Devices Channels Reference

Organize device communications in channels.

Devices Channels (Reference) encapsulate protocol configurations and settings, providing communication pathways between the HMI and field devices through various network interfaces.

Device Channels provide:

- Protocol encapsulation
- Interface configuration (Serial, TCP/IP)
- Connection management
- Timeout handling
- Remote execution capability
- Diagnostic monitoring

Each channel represents a configured communication path using a specific protocol to connect with PLCs or field devices.

Creating Channels
Channel Properties
Interface Settings
Channel States
Pending Write Limit
Remote Channels
Runtime Control
Common Configurations
Best Practices Checklist
Troubleshooting
Diagnostics
Diagnosing a channel reading TotalPoints = 0
Driver-health signals

Creating Channels

> ⚠ **Online Configuration limitation:** When a new channel is added to a running solution via Online Configuration, the channel registers in the namespace immediately, but tag polling does not start until the next runtime restart. Stop and restart the runtime to bring the new channel into live communication. Existing channels are not affected.

1. Navigate to **Devices Channels**
2. Click **New Channel** or **Plus** button
3. Select protocol from list
4. Configure:
    - **Channel Name** - Unique identifier
    - **Description** - Documentation
5. Click **OK**

Channel Properties

Core Settings

Property	Description	Required
Name	Channel Name	Yes
Protocol	Selected Protocol	Yes
Description	Object Description	No
Category	Editable classification label you assign to organize and tag channels by category — the same classification field available on Device Nodes and Device Access Types. Categories assigned to a channel help filter and organize channels, for example in the Device module diagnostics views and in Historian and Dataset data views.	No
Protocol Options	Protocol-specific settings	No
Interface	Communication type. Serial, Multi Serial, TCP/IP, Custom	No
InitialState	Channel startup behavior — Enabled, Disabled, Remote, Reserved	No
Timeout	Response timeout	No
RemoteSettings	Remote execution target. Used only when InitialState = Remote. IP:Port of the remote node's TWebServices (Solution Center) service — the node physically connected to the field device. Default port 10108 (older builds may differ; operator-configurable).	No
DriverVersion	Current driver version. Read-only	No

Interface Types

Serial

- RS-232/485 communication
- Single port configuration
- Hardware handshaking options

Multi Serial

- Multiple RS-232 ports
- Port count configuration
- Independent port settings

TCP/IP

- Ethernet/wireless networks
- Connection pooling
- Multiple simultaneous connections

Interface Settings

TCP/IP Settings

Setting	Description	Default
Listening Port	TCP port for slave connections	502 (Modbus)
Node Connection	Parallel requests per node	1
Max Simultaneous	Concurrent connections limit	10
Share Node Same IP	Multiple slaves on single IP	False
Use Single Thread	Single thread for same IP	False
Use Ping Check	Verify connection before send	False
Accept Unsolicited	Allow unsolicited messages	False

Serial Settings

- **Baud Rate** - Communication speed
- **Data Bits** - 7 or 8
- **Parity** - None, Even, Odd
- **Stop Bits** - 1 or 2
- **Handshake** - None, RTS/CTS, XON/XOFF

Channel States

State	Description	Use Case
Enabled	Active at startup	Normal operation
Disabled	Loaded but inactive	Maintenance mode
Remote	Runs on remote computer	Distributed architecture
Reserved	Not loaded	Temporarily excluded

Pending Write Limit

Each channel queues the writes it has not sent yet. Below the limit, the channel sends every write, so a gateway forwarding a burst of values for the same point keeps every intermediate value. Above the limit, writes of the same point are combined, and memory stays bounded at about 42 MB for a saturated channel.

From FrameworX 10.1.6, the default limit is 100,000 pending writes per channel. To set another limit, edit **WriteQueueLimit** in **MachineSettings\TServer.json**, for example to shorten the delay on a busy channel:

```
{
  "appSettings": {
    "Device": {
      "WriteQueueLimit": 10000
    }
  }
}
```

Accepted values are 1 to 1,000,000. Any other value uses the default and writes a warning to the trace. The runtime reads the setting when the channel starts. On a default installation, the MachineSettings folder is `C:\Users\Public\Documents\FrameworX\MachineSettings`.

> **Changed in FrameworX 10.1.6**
>
> The default pending-write limit rose from 1,000 to 100,000. A WriteQueueLimit already present in MachineSettings\TServer.json overrides the default. If the file on the machine sets 1000 and you want the new default, change the value to 100000 or remove the key.

Remote Channels

Execute Device Module on remote computers for gateway and tunneling applications.

How it works

The **main server** (where the solution runs and where you want the tags to update) reads each channel whose InitialState is **Remote**, connects to the **remote node** named in RemoteSettings, and launches a **TRemoteDevice** process **on that remote node only**. TRemoteDevice runs the protocol driver locally on the node, talks to the field device, and connects back to the main server to deliver the data. The remote node is the computer physically wired to the device; the main server never runs TRemoteDevice itself.

System Requirements

- Main server and remote node reachable over the network
- FrameworX installed on both computers, including the channel's protocol driver on the remote node
- **The TWebServices (Solution Center) service must be running on the remote node** and listening on its configured port (default 10108). This is the service that receives the remote-launch request — if it is not running or the port is wrong, no TRemoteDevice starts and no values arrive.
- Firewall open **both directions**: main server remote node on the TWebServices port, and remote node main server on the TServer port (the remote device connects back to the server)

Configuration

1. On the **main server's** solution, set the channel's **InitialState** to **Remote**
2. Set **RemoteSettings** to the **remote node's** address as **IP:Port**, where the port is that node's TWebServices service port (default 10108):

```
192.168.1.100:10108
```

Required Ports

Enable in firewall (Inbound/Outbound):

- TWebServices (Solution Center) on the remote node — default 10108. **This is the port you put in RemoteSettings.**
- TServer on the main server — default 3101. The remote device connects back to this.
- Custom ports as configured

Remote Execution Process

1. The main server (TServer/TStartup) connects to the **TWebServices (Solution Center) service on the remote node** at the RemoteSettings address
2. **TRemoteDevice launches on the remote node only** — it does not run on the main server
3. TRemoteDevice runs the channel's protocol driver locally on the remote node and connects back to the main server
4. Field data is transmitted back to the main server, where the channel's tags update

Special Considerations

⚠ Protocol drivers with specific permissions (PI, Tundra, Prediktor) must be executed by Windows Users. TWebServices or IIS must run under the same Windows User account.

Runtime Control

Start/Stop Operations

csharp

```
// Stop channel
@Device.Channel.ChannelName.Stop();

// Start channel
@Device.Channel.ChannelName.Start();

// Stop with timeout
@Device.Channel.ChannelName.Stop(10);
```

Script Example

csharp

```
public class ChannelControl
{
    public void RestartChannel(string channelName)
    {
        // Stop channel
        @Device.Channel[channelName].Stop(5);

        // Wait for stop
        Thread.Sleep(1000);

        // Start channel
        @Device.Channel[channelName].Start();
    }
}
```

Common Configurations

Modbus TCP

```
Protocol: Modbus
Interface: TCP/IP
Port: 502
Node Connection: 1
Timeout: 1000ms
```

Serial RS-485

```
Protocol: Modbus RTU
Interface: Serial
Baud: 19200
Data Bits: 8
Parity: Even
Stop Bits: 1
```

OPC UA

```
Protocol: OPC UA
Interface: TCP/IP
Port: 4840
Security: Basic256
Max Connections: 5
```

Best Practices Checklist

- [ ]

Name Channels Clearly - Include protocol and purpose

- [ ] **Document Settings** - Use description field
- [ ] **Set Appropriate Timeouts** - Match network conditions
- [ ] **Configure Connection Limits** - Prevent overload
- [ ] **Use Remote Channels** - For distributed systems
- [ ] **Monitor Channel Status** - Track communication health
- [ ] **Test Before Production** - Verify all settings

Troubleshooting

Channel won't start:

- Check InitialState setting
- Verify protocol license
- Review interface configuration
- Check port availability

Communication timeouts:

- Increase timeout value
- Check network latency
- Verify device response time
- Review connection settings

Connection failures:

- Verify IP addresses
- Check firewall rules
- Test network connectivity
- Review security settings

Remote channel issues (no TRemoteDevice / no values):

- **Confirm the TWebServices (Solution Center) service is running on the remote node** and reachable on the RemoteSettings port (default 10108) — this is the most common cause
- Verify RemoteSettings uses the TWebServices port, **not** the TServer port (3101) and **not** the field-device port
- Confirm the reverse path is open: the remote node must reach the main server's TServer port
- Verify TRemoteDevice and the channel's protocol driver are present in the remote node's install
- Review firewall exceptions in both directions

Diagnostics

Remote Channel Monitoring

Verification Steps

1. On the main server, the TStartup/TServer trace shows `Starting Module Device (Channel: <name>) :: Remote: <ip>`
2. Confirm a **TRemoteDevice** process is running on the **remote node** (it does not appear on the main server)
3. If it did not start, the server trace shows `Module Device (Channel: <name> failed` with the connection error — check that the remote node's TWebServices service is running and reachable on the RemoteSettings port
4. Once running, confirm the channel's `IsRunning = 1` and WatchDog advancing (see Driver-health signals below)

Monitor channel performance:

csharp

```
// Get channel run state
int isRunning = @Device.Channel.ChannelName.IsRunning; // 1 = running, 0 = stopped

// Check last error code
int lastErrorCode = @Device.Channel.ChannelName.LastErrorCode;

// Get statistics
uint successAmount = @Device.Channel.ChannelName.SuccessAmount;
uint failAmount = @Device.Channel.ChannelName.FailAmount;
```

Diagnosing a channel reading TotalPoints = 0

TotalPoints is the count of device points (Tag-to-address mappings) actually bound to a channel's live driver across every Node and AccessType beneath it. The runtime writes it from the driver's resolved point list when the Device Module initializes the channel — it is **not** a count of rows in the configuration. A channel showing **TotalPoints = 0** therefore has three distinct causes that look identical on a dashboard but call for different responses:

Cause	What is happening	How to confirm	Fix
Driver has no live point list	Points are defined in the configuration, but the driver's resolved point list is empty, so the runtime wrote TotalPoints = 0. Tags bound to the channel hold their **StartValue** (the value assigned on runtime startup) because no live binding is feeding them.	Tag values sit unchanged at their configured StartValue; the channel reports no read/write traffic. Check that the solution was built and that the points loaded for this channel.	Rebuild and start the runtime so the points load and bind to the driver. This is a configuration/startup state, not a field-communication fault — do not start by suspecting the physical link.
Server / slave-role driver, no master connected	A driver running in **server** or **unsolicited-message** role (for example a Modbus slave, an OPC UA server endpoint, or an MQTT broker-side configuration) does not poll. It listens and accepts incoming connections, then services requests passively. With no external master/client connected yet, zero points and zero SuccessAmount are the **normal, healthy** idle state.	The channel is configured with **Accept Unsolicited** / server-mode behavior and is listening on its port. IsRunning = 1 with no errors. Traffic begins only once a master /client connects.	None required — the driver is waiting passively by design. Confirm the remote master/client is configured to connect to this endpoint.
Communication was live, then went Bad	Points were bound and reading, then quality went Bad and traffic stopped — a genuine communication failure (cable, endpoint, device power, network path). This is the only one of the three that is an actual fault.	FailAmount rises; **Status** shows a non-zero error code; **LastErrorCode** / **LastErrorDateTime** carry the most recent failure and its timestamp.	Investigate the physical link or endpoint — **not** the configuration. The config was already proven by the period of live communication.

Because all three present as "TotalPoints = 0," do not diagnose by that number alone. Cross-check against the affirmative driver-health signals below before deciding the channel is faulty.

Driver-health signals

Judge whether a channel's driver is actually alive by the affirmative signals it maintains, rather than by TotalPoints or SuccessAmount — both of which are legitimately zero for a healthy server/slave-role channel with no master connected, or for a master channel that has not yet completed a poll.

Signal	Healthy reading	Meaning
IsRunning	1	The channel's protocol driver thread is active and ready to handle requests. 0 means the channel has not started, has been stopped, or failed to start.
ProcessPID	Non-zero	The OS process ID of the process hosting the channel. The runtime clears it to 0 when the channel stops, so a non-zero PID confirms a live host process. Pairs with **ProcessName**.
WatchDog	Advancing	While the driver's execution loop is running, the runtime increments this counter on roughly a one-second cadence. A value that keeps advancing on successive reads is a positive heartbeat; a value frozen across several seconds indicates the loop has stalled. Because the watchdog ticks independently of field traffic, it is a reliable "alive" signal even for a passive server/slave channel that shows no SuccessAmount.

Give the driver a settle window before judging it dead. These signals populate during the channel's after-start phase, which runs *after* the runtime reports the channel started — IsRunning and ProcessPID are set, and the watchdog begins ticking, slightly behind the initial start. The synchronous `Start(timeoutInSeconds)` call exists precisely for this reason: it polls IsRunning until the driver comes up or the timeout elapses. When checking from a script or dashboard, allow a short settle window after start before concluding a channel is down; re-read WatchDog twice a couple of seconds apart and confirm it advanced.

In this section...
