Using the Work Queue Manager

Version 2020.3
2021-02-04

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using the Work Queue Manager
InterSystems IRIS Data Platform Version 2020.3 2021-02-04

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

Using the Work Queue Manager 1
1 Background 1
1.1 ObjectScript CPU Utilization 13
1.2 Features of the Work Queue Manager 13
2 About Units of Work 3
3 About Worker Jobs 3
4 Basic Workflow 4
4.1 Basic Methods 5
4.2 Properties of Work Queues 7
5 Managing Categories 7
6 Using Callbacks 8
6.1 Including Callbacks for Work Items 9
6.2 Including Callbacks to Determine Completion 9
7 Controlling Output to the Current Device 9
8 Pausing and Resuming a Work Queue 10
9 Detaching and Attaching a Work Queue 10
10 Stopping a Work Queue and Removing Work Items 11
11 Specifying Setup and Teardown Processing 11
12 Retrieving Workload Metrics 12

Using the Work Queue Manager

The work queue manager is a feature of InterSystems IRIS that enables you to improve performance by distributing work to multiple concurrent processes programmatically. Prior to the introduction of the work queue manager, you may have used the JOB command to start multiple processes in your application and managed the processes (and any resulting failures) using custom code. The work queue manager provides an efficient and straightforward API that enables you to off-load process management.

InterSystems code uses the work queue manager internally in several places. You can use it for your own needs as described at a high level in the following sections.

For more information, see the %SYSTEM.WorkMgr class in the class reference.

1 Background

Recent innovations in the development of computer hardware have trended toward high-performance, multi-processor or multi-core architectures. At the same time, the speed of memory and network devices has only slowly improved. InterSystems developed the work queue manager in response to these trends and according to the following principals:

- Hardware resources, including CPUs and I/O, memory, and networking devices, are fixed.
- InterSystems IRIS must use hardware resources as efficiently as possible to maximize the speed of the business tasks it performs.
- To achieve maximum efficiency, the work queue manager must improve the under-utilization of CPUs that can occur when executing InterSystems ObjectScript code.
- The means to address under-utilization of CPUs includes queuing and prioritization.

While the InterSystems IRIS data platform as a whole is designed to utilize the hardware resources in your system as efficiently as possible, the work queue manager feature of the platform is specifically designed to tap into the additional CPU resources available in modern hardware configurations. The work queue manager serves two key purposes:

- Provide a framework that enables you to break up large programmatic tasks into smaller chunks to be executed in multiple, concurrent processes. By using more than one CPU at a time, the work queue manager significantly reduces the time it takes to process large workloads.
- Control the total CPU load on a system by managing the number of InterSystems jobs that are active at one time for system tasks. The number of active jobs is limited by category. For more information, see Managing Categories.

1.1 ObjectScript CPU Utilization

Generally, ObjectScript code runs in a single process and uses only one processor core. For transactional database applications that process relatively few instructions and global references between transactions, this approach works well. In fact, a key feature of the InterSystems IRIS data platform is massive scalability of transactional workloads. The platform optimizes the processing of very large numbers of relatively small *units of work* requested by a very large number of users at once.

Some newer types of workloads (for example, analytic workloads) differ from the workloads that InterSystems IRIS was initially built to optimize. For example, a newer workload may involve processing one SQL query that requires performing various operations over millions of rows. To speed up the processing of such a workload, InterSystems developed the work queue manager, which breaks up monolithic workloads into smaller chunks, processes the chunks in parallel, and relays

the results of each chunk back to a parent process, which can then relay the results back to you. In other words, the work queue manager is a mechanism similar to InterProcess Queues that enables developers who build their applications in InterSystems IRIS to break up large tasks into smaller tasks to be processed in parallel.

1.2 Features of the Work Queue Manager

The work queue manager includes several key features:

- Low latency and overhead
- Scalability
- Cooperation with the operating system
- Flexibility
- High levels of control and reporting

1.2.1 Low Latency and Overhead

The work queue manager is designed for low latency and low overhead. For example, consider a programmatic task that it takes your system 10 minutes to process sequentially. If your system has 10 cores, it would be more efficient to split up the task and process one tenth of the work on each core in parallel. In fact, if the overhead involved in splitting up the task, queueing each piece of the task, starting worker jobs, and collecting the notifications that each piece of the task was completed did not required any additional time, you could arrive at the result 10 times faster. The work queue manager is designed so that overhead tasks result in low latency.

1.2.2 Scalability

To maximize performance, the work queue manager is capable of processing a single task using all the CPU resources on your system. In practice, the work queue manager restricts the number of cores that a given type of task can use to ensure that all of the workloads on your system can be processed efficiently.

1.2.3 Cooperation with the Operating System

With traditional ObjectScript code for a massive transactional database application, your operating system may spend a significant number of resources switching between programmatic tasks, which is sometimes called context switching. With the work queue manager employing a queueing mechanism on each core, the need for context switching is greatly reduced. Only when the number of active jobs managed by the work queue manager exceeds the number of available cores does the operating system need to context switch. In this way, queueing work generally improves performance.

1.2.4 Flexibility

A unit of work is class method or subroutine that takes a set of arguments and meets the requirements described in About Units of Work. Any logic that can be represented within those constraints can be processed by the work queue manager, providing you with tremendous flexibility.

1.2.5 High Levels of Control and Reporting

The work queue manager provides you with a high-level of control over how the CPU resources on your system are used. For example, you can create categories of work and define the number of worker jobs assigned to the categories. Additionally, the work queue manager provides work load metrics so that you can monitor the load on your system in real time.

2 About Units of Work

The work queue manager functions by processing *units of work* (also called *work items*), which are ObjectScript class methods or subroutines that meet the following requirements:

- The class method or subroutine can be processed independently. For example, a unit of work cannot rely on output from a different unit of work. Independence is required since units of work may be processed in any order. However, you can use callbacks to execute work sequentially if needed. For more information, see Using Callbacks.
- The class method or subroutine is on the order of thousands of lines of ObjectScript code in size. This requirement ensures that the overhead of the framework is not a significant factor.

  Furthermore, it is preferable to use a large number (for example, 100) of smaller units of work rather than a small number of very large units of work (for example, 4). Distributing the work in such a way permits the system to scale up when more CPU cores are available.
- The code return a %Status value to indicate success or failure so that the **WaitForComplete()** method can return a %Status value to indicate overall success or failure. Alternatively, the unit of work can throw an exception that is trapped, converted to a %Status value, and returned in the master process.
- If the code changes the same global as a different unit of work, you must employ a locking strategy to ensure that one worker job cannot change the global while another worker is reading it.
- The code does not include exclusive news, kills, or unlocks since these interfere with the framework.
- If the code includes process-private globals for storing data, these process-private globals are not accessed from the master process or from any other chunk. This requirement is necessary since multiple jobs process each chunk.
- Any logic called as part of the class method or subroutine is correctly cleaned up such that no variables, locks, process-private globals, or other artifacts remain in the partition. This requirement is important since the same process will be used to subsequently process completely separate work items.

To use the work queue manager, you must divide some amount of programmatic work into units of work.

3 About Worker Jobs

Worker jobs are the processes that complete units of work for the work queue manager. You can view, manage, and monitor worker jobs like other processes by using the %SYSTEM.Process class. If you need to need to know whether a given process is a worker job, you can call **$system.WorkMgr.IsWorkerJob()** from within the process; that is, you can call the **IsWorkerJob()** method of the %SYSTEM.WorkMgr class.

The work queue manager directs worker jobs using the controller process, which is a dedicated process that performs the several operations:

- Starts up worker jobs
- Manages the number of worker jobs
- Detects and report on halted worker jobs
- Records workload metrics
- Detects inactive work queues
- Deletes work queues

A worker job can be in any of the following states:

- Waiting to attach to a work queue
- Waiting for units of work. A worker job can be in this state for only a short period of time before it is released.
- Active. A worker job is active only when it is making forward process while executing a unit of work.
- Blocked by a lock or event while processing a unit of work. A worker job that is blocked is not active. If a worker becomes blocked and there is additional work in the work queue, the work queue manager may activate a retired worker or start up a new worker. When a worker job is no longer blocked, the number of active workers may exceed the maximum number of active workers specified for the work queue. If this occurs, the controller process retires the next worker that completes a unit of work. Consequently, there may be short periods of time when the active number of worker jobs exceeds the maximum number of worker jobs specified for a given work queue.
- Retired and available to be activated rapidly

Unused worker jobs remain available for use by other work queue manager queues for a short period of time. The timeout period is subject to change and is deliberately not documented. After the timeout period expires, the worker is removed.

If a worker job is actively processing a work item for a queue that has been deleted or cleared, the system waits a very short period of time before issuing an `EXTERNAL INTERRUPT` error. If the worker job continues processing after the error, the system waits for the number of seconds specified in the DeleteTimeout property before forcibly terminating the worker and starting up a new worker to process the unit of work.

The superserver starts the worker jobs, which means that they run under the name of the operating system user used by the superserver process. This username may be different from the currently logged-in operating system user.

4 Basic Workflow

You can employ the work queue manager by performing the following steps:

1. Divide your ObjectScript code into *units of work*, which are class methods or subroutines that meet certain requirements. For more information, see About Units of Work.
2. Create a *work queue*, which is an instance of the %SYSTEM.WorkMgr class. To do so, call the **%New()** method of the %SYSTEM.WorkMgr class. The method returns a work queue.

   You can specify the number of parallel worker jobs to use, or you can use the default, which depends on your machine and operating system. Additionally, if you have created categories, you can specify the category that the jobs should be taken from.

   When you create a work queue, the work queue manager creates the following artifacts:

   - A global that contains information about the work queue such as what namespace the work queue runs in
   - A location and an event queue for the serialized units of work that the work queue must process
   - A location and an event queue for completion events that are created as the work queue finishes processing units of work

3. Add units of work (also called work items) to the work queue. To do so, you can call the **Queue()** or **QueueCallback()** method. As arguments, you pass the name of a class method (or subroutine) and any corresponding arguments.

   Processing begins immediately on items added to the queue.

   If there are more items in the queue than there are worker jobs available to the queue, then the jobs compete to empty the queue. For example, if there are 100 items and four jobs, each job removes an item from the head of the queue,

processes it, and then returns to the head of the queue to remove and process another item. This pattern continues until the queue is empty.

The work queue manager uses the security context of the caller when running a work item.

When you queue work items, the work queue manager performs the following tasks:

- Serializes the arguments, security context, and class method or subroutine that comprises the unit of work, and then inserts the serialized data into the global that lists the units of work associated with the work queue
- Signals an event on the work queue
- If additional worker jobs are required and available to process the units of work, causes a worker job to attach to the work queue and decrements the number of available worker jobs

4. Wait for the work to be completed. To do so, you can call the **WaitForComplete()** method of the work queue.

   The work queue manager then performs the following tasks:

   - Waits for a completion event
   - Displays output such as workload metrics to the terminal
   - Collects any errors related to the unit of work
   - If you added units of work to the work queue using the **QueueCallback()** method, runs the callback code

5. Continue processing as appropriate for your application.

The following example shows these basic steps:

```
Set queue=##class(%SYSTEM.WorkMgr).%New(,.sc)
If $$$ISERR(sc) {
   Return sc
}
For i = 1:1:filelist.Count() {
    Set sc=queue.Queue("..Load",filelist.GetAt(i))
    If $$$ISERR(sc) {
        Return sc
    }
}
Set sc=queue.WaitForComplete()
If $$$ISERR(sc) {
    Return sc
}
```

The code initializes the work queue manager and then iterates through a list of files. For each file, the code adds a work queue item that loads a file. After adding all the work queue items, the code waits for the work to be completed.

Note: The %SYSTEM.WorkMgr class supports more complex workflows with the methods described later in this document.

4.1 Basic Methods

To complete the steps described in the previous section, you can use the following three methods of the %SYSTEM.WorkMgr class:

%New()

```
classmethod %New(qspec As %String = "", ByRef sc As %Status, numberjobs As %Integer, category)
 as WorkMgr
```

Creates, initializes, and returns a *work queue*, which is an instance of the %SYSTEM.WorkMgr class that you can use to perform parallel processing. The method accepts the following arguments:

qspec

A string of compiler flags and qualifiers that affect code running within this work queue. See "Viewing Class Compiler Flags and Qualifiers" in the chapter "Defining and Compiling Classes" in *Defining and Using Classes*.

sc

A %Status value that is returned by reference and indicates whether the system was successful when it created and initialized this work queue.

numberjobs

The maximum number of parallel worker jobs to use in this work queue. The default depends on the characteristics of the machine and operating system.

category

The name of the category that supplies the worker jobs to use in this work queue. For more information, see Managing Categories.

The system does not allocate any worker jobs to the queue upon creation. Worker jobs are allocated only after you add a unit of work to the work queue.

Queue()

```
method Queue(work As %String, args... As %String) as %Status
```

Adds a unit of work to a work queue. The method accepts the following arguments:

work

The code to execute. In general, the code should return a %Status value to indicate success or failure.

If the code returns a %Status value, you can use the following syntax:

- `##class(Classname).ClassMethod` for a class method, where *Classname* is the fully qualified name of the class and *ClassMethod* is the name of the method.

  If the method is in the same class, you can use the syntax `..ClassMethod` as shown in the example.
- `$$entry^rtn` for a subroutine, where *entry* is the name of the subroutine and *rtn* is the name of the routine.

If the code does not return a %Status value, use the following syntax instead:

- `=##class(Classname).ClassMethod` for a class method (or `=..ClassMethod` if the method is in the same class)
- `entry^rtn` for a subroutine

See About Units of Work for information about the requirements for units of work.

args

A comma-separated list of arguments for the class method or subroutine. To pass a multidimensional array as an argument, precede that argument with a period as usual so that it is passed by reference.

The size of the data passed in these arguments should be relatively small to make the most of the framework. To pass a large amount of information, use a global instead of an argument.

As you queue units of work, the system allocates worker jobs one at a time up to the *numberjobs* value that you specified when you created the work queue or up to the default value. Additionally, the security context of the caller is recorded, and each work item runs within that security context.

WaitForComplete()

```
method WaitForComplete(qspec As %String, errorlog As %String) as %Status
```

Waits for the work queue to complete all the items and then returns a %Status value to indicate success or failure. The %Status value contains information from all %Status values returned by the work items. The method accepts the following arguments:

qspec

A string of compiler flags and qualifiers. See "Viewing Class Compiler Flags and Qualifiers" in the chapter "Defining and Compiling Classes" in *Defining and Using Classes*.

errorlog

A string of any error information, which is returned as output.

4.2 Properties of Work Queues

Each work queue (or instance of %SYSTEM.WorkMgr) has the following properties:

NumWorkers

The number of worker jobs assigned to the work queue.

NumActiveWorkers

The number of currently active workers.

DeleteTimeout

The number of seconds after you make a call to delete the work queue or the work queue goes out of scope that the system waits before forcefully shutting down any worker jobs that are processing work items. The default value is five, and the value must be greater than zero. If you set the value to zero, the system automatically changes the value to one.

5 Managing Categories

A *category* is an independent pool of worker jobs. When you initialize a set of worker jobs, you can specify the category that supplies the workers. If any of the worker jobs in the set request additional worker jobs while executing work items, then the new worker jobs are from the same category.

Each category is associated with a maximum number of worker jobs, which is stored in the `^%SYS("WQM","GroupDefaultMax",`*`category`*`)` global and can be modified using the **ModifyCategory()** method described below. Additionally, each work queue is associated with a maximum number of worker jobs as described in Properties of Work Queues. Therefore, if a category is assigned 40 worker jobs and you create four work queues in the same category with 10 worker jobs each, then the four work queues can process work in parallel. If one work queue in the category is moving slowly, the work of the other three work queues is unaffected.

The system includes three categories that you cannot delete: SQL, SYS, and Default. The SQL category is for any SQL processing performed by the system, including parallel processing of queries. The SYS category is for system processes and is not intended for end users. The Default category supplies worker jobs when you initialize a set of worker jobs without specifying a category. By default, the maximum number of workers for these categories is `Dynamic`, which is equivalent to twice the number of cores available to your system.

You can create additional categories as needed in the InterSystems IRIS Management Portal. For more information, see Configuring Work Queue Manager Categories.

You can also call the following methods of the WQM.API class from the %SYS namespace to manage existing categories programmatically:

MaxWorkersAvailable()

Returns the maximum number of workers on the system.

IsUserCategory()

Indicates whether a given category name is the name of an existing user-defined category.

IsValidCategoryName()

Indicates whether a given category name is a valid user-defined category name. Category names must be unique and are case sensitive. Additionally, category names can include only letters, numbers, and periods, and have a maximum of 50 characters, including the prefix `User.`. To distinguish user-specified categories from the default categories provided by the system, user-specified categories must include this prefix.

GetWorkersForCategory()

Returns the maximum number of workers assigned to a given category.

ModifyCategory()

Enables you to modify the maximum number of workers assigned to a given category. To reset a system category to a `Dynamic` number of maximum workers, you can call `ModifyCategory(`*`category`*`,0)`, where *category* is the name of the system category. If you set the maximum number of workers for a user-defined category to zero, InterSystems IRIS deletes the category.

You must have the `%Admin_Manage:USE` privilege to invoke each method except **IsUserCategory()** and **IsValidCategoryName()**.

To list all the existing categories on the system and the maximum number of workers associated with them, you can use the **MaxActiveWorkers** query.

6 Using Callbacks

A callback is code that the work queue manager must execute after completing a work item. You can use callbacks for two reasons:

- To perform work that is dependent on the completion of a work item
- To signal that all queued work is completed if you choose to complete work items asynchronously

6.1 Including Callbacks for Work Items

To add a callback, you call the **QueueCallback()** method instead of the **Queue()** method when adding work items to the work queue:

```
method QueueCallback(work As %String, callback As %String, args... As %String) as %Status
```

The *work* and *args* methods are the same as for the **Queue()** method. However, the *callback* argument specifies the callback code to execute using the following syntax:

- `##class(Classname).ClassMethod` for a class method
- `$$entry^rtn` for a subroutine

The class method or subroutine must accept the same arguments, in the same order, as the main work item. The master process passes the same arguments to the main work item and to the callback code.

The callback code can access the following public variables:

- *%job*, which contains the job ID of the process that actually did the work
- *%status*, which contains the %Status value returned by the unit of work
- *%workqueue*, which is the OREF of the work queue instance

These public variables are available within the callbacks but not within the work items.

6.2 Including Callbacks to Determine Completion

Instead of using the **WaitForComplete()** method to wait for all the queued work in a work queue to be completed before returning to the master process, you can poll the work queue manager to determine completion as follows:

- Use the **QueueCallback()** method instead of the **Queue()** method to add work items to the work queue as described in the previous section.
- When the work is completed for all work items, set the public variable *%exit* to 1 in the callback code.
- Use the **Wait()** method instead of the **WaitForComplete()** method:

  ```
  method Wait(qspec As %String, byRef AtEnd As %Boolean) as %Status
  ```

The **Wait()** method waits for a signal from a callback to exit back to the caller. Specifically, it waits for the callback code to set the public variable *%exit* equal to 1. **Wait()** returns *AtEnd* by reference. When *AtEnd* is 1, all the work is completed. Alternatively, if *AtEnd* is 0, one or more work items are not completed.

7 Controlling Output to the Current Device

By default, if work items generate output (**`WRITE`** statements) to the current device, the work queue saves the output in a buffer until the end of **WaitForComplete()** or **Wait()**. If you want a work item to generate output earlier, have that work item call the **Flush()** class method of the %SYSTEM.Context.WorkMgr class, for example:

```
set sc = $system.Context.WorkMgr().Flush()
```

When the work item calls this method, that causes the parent work queue to write all saved output for the work item.

Additionally, you can use the `-d` flag to suppress all output to the current device. In this case, the **Flush()** method does nothing, because there is no output.

8 Pausing and Resuming a Work Queue

The %SYSTEM.WorkMgr class provides methods you can use to pause and resume work within a work queue:

Pause()

```
method Pause(timeout As %Integer, ByRef completed As %Boolean = 0) as %Status
```

Prevents the worker jobs associated with this work queue from accepting *additional* items from this work queue. The **Pause()** method also stops any work items that are in progress.

The *timeout* argument represents the amount of time in seconds that the method waits before stopping work items that are in progress. After the timeout period, the method returns the *completed* value, which indicates whether the work items that were in progress when you called the **Pause()** method were completed. Consequently, you can pass in a *timeout* value of `0` to know immediately whether the worker jobs completed all the work items in the work queue.

Resume()

```
method Resume() as %Status
```

Resumes work in this work queue, if it had previously been paused using the **Pause()** method. Specifically, this method enables the work queue processes to accept and start any additional items in the work queue.

For information on halting work completely, see Stopping a Work Queue and Removing Work Items.

9 Detaching and Attaching a Work Queue

Typically, you initialize a set of worker jobs, queue work items, and then wait for the worker jobs to complete the work items. However, you may encounter situations where worker jobs are taking longer than expected to complete work items or you cannot dedicate a single process to waiting. Consequently, the work queue manager enables you to detach a work queue from a process and subsequently attach the work queue to the same process or to a different process.

For example, suppose that `queue` references a work queue that you initialized. Also suppose that you added several work items to the work queue. Before calling the **Wait()** or **WaitForComplete()** to determine the status of the work being processed, you could employ the following methods:

Detach()

```
method Detach(ByRef token As %String, timeout As %Integer=86400) as Status
```

Detaches the work queue object from the object reference that you created when you initialized the work queue. The **Detach()** method enables any work in progress to continue and preserves the current state of the work queue.

The *token* argument represents a secure token that you can use to subsequently attach the work queue to another process. The *timeout* argument is optional and indicates the amount of time in seconds that the system retains the detached work queue object. After the timeout period has elapsed, the system removes any worker jobs and information associated with the work queue. The default value of *timeout* is one day.

After you call the **Detach()** method, most calls on the detached object reference return errors. However, the **NumActiveWorkers()** and **NumWorkers()** methods return `-1`.

Attach()

```
Attach(token, ByRef sc As %Status) as WorkMgr
```

Attaches a new object reference to a previously detached work queue object if the work queue object is still in memory. The **Attach()** method returns a new instance of the work queue manager associated with the work queue. You can subsequently call methods on the work queue. For example, you can call the **Wait()** method with a *timeout* value of `0` to determine whether the queue had completed any work items before being detached.

The *token* argument represents the secure token returned by the **Detach()** method that you previously called on the work queue.

For example, you could detach and then attach a work queue as follows:

```
Set sc=queue.Detach(.token,60)
If $$$ISERR(sc) {
       Return sc
    }
Set queue=$system.WorkMgr.Attach(token,.sc)
If $$$ISERR(sc) {
       Return sc
    }
```

10 Stopping a Work Queue and Removing Work Items

You can stop a work queue, interrupting any work items in progress and removing any queued work items. To do so, call the **Clear()** method of the work queue.

```
method Clear(timeout As %Integer = 5) as %Status
```

Given the timeout period *timeout* (in seconds), this method waits for the worker jobs to finish their current tasks, and then kills the jobs. The system removes and then recreates the work queue, without any work items attached. Afterword, the system returns immediately from **Wait()** or **WaitForComplete()**.

11 Specifying Setup and Teardown Processing

Each work queue typically has multiple worker jobs. If there are more work items than worker jobs, then a worker job will perform multiple work items, one at a time. It is useful to identify any setup steps needed before these work items start, and invoke all such logic before adding the work items to the queue.

The %SYSTEM.WorkMgr class provides methods, **Setup()** and **TearDown()**, that you can use to define the setup activity and the cleanup activity for the worker jobs. For example, use **Setup()** to set public variables for use within the worker job, and use **TearDown()** to kill those variables. You can also use **Setup()** to take out locks and to set process-private globals, and you would use **TearDown()** to release those locks and remove those globals.

In either case, you must call **Setup()**, **TearDown()**, or both before calling **Queue()** or **QueueCallback()**. The **Setup()** and **TearDown()** methods save information in internal globals used only by the work queue manager. When any worker job starts its first work item from this queue, that worker job first checks the work manager queue globals to see if there is any setup logic. If so, the worker job executes that logic and then starts the work item. The worker job does not execute the setup logic again. Similarly, after any worker job finishes its last work item from the queue, that worker job checks to see if there is any teardown logic. If so, the worker job executes that logic.

The following provides details for these methods:

Setup()

```
method Setup(work As %String, args... As %String) as %Status
```

Specifies the code for a worker process to call before processing its first item from the queue. If you use this method, you must call it before calling the **Queue()** or **QueueCallback** method. **Setup()** accepts the following arguments:

work

The setup code to execute. The supported syntax for this argument is the same as the supported syntax for the *work* argument of the **Queue()** method, which is described in a previous section.

args

A comma-separated list of arguments for this code. To pass a multidimensional array as an argument, you can precede that argument with a period so that it is passed by reference.

You should keep the size of the data passed in these arguments relatively small. To provide a large amount of information, you can use a global instead of passing arguments.

TearDown()

```
method TearDown(work As %String, args... As %String) as %Status
```

Specifies the code for a worker process to call to restore the process to its previous state, after processing its last item from a queue.If you use this method, you must call it before calling the **Queue()** or **QueueCallback** method.

TearDown() accepts the same arguments as the **Setup()** method. However, the *work* argument specifies the teardown code to execute.

12 Retrieving Workload Metrics

The work queue manager compiles various metrics for work performed in the context of a programmatic task:

CommandsExecuted

The number of commands the process executed, which is an approximate measure of CPU usage.

DiskReadMilliseconds

The amount of time in milliseconds spent performing disk reads, which is a measure of latency.

GlobalReferences

The number of references the process made to a specific global node or subtree, which is an approximate measure of database activity.

GlobalUpdates

The number of updates — specifically, **`set`** and **kill** operations — that the process executed on standard globals.

JournalEntries

The number of entries added to the journal.log file by the process. For more information about journaling in InterSystems IRIS, see "Journaling" in the *Data Integrity Guide*.

ProcessPrivateUpdates

The number of updates — specifically, **set** and **kill** operations — that the process executed on process-private globals. A process-private global is an array variable that is accessible only to the process that created it.

The metrics are aggregated each time the **Wait()** or **WaitForComplete()** method of the work queue manager is called and are stored in the **$system.Context.WorkMgr()** special object. You can retrieve the metrics programmatically as follows:

1. Set a variable equal to the $system.Context.WorkMgr() special object, for example:

   ```
   Set context=$system.Context.WorkMgr()
   ```

2. Define variables for collecting each metric of interest, for example:

   ```
   Set gbls=$zutil(67,43,$job)+context.GlobalReferences
   ```

   In this example, *gbls* collects the number of references made to a specific global node or subtree by the worker jobs associated with this work queue manager instance.

3. Perform work queue manager work, ensuring that you call either **WaitForComplete()** or **Wait()**.

4. Report the metrics of interest using the variables you created, for example:

   ```
   Write "Total globals : ",$zutil(67,43,$job)+context.GlobalReferences-gbls,!
   ```