FACTORYTALK OPTIX UNIVERSITY — STUDENT REFERENCE SHEET

C# NetLogic for PLC Programmers Cheat Sheet

Translating Studio 5000 Ladder & Structured Text concepts into event-driven FactoryTalk Optix C# scripts and IT/OT REST APIs.

01 The PLC-to-Optix C# Rosetta Stone

Unlike a PLC that executes a continuous top-to-bottom scan every few milliseconds, FactoryTalk Optix NetLogic is **strictly event-driven and timer-driven**. Map your existing Studio 5000 mental model directly to these C# constructs:

Studio 5000 / PLC Concept	Optix C# NetLogic Equivalent	Execution Behavior in Optix
First-Scan Bit (`S:FS`) Power-Up Handler	`public override void Start()`	Executes once when the NetLogic node initializes (at project startup or when its parent UI screen opens).
Shutdown / Pre-Scan Cleanup	`public override void Stop()`	Executes when the parent screen closes or runtime stops. Always dispose of timers and event listeners here.
AOI Input/Output Parameters Local / InOut AOI Tags	**Child Variables under NetLogic Node** `LogicObject.GetVariable("TagName")`	Exposes configurable pins in the Optix Studio properties inspector so engineers never hardcode values in C#.
JSR (Jump to Subroutine) HMI Button Macro	`[ExportMethod]` attribute above a C# method	Exposes the C# function to Studio so UI button clicks, alarm events, or tag transitions can invoke it.
Timer On-Delay (`TON`) Periodic Task / Scan Loop	`DelayedTask` (One-shot delay) `PeriodicTask` (Repeating interval)	Executes a callback method after X ms or every X ms without freezing the HMI user interface thread.
One-Shot Rising (`ONS`) Tag Change Trigger	`VariableChange` Event Handler or `VariableTransitionEventDispatcher`	Fires C# logic only when a monitored variable transitions (e.g., rising edge `0 → 1` of an alarm bit).
CIP `MSG` Instruction `.EN`, `.DN`, and `.ER` status bits	`Task.Run(async () => ...)` + `await httpClient.PostAsync(...)`	Dispatches network/API packets on a background thread and awaits HTTP status without halting HMI execution.

02 AOI-Style Parameter Binding & Background PLC Tag Polling

Crucial Optix Rule: By default, Optix optimizes network traffic by only polling PLC tags currently displayed on the active screen. To read/write PLC tags reliably inside a background C# script, bind child variables and synchronize them via `VariableSynchronizer`:

```
public class PumpMonitorLogic : BaseNetLogic
{
    private IUAVariable pumpSpeedVar;
    private IUAVariable faultBitVar;
    private VariableSynchronizer tagSync;

    public override void Start()
    {
        // 1. Grab references to Child Variables attached to this NetLogic node
        pumpSpeedVar = LogicObject.GetVariable("PumpSpeedRPM");
        faultBitVar  = LogicObject.GetVariable("PumpOverloadBit");

        // 2. Force continuous PLC polling even when the operator is on another
screen
        tagSync = new VariableSynchronizer();
        tagSync.Add(pumpSpeedVar);
        tagSync.Add(faultBitVar);

        // 3. Attach an ONS-style change listener to the fault bit
        faultBitVar.VariableChange += OnFaultBitChanged;
    }

    private void OnFaultBitChanged(object sender, VariableChangeEventArgs e)
    {
        if ((bool)e.NewValue == true && (bool)e.OldValue == false) // Rising Edge
(ONS)
            Log.Warning("PumpMonitor", $"Overload tripped at {pumpSpeedVar.Value}
RPM!");
    }

    public override void Stop()
    {
        faultBitVar.VariableChange -= OnFaultBitChanged;
        tagSync?.Dispose(); // Clean up PLC polling subscription
    }
}
```

03 Timers & Periodic Execution (`PeriodicTask`, `DelayedTask`, `LongRunningTask`)

Never call `Thread.Sleep()` on the main NetLogic thread—it acts like a PLC watchdog fault and freezes the HMI touchscreen. Always select the matching native Optix task class and dispose of it inside `Stop()`:

Optix Task Class	PLC Equivalent	C# Instantiation & Lifecycle Pattern
`PeriodicTask` Repeating cyclic execution	Studio 5000 **Periodic Task**	`myPeriodic = new PeriodicTask(OnTickMethod, 5000, LogicObject);` `myPeriodic.Start();` — Calls `OnTickMethod()` every 5,000 ms.
`DelayedTask` Single delayed execution	Ladder `TON` **Timer**	`myDelay = new DelayedTask(OnTimerDone, 1500, LogicObject);` `myDelay.Start();` — Waits 1,500 ms once, then executes `OnTimerDone()`.
`LongRunningTask` Heavy file/DB work	Background **Co-Processor**	`myLongTask = new LongRunningTask(ProcessCsvFile, LogicObject);` `myLongTask.Start();` — Runs intensive loops on a separate worker thread.

04 Outbound IT/OT Webhook Pattern (`HTTP POST` — "The IT CIP MSG Write")

Use a single `static readonly HttpClient` instance and `Task.Run` to push real-time JSON alerts to Microsoft Teams, Slack, or enterprise webhooks without blocking the HMI thread:

```
using System.Net.Http;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;

// Inside your BaseNetLogic class:
private static readonly HttpClient httpClient = new HttpClient();

[ExportMethod]
public void SendWebhookAlert(string message, int severity)
{
    string url = LogicObject.GetVariable("WebhookUrl").Value;
    string asset = LogicObject.GetVariable("MachineName").Value;

    // Offload HTTP POST like an asynchronous PLC MSG instruction
    Task.Run(async () =>
    {
        try
        {
            var payload = new {
                timestamp = DateTime.UtcNow.ToString("o"),
                machine = asset,
                text = $"**[{asset}] Alert (Sev {severity}):** {message}"
            };
            string json = JsonSerializer.Serialize(payload);
            using var content = new StringContent(json, Encoding.UTF8,
"application/json");

            HttpResponseMessage res = await httpClient.PostAsync(url, content);
            LogicObject.GetVariable("LastStatus").Value = $"HTTP
{(int)res.StatusCode} {res.ReasonPhrase}";
        }
        catch (Exception ex) { Log.Error("Webhook", ex.Message); }
    });
}
```

05 Inbound REST API Poller Pattern (`HTTP GET` + Surgical JSON Parsing)

Use `JsonDocument.Parse` and `TryGetProperty` to cherry-pick nested values from a REST API JSON response (like weather or utility tariff data) and write them straight into Optix variables linked to PLC tags:

```
[ExportMethod]
public void FetchApiDataNow()
{
    Task.Run(async () =>
    {
        try
        {
            string url = "https://api.open-meteo.com/v1/forecast?
latitude=29.95&longitude=-90.07" +

"&current=temperature_2m,relative_humidity_2m&temperature_unit=fahrenheit";

            string json = await httpClient.GetStringAsync(url);

            // Traverse JSON tree safely: root -> "current" -> "temperature_2m"
            using JsonDocument doc = JsonDocument.Parse(json);
            if (doc.RootElement.TryGetProperty("current", out JsonElement current))
            {
                if (current.TryGetProperty("temperature_2m", out JsonElement temp))
                    LogicObject.GetVariable("TemperatureF").Value =
temp.GetDouble();

                if (current.TryGetProperty("relative_humidity_2m", out JsonElement
rh))
                    LogicObject.GetVariable("RelativeHumidity").Value =
rh.GetInt32();
            }
        }
        catch (Exception ex) { Log.Error("ApiPoller", $"GET Failed: {ex.Message}");
}
    });
}
```

Five Golden Rules for C# NetLogic Debugging & Reliability

1. Use `Log.Info("Category", "Message")` as your Watch Window: Inspect real-time output in the Optix Studio Log pane. **2. Never Name a NetLogic Node After a System Class:** Naming a script node `HttpClient` or `SerialPort` causes C# namespace collisions. **3. Always Use `CultureInfo.InvariantCulture` for URL Decimals:** Prevents European OS locales from replacing decimal points with commas in URL coordinates. **4. Null-Check Variables:** Verify `LogicObject.GetVariable("Name") != null` in `Start()`. **5. Dispose Tasks in `Stop()`:** Call `?.Dispose()` on every `PeriodicTask`, `DelayedTask`, and `VariableSynchronizer` to prevent memory leaks when switching screens.