Alarms Engine Reference

Understand the alarms engine architecture.

Reference **Modules** **Alarms** UI | Engine

Alarms Engine (Reference) manages the execution, evaluation, and lifecycle of all alarm conditions within the FrameworX runtime environment. The Alarms Engine orchestrates:

The Alarms Engine orchestrates:

- Continuous condition evaluation
- State machine transitions
- Event generation and logging
- Notification dispatching
- Database synchronization
- Multi-threaded processing

Understanding the engine helps when:

- Optimizing high-volume alarm systems
- Debugging complex alarm behaviors
- Implementing custom notification schemes
- Managing distributed alarm architectures

Alarm Lifecycle
Evaluation Process
Condition Evaluation
Deadband Processing
Database Operations
Notification Pipeline
Performance Optimization
Startup Sequence
Shutdown Sequence
Advanced Configuration
Troubleshooting
Best Practices

Alarm Lifecycle

State Machine

Each alarm item follows this state progression:

- **Normal** - Condition not met
  - Active (condition triggered)
- **Active** - Condition met, not acknowledged
  - Acknowledged (operator ack)
  - Normalized (condition cleared)
- **Acknowledged** - Condition met, acknowledged
  - Normal (condition cleared)
- **Normalized** - Condition cleared, not acknowledged
  - Normal (operator ack)

State Transitions

csharp

```
// State transition events
OnActive()      // Normal  Active
OnAcknowledge() // Active  Acknowledged
OnNormalize()   // Active  Normalized
OnReturn()      // Any  Normal
```

Evaluation Process

Scan Cycle

The engine evaluates alarms in cycles:

1. **Tag Value Update** - Tag system notifies engine
2. **Condition Check** - Evaluate against limits
3. **Deadband Verification** - Apply time/value deadbands
4. **State Evaluation** - Determine state change
5. **Event Generation** - Create alarm event
6. **Notification Dispatch** - Trigger notifications

Multi-Threading

- **Main Thread** - Coordinates evaluation
- **Evaluation Threads** - Pool for condition checking
- **Database Thread** - Async logging

- **Notification Threads** - Parallel notification dispatch

Thread pool size: Min(CPU cores * 2, Active alarm count / 100)

Condition Evaluation

Standard Conditions

For each scan cycle:

```
IF (EvaluateCondition(TagValue, Limit))
  AND (NOT InDeadband())
  AND (EnableCondition == True)
  AND (Group.Enabled == True)
THEN
  TriggerAlarm()
```

Rate of Change

Special handling for RateOfChange:

```
CurrentRate = (CurrentValue - PreviousValue) / TimeDelta
IF (CurrentRate >= Limit)
  TriggerAlarm()
```

Sampling: Every 100ms for RateOfChange conditions

Deviation Alarms

```
Deviation = ABS(TagValue - Setpoint)
IF (Deviation > Limit + SetpointDeadband)
  TriggerAlarm()
```

Deadband Processing

Time Deadband

Prevents re-triggering within time window:

- **Activation** - Alarm triggers at T0
- **Deadband Period** - T0 to (T0 + TimeDeadband)
- **Blocked** - Cannot re-trigger during period
- **Clear** - After deadband expires

Value Deadband

Hysteresis to prevent chattering:

- **Hi Alarm Example**:
  - Trigger: Value >= 100
  - Clear: Value < (100 - Deadband)
  - Re-trigger: Value >= (100 + Deadband)

Combined Deadbands

Both time AND value must be satisfied:

```
CanRetrigger = (TimeSinceLastAlarm > TimeDeadband)
            AND (ValueChange > ValueDeadband)
```

Database Operations

Write Queue

Asynchronous database writes:

- **Queue Size**: 10,000 events (configurable)
- **Batch Size**: 100 events per write

- **Flush Interval**: 1 second or when batch full

Event Types Logged

- **Alarm Active** - Condition triggered
- **Alarm Acknowledged** - Operator acknowledgment
- **Alarm Normalized** - Condition cleared
- **Alarm Return** - Return to normal
- **Configuration Change** - Runtime modifications

Database Schema

sql

```
AlarmEvents (
  EventID BIGINT PRIMARY KEY,
  Timestamp DATETIME,
  TagName VARCHAR(255),
  AlarmGroup VARCHAR(100),
  AlarmArea VARCHAR(255),
  EventType INT,
  Value FLOAT,
  Limit FLOAT,
  Message TEXT,
  UserName VARCHAR(100),
  AckTime DATETIME
)
```

Store and Forward Batch Sizes

Available from FrameworX 10.1.5c. When an Alarm Logging Database uses Store and Forward, the number of rows replayed per synchronization cycle is configurable server-side in `TServer.json` (in the `FrameworX\MachineSettings` folder), under the `appSettings` object. Two independent Alarm keys control it:

Key	Default	Drives
`Alarm.StoreAndForwardRecoveryBatchSize`	100	Recovery pass (KeepLocalCopy = off) — drains the local cache after the remote database reconnects.
`Alarm.StoreAndForwardNotSyncBatchSize`	1000	Replay pass (KeepLocalCopy = on) — replays flagged rows while preserving the local copy.

Both validate to the range [50, 50000]; out-of-range or missing values fall back to the legacy default. For the full procedure, the JSON example, and the equivalent Historian key, see Configuring Store and Forward Batch Sizes (Historian Engine Reference).

Note: this is distinct from the alarm-event **Write Queue** batch above, which governs how alarm events are flushed to the logging database during normal operation.

Redundancy: Store and Forward and database redundancy replication of alarm history are mutually exclusive, so choose one of the two. The Designer blocks the combination in Alarms > Global Settings, and if a solution has both, the runtime turns Store and Forward off at startup and logs the reason.

Alarm History Query Row Ceiling

Available from FrameworX 10.1.5e. Alarm history queries load their result set into runtime memory before returning it. On a large alarm database, or when the logging database responds slowly, a single wide query can hold millions of rows at once and drive the runtime process memory up sharply. An optional server-side ceiling bounds how many rows any one alarm history query may load.

The ceiling is configured in `TServer.json` (in the `FrameworX\MachineSettings` folder), under the `appSettings` object:

Key	Default	Range	Drives
`Alarm.HistoryQueryMaxRows`	100000	64000 to 5000000	Maximum number of alarm history rows a single query may load into runtime memory. Applies to every alarm history request served by the runtime: AlarmViewer history views, TrendChart and DrillingChart annotation requests, and scripts that query alarm history.

The key is optional. When it is absent the default of 100000 applies, so existing installations need no action and no `TServer.json` edit. A value outside the valid range is ignored: the runtime logs a Warning in the TraceWindow and keeps the default. The minimum of 64000 exists because TrendChart and DrillingChart request exactly 64000 rows of alarm history for their annotations, and a lower ceiling would degrade trend annotations.

Behavior When a Query Exceeds the Ceiling

Exceeding the ceiling is not an error. The query succeeds and returns the **newest** rows up to the ceiling. The runtime logs a Warning naming the requested row count, the applied ceiling and this configuration key, at most once every 60 seconds so that repeated polling does not flood the TraceWindow.

Note for display engineers: an AlarmViewer or an alarm report whose `MaxLines` property is configured above the ceiling now shows at most the ceiling, the newest rows, without an error banner. Sites that genuinely need to export more rows in a single view should raise `HistoryQueryMaxRows` deliberately.

Configuring the Alarm History Row Ceiling

The setting is machine-wide and applies to every solution running on the machine. Each solution runs its own TServer process, and all of them read the same `TServer.json` configuration file.

1. Open `TServer.json` in the `FrameworX\MachineSettings` folder.
2. Inside the `appSettings` object, add or locate the `Alarm` section and set the value.
3. Save the file and restart the solution runtime. No machine reboot and no reinstall are needed.

```
{
  "appSettings": {
    "Alarm": {
      "HistoryQueryMaxRows": 100000
    }
  }
}
```

Raise the ceiling only when a site has a genuine need to return more rows in one query, and size it against the memory available to the runtime server. Lowering it reduces peak memory per query at the cost of showing fewer history rows in a single view.

Notification Pipeline

Dispatch Sequence

1. **Group Notification** - Check NotificationMethod
2. **Sound Alert** - Play configured sound
3. **Visual Update** - Update display elements
4. **Script Execution** - Call custom methods
5. **External Systems** - API calls/integrations

Priority Handling

High-priority alarms bypass queue:

```
Priority >= 900: Immediate dispatch
Priority 500-899: High priority queue
Priority 100-499: Normal queue
Priority < 100: Low priority queue
```

Performance Optimization

Evaluation Strategies

Lazy Evaluation:

- Only evaluate on tag change
- Skip disabled alarms
- Cache unchanging conditions

Batch Processing:

- Group related alarms
- Evaluate in parallel
- Single database transaction

Memory Management

- **Alarm Pool**: Pre-allocated alarm objects
- **Event Buffer**: Circular buffer for events
- **Cache Size**: Last 1000 evaluations
- **String Interning**: Reuse message strings

CPU Optimization

- **Condition Caching**: Store compiled expressions
- **SIMD Operations**: Vectorized limit checks

- **Thread Affinity**: Pin threads to cores

Startup Sequence

1. **Load Configuration** - Read alarm definitions
2. **Initialize Database** - Connect and verify schema
3. **Compile Conditions** - Pre-compile expressions
4. **Cache Warmup** - Load recent alarm states
5. **Apply Disable Time** - Suppress per settings
6. **Begin Evaluation** - Start scan cycles

Shutdown Sequence

1. **Stop Evaluation** - Halt new scans
2. **Process Queue** - Complete pending evaluations
3. **Flush Database** - Write all events
4. **Save State** - Cache current states
5. **Close Connections** - Cleanup resources

Advanced Configuration

Engine Tuning Parameters

Access via configuration file:

xml

```xml
<AlarmEngine>
  <ThreadPoolSize>16</ThreadPoolSize>
  <EvaluationInterval>100</EvaluationInterval>
  <DatabaseBatchSize>100</DatabaseBatchSize>
  <QueueSize>10000</QueueSize>
  <CacheSize>1000</CacheSize>
  <UseParallelEvaluation>true</UseParallelEvaluation>
</AlarmEngine>
```

Runtime Metrics

Monitor engine performance:

csharp

```csharp
var metrics = @Alarm.Engine.Metrics;
int evaluationsPerSec = metrics.EvaluationRate;
int queueDepth = metrics.QueueDepth;
double avgEvalTime = metrics.AverageEvaluationTime;
int activeThreads = metrics.ActiveThreadCount;
```

Troubleshooting

High CPU Usage:

- Reduce evaluation frequency
- Optimize complex conditions
- Increase deadband values
- Check for alarm storms

Database Lag:

- Increase batch size
- Optimize connection string
- Check database performance
- Archive old data

Memory Growth:

- Reduce cache size
- Check for memory leaks
- Monitor queue depth

- Clear old events

Missed Alarms:

- Verify evaluation interval
- Check thread pool size
- Review condition logic
- Monitor system resources

Best Practices

1. **Size thread pool appropriately** - 2x CPU cores typical
2. **Use deadbands effectively** - Prevent evaluation storms
3. **Optimize conditions** - Simple expressions evaluate faster
4. **Batch database writes** - Reduce I/O overhead
5. **Monitor queue depth** - Indicates processing bottlenecks
6. **Regular maintenance** - Archive and optimize database
7. **Profile before optimizing** - Measure actual bottlenecks

---

In this section...
