AThread – Managed Thread Lifecycle Utility

AThread provides a safe, registered, and controllable wrapper for background thread execution in .NET/C#. It integrates automatic registration with a central ThreadsManager, cooperative cancellation, and logging. This ensures that all threads can be monitored, stopped, and cleaned up—especially useful in modular, reloadable apps.

# Key Features

- Named thread instances for traceability  
- Registration and deregistration with ThreadsManager  
- Cooperative cancellation via CancellationTokenSource  
- Clean start/stop methods and status queries  
- Exception handling and logging for thread lifecycle events

# AThread Code Example

Here is the class implementation:

public class AThread  
{  
 public string InstanceName { get; init; }  
 private Thread \_thread;  
 private CancellationTokenSource \_cts = new();  
 public bool IsRunning => \_thread.IsAlive && !\_cts.IsCancellationRequested;  
  
 public AThread(string instanceName, Action work, bool isBackground = true)  
 {  
 InstanceName = instanceName;  
  
 \_thread = new Thread(() =>  
 {  
 try { work(); }  
 catch (OperationCanceledException)  
 {  
 Logger.Log($"[AThread] {InstanceName} cancelled.");  
 }  
 catch (Exception ex)  
 {  
 Logger.Log($"[AThread] {InstanceName} error: {ex.Message}");  
 }  
 });  
 \_thread.IsBackground = isBackground;  
  
 ThreadsManager.Register(this);  
 Logger.Log($"[AThread] Registered: {InstanceName}");  
 }  
  
 public void Start()  
 {  
 if (\_thread.ThreadState == ThreadState.Unstarted)  
 {  
 \_thread.Start();  
 Logger.Log($"[AThread] Started: {InstanceName}");  
 }  
 }  
  
 public void Stop()  
 {  
 if (\_thread == null || !\_thread.IsAlive)  
 return;  
  
 Logger.Log($"[AThread] Stop requested: {InstanceName}");  
  
 \_cts.Cancel();  
 if (!\_thread.Join(2000))  
 {  
 Logger.Log($"[AThread] Still running after Cancel: {InstanceName} — trying Interrupt...");  
 \_thread.Interrupt();  
  
 if (!\_thread.Join(1000))  
 {  
 Logger.Log($"❌ [AThread] Cannot stop thread {InstanceName} cleanly.");  
 throw new InvalidOperationException($"Thread {InstanceName} refused to stop.");  
 }  
 }  
  
 Logger.Log($"✅ [AThread] Cleanly stopped: {InstanceName}");  
 ThreadsManager.Deregister(this);  
 }  
}

# How to Use

1. Create a new AThread with a name and a work method:  
 var myThread = new AThread("Worker1", () => { while(IsRunning) { ... } });  
2. Start the thread with myThread.Start();  
3. Stop it any time with myThread.Stop();  
All instances are automatically registered for management and can be safely terminated on app shutdown or project rebuild.

# Best Practices

- Always check IsRunning in long-running work loops to allow cooperative cancellation.  
- Use ThreadsManager to monitor and stop all threads if required.  
- Prefer AThread over direct Thread usage for resource safety and app lifecycle management.

# Diagram: Thread Lifecycle

TBA

# Summary

AThread is designed to make multithreaded code safer and more maintainable in dynamic, modular apps.