ATask – Managed Task Lifecycle Utility

ATask provides a managed, registered, and cancellable abstraction for background task execution in .NET/C#. It supports both fire-and-forget and result-returning asynchronous operations, with centralized registration for lifecycle management and safe cleanup.

# 1. Key Features

- Named task instances for traceability  
- Registration/deregistration with TaskManager  
- Cooperative cancellation via CancellationToken  
- Both fire-and-forget and awaitable generic variants  
- Exception handling and logging

# 2. Variants

## ATask – Fire-and-Forget

public class ATask  
{  
 public string InstanceName { get; }  
 private CancellationTokenSource \_cts = new();  
 private Task \_task;  
  
 public bool IsRunning => !\_task.IsCompleted && !\_cts.IsCancellationRequested;  
  
 public ATask(string instanceName, Action<CancellationToken> work)  
 {  
 InstanceName = instanceName;  
 \_task = new Task(() => work(\_cts.Token), \_cts.Token);  
 TaskManager.Register(this);  
 }  
  
 public void Start()  
 {  
 if (\_task.Status == TaskStatus.Created)  
 \_task.Start();  
 }  
  
 public void Stop()  
 {  
 \_cts.Cancel();  
 TaskManager.Deregister(this);  
 }  
}

Usage Example (Fire-and-Forget):

var myTask = new ATask("Worker1", token => {  
 while (!token.IsCancellationRequested)  
 {  
 DoSomeWork();  
 }  
});  
myTask.Start();  
// ... later:  
myTask.Stop();

## ATask<TResult> – Awaitable with Return Value

public class ATask<TResult>  
{  
 public string InstanceName { get; }  
 private CancellationTokenSource \_cts = new();  
 private Task<TResult> \_task;  
  
 public bool IsRunning => !\_task.IsCompleted && !\_cts.IsCancellationRequested;  
  
 public ATask(string instanceName, Func<CancellationToken, TResult> work)  
 {  
 InstanceName = instanceName;  
 \_task = new Task<TResult>(() => work(\_cts.Token), \_cts.Token);  
 TaskManager.Register(this);  
 }  
  
 public void Start()  
 {  
 if (\_task.Status == TaskStatus.Created)  
 \_task.Start();  
 }  
  
 public async Task<TResult> WaitAsync()  
 {  
 return await \_task;  
 }  
  
 public void Stop()  
 {  
 \_cts.Cancel();  
 TaskManager.Deregister(this);  
 }  
}

Usage Example (Result/awaitable):

var calcTask = new ATask<int>("Adder", token => {  
 int sum = 0;  
 for (int i = 0; i < 100 && !token.IsCancellationRequested; i++)  
 sum += i;  
 return sum;  
});  
calcTask.Start();  
int result = await calcTask.WaitAsync(); // returns 4950

# 3. Best Practices

- Always use the provided CancellationToken to exit loops and support cooperative cancellation.  
- Register all tasks for lifecycle management and shutdown.  
- Prefer ATask and ATask<TResult> over direct Task usage for better safety and app management.

# 4. Task Lifecycle Diagram

TBA

# 5. Summary

ATask makes asynchronous and background code safer and more maintainable in dynamic, modular applications.