Script Engine Reference

Understand the scripting engine architecture.

Reference **Modules** **Scripts** UI | Engine

Script Engine (Reference) manages the execution, synchronization, and optimization of all scripts, tasks, and expressions within the FrameworX runtime environment.

Advanced Topic: This document provides deep technical insight into the Scripts module execution engine. Most solutions don't require this level of understanding - the default engine behavior handles typical automation requirements automatically.

Overview
Task Execution Model
Diagnostic Properties
Data Synchronization
Pre-Loading System
Performance Optimization
Advanced Scenarios
Troubleshooting
Best Practices

Overview

The Script Engine orchestrates:

- Multi-threaded task execution
- Data synchronization across distributed systems
- Memory management and caching
- Exception protection and diagnostics
- Pre-loading and performance optimization

Understanding the engine helps when:

- Optimizing large-scale solutions
- Debugging complex timing issues
- Managing distributed architectures
- Implementing high-performance scripting

Task Execution Model

Threading Architecture

Each Script Task runs in its own managed thread:

- **Independent execution**: Tasks don't block each other
- **Thread pool management**: Automatic allocation and recycling
- **CPU monitoring**: Track execution time per task
- **Exception isolation**: Failures don't cascade

csharp

```
// Each task automatically wrapped with:
try
{
    // Your task code executes here
    // In its own thread context
}
catch (Exception ex)
{
    // Automatic exception handling
    // Logged but doesn't crash system
}
```

Execution Triggers

Trigger Type	Description	Thread Behavior
Tag Change	Executes when tag value changes	New thread per change
Period	Runs at fixed intervals	Reuses thread if available
Startup	Runs once at module start	Priority thread
Expression	Inline execution	Thread pool shared

CPU Management

The engine monitors CPU usage to prevent system overload:

- Tasks yielding excessive CPU are flagged
- Long-running tasks don't block others
- Automatic thread priority adjustment

Diagnostic Properties

Access runtime metrics via `@Script.Task.[TaskName]` namespace:

Property	Type	Description
`LastExecution`	DateTime	When task last ran
`LastCPUTime`	TimeSpan	Duration of last execution
`PeakCPUTime`	TimeSpan	Maximum execution time recorded
`ExecutionCount`	Integer	Total number of executions
`State`	Enum	Running, Idle, or Error
`LastException`	String	Most recent error message

Monitor these in:

- [Scripts Monitor (Reference)] - Real-time dashboard
- PropertyWatch - During debugging
- Custom displays - For operator visibility

Data Synchronization

Distributed Architecture

The engine maintains data consistency across:

- **Server** (TServer.exe) - Central data authority
- **Clients** (TRichClient.exe, Web) - Remote displays
- **Modules** - Scripts, Datasets, Reports running remotely

Synchronization Queue

Each server-module connection maintains an internal queue:

```
[Server]  [Queue]  [Module/Client]

    Tag Updates
    Method Calls
    State Changes
```

Queue Optimization:

- Each tag occupies ONE queue position
- Rapid changes update value in-place
- Prevents memory overflow
- Latest value prioritized

RaiseAllChanges Property

<ac:structured-macro ac:name="warning"> ac:rich-text-body **Use with Caution**: Enabling `RaiseAllChanges` forces ALL intermediate values into the queue, potentially causing:

- Memory exhaustion
- Queue overflow
- Performance degradation </ac:rich-text-body> </ac:structured-macro>

Enable only when:

- Audit trail requires all values

- Historical accuracy critical
- Processing can handle volume

csharp

```
// Example of potential issue
for (int i = 0; i < 10000; i++)
{
    @Tag.Counter = i;  // With RaiseAllChanges, queues 10,000 updates!
}
```

Synchronization Timeout

- **Default**: 2 minutes
- **Impact**: Large objects (DataTables) must sync within timeout
- **Failure**: Connection drops, requires reconnection

Shutdown Sequence

1. Module receives shutdown signal
2. Current executions complete
3. Synchronization queue flushes
4. Module disconnects
5. Server proceeds with shutdown

Pre-Loading System

Automatic Pre-Loading

The engine implements two pre-loading strategies:

1. Cache-Based Pre-Loading

First Execution:

- No cache exists
- Minimal pre-load
- Objects cached as used

Subsequent Executions:

- Load from saved cache
- Contains previously used objects
- Stabilizes over time

Cache Invalidation:

- Project changes clear cache
- Module updates reset cache
- Starts fresh after changes

2. Anticipatory Pre-Loading

During execution, the engine predicts object usage:

- Analyzes script patterns
- Pre-fetches likely objects
- Reduces first-access delay

Limitations:

- Cannot predict dynamic references
- `TK.GetObjectValue()` unpredictable
- Conditional logic uncertain

Manual Pre-Loading

Force objects into memory before use:

csharp

```
// In ServerStartup task
TK.PreloadTags("Critical.*");  // Preload all Critical tags
TK.PreloadObject(@Alarm.Area1); // Preload specific alarm area
```

Execution Order:

1. Cache-based pre-load completes
2. Manual pre-load executes
3. ServerStartup task runs
4. Other tasks enabled

Pre-Loading Trade-offs

Aspect	More Pre-Loading	Less Pre-Loading
Startup Time	Slower	Faster
First Execution	Faster	May pause for sync
Memory Usage	Higher	Lower
Predictability	Consistent	Variable

Performance Optimization

Script Task Optimization

1. **Minimize Blocking Operations**

csharp

```
// Avoid
Thread.Sleep(5000);  // Blocks thread

// Prefer
@Tag.DelayedAction = true;  // Event-driven
```

1. **Batch Tag Updates**

csharp

```
// Inefficient
for (int i = 0; i < 100; i++)
    @Tag["Item" + i] = value;

// Efficient
var updates = new Dictionary<string, object>();
// Prepare all updates
TK.SetMultipleTags(updates);  // Single sync
```

1. **Use Appropriate Triggers**
    - Change-based for reactive logic
    - Time-based for polling
    - Avoid high-frequency triggers

Expression Optimization

- Simple calculations only
- Complex logic in Script Classes
- Minimize tags per expression
- Use TIF instead of IIF for methods

Memory Management

Monitor and manage:

- Cache size growth
- Queue depth
- Thread count
- Object references

Advanced Scenarios

Large DataTable Synchronization

csharp

```csharp
// Split large tables to avoid timeout
var chunks = TK.SplitDataTable(largeTable, 1000);
foreach (var chunk in chunks)
{
    @Tag.PartialData = chunk;
    TK.Sleep(100);  // Allow queue processing
}
```

Dynamic Object Access

csharp

```csharp
// Pre-load dynamic references
string tagName = GetDynamicTagName();
TK.PreloadTag(tagName);  // Force sync
var value = TK.GetObjectValue("Tag." + tagName);
```

High-Frequency Updates

csharp

```csharp
// Buffer rapid changes
private DateTime lastUpdate;
private double bufferedValue;

public void ProcessHighFrequency(double value)
{
    bufferedValue = value;
    if ((DateTime.Now - lastUpdate).TotalMilliseconds > 100)
    {
        @Tag.Output = bufferedValue;
        lastUpdate = DateTime.Now;
    }
}
```

Troubleshooting

Task Not Executing

- Check `State` property
- Verify trigger conditions
- Review `LastException`
- Monitor CPU usage

Synchronization Issues

- Check queue depth
- Verify timeout settings
- Review network latency
- Monitor memory usage

Performance Problems

- Profile with Scripts Monitor
- Check `PeakCPUTime`
- Review pre-loading strategy
- Optimize synchronization

Best Practices

1. **Let the engine manage threads** - Don't create manual threads
2. **Trust automatic pre-loading** - Manual intervention rarely needed
3. **Monitor diagnostics** - Use built-in properties
4. **Design for distribution** - Consider network delays
5. **Test at scale** - Verify with production data volumes
6. **Profile before optimizing** - Measure actual bottlenecks

---

In this section...