minpool>1</minpool>");
sbConfig.Append(@"<maxpool>2</maxpool>");
sbConfig.Append(@"<timeout>" + ((60 * 1000).ToString()) + @"</timeout>");
//^ 1 minute in milliseconds
sbConfig.Append(@"<debug>1</debug>");
//^ timeout does not apply
sbConfig.Append(@"<visible>1</visible>");
sbConfig.Append(@"<wslocation>" + Path.Combine(testPath, wsName) + "</wslocation>");
sbConfig.Append(@"<wssize>1600000</wssize>");
sbConfig.Append(@"<evlevel>2</evlevel>");
sbConfig.Append(@"<busyid />");
sbConfig.Append(@"<user_state>started</user_state>");
sbConfig.Append(@"</workspace></workspaces></config>");
S.OpenConfigFromText(sbConfig.ToString());
```

Distributing the APLNext Supervisor with your application

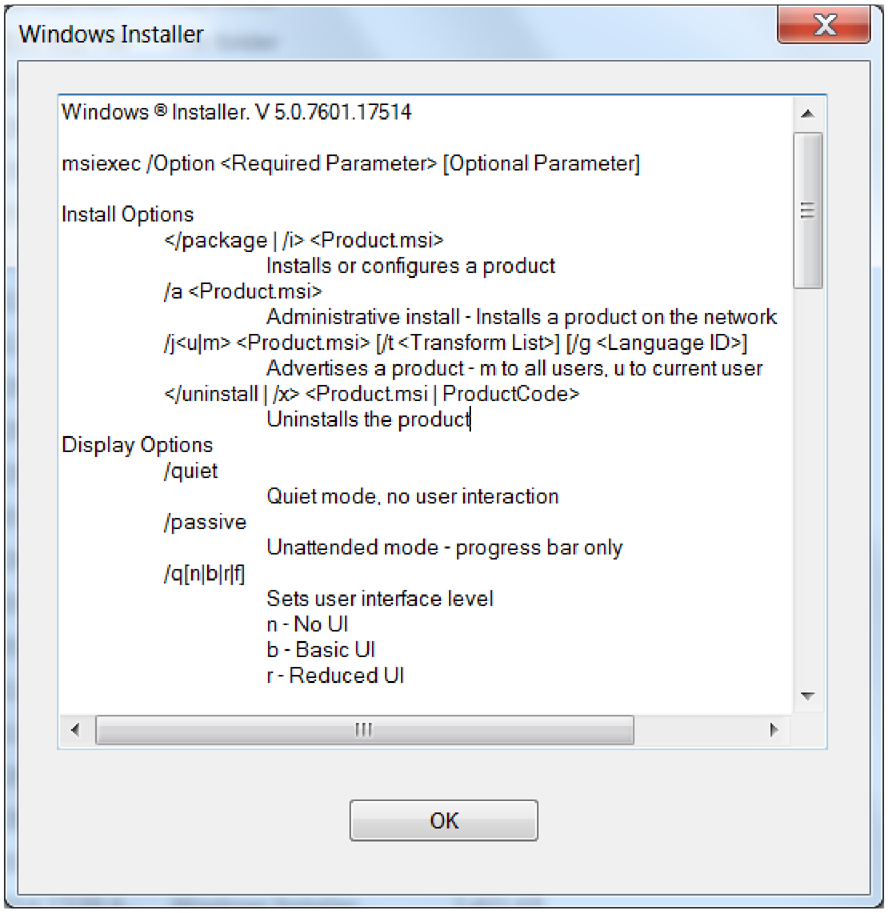
The APLNext Supervisor Assembly Deployment

The APLNext Supervisor is a .Net assembly (.dll file) with several dependencies (additional .dll files). The APLNext Supervisor also incorporates an ActiveX interface so that it can be used when APL+Win is the 'controlling application'. The APLNext Supervisor installation tool (.msi) should generally be used to deploy the APLNext Supervisor so that the ActiveX interface is properly registered on the target workstation. This .msi includes command line options for installation logging, silent installation, etc. which are documented if the .msi is executed from the Windows Command Prompt with the '?' switch.



```
C:\Test>APLNextSupervisorSetup_v1.6.34.msi  ?

C:\Test>_
```



```
Windows ® Installer. V 5.0.7601.17514

msiexec /Option <Required Parameter> [Optional Parameter]

Install Options
        </package | /i> <Product.msi>
                Installs or configures a product
        /a <Product.msi>
                Administrative install - Installs a product on the network
        /j<u|m> <Product.msi> [/t <Transform List>] [/g <Language ID>]
                Advertises a product - m to all users, u to current user
        </uninstall | /x> <Product.msi | ProductCode>
                Uninstalls the product
Display Options
        /quiet
                Quiet mode, no user interaction
        /passive
                Unattended mode - progress bar only
        /q[n|b|r|f]
                Sets user interface level
                n - No UI
                b - Basic UI
                r - Reduced UI
```

APLNext Supervisor for APL+Win

Microsoft .Net Framework 4.5
The APLNext Supervisor is implemented using the Microsoft .Net Framework 4.0. This framework must be must be installed on the target machine. The Microsoft .Net 4.5 Framework (full version) is available at no charge.

Controlling Application Deployment
If the 'controlling application' is APL+Win, the APLNext Supervisor installer (.msi) should be distributed to the end user to install the APLNext Supervisor software on the target machine. This installer will check that the proper Microsoft .Net Framework is already installed on the target machine and register the APLNext Supervisor as an ActiveX component on the target machine. The APLNext Supervisor installer is included with a current subscription to the APL+Win software.

If the 'controlling application' is a .Net language, such as C#, then adding a reference to the APLNext.Supervisor in the C# project will include the APLNext.Supervisor in the solution when the project is compiled in Visual Studio. Separate installation and registration of the APLNext Supervisor as an ActiveX component on the target machine is not necessary in this case.

Runtime APL+Win
When deploying an application system which incorporates the APLNext Supervisor technology, the appropriate version of the APL+Win runtime executable should be used.

The APLNext Supervisor will execute the application system 'kernel' functions using instances of APL+Win as an ActiveX server, so no matter what programming environment is used for the 'controlling application', when the application system is deployed to an end user, a properly licensed APL+Win runtime version must be registered as an ActiveX component on the target machine.

APL+Win .adf File
The APLNext Supervisor tracks the independent thread instances of APL+Win ActiveX server by their Windows operating system process id. For the APLNext Supervisor to obtain the Windows operating system 'process id' of an APL+Win ActiveX server instance, the APL+Win '.adf' file must be installed with APL+Win. For the run-time version of APL+Win, the application system developer names this file in association with the APL+Win runtime executable, e.g. myRuntimeAPLWin.exe (from aplwr.exe) and myRunTimeAPLWin.adf from (aplwr.adf). Failure to install the applicable '.adf' file will result in an error when the controlling application attempts to execute the APLNext Supervisor 'Start' method.

APL+Win 'kernel' function workspaces
The APL+Win runtime version can load the same 'kernel' function workspaces that were designed and implemented with the APL+Win developer version.

Error Handling using the APLNext Supervisor

PauseThreadsOnError

Refer to the documentation of the APLNext Supervisor 'PauseThreadsOnError' property a description of how the Supervisor can be configured to pause all remaining threads upon an error condition that is detected by the APLNext Supervisor.

Error Conditions Outside of the Domain of the APLNext Supervisor

The APLNext Supervisor cannot detect error conditions within:

- The 'controlling application' which starts and stops an APLNext Supervisor instance and uses the methods properties and events of that APLNext Supervisor instance. The 'controlling application' should use appropriate exception handling such as a try/catch control structure. The 'controlling application' should also check the result value of each APLNext Supervisor method used by the 'controlling application' for an error condition.
- The event handler functions of the 'controlling application' which subscribed to the APLNext Supervisor ProcessCompleteCallback, ProcessAllDoneCallback or ProcessProgressCallback events. The event handler functions should use appropriate exception handling such as a try/catch control structure. The event handler functions should also check the event data, if any, made available to them by the APLNext Supervisor.
- An APL+Win 'kernel' function scheduled to be executed by the APLNext Supervisor BeginCall method. An 'unhandled' error condition within a 'kernel' function can result in a processing suspension in the associated instance of the APL+Win ActiveX server. If the APLNext Supervisor configuration timeout element applies, the APLNext Supervisor will abort the thread associated with this instance of the APL+Win ActiveX server.

  The 'kernel' function should use appropriate exception handling such as a try/catch control structure. An error condition occurring within a 'kernel' function can be handled:
  - Directly within the 'kernel' function assuming an appropriate recovery or error recording processes are implemented in that function.
  - In the 'ProcessCompleteCallback' event handler function using information provided in the result of the 'kernel' function.

APLNext Supervisor Error Messages

Errors which may occur when the 'controlling application' uses APLNext Supervisor properties, methods or events are reported by the Supervisor to that 'controlling application'. When the controlling application is APL+Win-based such errors are reported using the APL+Win ⎕dm system variable. When the controlling application is C#-based such errors are reported using the Microsoft .Net exception object.

The APLNext Supervisor creates instances of the APL+Win ActiveX Engine and uses the methods, properties and events of that engine. These directives operate in the context of the target workstation's user permissions and limits on memory and processors. An error that occurs because of these constraints on APLNext Supervisor directives to the APL+Win ActiveX Engine are reported to the controlling application with additional information provided by the .Net Framework Exception object. The log file specified by Supervisor 'SetLogFileName' method may also contain these error messages.

Debugging an Application System which uses the APLNext Supervisor

APLNext Supervisor Implementation Background

The APLNext Supervisor can be implemented by an application system programmer with all APL+Win code or a mixture of C# (or other .Net programming language) and APL+Win code. The 'controlling application' and APLNext Supervisor event handle methods can be written in APL+Win or C# (or any other .Net language). The 'Kernel' portion of the application system is always written in APL+Win.

To avoid unanticipated behavior when debugging an application system based on the APLNext Supervisor:

- Set the 'debug' element of that configuration to '1' so that while debugging the 'timeout' will not apply.
- Set the 'PauseThreadsOnError' property value of the APLNext Supervisor instance to true/1 to pause uncompleted threads if an error condition is detected by the APLNext Supervisor.

The APLNext Supervisor 'Pause' and 'Resume' methods are also helpful when debugging an application system based on the APLNext Supervisor.

It may also be helpful during application system debugging of the 'Kernel' workspace to set the value of the 'minpool' and 'maxpool' elements of the APLNext Supervisor configuration to '1'.

It may also be helpful to have the 'controlling application' make only one use of the 'BeginCall' method of the APLNext Supervisor while debugging.

For a .Net-based 'controlling application', Microsoft Visual Studio provides tools to incorporate program stops, watches and halt/step debugging.

APL+Win Session Code Walker Debugging Tool

The 'kernel' functions are always written in APL+Win and so are the 'controlling application' and event handler functions when APL+Win is the 'controlling application'.

The APL+Win developer version 'code walker' for inserting program stops and performing 'halt/step' debugging can be used with the APL+Win 'kernel' functions, the 'controlling application' function and the event handler functions when the APLNext Supervisor is in 'async' mode.

Debugging the 'Kernel' Function

- Since a 'kernel' function is written in APL+Win and the APLNext Supervisor uses an instance of the APL+Win ActiveX engine to execute a 'kernel' function, register the APL+Win developer version ActiveX engine on the programmer's machine while debugging the application system.
- Since the 'kernel' function is run via instances of the APL+Win ActiveX engine, set the 'visible' element value of the APLNext Supervisor configuration to '1' so that the APL+Win developer session will be visible when the 'kernel' function is running.

Supervisor Log File

The APLNext Supervisor 'SetLogFileName' method can be used to direct the Supervisor event log entries to a specific file.