Overview

› Relevant source files

This document provides a high-level introduction to IndigoSCADA, a comprehensive industrial SCADA (Supervisory Control and Data Acquisition) platform. It covers the system's purpose, key capabilities, architectural components, and deployment requirements. For detailed information about specific subsystems, refer to the specialized pages: system architecture (2), protocol drivers (3), data acquisition (4), HMI (5), configuration (6), and build system (7).

What is IndigoSCADA?

IndigoSCADA is an open-source SCADA platform designed for industrial monitoring and control applications. The system acquires data from field devices using industry-standard protocols, processes alarm conditions, stores historical data, and provides operator interfaces for visualization and control. It runs as a multi-process distributed system with process isolation for robustness.

Sources: `src/LICENSE.txt` 1-45 `src/README.txt` 1-30

Key Capabilities

IndigoSCADA provides the following core capabilities:

Capability	Description
Protocol Support	Native drivers for IEC 60870-5-104, Modbus RTU/TCP, DNP3, OPC DA/AE/HDA, OPC UA, MQTT, IEC 61850, and others
Data Acquisition	Real-time data collection from multiple field devices with configurable sample points
Alarm Management	Threshold-based alarming with upper/lower warning and alarm limits, alarm groups, and acknowledgment
Historical Storage	Time-series data archiving with efficient compression and retrieval
HMI	Qt-based graphical interface with custom SCADA widgets (LEDs, meters, plots, tanks, switches)
Calculated Values	Embedded C interpreter (EiC) for executing custom calculations on acquired data
OPC Server	OPC DA 2.0 server for exporting data to external OPC clients
Multi-Line Support	Multiple instances of each protocol driver for parallel communication lines

Sources: `src/daq/daq.dsw` 1-227 `src/CHANGES.txt` 1-167

Supported Industrial Protocols

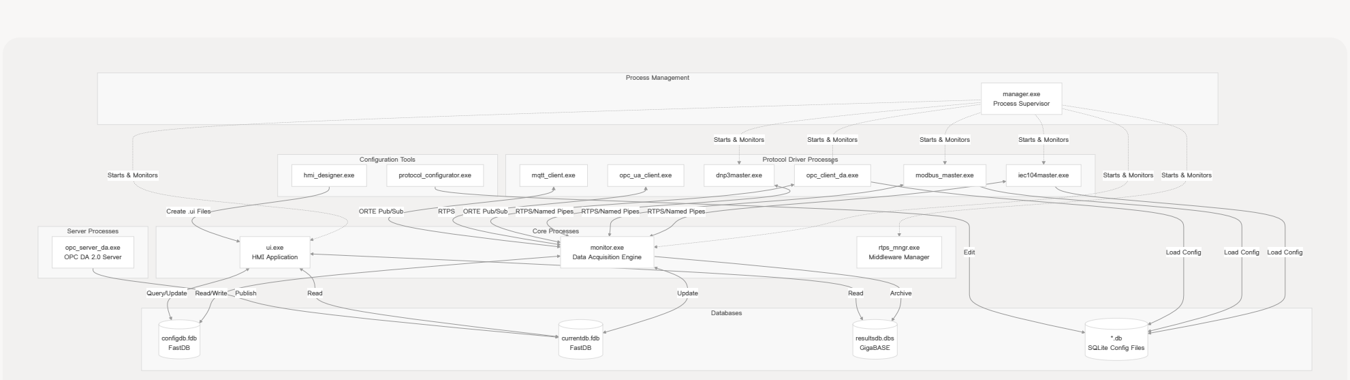
The following table lists all protocol drivers available in IndigoSCADA:

Protocol	Driver Executable	Driver DLL	Description
IEC 60870-5-104	`iec104master.exe`	`iec104driver.dll`	IEC 104 telecontrol protocol (TCP/IP)
IEC 60870-5-101	`iec101master.exe`	`iec101driver.dll`	IEC 101 telecontrol protocol (serial)
IEC 60870-5-103	`iec103master.exe`	`iec103driver.dll`	IEC 103 protection equipment protocol
Modbus	`modbus_master.exe`, `modbus_slave.exe`	`modbus_driver.dll`	Modbus RTU/TCP master and slave
DNP3	`dnp3master.exe`, `dnp3slave.exe`	`dnp3driver.dll`	DNP3 distributed network protocol
OPC DA	`opc_client_da.exe`	`opc_client_da_driver.dll`	OPC Data Access 2.0 client
OPC AE	`opc_client_ae.exe`	`opc_client_ae_driver.dll`	OPC Alarms & Events 1.10 client
OPC HDA	`opc_client_hda.exe`	`opc_client_hda_driver.dll`	OPC Historical Data Access 1.20 client
OPC UA	`opc_ua_client.exe`	`opcua_driver.dll`	OPC Unified Architecture client
MQTT	`mqtt_client.exe`, `mqtt_client_publisher.exe`	`mqtt_client_driver.dll`	MQTT subscriber and publisher
IEC 61850	`iec61850client.exe`, `iec61850server.exe`	`iec61850driver.dll`	IEC 61850 substation automation
RFC1006	`rfc1006client.exe`	`rfc1006driver.dll`	ISO transport over TCP
DLMS	`dlms_client.exe`, `dlms_server.exe`	N/A	DLMS/COSEM for smart metering

Sources: `src/daq/daq.dsw` 105-227 `src/CHANGES.txt` 14-93

System Architecture Overview

The following diagram shows the main executable processes and their relationships:


Process Management
manager.exe
Process Supervisor
Starts & Monitors
Starts & Monitors
Starts & Monitors
Starts & Monitors
Starts & Monitors
Starts & Monitors
Starts & Monitors
Configuration Tools
hmi_designer.exe
protocol_configurator.exe
Protocol Driver Processes
mqtt_client.exe
opc_ua_client.exe
dnp3master.exe
opc_client_da.exe
modbus_master.exe
iec104master.exe
Create .ui Files
ORTE Pub/Sub
RTPS
ORTE Pub/Sub
RTPS/Named Pipes
RTPS/Named Pipes
RTPS/Named Pipes
Server Processes
opc_server_da.exe
OPC DA 2.0 Server
ui.exe
HMI Application
Core Processes
monitor.exe
Data Acquisition Engine
rtps_mngr.exe
Middleware Manager
Edit
Load Config
Load Config
Load Config
Query/Update
Read/Write
Publish
Read
Update
Read
Archive
Databases
configdb.fdb
FastDB
currentdb.fdb
FastDB
resultsdb.dbs
GigaBASE
*.db
SQLite Config Files

Sources: `src/daq/daq.dsw` 51-227 `src/CHANGES.txt` 86-167

Core Components

manager.exe - Process Manager

The `manager.exe` process acts as the system supervisor, responsible for starting, monitoring, and restarting all other IndigoSCADA processes based on configuration in `manager.ini`.

Sources: `src/daq/daq.dsw` 87-88

monitor.exe - Data Acquisition Engine

The `monitor.exe` process is the central data acquisition engine. It coordinates all protocol drivers, processes incoming data through the `Results` class, evaluates alarm conditions, and files historical data. The `Results` class maintains the `EnabledPoints` dictionary of active sample points with their current values and alarm states.

Sources: `src/daq/daq.dsw` 63-64 `src/database/results.h` 396-457

ui.exe - Human-Machine Interface

The `ui.exe` process provides the operator interface built with Qt 3.3.8 and the qwt 4.2.0 widget library. It displays real-time data, alarm lists, trend plots, and allows operators to issue control commands.

Sources: `src/daq/daq.dsw` 75-76 `src/hmi_designer/designer/app/designer.dsp` 1-117

Protocol Drivers

Each protocol driver follows a common architecture pattern:

- **DriverInstance** class: Configuration and high-level coordination
- **DriverThread** class: Optional child process management
- **Protocol implementation**: Native protocol handling (e.g., `modbus_imp`, `iec103_imp`, `dnp3_imp`)
- **SQLite database**: Driver-specific configuration (e.g., `modbus_databaseN.db`)

Sources: `src/drivers/modbus/protocol/modbus_master.dsp` 1-147 `src/drivers/iec103/protocol/iec103master.dsp` 1-137

Middleware Communication

Inter-process communication uses multiple mechanisms:

- **RTPS (Real-Time Publish-Subscribe)**: Managed by `rtps_mngr.exe` process
- **ORTE Domain**: Shared memory pub/sub
- **Named Pipes**: Parent-child process communication
- **FIFO Buffers**: Local buffering within drivers

The `iec_item` structure serves as the universal data format exchanged between processes.

Sources: `src/middleware/rtps/manager/mcvs/rtps_mngr.dsp` 1-108 `src/CHANGES.txt` 96-103

Database Architecture

IndigoSCADA uses three database engines optimized for different workloads:

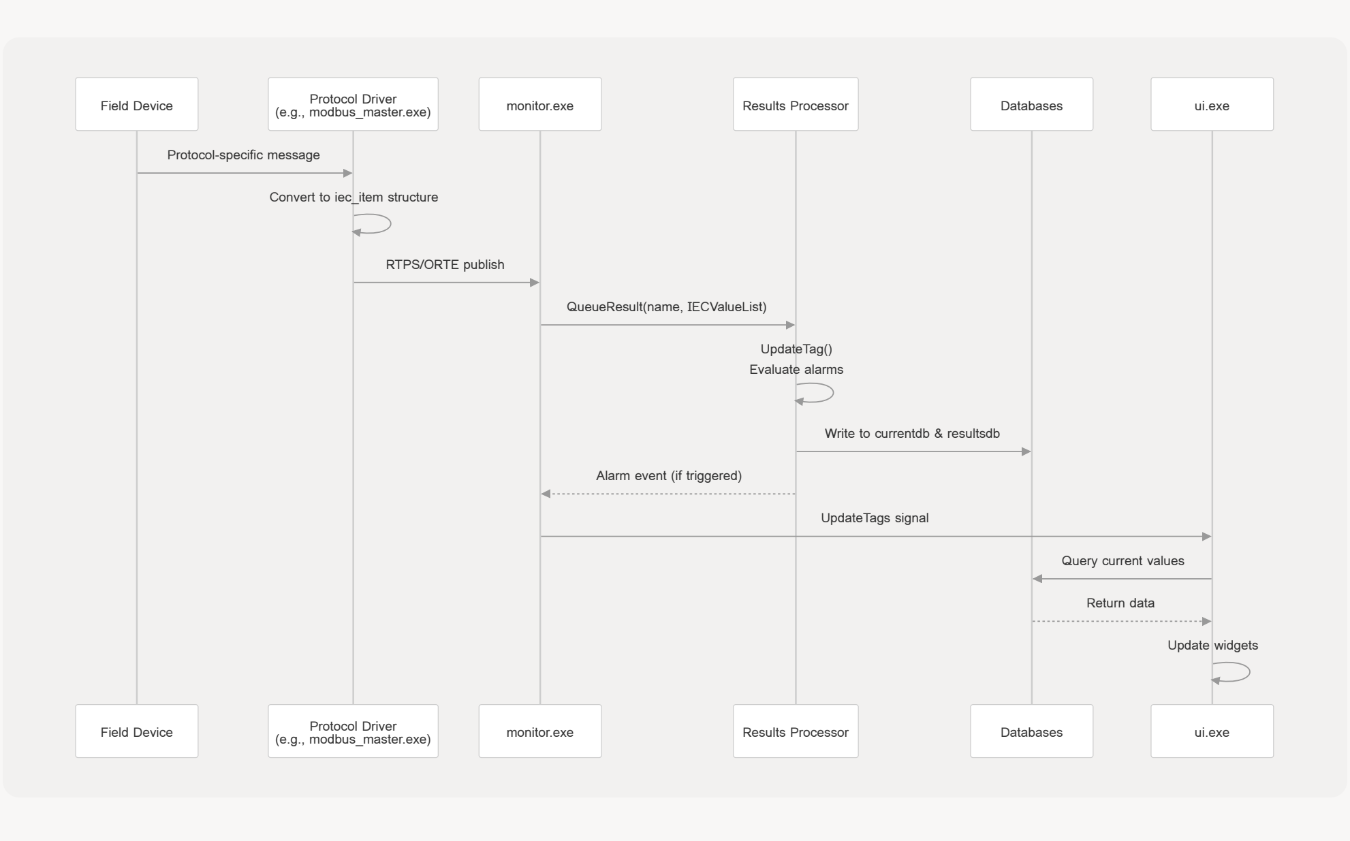
Database	Engine	Purpose	Tables
`configdb.fdb`	FastDB 3.56	System configuration	`UNITS`, `SAMPLE`, `TAGS`, `PROPS`, `USERS`, `SCHEDULE`
`currentdb.fdb`	FastDB 3.56	Real-time values	`TAGS_DB`, `CVAL_DB`
`resultsdb.dbs`	GigaBASE 3.37	Historical data	`SamplePointN_results` (one per sample point)
`*_databaseN.db`	SQLite 3	Protocol configs	Protocol-specific item mappings

The `TAGS` table contains the `IOA` (Information Object Address) field that links sample points to protocol driver configurations.

Sources: `src/database/results.h` 1-460 `src/CHANGES.txt` 50-73

Data Flow

The following sequence illustrates how data flows from a field device to the HMI:


Field Device
Protocol Driver
(e.g., modbus_master.exe)
monitor.exe
Results Processor
Databases
ui.exe
Protocol-specific message
Convert to iec_item structure
RTPS/ORTE publish
QueueResult(name, IECValueList)
UpdateTag()
Evaluate alarms
Write to currentdb & resultsdb
Alarm event (if triggered)
UpdateTags signal
Query current values
Return data
Update widgets
Field Device
Protocol Driver
(e.g., modbus_master.exe)
monitor.exe
Results Processor
Databases
ui.exe

Sources: `src/database/results.h` 440-441 `src/database/results.h` 389

Sample Point and Tag Configuration

The system organizes data points hierarchically:

- **Sample Point**: A logical collection of related tags (defined in `SAMPLE` table)
- **Tag**: An individual measurement or control point (defined in `TAGS` table)
- **IOA (Information Object Address)**: Links tags to protocol driver items

The `SamplePoint` class maintains a `Tags` dictionary of `TagItem` objects, each containing the current value, alarm limits, and statistics.

Sources: `src/database/results.h` 313-390 `src/database/results.h` 294-311

System History and Platform Support

IndigoSCADA originated in 2000 as a Linux-based system. In 2001, it was ported to Windows 2000 Professional, and since then development has focused primarily on Windows. The codebase contains extensive `#ifdef WIN32` conditionals, and porting back to Linux would require significant effort to address platform-specific code.

Sources: `src/HISTORY.txt` 1-13 `src/CHANGES.txt` 29

Build Requirements and Dependencies

Windows Build Environment

- **IDE**: Microsoft Visual C++ 6.0 or later
- **Workspace**: `src/daq/daq.dsw`
- **Qt Framework**: Qt 3.3.8 for Windows (non-commercial)
- **Location**: Source installed at `C:\scada\`, binaries output to `C:\scada\bin\`

Third-Party Libraries

Library	Version	Purpose
Qt	3.3.8	GUI framework and core utilities
FastDB	3.56	In-memory database for config and real-time data
GigaBASE	3.37	Time-series historical database
EiC	4.4.2	Embedded C interpreter for calculated driver
qwt	4.2.0	SCADA widget library
SQLite	3.x	Protocol configuration storage
libmodbus	N/A	Modbus protocol implementation
pthreads-w32	2.8.0	POSIX threads for Windows

Sources: `src/README.txt` 1-30 `src/CREDITS.txt` 1-7 `src/CHANGES.txt` 133-135

Installation Directory Structure

The standard installation layout is:

```
C:\scada\
├── bin\              # Executables and DLLs
├── project\          # Project-specific files (.ini, .db, .fdb, .dbs, .ui)
├── init_db\          # Database initialization SQL scripts
├── lib\              # Static libraries for development
├── src\              # Source code (optional)
└── Help\             # Documentation
```

The `project.ini` file in `bin\` directory configures the project directory location, defaulting to `C:\scada\project\`.

Sources: `src/USAGE.txt` 1-24 `src/common/common.cpp` 72-149

License

IndigoSCADA is licensed under the GNU Lesser General Public License v3 (LGPL v3). It is free software distributed without warranty. The software includes third-party libraries, each subject to their respective license terms.

Sources: `src/LICENSE.txt` 1-48 `src/USAGE.txt` 12-13

Getting Started

To begin using IndigoSCADA:

1. Install the binary distribution to `C:\` drive (creates `C:\scada\` directory)
2. Launch `C:\scada\bin\manager.exe` to start all system processes
3. Configure protocol drivers using `protocol_configurator.exe`
4. Define sample points and tags through the UI configuration dialogs
5. Design custom HMI screens using `hmi_designer.exe`

For detailed configuration procedures, see Configuration and Setup. For protocol-specific setup, see individual driver pages under Protocol Drivers.

Sources: `src/USAGE.txt` 1-24

System History and Evolution

› Relevant source files

Purpose and Scope

This document chronicles the development history and architectural evolution of IndigoSCADA from its origins in 2000 to present day. It covers major milestones, protocol driver additions, middleware transitions, database schema changes, and the shift from Linux to Windows as the primary platform. For information about the current system architecture and components, see System Architecture. For build system details and platform-specific information, see Platform Support and Porting.

Origins and Initial Development (2000-2001)

IndigoSCADA originated as a Linux-based SCADA system in the year 2000. The initial development targeted Linux operating systems, with the codebase structured around Unix conventions and POSIX APIs. Evidence of this early Linux heritage can be found in generated UI files dated from this period, such as `src/drivers/iec101/iec101driverCommandData.cpp 4` which shows a generation timestamp of `Thu Jan 4 16:13:32 2001`.

In 2001, the system underwent a critical platform migration, being ported to Windows 2000 Professional. This transition was documented in `src/HISTORY.txt 1-3` :

> *The codebase origin goes back to 2000, when it was developed under Linux OS. Then in 2001 it was ported to Window 2000 Pro.*

This migration decision would have lasting implications for the system's architecture and maintenance trajectory.

Sources: `src/HISTORY.txt 1-11` `src/drivers/iec101/iec101driverCommandData.cpp 1-10`

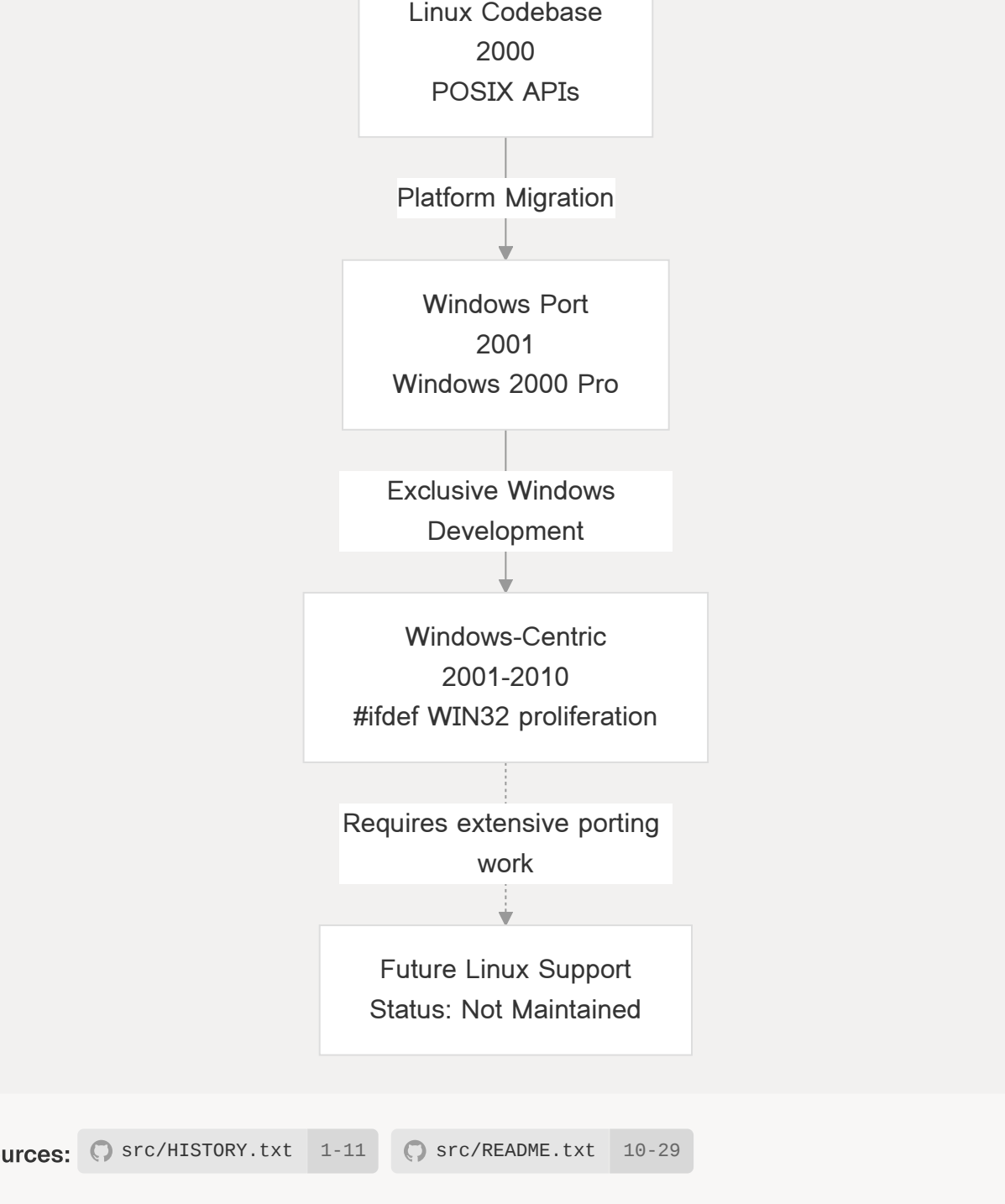
The Windows-Centric Era (2001-2010)

Following the 2001 port to Windows, IndigoSCADA evolved exclusively on the Windows platform. According to `src/HISTORY.txt 4-5` "Since 2001 we had no time and requests to port the codebase back to Linux again." During this period, the codebase accumulated extensive Windows-specific code through the use of `#ifdef WIN32` preprocessor directives throughout the source tree.

The system standardized on the following Windows-based development stack:

- **Visual Studio C++ 6.0** as the primary compiler (`src/README.txt 14`)
- **Qt 3.3.8 for Windows** for the GUI framework (`src/README.txt 19`)
- **Windows-specific IPC mechanisms** including named pipes

By the end of this period, the codebase required substantial `#ifdef WIN32` or `#ifdef UNIX` conditionals to enable Linux compilation, as noted in `src/HISTORY.txt 7`



Sources: `src/HISTORY.txt 1-11` `src/README.txt 10-29`

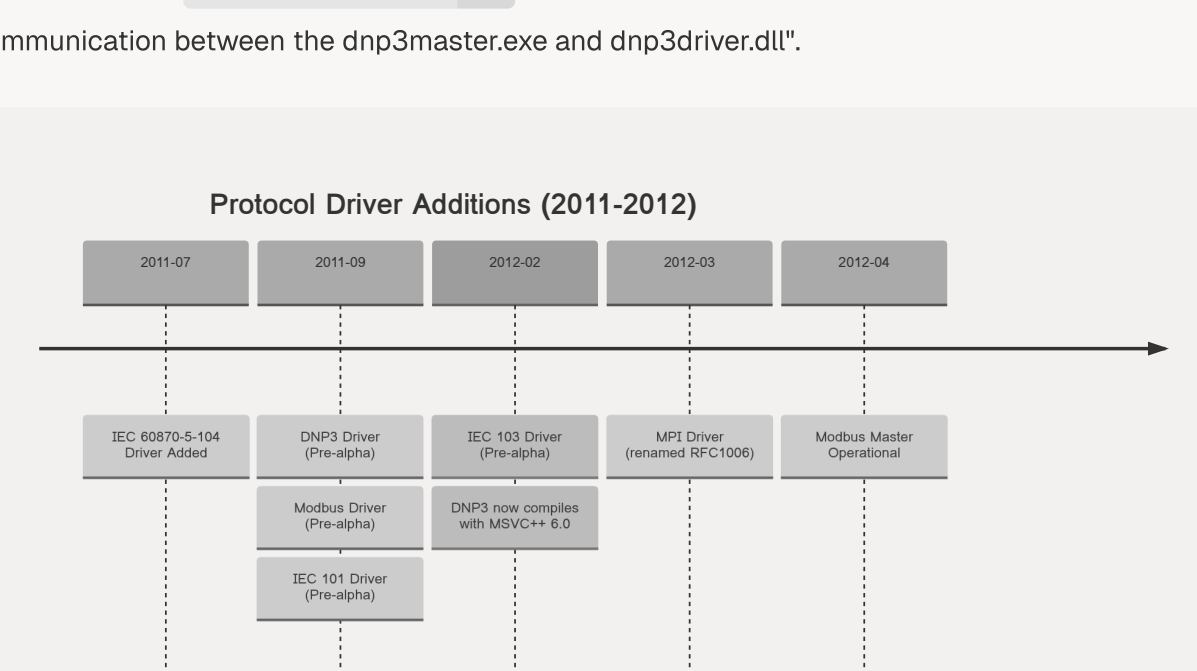
Major Protocol Expansion (2011-2012)

The years 2011-2012 marked a significant expansion in protocol support, with multiple industrial protocol drivers added to the system. This rapid growth transformed IndigoSCADA from a basic SCADA platform into a comprehensive multi-protocol system.

Protocol Driver Timeline

Date	Protocol Driver	Status	Source
30-07-2011	IEC 60870-5-104	Protocol implementation pending	`src/CHANGES.txt 14`
06-09-2011	DNP3	Pre-alpha state	`src/CHANGES.txt 23`
06-09-2011	Modbus	Pre-alpha state	`src/CHANGES.txt 24`
17-09-2011	IEC 101	Pre-alpha state	`src/CHANGES.txt 32`
05-02-2012	IEC 103	Pre-alpha state	`src/CHANGES.txt 38`
12-03-2012	MPI (RFC1006)	Initial addition	`src/CHANGES.txt 77`
14-03-2012	RFC1006	MPI driver renamed	`src/CHANGES.txt 83`
21-04-2012	Modbus Master	Operational (needs RTU testing)	`src/CHANGES.txt 89`

During this period, the protocol drivers followed a common pattern where protocol-specific executables (`xxx_master.exe`) were paired with driver DLLs (`xxx_driver.dll`). Inter-process communication between these components was initially identified as missing and became a focus area, as noted in `src/CHANGES.txt 47` for DNP3: "It is still missing the Inter Process Communication between the dnp3master.exe and dnp3driver.dll".



Sources: `src/CHANGES.txt 14-93`

Middleware Architecture Evolution (2012-2014)

A fundamental architectural shift occurred during 2012-2014 with the introduction and evolution of middleware components for inter-process communication.

RTPS Middleware Introduction (May 2012)

On May 2, 2012, Real Time Publisher Subscriber (RTPS) was introduced as the primary middleware protocol for inter-process communication between SCADA processes (`src/CHANGES.txt 96`):

> *Added Real Time Publisher Subscriber (RTPS) as middleware protocol for inter process communication between SCADA processes*

This addition addressed the IPC gaps that had been noted in earlier protocol driver implementations. By May 30, 2012, RTPS was integrated with protocol processes (`src/CHANGES.txt 102`).

Protocol Configurator and SQLite Integration (May 2012)

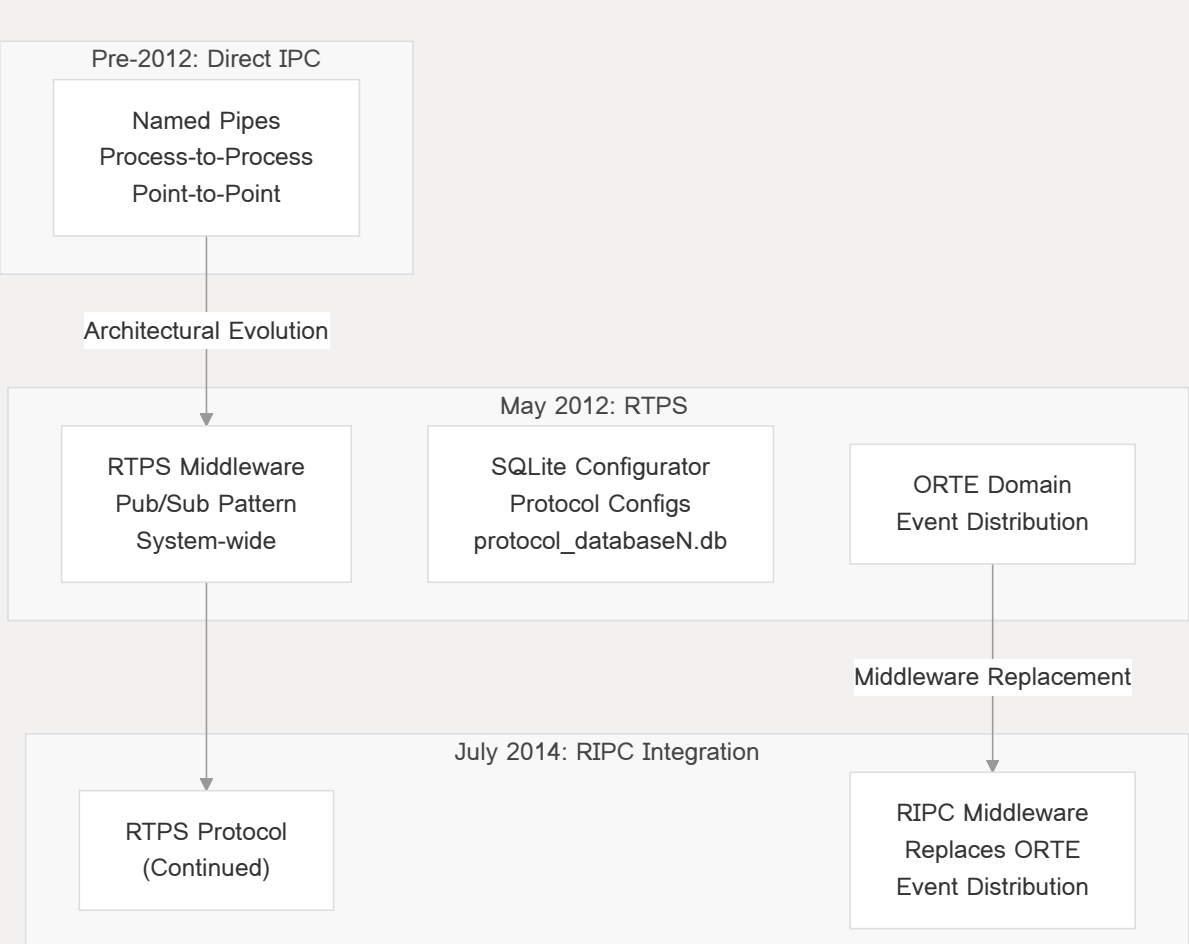
Simultaneously with RTPS, a protocol configurator based on SQLite 3 database files was introduced (`src/CHANGES.txt 99`). This allowed protocol-specific configurations to be stored separately from the main FastDB configuration database. The configurator was implemented for OPC clients by June 4, 2012 (`src/CHANGES.txt 103`).

ORTE to RIPC Transition (2014)

A significant middleware change occurred on July 6, 2014, when ORTE (Open Real-Time Ethernet) was removed and RIPC (Real-time IPC) was integrated (`src/CHANGES.txt 164`):

> *Removed ORTE and integration of RIPC*

This transition represents a shift in the underlying middleware technology while maintaining the publish-subscribe communication patterns established with RTPS.



Sources: `src/CHANGES.txt 96-164`

Database Schema Evolution

The configuration database schema underwent significant changes during the 2011-2012 development period to support the expanding protocol driver ecosystem.

TAGS Table Evolution

The `TAGS` table in the FastDB configuration database (`configdb.fdb`) received critical additions:

February 21, 2012: UNIT Column Addition

A new `UNIT` column was added to associate tags with specific driver instances (`src/CHANGES.txt 50-73`):

```
create table TAGS
(
NAME string,
TAG string,
UPPERALARM real8,
UPPERWARN real8,
LOWERWARN real8,
LOWERALARM real8,
UAENABLE int4,
UWENABLE int4,
LWENABLE int4,
LAENABLE int4,
RECEIPE string,
ENABLED int4,
PARAMS string,
UNIT string  -- ADDED: Links tag to driver instance
);
```

This change required database recreation using the updated schema from `C`

June 4, 2012: IOA Column Addition

The Information Object Address (IOA) column was added to support protocol addressing (`src/CHANGES.txt 107-125`):

```
create table TAGS
(
NAME string,
TAG string,
UPPERALARM real8,
UPPERWARN real8,
LOWERWARN real8,
LOWERALARM real8,
UAENABLE int4,
UWENABLE int4,
LWENABLE int4,
LAENABLE int4,
RECEIPE string,
ENABLED int4,
IOA int4,      -- ADDED: Information Object Address
PARAMS string,
UNIT string
);
```

This IOA column became central to mapping between protocol-specific addresses and IndigoSCADA's internal tag system, enabling the linkage described in `src/CHANGES.txt 128` : "OPC client DA now is working with multiple dedicated lines".

Multi-Database Architecture

By this period, the system had established its three-database architecture:

- **FastDB 3.56** for `configdb.fdb` (configuration) and `currentdb.fdb` (real-time values)
- **GigaBASE 3.37** for `resultsdb.dbs` (historical data)
- **SQLite 3** for protocol-specific configurations (`protocol_databaseN.db`)

Sources: `src/CHANGES.txt 50-128` `src/README.txt 3-8`

HMI System Development (2012-2013)

The Human-Machine Interface underwent significant development during 2012-2013, transforming from basic displays to a comprehensive HMI designer system.

HMI Designer and Widget System (July 2012)

On July 15, 2012, fundamental HMI infrastructure was added (`src/CHANGES.txt 133-139`):

- **pthreads-w32-2-8-0** for Windows threading support
- **qwt-4.2.0** for scientific widgets and plotting
- Initial HMI designer components

By July 21, 2012, the complete `HMI_designer.exe` codebase was added (`src/CHANGES.txt 142`). This designer tool, based on Qt Designer, allowed operators to create custom SCADA screens with industrial widgets.

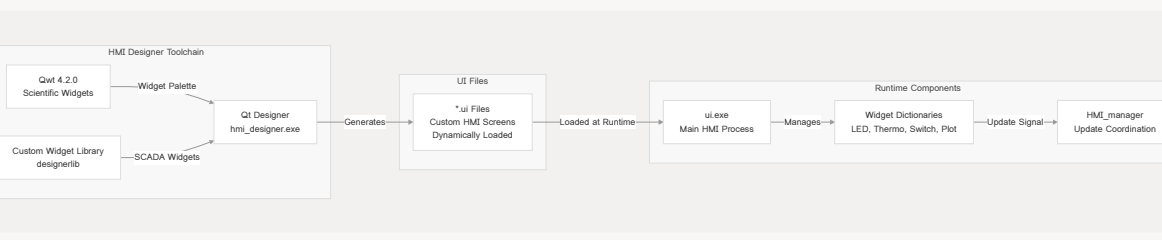
SCADA Widget Expansion (August 2012)

On August 19, 2012, additional SCADA widgets and popup window capabilities were added (`src/CHANGES.txt 145-146`):

- Extended widget library (LEDs, thermometers, tanks, switches, plots, meters)
- Popup windows for detailed widget inspection
- Right-click context menus

Multiple HMI Windows (November 2012)

Support for multiple simultaneous HMI windows was added on November 26, 2012 (`src/CHANGES.txt 149`), enabling operators to view multiple process areas concurrently.



Sources: `src/CHANGES.txt 133-149`

OPC Client Maturation (2011-2012)

The OPC client implementations evolved from limitations to multi-server support during 2011-2012.

Initial OPC DA Client Limitations (March 2011)

In March 2011, significant limitations were documented in the OPC DA client's server browsing feature (`src/CHANGES.txt 4-11`):

- Browsing limited to 40,000 ItemIDs
- Required manual subset selection
- Needed external configurator for `.xml` or `.xls` import

Multi-Line OPC DA Support (June 2012)

By June 4, 2012, OPC client DA achieved "working with multiple dedicated lines" status (`src/CHANGES.txt 127`), allowing simultaneous connections to multiple OPC servers, each configured as a separate "line" with its own configuration database.

Class ID Field Addition (February 2012)

Configuration enhancements included the addition of "Class ID" fields to OPC DA, OPC AE, and OPC HDA configuration windows (`src/CHANGES.txt 42`), enabling more precise OPC server identification.

Sources: `src/CHANGES.txt 1-127`

Protocol Driver Maturation

Modbus Driver Beta Release (March 2013)

The Modbus driver reached beta-level maturity on March 16, 2013 (`src/CHANGES.txt 155`), representing the completion of a 1.5-year development cycle from its initial pre-alpha state in September 2011.

DNP3 IPC Integration (June 2012)

The DNP3 driver achieved monitor direction connectivity on June 12, 2012 (`src/CHANGES.txt 130`): "dnp3master.exe is now connected with dnp3driver.dll through RTPS in monitoring direction", demonstrating successful RTPS middleware integration.

Sources: `src/CHANGES.txt 130-155`

Project Structure Reorganization (2013)

On October 21, 2013, a major reorganization created a dedicated project directory (`src/CHANGES.txt 161`):

> *Created project directory /scada/project with all project files: .ini .db .fdb .dbs .ui*

This reorganization separated runtime configuration and project files from the binary installation, establishing the directory structure that persists today:

- `C:\scada\bin\` - Executables and DLLs
- `C:\scada\project\` - Configuration files, databases, UI files
- `C:\scada\init_db\` - Database initialization scripts
- `C:\scada\Help\` - Documentation

Sources: `src/CHANGES.txt 161`

Gateway Functionality (2018)

The most recent major feature addition occurred on April 17, 2018, when gateway functionality was added (`src/CHANGES.txt 166-167`). This feature enables IndigoSCADA to act as a protocol gateway, bridging between different industrial protocols (e.g., Modbus to OPC DA, or IEC 104 to OPC UA).

Sources: `src/CHANGES.txt 166-167`

Current State and Outstanding Items

As of the last documented update (August 26, 2014), several items remained in the TODO list (`src/TODO.txt 1-14`):

Item	Description	Status
Dynamic Alarm Lists	Current alarm list is static	Not Implemented
Calculated Driver	Softlogic improvements needed	Ongoing
Linux Porting	Port to current Linux distributions	Not Maintained
Database Upgrade	Upgrade FastDB/GigaBASE versions	Pending
Documentation	Update system documentation	Ongoing
Internationalization	Translate user interface	Contribution Needed

Linux Portability Status

The current state of Linux support is explicitly documented in `src/HISTORY.txt 9` : "I am looking for volunteers interested in porting it back to Linux." The extensive Windows-specific code accumulated since 2001 presents a significant porting challenge.

Sources: `src/TODO.txt 1-14` `src/HISTORY.txt 1-11`

Licensing and Copyright Evolution

IndigoSCADA has maintained consistent licensing since at least 2002:

- **Copyright:** 2002-2014 Enscada Limited (`src/LICENSE.txt 3`)
- **License:** GNU Lesser General Public License v3 (`src/LICENSE.txt 15`)
- **Status:** Free software with NO WARRANTY (`src/LICENSE.txt 30-40`)

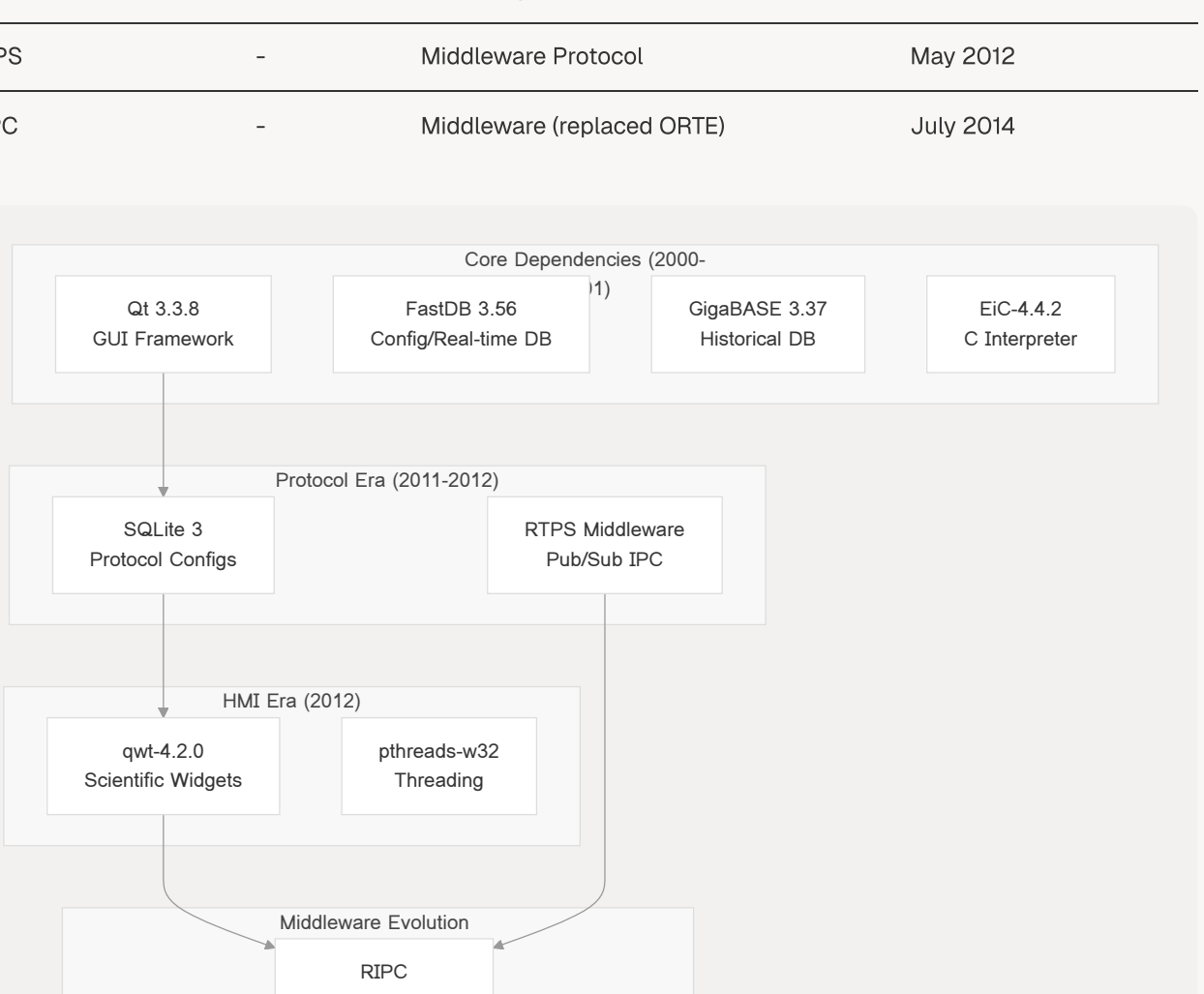
The system includes third-party libraries (EiC, FastDB, GigaBASE, RIPC) that maintain their respective licenses (`src/CREDITS.txt 3-4`).

Sources: `src/LICENSE.txt 1-886` `src/CREDITS.txt 1-7`

Technology Stack Evolution

The following table summarizes the major technology dependencies and their versions:

Component	Version	Purpose	Added/Version
Qt	3.3.8	GUI Framework	Initial (2000)
FastDB	3.56	Configuration/Real-time DB	Initial
GigaBASE	3.37	Historical Database	Initial
EiC	4.4.2	C Interpreter (Calculated Driver)	Initial
Visual Studio C++	6.0+	Compiler	2001 (Windows port)
SQLite	3.x	Protocol Configuration	May 2012
qwt	4.2.0	Scientific Widgets	July 2012
pthreads-w32	2-8-0	Threading (Windows)	July 2012
RTPS	-	Middleware Protocol	May 2012
RIPC	-	Middleware (replaced ORTE)	July 2014



Sources: `src/README.txt 3-29` `src/CHANGES.txt 96-164`

Visual Studio Build System

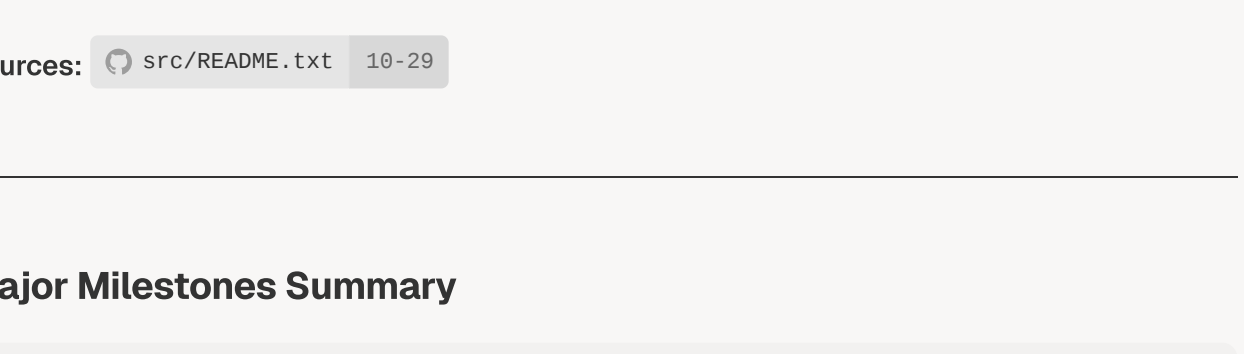
The build system has remained consistent since the Windows port, using Visual Studio C++ 6.0 workspace files:

- **Workspace:** `daq.dsw` (`src/README.txt 14`)
- **Project Files:** `.dsp` format for individual components
- **Build Target:** "all" project compiles entire system (`src/README.txt 15-16`)

The workspace is located at `C:\scada\src\daq` directory and includes all protocol drivers, monitor process, UI components, and utility tools.

Sources: `src/README.txt 10-29`

Major Milestones Summary



This timeline represents approximately 18 years of development from initial Linux origins to a comprehensive Windows-based multi-protocol SCADA platform with advanced HMI capabilities, flexible middleware, and gateway functions.

Sources: src/HISTORY.txt 1-11 src/CHANGES.txt 1-167 src/TODO.txt 1-14 src/LICENSE.txt 1-886

Key Components

> Relevant source files

This page provides an overview of the major components that comprise the IndigoSCADA system, including their responsibilities, relationships, and key code entities. For detailed architectural patterns and design principles, see System Architecture. For specific protocol driver implementations, see Protocol Drivers.

System Overview

IndigoSCADA is organized as a multi-process SCADA platform where each major function runs as a separate executable. The system follows a layered architecture with process isolation, centralized data acquisition, middleware-based communication, and a multi-database storage strategy.

Component Categories

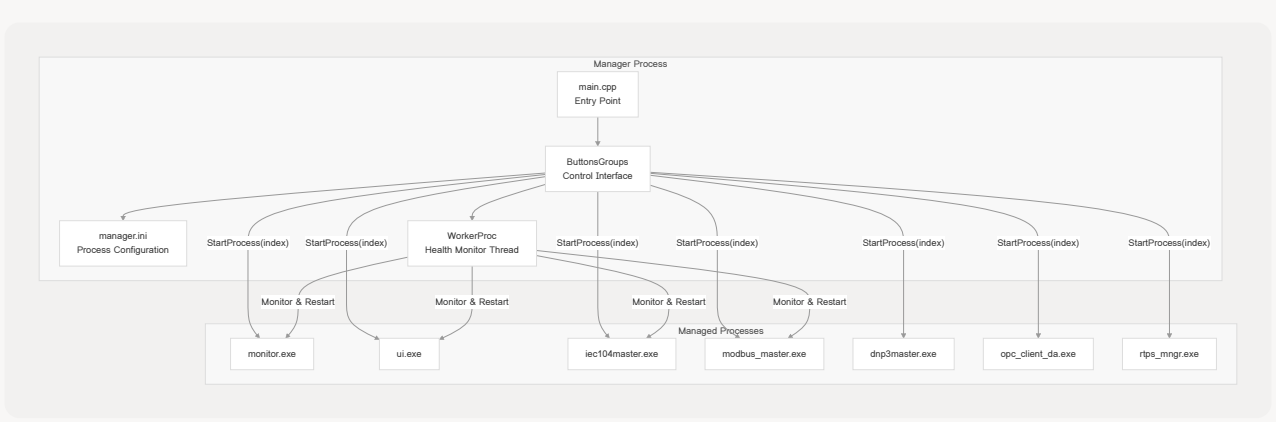
The system components are organized into the following functional categories:

Category	Components	Purpose
Process Management	`manager.exe`	Orchestrates startup, monitoring, and restart of all processes
Data Acquisition	`monitor.exe`, `Calculated` driver	Central data collection, routing, and script execution
Protocol Drivers	`iec104master.exe`, `modbus_master.exe`, `dnp3master.exe`, `opc_client_da.exe`, `opc_ua_client.exe`, `mqtt_client.exe`, etc.	Communicate with field devices using native protocols
Middleware	`rtps_mngr.exe`, ORTE domain, Named Pipes	Inter-process communication and data distribution
Databases	FastDB, GigaBASE, SQLite	Configuration, real-time values, historical data, protocol configs
User Interface	`ui.exe`, `hmi_designer.exe`	Operator interface and screen design tools
Configuration	`protocol_configurator.exe`, configuration dialogs	System setup and administration
Data Processing	Results processor, Alarm engine	Alarm evaluation and historical data filing
Data Export	`opc_server_da.exe`	Expose data to external OPC clients

Sources: `src/daq/daq.dsw` 1-1046

Core Processes

Process Manager (manager.exe)


Manager Process
main.cpp
Entry Point
ButtonsGroups
Control Interface
manager.ini
Process Configuration
StartProcess(index)
WorkerProc
Health Monitor Thread
Monitor & Restart
Managed Processes
monitor.exe
ui.exe
iec104master.exe
modbus_master.exe
dnp3master.exe
opc_client_da.exe
rtps_mngr.exe

The `manager.exe` process serves as the system supervisor, responsible for:

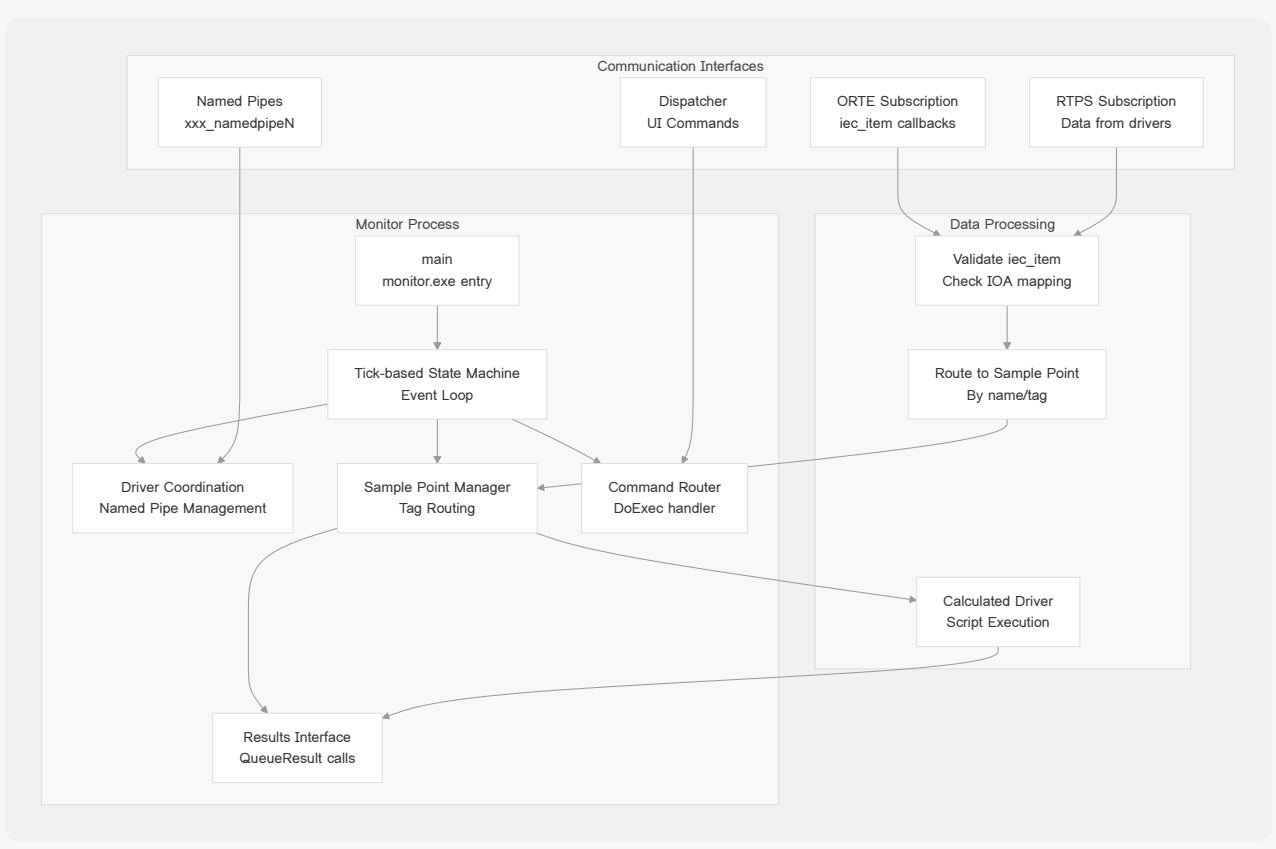
- **Process Orchestration**: Reads `manager.ini` from the project directory to determine which processes to start and their configuration
- **Startup Sequencing**: Calls `StartProcess(int nIndex)` for each configured process with appropriate pause intervals
- **Health Monitoring**: Runs `ProcessRestarter()` in a worker thread to periodically check process health using `GetExitCodeProcess()`
- **Automatic Restart**: If a process with `Restart=Yes` configuration exits, it is automatically restarted
- **User Control**: Provides GUI (`ButtonsGroups` class) or runs headless for starting/stopping the entire system

Key configuration parameters in `manager.ini`:

- `Process0..ProcessN`: Individual process definitions
- `CommandLine` or `Process`: Executable to launch
- `WorkingDir`: Working directory for the process
- `PauseStart`: Milliseconds to wait after starting
- `PauseEnd`: Milliseconds to wait before forcing termination
- `Restart`: Whether to automatically restart on failure
- `UserInterface`: Whether the process has a GUI (affects desktop assignment)
- `CheckProcessSeconds`: Health check interval

Sources: `src/manager/buttongroups.cpp` 1-624 `src/manager/buttongroups.h` 1-58 `src/manager/main.cpp` 1-99

Monitor Process (monitor.exe)


Communication Interfaces
Named Pipes
xxx_namedpipeN
Dispatcher
UI Commands
ORTE Subscription
iec_item callbacks
RTPS Subscription
Data from drivers
Monitor Process
main
monitor.exe entry
Tick-based State Machine
Event Loop
Driver Coordination
Named Pipe Management
Sample Point Manager
Tag Routing
Command Router
DoExec handler
Results Interface
QueueResult calls
Data Processing
Validate iec_item
Check IOA mapping
Route to Sample Point
By name/tag
Calculated Driver
Script Execution

The `monitor.exe` process is the central data acquisition and routing engine:

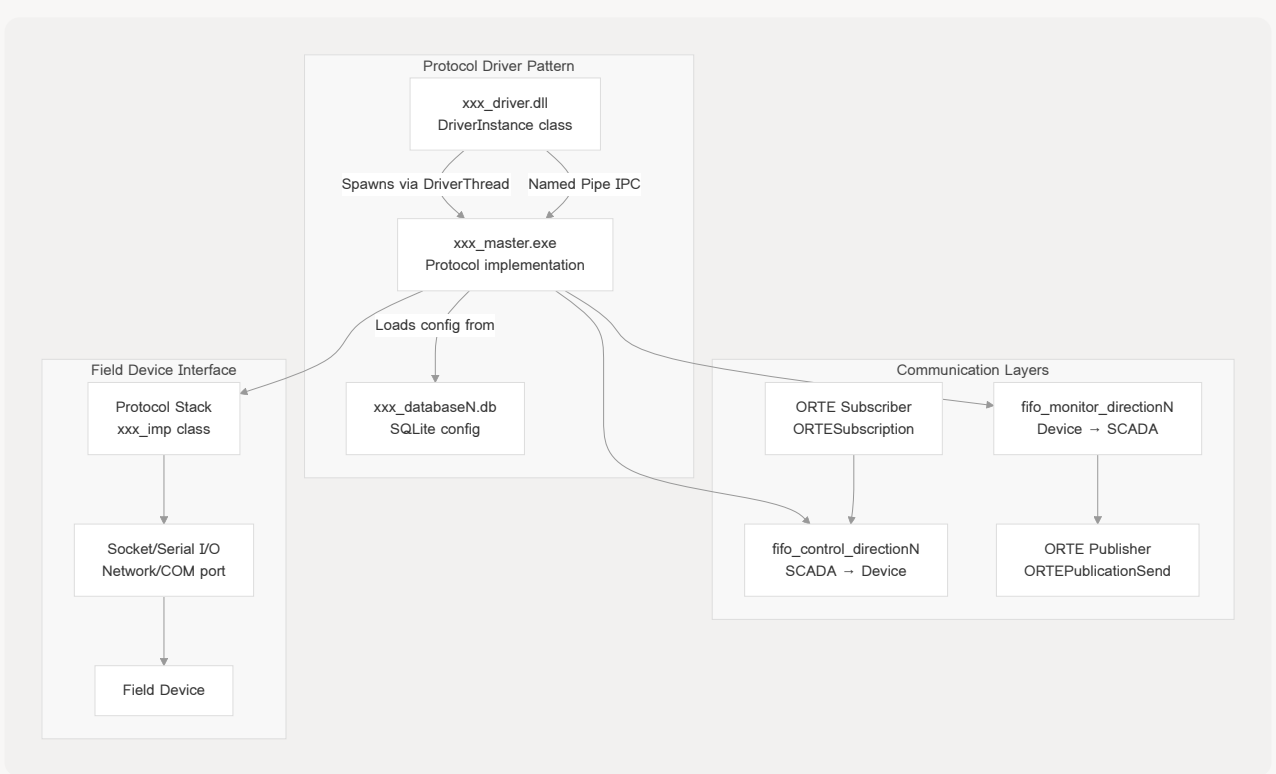
- **Driver Coordination**: Manages communication with all protocol driver child processes via named pipes
- **Data Collection**: Subscribes to ORTE/RTPS middleware to receive `iec_item` messages from drivers
- **Tag Routing**: Maps Information Object Addresses (IOAs) to sample points and routes data accordingly
- **Command Distribution**: Receives commands from UI via dispatcher and routes them to appropriate drivers
- **Sample Point Management**: Maintains current values and triggers results processing
- **Calculated Values**: Loads and executes the Calculated driver DLL for computed data points
- **Configuration Loading**: Reads from `configdb` (FastDB) to understand system topology

The monitor operates on a tick-based event loop, processing queued messages and maintaining real-time state.

Sources: `src/daq/daq.dsw` 639-659

Protocol Driver Processes

All protocol drivers follow a common architectural pattern with driver-specific implementations:


Protocol Driver Pattern
xxx_driver.dll
DriverInstance class
Spawns via DriverThread
Named Pipe IPC
xxx_master.exe
Protocol implementation
Loads config from
xxx_databaseN.db
SQLite config
Field Device Interface
Protocol Stack
xxx_imp class
Socket/Serial I/O
Network/COM port
Field Device
Communication Layers
ORTE Subscriber
ORTESubscription
fifo_monitor_directionN
Device → SCADA
fifo_control_directionN
SCADA → Device
ORTE Publisher
ORTEPublicationSend

Driver Components

Each protocol driver typically consists of three components:

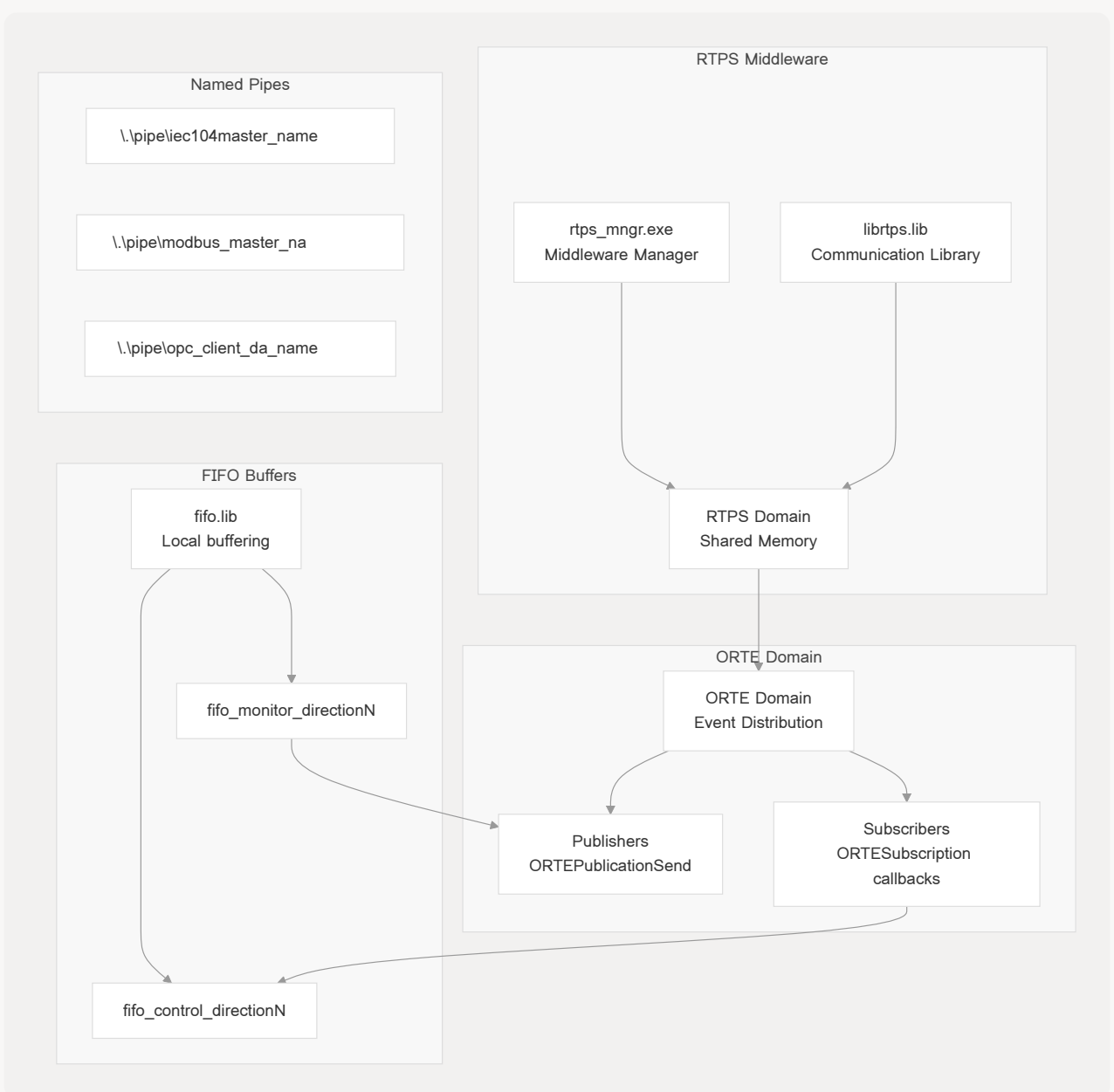
1. **Driver DLL** (`xxx_driver.dll`):
    - Contains `DriverInstance` class that interfaces with monitor
    - Manages driver lifecycle and configuration
    - Spawns child process if needed
2. **Master Executable** (`xxx_master.exe`):
    - Implements protocol-specific communication
    - Runs as child process (for IEC104, DNP3, Modbus, OPC clients)
    - Converts protocol data to/from `iec_item` structures
3. **SQLite Configuration** (`xxx_databaseN.db`):
    - Stores protocol-specific mappings (IOA to register/item)
    - Loaded at startup by both DLL and executable
    - Line number (N) distinguishes multiple instances

Available Protocol Drivers

Driver	Master Executable	Description
IEC 60870-5-104	`iec104master.exe`, `iec104slave.exe`	TCP-based SCADA protocol
Modbus	`modbus_master.exe`, `modbus_slave.exe`	Modbus TCP and RTU
DNP3	`dnp3master.exe`, `dnp3slave.exe`	Distributed Network Protocol 3
OPC DA	`opc_client_da.exe`	OPC Data Access 2.0 client
OPC AE	`opc_client_ae.exe`	OPC Alarms & Events client
OPC HDA	`opc_client_hda.exe`	OPC Historical Data Access client
OPC UA	`opc_ua_client.exe`, `opc_ua_server.exe`	OPC Unified Architecture
MQTT	`mqtt_client.exe`, `mqtt_client_publisher.exe`	MQTT publish/subscribe
IEC 60870-5-101	`iec101master.exe`, `iec101slave.exe`	Serial-based SCADA protocol
IEC 60870-5-103	`iec103master.exe`, `iec103slave.exe`	Protection device protocol
IEC 61850	`iec61850client.exe`, `iec61850server.exe`	Substation automation
RFC1006	`rfc1006client.exe`, `rfc1006_server.exe`	ISO transport over TCP
DLMS	`dlms_client.exe`, `dlms_server.exe`	Metering protocol

Sources: `src/daq/daq.dsw` 6-227 `src/drivers/modbus/protocol/modbus_master.dsp` 1-147 `src/drivers/iec103/protocol/iec103master.dsp` 1-137 `src/drivers/opc_client_hda/protocol/opc_client_hda.dsp` 1-133 `src/drivers/opc_client_ae/protocol/opc_client_ae.dsp` 1-133

Middleware and IPC


Named Pipes
\\.\pipe\iec104master_name
\\.\pipe\modbus_master_na
\\.\pipe\opc_client_da_name
RTPS Middleware
rtps_mngr.exe
Middleware Manager
librtps.lib
Communication Library
RTPS Domain
Shared Memory
FIFO Buffers
fifo.lib
Local buffering
fifo_monitor_directionN
ORTE Domain
ORTE Domain
Event Distribution
Publishers
ORTEPublicationSend
Subscribers
ORTESubscription
callbacks
fifo_control_directionN

IndigoSCADA uses multiple IPC mechanisms for different purposes:

RTPS/ORTE Middleware

- **Purpose**: Publish-subscribe data distribution between processes
- **Manager**: `rtps_mngr.exe` coordinates the middleware domain
- **Library**: `librtps.lib` provides API for publishers and subscribers
- **Data Type**: `iec_item` structures (universal data format)
- **Directions**:
    - `fifo_monitor_directionN`: Device → SCADA (measurements)
    - `fifo_control_directionN`: SCADA → Device (commands)

Windows Named Pipes

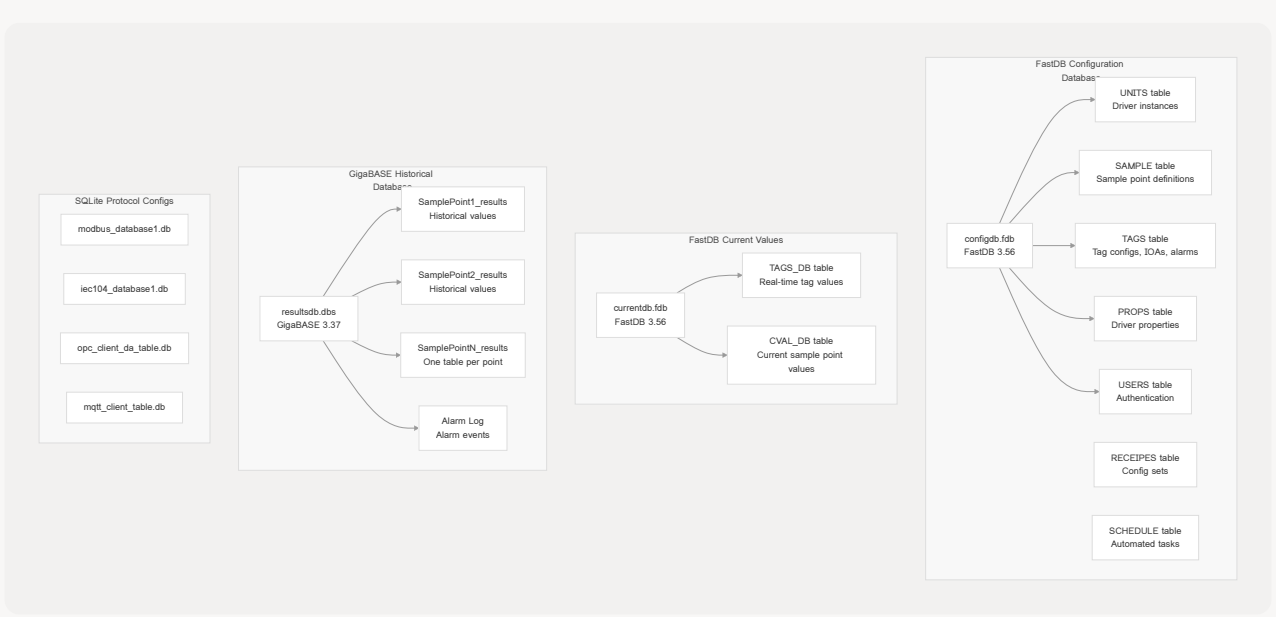
- **Purpose**: Parent-child process communication and keep-alive
- **Naming**: `\\.\pipe\{processname}_namedpipe{linenum}`
- **Usage**: Driver DLLs communicate with spawned master executables
- **Data**: Command/control messages and health checks

FIFO Buffers

- **Purpose**: Local buffering within driver processes
- **Library**: `fifo.lib` provides queue management
- **Function**: Decouples I/O threads from middleware publication

Sources: `src/middleware/rtps/manager/mcvs/rtps_mngr.dsp` 1-108 `src/daq/daq.dsw` 552-563 `src/daq/daq.dsw` 183-184

Database Components


SQLite Protocol Configs
modbus_database1.db
iec104_database1.db
opc_client_da_table.db
mqtt_client_table.db
GigaBASE Historical Database
resultsdb.dbs
GigaBASE 3.37
SamplePoint1_results
Historical values
SamplePoint2_results
Historical values
SamplePointN_results
One table per point
Alarm Log
Alarm events
FastDB Current Values
currentdb.fdb
FastDB 3.56
TAGS_DB table
Real-time tag values
CVAL_DB table
Current sample point values
FastDB Configuration Database
configdb.fdb
FastDB 3.56
UNITS table
Driver instances
SAMPLE table
Sample point definitions
TAGS table
Tag configs, IOAs, alarms
PROPS table
Driver properties
USERS table
Authentication
RECEIPES table
Config sets
SCHEDULE table
Automated tasks

Configuration Database (configdb)

- **Engine**: FastDB 3.56 (in-memory, transactional)
- **Location**: Project directory
- **Key Tables**:
    - `UNITS`: Driver instance definitions and settings
    - `SAMPLE`: Sample point definitions
    - `TAGS`: Tag configurations with IOA mappings and alarm limits
    - `PROPS`: Driver-specific properties
    - `USERS`: User authentication and privileges
    - `RECEIPES`: Named configuration sets
    - `SCHEDULE`: Automated task scheduling
- **Initialization**: Created from `init_db/configdb.sql` schema

Current Values Database (currentdb)

- **Engine**: FastDB 3.56 (in-memory)
- **Location**: Project directory
- **Key Tables**:
    - `TAGS_DB`: Real-time tag values
    - `CVAL_DB`: Current sample point values with quality and timestamps
- **Purpose**: High-speed read access for HMI and OPC server

Historical Database (resultsdb)

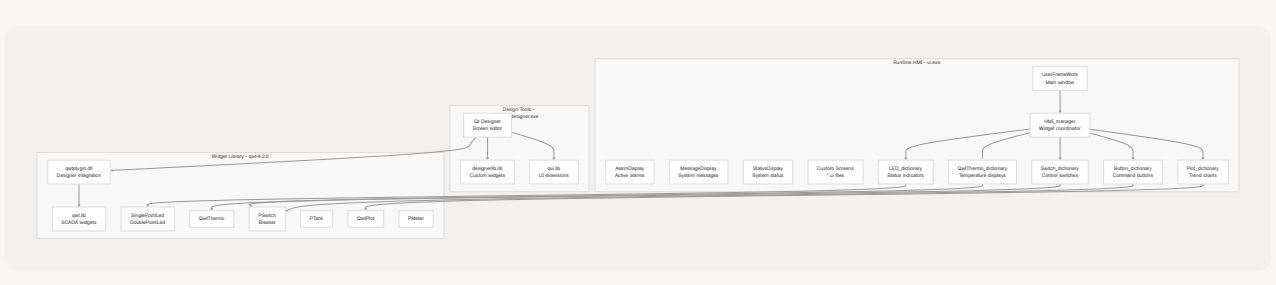
- **Engine**: GigaBASE 3.37 (persistent, compressed)
- **Location**: Project directory
- **Structure**: One table per sample point (`{name}_results`)
- **Contents**: Time-series data with timestamps, values, and quality
- **Purpose**: Long-term storage and historical queries

Protocol Configuration Databases

- **Engine**: SQLite 3
- **Naming**: `{protocol}_database{line}.db`
- **Purpose**: Protocol-specific mappings (IOA to register, Item ID, topic, etc.)
- **Usage**: Loaded by driver processes at startup

Sources: `src/daq/daq.dsw` 336-344 `src/daq/daq.dsw` 360-368 `src/database/results.h` 1-459

HMI Components



Runtime HMI (ui.exe)

- **Framework**: Qt 3.3.8 with custom SCADA extensions
- **Main Class**: `UserFrameWork` - manages main window and tabs
- **Coordinator**: `HMI_manager` - manages widget updates and data binding
- **Widget Dictionaries**: Organize widgets by type for efficient batch updates
- **Custom Screens**: Dynamically loaded `.ui` files designed in `hmi_designer.exe`
- **Standard Displays**:
    - `AlarmDisplay`: Active alarms with acknowledgment
    - `MessageDisplay`: System messages and events
    - `StatusDisplay`: System health indicators
- **Data Binding**: Widgets keyed by sample point name (truncated at underscore)
- **Update Mechanism**:
    - `UpdateTags` signal for value changes
    - `UpdateSamplePoint` for alarm state changes

HMI Designer (hmi_designer.exe)

- **Base**: Qt Designer with IndigoSCADA extensions
- **Libraries**:
    - `designerlib.lib`: Custom widget support
    - `qui.lib`: UI library extensions
- **Custom Widgets**: SCADA-specific controls from `qwt-4.2.0` library
- **Output**: `.ui` files loaded at runtime by `ui.exe`

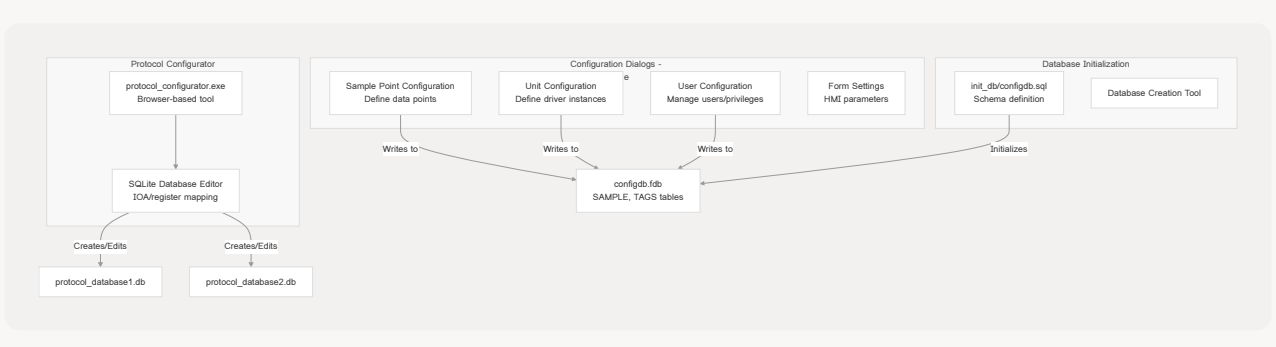
QWT Widget Library (qwt-4.2.0)

SCADA-specific widgets for visualization and control:

- **LEDs**: `SinglePointLed`, `DoublePointLed` - status indication with flashing for alarms
- **Thermometers**: `QwtThermo` - analog value display
- **Switches**: `PSwitch`, `Breaker` - binary control with confirmation
- **Tanks**: `PTank` - level indication
- **Plots**: `QwtPlot` - time-series trending
- **Meters**: `PMeter` - analog meters

Sources: `src/daq/daq.dsw` 999-1019 `src/daq/daq.dsw` 870-879 `src/hmi_designer/designer/app/designer.dsp` 1-118 `src/hmi_designer/designer/designer/designerlib.dsp` 1-151 `src/hmi_designer/designer/uilib/qui.dsp` 1-152

Configuration Tools


Protocol Configurator
protocol_configurator.exe
Browser-based tool
SQLite Database Editor
IOA/register mapping
Creates/Edits
protocol_database1.db
protocol_database2.db
Configuration Dialogs
Sample Point Configuration
Define data points
Unit Configuration
Define driver instances
User Configuration
Manage users/privileges
Form Settings
HMI parameters
Writes to
configdb.fdb
SAMPLE, TAGS tables
Database Initialization
init_db/configdb.sql
Schema definition
Database Creation Tool
Initializes

Protocol Configurator (protocol_configurator.exe)

- **Purpose**: Configure protocol-specific mappings
- **Interface**: Qt-based browser for SQLite databases
- **Functions**:
    - Create/edit `protocol_databaseN.db` files
    - Map IOAs to protocol-specific addresses (registers, item IDs, topics)
    - Define data types and scaling
    - Configure communication parameters
- **Workflow**: Configuration done offline, drivers load config at startup

Configuration Dialogs (in ui.exe)

Integrated configuration screens for system setup:

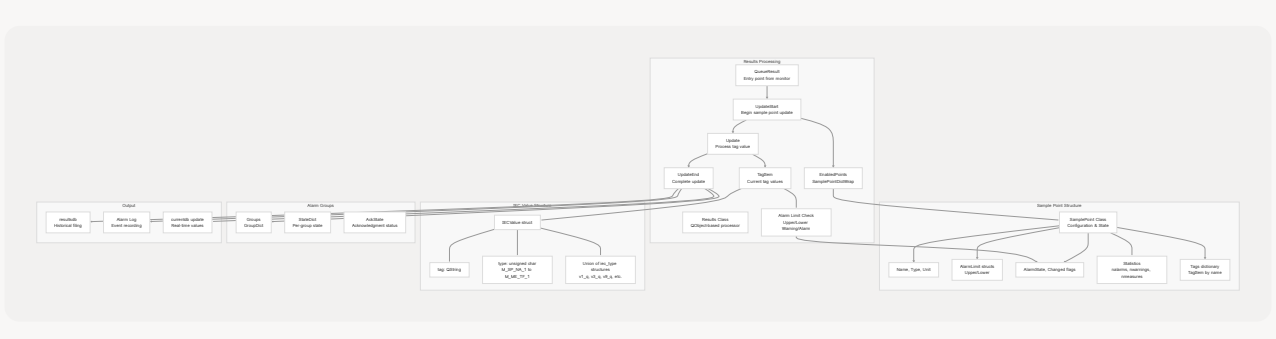
- **Sample Point Configuration**:
    - Define sample points (data points to monitor)
    - Set alarm limits (upper/lower alarm/warning)
    - Configure scaling and engineering units
    - Create historical results tables
    - Associate with driver units
- **Unit Configuration**:
    - Define units (driver instances)
    - Select unit type (protocol)
    - Configure driver-specific properties
    - Set communication parameters
    - Assign sample points to units
- **User Configuration**:
    - Manage user accounts
    - Set privilege levels (`PRIVS_ALL`, `PRIVS_ACK_ALARMS`, `PRIVS_FACTORY_ADMIN`)
    - Configure passwords
    - Define access restrictions

Database Initialization

- **Schema Files**: `init_db/configdb.sql` defines database structure
- **Initialization**: Creates empty FastDB configuration database
- **Versioning**: Schema evolution tracked for migrations

Sources: `src/daq/daq.dsw` 834-843

Data Processing Components



Results Processor

The Results class manages alarm evaluation and historical data filing:

Key Classes:

- `Results`: Main processor class (QObject-based for signals/slots)
- `SamplePoint`: Holds sample point configuration and current state
- `TagItem`: Current value and alarm state for a single tag
- `IECValue`: Universal value structure supporting all IEC 60870-5 types
- `AlarmLimit`: Upper/lower alarm/warning threshold configuration

Processing Flow:

1. **Queue Entry**: `QueueResult(name, IECValueList)` called by monitor
2. **Update Start**: `UpdateStart(name)` locks sample point
3. **Tag Processing**: `Update(name, tag, value)` for each tag:
    - Evaluates alarm limits (`AlarmLimit::Check()`)
    - Updates statistics
    - Determines alarm state changes
    - Sets `changed` and `fAckTriggered` flags
4. **Update End**: `UpdateEnd(name)`:
    - Files to historical database if `Fileable` is true
    - Logs alarm events
    - Updates current values database
    - Updates alarm groups

Alarm States:

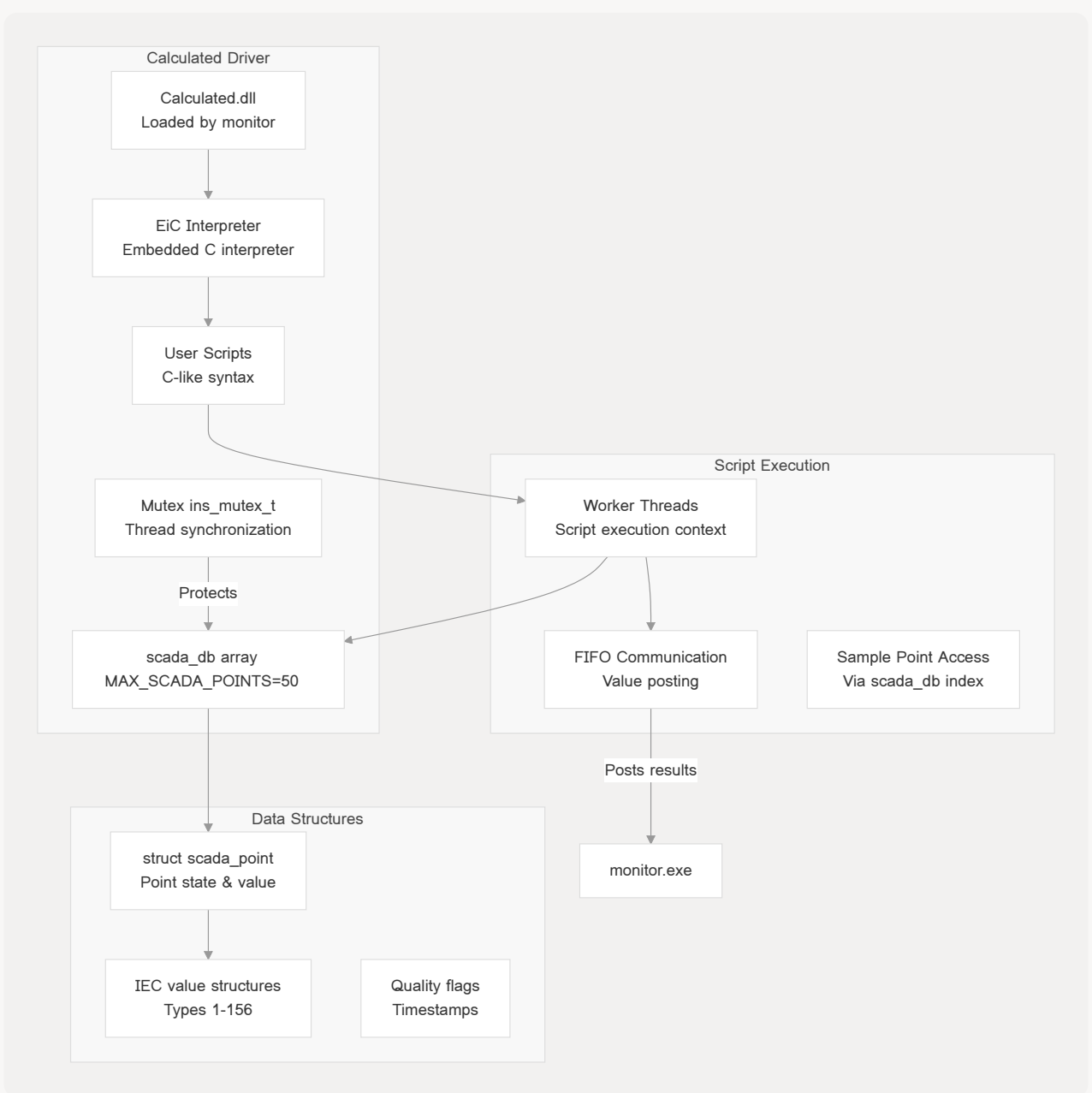
- `OkLevel` (0): Normal operation
- `WarningLevel` (1): Warning threshold exceeded
- `AlarmLevel` (2): Alarm threshold exceeded
- `FailureLevel` (3): Communication or quality failure

Alarm Groups:

- Organize sample points into logical groups
- Aggregate state across multiple points
- Support group-level acknowledgment
- Cached in memory (`Groups` dictionary) for performance

Sources: `src/database/results.h` 1-459

Calculated Driver


Calculated Driver
Calculated.dll
Loaded by monitor
EiC Interpreter
Embedded C interpreter
User Scripts
C-like syntax
Mutex ins_mutex_t
Thread synchronization
Protects
scada_db array
MAX_SCADA_POINTS=50
Script Execution
Worker Threads
Script execution context
FIFO Communication
Value posting
Sample Point Access
Via scada_db index
Posts results
monitor.exe
Data Structures
struct scada_point
Point state & value
IEC value structures
Types 1-156
Quality flags
Timestamps

The Calculated driver enables computed/derived data points:

Architecture:

- **Interpreter**: EiC-4.4.2 embedded C interpreter
- **Loading**: DLL loaded by monitor process at runtime
- **Shared Memory**: `scada_db` array accessible from scripts
- **Thread Safety**: `ins_mutex_t` protects concurrent access
- **Communication**: FIFO for posting calculated values back to monitor

Capabilities:

- Execute C-like scripts to compute values from other points
- Access real-time sample point data via `scada_db` array
- Support mathematical expressions, conditionals, loops
- Post calculated values as new sample points

Use Cases:

- Derived measurements (e.g., power = voltage × current)
- Unit conversions
- Complex calculations requiring multiple inputs
- Conditional logic based on multiple point states

Sources: `src/daq/daq.dsw` 6-17 `src/database/results.h` 24-38

Directory Structure

The IndigoSCADA installation follows a standard directory layout:

```
C:\scada\
├── bin\                   # All executables and DLLs
│   ├── manager.exe
│   ├── monitor.exe
│   ├── ui.exe
│   ├── hmi_designer.exe
│   ├── protocol_configurator.exe
│   ├── iec104master.exe
│   ├── modbus_master.exe
│   ├── dnp3master.exe
│   ├── opc_client_da.exe
│   ├── rtps_mngr.exe
│   └── ... (all other executables and DLLs)
│
├── project\               # Project-specific files (configurable)
│   ├── manager.ini        # Process configuration
│   ├── configdb.fdb       # Configuration database
│   ├── currentdb.fdb      # Real-time values database
│   ├── resultsdb.dbs      # Historical database
│   ├── iec104_database1.db # Protocol configs
│   ├── modbus_database1.db
│   └── ... (other database files)
│
├── init_db\               # Database initialization
│   ├── configdb.sql       # Schema definition
│   └── ... (other schemas)
│
├── lib\                   # Import libraries
│   ├── fastdb.lib
│   ├── gigabase.lib
│   ├── utilities.lib
│   ├── rtps.lib
│   └── ... (other libraries)
│
├── logs\                  # Log files
│   └── manager.log        # Process manager log
│
└── Help\                  # Documentation
    └── ... (help files)
```

Project Directory Configuration:

- Configurable via `bin\project.ini`
- Key: `[project_directory] path=...`
- Default: `C:\scada\project`
- Functions: `GetScadaProjectDirectory()` and `SetScadaProjectDirectory()`

Sources: `src/manager/buttongroups.cpp` 26-138 `src/common/common.cpp` 38-153

Build System

IndigoSCADA uses Microsoft Visual Studio 6.0 build system:

Workspace File: `src/daq/daq.dsw`

- Central workspace containing all projects
- Defines build dependencies between projects
- "all" meta-project builds entire system

Key Library Projects:

- `fastdb`: FastDB 3.56 database engine
- `gigabase`: GigaBASE 3.37 database engine
- `utilities`: Common utility functions
- `fifo`: FIFO queue implementation
- `librtps`: RTPS middleware library
- `scada`: Core SCADA library
- `qwt`: QWT 4.2.0 widget library
- `qui`: Qt UI extensions
- `designerlib`: Designer extensions

Executable Projects:

- Each driver, tool, and core process has dedicated .dsp file
- Output directory: `c:\scada\bin`
- Dependencies automatically managed

Third-Party Dependencies:

- Qt 3.3.8: UI framework
- pthreads-w32: Threading library (Windows)
- SQLite 3: Embedded database
- Various protocol libraries (libmodbus, OPC toolkits, etc.)

Sources: `src/daq/daq.dsw` 1-1046

Protocol Driver Architecture

> Relevant source files

This document describes the common architectural patterns, classes, and interfaces used across all protocol drivers in IndigoSCADA. It covers the `DriverInstance` class hierarchy, child process management patterns, state machines, configuration loading, and inter-process communication mechanisms. For details on specific protocol implementations, see Protocol Drivers. For middleware and IPC implementation details, see Middleware and IPC.

Overview of Common Pattern

All protocol drivers in IndigoSCADA follow a repeating architectural pattern consisting of three main components:

Component	Class/Process	Purpose	Lifecycle
Parent Coordinator	`DriverInstance` subclass	Configuration, state machine, database queries	Runs in `monitor.exe` main thread
Process Manager	`DriverThread` subclass	Child process spawn/monitor	Optional thread in `monitor.exe`
Protocol Implementation	`xxx_imp` class in `xxx_master.exe`	Protocol-specific I/O and logic	Separate child process

The parent component runs within the main `monitor.exe` process and coordinates overall driver operation through a tick-based state machine. Many drivers optionally spawn a child process for the actual protocol implementation, managed by a `DriverThread`. Communication between components uses multiple channels: ORTE/RTPS pub/sub for system-wide messages, local FIFOs for buffering, and Windows named pipes for parent-child coordination.

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 1-150
`src/drivers/modbus/modbus_driver_instance.cpp` 1-150
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 1-150

DriverInstance Class Hierarchy

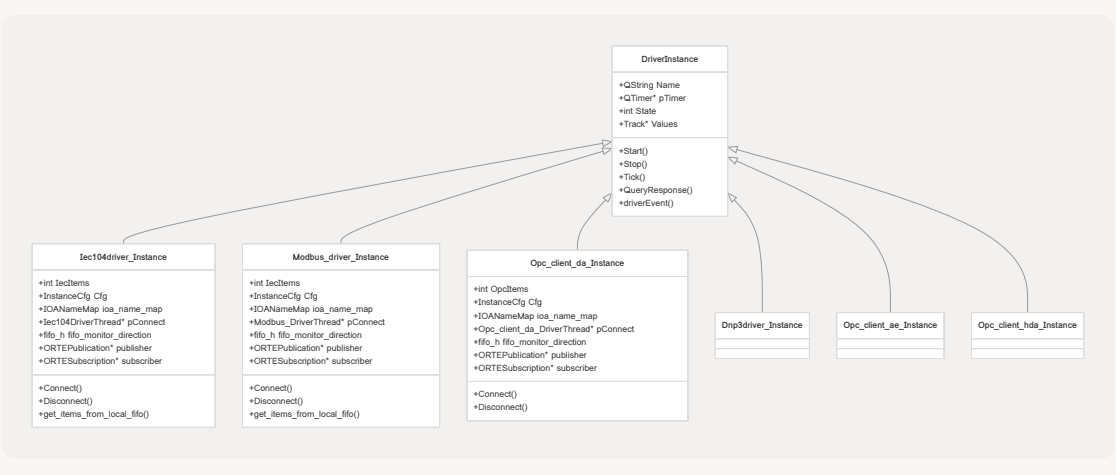

DriverInstance
+QString Name
+QTimer* pTimer
+int State
+Track* Values
+Start()
+Stop()
+Tick()
+QueryResponse()
+driverEvent()
Iec104driver_Instance
+int IecItems
+InstanceCfg Cfg
+IOANameMap ioa_name_map
+Iec104DriverThread* pConnect
+fifo_h fifo_monitor_direction
+ORTEPublication* publisher
+ORTESubscription* subscriber
+Connect()
+Disconnect()
+get_items_from_local_fifo()
Modbus_driver_Instance
+int IecItems
+InstanceCfg Cfg
+IOANameMap ioa_name_map
+Modbus_DriverThread* pConnect
+fifo_h fifo_monitor_direction
+ORTEPublication* publisher
+ORTESubscription* subscriber
+Connect()
+Disconnect()
+get_items_from_local_fifo()
Opc_client_da_Instance
+int OpcItems
+InstanceCfg Cfg
+IOANameMap ioa_name_map
+Opc_client_da_DriverThread* pConnect
+fifo_h fifo_monitor_direction
+ORTEPublication* publisher
+ORTESubscription* subscriber
+Connect()
+Disconnect()
Dnp3driver_Instance
Opc_client_ae_Instance
Opc_client_hda_Instance

Diagram: DriverInstance Class Hierarchy and Common Members

Each protocol driver implements a `DriverInstance` subclass that inherits from the base `DriverInstance` class. The base class provides:

- **State Machine**: An integer `State` variable and `Tick()` method called periodically by `pTimer`
- **Configuration Management**: `QueryResponse()` handles database callbacks for loading configuration
- **Sample Point Tracking**: `Track* Values` array holds per-sample-point metadata
- **Event Handling**: `driverEvent()` processes asynchronous events from child threads

All subclasses add protocol-specific members including:

- `IecItems` or `OpcItems`: Number of data points to poll
- `Cfg`: Structure holding protocol parameters (IP address, port, etc.)
- `ioa_name_map`: Maps Information Object Address (IOA) integers to sample point names
- `fifo_monitor_direction`: Local FIFO buffer for incoming protocol data
- `publisher` / `subscriber`: ORTE middleware handles for pub/sub communication
- `pConnect`: Pointer to optional `DriverThread` for child process management

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 122-293
`src/drivers/modbus/modbus_driver_instance.h` 59-103
`src/drivers/opc_client_da/opc_client_da_instance.h` 60-140

Child Process Architecture

Many protocol drivers use a two-process architecture where the parent `DriverInstance` (running in `monitor.exe`) spawns a child process for protocol I/O. This provides isolation and allows protocol code to be restarted independently.

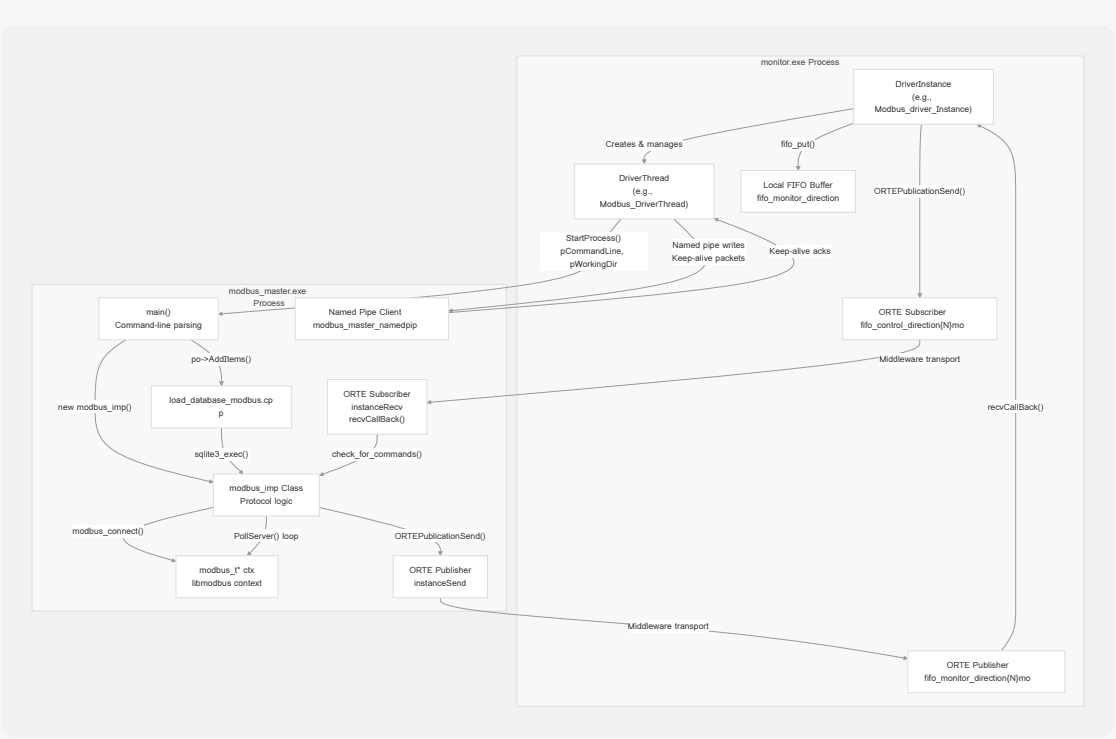

monitor.exe Process
DriverInstance (e.g., Modbus_driver_Instance)
Creates & manages
fifo_put()
DriverThread (e.g., Modbus_DriverThread)
Local FIFO Buffer fifo_monitor_direction
ORTEPublicationSend()
StartProcess() pCommandLine, pWorkingDir
Named pipe writes Keep-alive packets
Keep-alive acks
modbus_master.exe Process
main() Command-line parsing
Named Pipe Client modbus_master_namedpipe
ORTE Subscriber
Middleware transport
po->AddItems()
new modbus_imp()
load_database_modbus.cpp
ORTE Subscriber instanceRecv recvCallBack()
sqlite3_exec()
check_for_commands()
modbus_imp Class Protocol logic
recvCallBack()
modbus_connect()
PollServer() loop
ORTEPublicationSend()
modbus_t* ctx libmodbus context
ORTE Publisher instanceSend
Middleware transport
ORTE Publisher

Diagram: Child Process Pattern for Modbus Driver

Parent Process Responsibilities

The parent `DriverInstance` coordinates the system by:

1. **Spawning Child**: The `Connect()` method creates a `DriverThread` which calls `StartProcess()` with command-line arguments
2. **State Management**: The `Tick()` method implements a state machine transitioning through initialization, general interrogation, and running states
3. **FIFO Consumption**: `get_items_from_local_fifo()` reads messages from the local FIFO buffer populated by middleware callbacks
4. **Command Publishing**: When the HMI issues commands, `QueryResponse()` with transaction ID `tGetIOAfromSamplePointName` constructs command messages and publishes them

Child Process Responsibilities

The child process (e.g., `modbus_master.exe`) handles protocol I/O:

1. **Configuration Loading**: `AddItems()` opens the SQLite database (e.g., `modbus_database1.db`) and loads protocol mappings
2. **Protocol I/O**: `PollServer()` enters an infinite loop calling `PollItems()` to read from field devices via `modbus_read_registers()`, `modbus_read_bits()`, etc.
3. **Data Publishing**: Each polled value is packaged into an `iec_item` struct and published via `ORTEPublicationSend(publisher)`
4. **Command Handling**: The `recvCallBack()` function receives control direction messages and invokes `check_for_commands()` to execute writes

Named Pipe Keep-Alive

A Windows named pipe provides heartbeat monitoring between parent and child:

- Pipe name format: `\\.\pipe\{driver}_master_namedpipe{N}` where `{N}` is the line number
- Parent sends periodic packets (every 2 seconds) with address 4004 and data byte 33
- Child must respond with acknowledgment; if connection lost for >60 seconds, child exits
- On child crash detection, parent restarts the process automatically

Sources: `src/drivers/modbus/modbus_driverthread.cpp` 22-135
`src/drivers/modbus/protocol/main_master.cpp` 72-355
`src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 22-145
`src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 22-149

State Machine Architecture

All `DriverInstance` implementations use a tick-based state machine driven by a 1-second `QTimer`. The `Tick()` method is called periodically and switches on the `State` variable to perform state-specific actions.

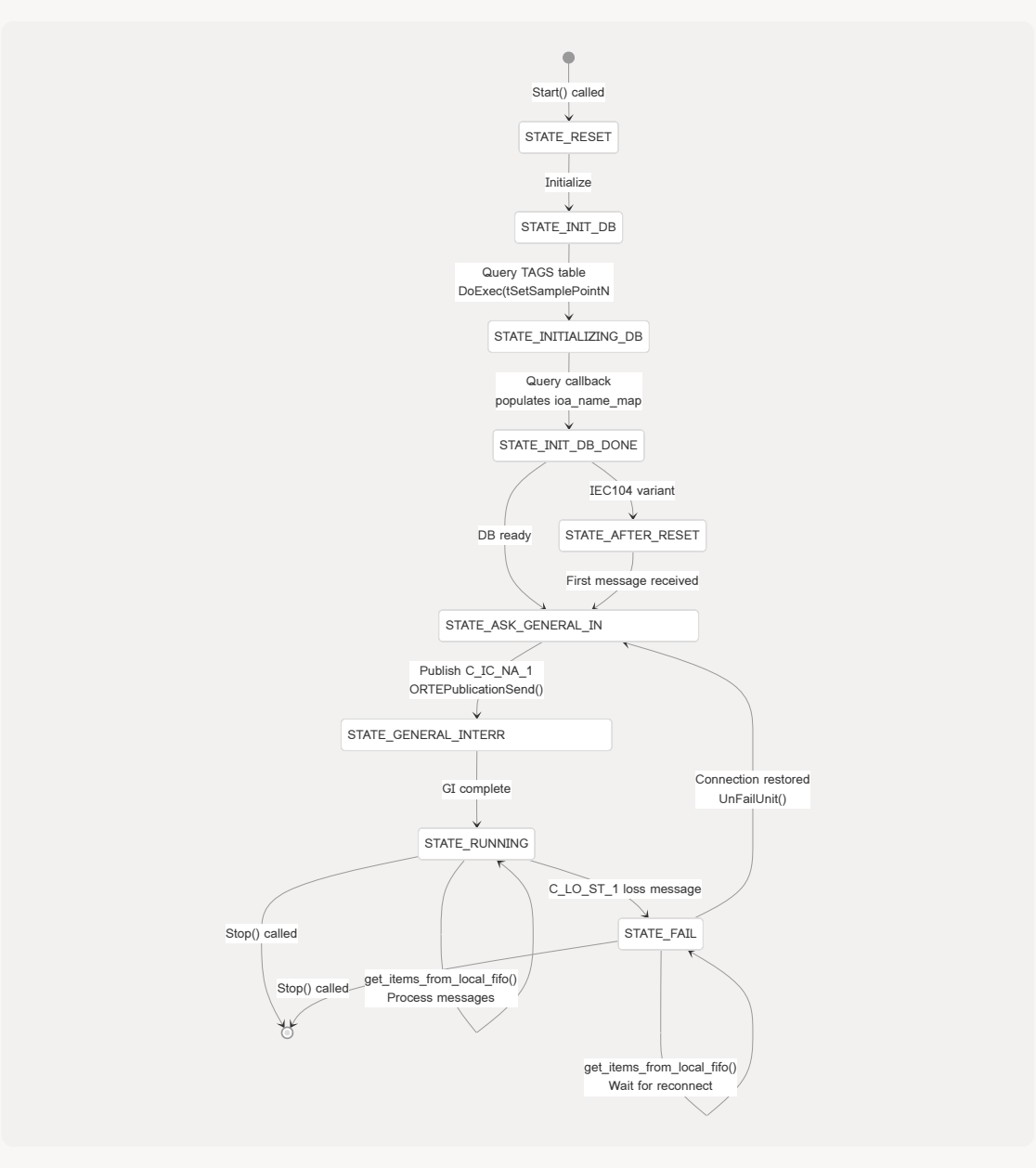

Start() called
STATE_RESET
Initialize
STATE_INIT_DB
Query TAGS table DoExec(tSetSamplePointN
STATE_INITIALIZING_DB
Query callback populates ioa_name_map
STATE_INIT_DB_DONE
IEC104 variant
DB ready
STATE_AFTER_RESET
First message received
STATE_ASK_GENERAL_IN
Publish C_IC_NA_1 ORTEPublicationSend()
STATE_GENERAL_INTERR
Connection restored UnFailUnit()
GI complete
STATE_RUNNING
C_LO_ST_1 loss message
Stop() called
STATE_FAIL
Stop() called
get_items_from_local_fifo() Process messages
get_items_from_local_fifo() Wait for reconnect

Diagram: Common Driver State Machine Flow

State Descriptions

State	Purpose	Actions	Transitions
`STATE_RESET`	Initial entry	Set state to `STATE_INIT_DB`	→ `STATE_INIT_DB`
`STATE_INIT_DB`	Load IOA mappings	Query TAGS table: `"select * from TAGS where UNIT='...'"`	→ `STATE_INITIALIZING_DB`
`STATE_INITIALIZING_DB`	Waiting for DB	No action, wait for `QueryResponse()`	→ `STATE_INIT_DB_DONE`
`STATE_INIT_DB_DONE`	DB ready	Populate `ioa_name_map` from query results	→ `STATE_ASK_GENERAL_INTERROGATION` or `STATE_AFTER_RESET`
`STATE_AFTER_RESET`	Wait for child	Call `get_items_from_local_fifo()`, wait for first message	→ `STATE_ASK_GENERAL_INTERROGATION`
`STATE_ASK_GENERAL_INTERROGATION`	Request full data	Construct `C_IC_NA_1` message, publish to control direction	→ `STATE_GENERAL_INTERROGATION_DONE`
`STATE_GENERAL_INTERROGATION_DONE`	GI pending	Single tick delay	→ `STATE_RUNNING`
`STATE_RUNNING`	Normal operation	Process FIFO messages, post values to database	→ `STATE_FAIL` on loss
`STATE_FAIL`	Connection lost	Call `FailUnit()`, continue reading FIFO for recovery	→ `STATE_ASK_GENERAL_INTERROGATION`

State Implementation Example

The `Tick()` method structure is consistent across all drivers:

```
void Xxx_driver_Instance::Tick()
{
    switch(State)
    {
        case STATE_RESET:
            State = STATE_INIT_DB;
            break;

        case STATE_INIT_DB:
            State = STATE_INITIALIZING_DB;
            QString cmd = "select * from TAGS where UNIT='"+ Name + "';";
            GetConfigureDb()->DoExec(this, cmd, tSetSamplePointNameIOAMap);
            break;

        case STATE_INIT_DB_DONE:
            State = STATE_ASK_GENERAL_INTERROGATION;
            break;

        case STATE_ASK_GENERAL_INTERROGATION:
            // Construct C_IC_NA_1 message
            struct iec_item item_to_send;
            memset(&item_to_send, 0x00, sizeof(struct iec_item));
            item_to_send.iec_type = C_IC_NA_1;
            item_to_send.iec_obj.ioa = 0;
            item_to_send.iec_obj.o.type100.qoi = 1;
            // Publish message
            ORTEPublicationSend(publisher);
            State = STATE_GENERAL_INTERROGATION_DONE;
            break;

        case STATE_RUNNING:
            get_items_from_local_fifo();
            break;
    }
}
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 661-750
`src/drivers/modbus/modbus_driver_instance.cpp` 519-600
`src/drivers/dnp3/dnp3driver_instance.cpp` 416-475

Configuration Loading

Protocol drivers load configuration from two sources: the FastDB configuration database (for general SCADA settings) and protocol-specific SQLite databases.

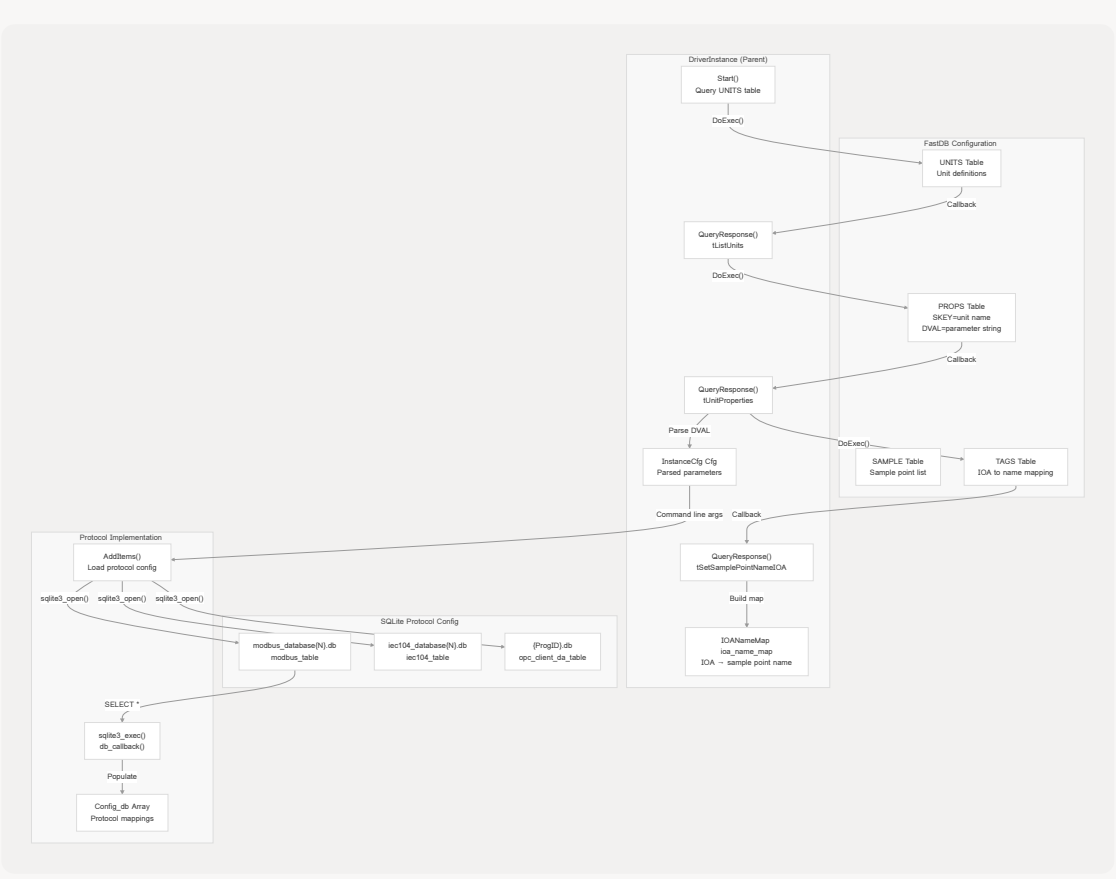

Query UNITS table
DoExec()
FastDB Configuration
UNITS Table Unit definitions
Callback
QueryResponse() tListUnits
DoExec()
PROPS Table SKEY=unit name DVAL=parameter string
Callback
QueryResponse() tUnitProperties
Parse DVAL
DoExec()
InstanceCfg Cfg Parsed parameters
SAMPLE Table Sample point list
TAGS Table IOA to name mapping
Command line args
Callback
QueryResponse() tSetSamplePointNameIOA
Build map
IOANameMap ioa_name_map IOA → sample point name
Protocol Implementation
AddItems() Load protocol config
sqlite3_open()
SQLite Protocol Config
modbus_database[N].db modbus_table
iec104_database[N].db iec104_table
{ProgID}.db opc_client_da_table
SELECT *
sqlite3_exec() db_callback()
Populate
Config_db Array Protocol mappings

Diagram: Configuration Loading Flow

FastDB Configuration Queries

The parent `DriverInstance` loads configuration through a series of asynchronous database queries in `QueryResponse()`:

1. **tListUnits**: Retrieves the unit definition from the UNITS table matching the driver name
2. **tUnitProperties**: Fetches the PROPS table entry where SKEY is the unit name and extracts the DVAL text field
3. **tList**: Loads sample points from the SAMPLE table associated with this unit
4. **tSetSamplePointNameIOAMap**: Queries the TAGS table to build the IOA-to-name mapping

The DVAL field in PROPS contains a space-separated parameter string that is parsed using `QTextIStream`:

```
QString s = UndoEscapeSQLText(GetConfigureDb()->GetString("DVAL"));
QTextIStream is(&s);
is >> IecItems;              // Number of items
is >> Cfg.SampleTime;        // Polling period (ms)
is >> Cfg.ServerIPAddress;   // Protocol server IP
is >> Cfg.ServerIPPort;      // Protocol server port
// ... additional protocol-specific parameters
```

SQLite Protocol Configuration

Child processes load protocol-specific mappings from SQLite databases. The database filename is constructed from the line number and protocol name:

```
strcpy(database_name, project_dir);
strcat(database_name, "\\");
strcat(database_name, "modbus_database");
strcat(database_name, line_number);  // e.g., "1"
strcat(database_name, ".db");

rc = sqlite3_open(database_name, &db);
rc = sqlite3_exec(db, "select * from modbus_table;", db_callback, 0, &zErrMsg);
```

The `db_callback()` function populates a `Config_db` array with protocol mappings:

```
static int db_callback(void *NotUsed, int argc, char **argv, char **azColName)
{
    for(i = 0; i < argc; i++)
    {
        switch(i)
        {
            case 0: // slave_id
                gl_Config_db[gl_row_counter].slave_id = atoi(argv[i]);
                break;
            case 1: // modbus_function_read
                gl_Config_db[gl_row_counter].modbus_function_read = atoi(argv[i]);
                break;
            case 5: // ioa_control_center
                gl_Config_db[gl_row_counter].ioa_control_center = atoi(argv[i]);
                break;
            // ...
        }
    }
    gl_row_counter++;
    return 0;
}
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 150-289
`src/drivers/modbus/protocol/load_database_modbus.cpp` 173-254
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 174-313

Communication Interfaces

Protocol drivers use three distinct communication channels for different purposes:

ORTE/RTPS Middleware

The ORTE (Open Real-Time Ethernet) middleware provides publish/subscribe communication between all system components. Each driver creates two topics:

Publisher (Monitor Direction):

```
char fifo_monitor_name[150];
strcpy(fifo_monitor_name, "fifo_monitor_direction");
strcat(fifo_monitor_name, line_number);  // e.g., "1"
strcat(fifo_monitor_name, "modbus");     // Protocol name

publisher = ORTEPublicationCreate(
    domain,
    fifo_monitor_name,          // Topic: "fifo_monitor_direction1modbus"
    "iec_item_type",            // Type name
    &instanceSend,              // Data buffer
    &persistence,               // QoS
    strength,
    NULL, NULL, NULL);
```

Subscriber (Control Direction):

```
char fifo_control_name[150];
strcpy(fifo_control_name, "fifo_control_direction");
strcat(fifo_control_name, line_number);
strcat(fifo_control_name, "modbus");

subscriber = ORTESubscriptionCreate(
    domain,
    IMMEDIATE,
    BEST_EFFORTS,
    fifo_control_name,          // Topic: "fifo_control_direction1modbus"
    "iec_item_type",
    &instanceRecv,
    &deadline,
    &minimumSeparation,
    recvCallBack,               // Callback function
    this,                       // Callback parameter
    smIPAddress);
```

The `recvCallBack()` function is invoked when control messages arrive:

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void *recvCallBackParam)
{
    Xxx_driver_Instance * cl = (Xxx_driver_Instance*)recvCallBackParam;
    iec_item_type *item1 = (iec_item_type*)vinstance;

    switch (info->status)
    {
        case NEW_DATA:
            struct iec_item item2;
            rebuild_iec_item_message(&item2, item1);
            fifo_put(cl->fifo_monitor_direction, (char *)&item2, sizeof(struct iec_item));
            break;
    }
}
```

Local FIFO Buffers

Each driver maintains a local FIFO buffer (`fifo_monitor_direction`) that decouples middleware callbacks from the main `Tick()` processing loop:

```
// Initialization
fifo_monitor_direction = fifo_open(MAX_FIFO_SIZE);

// In recvCallBack()
fifo_put(cl->fifo_monitor_direction, (char *)&item2, sizeof(struct iec_item));

// In Tick() -> get_items_from_local_fifo()
for(int i = 0; (len = fifo_get(fifo_monitor_direction, (char*)buf,
                                sizeof(struct iec_item), wait_limit_ms)) >= 0; i++)
{
    p_item = (struct iec_item*)buf;
    // Process message
}
```

The FIFO provides buffering for up to 65535 messages and allows non-blocking reads with a timeout.

Windows Named Pipes

For drivers using child processes, a Windows named pipe provides heartbeat monitoring:

Parent Side (DriverThread):

```
strcpy(pipe_name, "\\\\.\\pipe\\modbus_master_namedpipe");
strcat(pipe_name, line_number);

int send_ack_to_child(int address, int data, char* pipeName)
{
    h_pipe = CreateFile(pipeName, GENERIC_WRITE, 0, NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    struct iec_item to_send;
    memset(&to_send, 0x00, sizeof(struct iec_item));
    to_send.iec_obj.ioa = address;         // 4004
    to_send.iec_obj.o.type61.sv = data;    // 33 for keep-alive

    WriteFile(h_pipe, (char *)&to_send, sizeof(struct iec_item), &written, NULL);
    CloseHandle(h_pipe);
}
```

Child Side (Protocol Implementation):

```
// In PipeWorker() thread
for(i = 0; i < N_PIPES; i++)
{
    pipeHnds[i] = CreateNamedPipe(
        pipe_name,
        PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,
        N_PIPES, 0, 0, 1000, NULL);

    ConnectNamedPipe(pipeHnds[i], &ovrp[i]);
}

// Wait for messages
rc = WaitForMultipleObjects(N_PIPES, evnt, FALSE, INFINITE);
ReadFile(pipeHnds[pipe_id], in_buffer[pipe_id], BUF_SIZE, &len, &ovrp[pipe_id]);

// Reset timeout counter
gl_timeout_connection_with_parent = 0;
```

The child process exits if `gl_timeout_connection_with_parent` exceeds a threshold (typically 60 seconds).

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 143-212
`src/drivers/iec104/iec104driver_instance.cpp` 42-114
`src/drivers/modbus/protocol/main_master.cpp` 361-454
`src/drivers/modbus/modbus_driverthread.cpp` 52-129

Data Flow Pattern

The complete data flow from field device to HMI follows this path through the driver architecture:

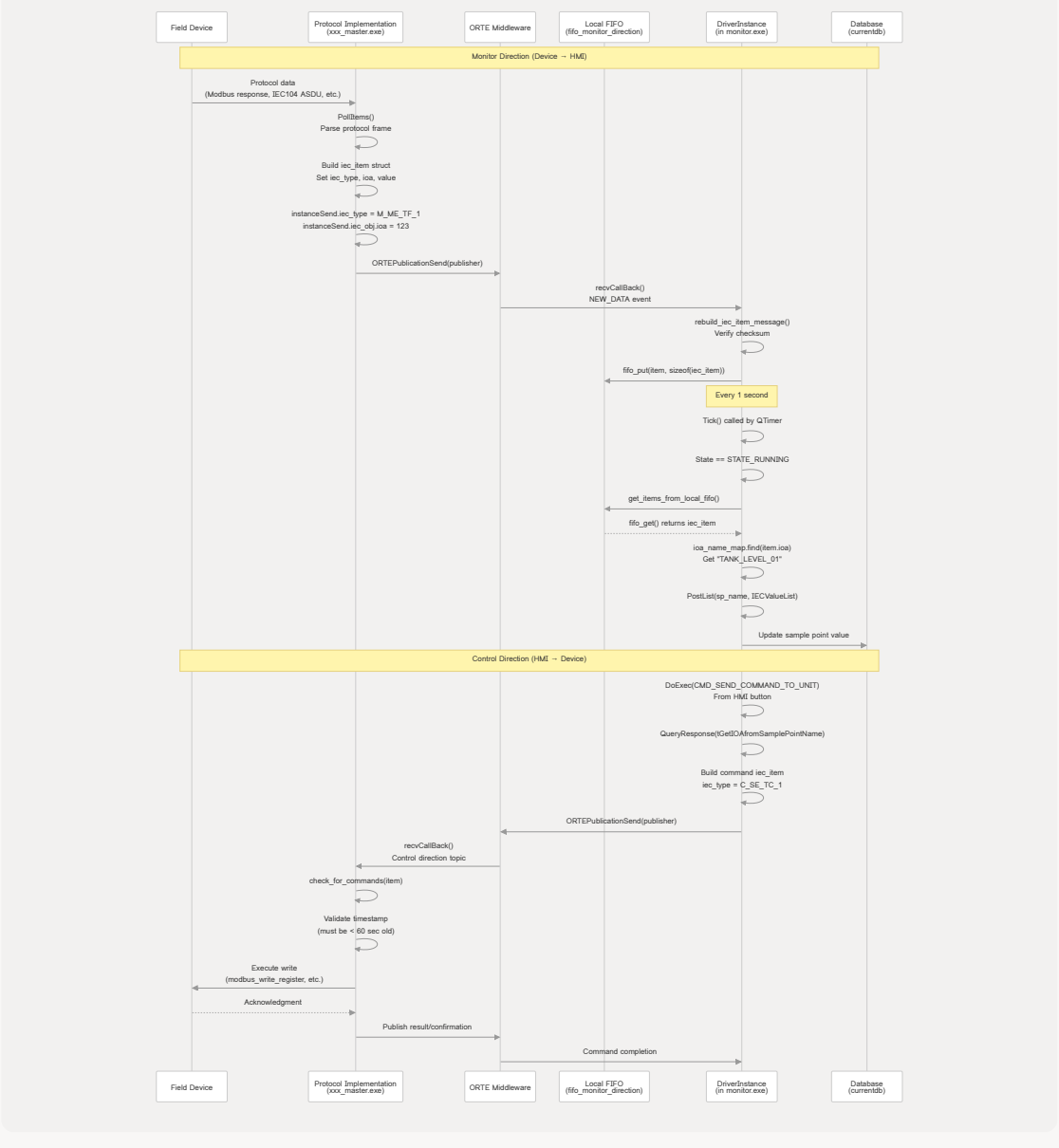

Field Device
Protocol Implementation (xxx_master.exe)
ORTE Middleware
Local FIFO (fifo_monitor_direction)
DriverInstance (in monitor.exe)
Database (currentdb)
Monitor Direction (Device → HMI)
Protocol data (Modbus response, IEC104 ASDU, etc.)
PollItems() Parse protocol frame
Build iec_item struct Set iec_type, ioa, value
ORTEPublicationSend(publisher)
recvCallBack() NEW_DATA event
rebuild_iec_item_message() Verify checksum
fifo_put(item, sizeof(iec_item))
Every 1 second
Tick() called by QTimer
State == STATE_RUNNING
get_items_from_local_fifo()
fifo_get() returns iec_item
Update sample point value
Control Direction (HMI → Device)
DoExec(CMD_SEND_COMMAND_TO_UNIT) From HMI button
ORTEPublicationSend(publisher)
recvCallBack() Control direction topic
check_for_commands(item)
Validate timestamp (must be < 60 sec old)
Execute write (modbus_write_register, etc.)
Acknowledgment
Publish result/confirmation
Command completion

Diagram: Complete Data Flow Through Driver Architecture

Monitor Direction Processing

When field data arrives, the protocol implementation follows these steps:

1. **Protocol I/O**: The `PollItems()` or equivalent method reads data using protocol-specific functions like `modbus_read_registers()` at `src/drivers/modbus/protocol/modbus_imp.cpp` 711
2. **IEC Item Construction**: Raw protocol data is converted to a normalized `iec_item` structure with appropriate IEC 60870-5 type (e.g., `M_ME_TF_1` for float, `M_SP_TB_1` for single point) at `src/drivers/modbus/protocol/modbus_imp.cpp` 756-766
3. **Middleware Publishing**: The `instanceSend` structure is populated and `ORTEPublicationSend(publisher)` broadcasts the message at `src/drivers/modbus/protocol/modbus_imp.cpp` 1214
4. **Callback Reception**: The parent `DriverInstance`'s `recvCallBack()` is invoked, validates the checksum, and writes to the local FIFO at `src/drivers/iec104/iec104driver_instance.cpp` 89-114
5. **Tick Processing**: The `Tick()` method calls `get_items_from_local_fifo()` which drains the FIFO buffer at `src/drivers/iec104/iec104driver_instance.cpp` 752-830
6. **Name Resolution**: The IOA is looked up in `ioa_name_map` to find the sample point name at `src/drivers/iec104/iec104driver_instance.cpp` 833-841
7. **Value Posting**: `PostList(sp_name, IECValueList)` updates the in-memory database with the new value

Control Direction Processing

When the HMI issues a command:

1. **Command Construction**: The UI calls `DoExec(CMD_SEND_COMMAND_TO_UNIT)` which triggers a database query at `src/drivers/iec104/iec104driver_instance.cpp` 314-339
2. **IOA Lookup**: The `QueryResponse()` handler with transaction ID `tGetIOAfromSamplePointName` retrieves the IOA for the target sample point
3. **Message Building**: A command `iec_item` is created with the appropriate IEC type (e.g., `C_SE_TC_1` for set point) at `src/drivers/iec104/iec104driver_instance.cpp` 483-496
4. **Control Publishing**: The command is published to the control direction topic via `ORTEPublicationSend(publisher)` at `src/drivers/iec104/iec104driver_instance.cpp` 529
5. **Command Validation**: The child process `check_for_commands()` verifies the command timestamp is less than 60 seconds old at `src/drivers/modbus/protocol/modbus_imp.cpp` 1472-1489
6. **Protocol Execution**: The protocol-specific write function is called (e.g., `modbus_write_register()`) at `src/drivers/modbus/protocol/modbus_imp.cpp` 1588-1606

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 562-1228
`src/drivers/iec104/iec104driver_instance.cpp` 752-1046
`src/drivers/modbus/protocol/modbus_imp.cpp` 1343-1846

Track Structure for Sample Point Metadata

Each `DriverInstance` maintains an array of `Track` structures that store per-sample-point metadata and statistics:

```
struct Track
{
    QString Name;              // Sample point name
    SampleStatistic Stats;     // Statistical accumulator
    double LastValue;          // Previous value for change detection
    bool fSpotValue;          // Report last value vs mean
    unsigned SamplePeriod;     // Polling interval (ms)
    QDateTime NextSample;      // Next scheduled sample time
    bool fFailed;             // Sample point failed flag

    void clear()
    {
        LastValue = 0.0;
        fSpotValue = false;
        NextSample = QDateTime::currentDateTime();
        fFailed = false;
        Stats.reset();
    }
};
```

The `Values` array is dynamically allocated after loading the `IecItems` count from configuration:

```
if(Values)
{
    delete[] Values;
    Values = NULL;
}
Values = new Track[IecItems+1];  // 1-indexed array
```

The array is indexed by `IPINDEX` from the SAMPLE table, allowing direct mapping from database records to runtime structures.

Sources: `src/drivers/modbus/modbus_driver_instance.h` 86-102
`src/drivers/iec104/iec104driver_instance.cpp` 267-273

Common Utility Functions

Several utility functions are shared across protocol driver implementations:

Checksum Validation

All `iec_item` messages include a CRC-8 checksum calculated over the entire structure:

```
void rebuild_iec_item_message(struct iec_item *item2, iec_item_type *item1)
{
    item2->iec_type = item1->iec_type;
    memcpy(&(item2->iec_obj), &(item1->iec_obj), sizeof(struct iec_object));
    item2->cause = item1->cause;
    item2->msg_id = item1->msg_id;
    // ... copy remaining fields

    unsigned char checksum = clearCrc((unsigned char *)item2, sizeof(struct iec_item));
    if(checksum != 0)
    {
        ExitProcess(0);  // Fatal error on checksum mismatch
    }
}
```

Timestamp Conversion

Protocol implementations convert between IEC 60870-5 CP56Time2a format and Unix epoch:

```
void get_utc_host_time(struct cp56time2a* time)
{
    epoch_in_ms = getTimeInMs();
    seconds = epoch_in_ms/1000;
    ptm = gmtime64((int64_t*)(&seconds));

    time->hour = ptm->tm_hour;
    time->min = ptm->tm_min;
    time->msec = ptm->tm_sec*1000 + (unsigned short)(epoch_in_ms%1000);
    time->mday = ptm->tm_mday;
    time->wday = (ptm->tm_wday == 0) ? ptm->tm_wday + 7 : ptm->tm_wday;
    time->month = ptm->tm_mon + 1;
    time->year = ptm->tm_year - 100;
}

int64_t epoch_from_cp56time2a(const struct cp56time2a* time)
{
    struct tm t;
    t.tm_hour = time->hour;
    t.tm_min = time->min;
    t.tm_sec = time->msec/1000;
    t.tm_mday = time->mday;
    t.tm_mon = time->month - 1;
    t.tm_year = time->year + 100;

    return mktime64(&t);
}
```

Process Management

The `DriverThread` base class provides standard process spawning:

```
bool StartProcess(char* pCommandLine, char* pWorkingDir)
{
    STARTUPINFO startUpInfo = { sizeof(STARTUPINFO), NULL, "", NULL, 0, 0, 0, 0, 0, 0,
                                STARTF_USESHOWWINDOW, 0, 0, NULL, 0, 0, 0 };

    if(CreateProcess(NULL, pCommandLine, NULL, NULL, TRUE, 0, NULL,
                     pWorkingDir, &startUpInfo, &pProcInfo[nIndex]))
    {
        return true;
    }
    return false;
}
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 52-87
`src/drivers/modbus/protocol/modbus_imp.cpp` 504-530

src/drivers/modbus/protocol/modbus_imp.cpp 532-557

src/drivers/opc_client_ae/opc_client_aedriverthread.cpp 138-148

Middleware and IPC

> Relevant source files

Purpose and Scope

This document describes the inter-process communication (IPC) mechanisms and middleware infrastructure used in IndigoSCADA to enable communication between distributed processes. The system employs multiple complementary IPC mechanisms to achieve different communication patterns: publish/subscribe event distribution (ORTE/RTPS), point-to-point process control (named pipes), and local buffering (FIFOs).

For information about the overall data flow pipeline and how these mechanisms fit into the larger architecture, see Data Flow Pipeline. For details on how individual protocol drivers are structured, see Protocol Driver Architecture.

Overview of IPC Mechanisms

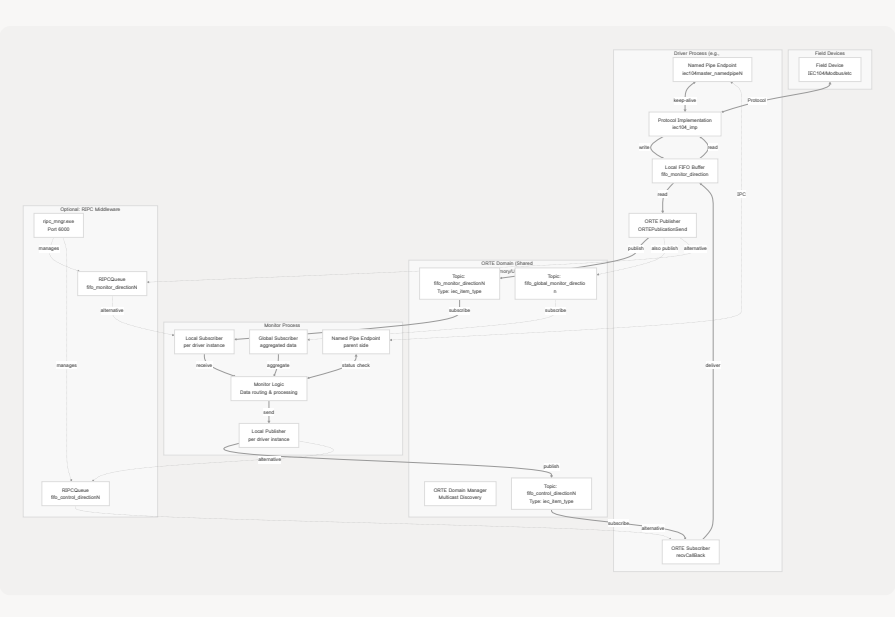
IndigoSCADA uses a layered IPC architecture with four distinct mechanisms working in concert:

Mechanism	Purpose	Scope	Direction	Implementation
ORTE/RTPS	Publish/Subscribe event distribution	System-wide	Bidirectional	UDP multicast, shared memory
RIPC Queues	Message queue middleware (optional)	System-wide	Bidirectional	TCP client/server with queues
Named Pipes	Parent-child process coordination	Process pair	Bidirectional	Windows named pipes
Local FIFOs	Intra-driver buffering	Single driver	Unidirectional	Circular buffer in shared memory

The system can be compiled with either ORTE or RIPC as the primary middleware, controlled by the `USE_RIPC_MIDDLEWARE` preprocessor flag. ORTE is the default and preferred mechanism.

Sources: `src/drivers/iec104/iec104driver_instance.h` 1-336 `src/drivers/iec104/iec104driver_instance.cpp` 1-100

IPC Architecture Diagram



Sources: `src/drivers/iec104/iec104driver_instance.cpp` 16-39 `src/drivers/iec104/iec104driver_instance.cpp` 41-114 `src/drivers/modbus/modbus_driver_instance.cpp` 16-114

Message Structure: iec_item

All IPC mechanisms transport the same fundamental data structure: `iec_item`, which encapsulates IEC 60870-5-104 protocol messages. This provides a universal abstraction layer that decouples protocol drivers from the middleware.

Core Structure

```
struct iec_item {
    unsigned char iec_type;        // IEC 104 type identifier (M_SP_NA_1, M_ME_NC_1, etc)
    struct iec_object iec_obj;     // Protocol-specific object with IOA and value
    unsigned char cause;           // Cause of transmission
    unsigned int msg_id;           // Sequence number for message tracking
    unsigned int ioa_control_center; // Information Object Address
    unsigned short casdu;          // Common Address of ASDU
    unsigned char is_neg;          // Negative confirmation flag
    unsigned char checksum;        // CRC-8 checksum for integrity
};
```

Checksum Validation

Every `iec_item` message includes a CRC-8 checksum calculated over the entire structure. Receiving processes validate this checksum and terminate on error:

```
// From rebuild_iec_item_message
checksum = clearCrc((unsigned char *)item2, sizeof(struct iec_item));
if(checksum != 0) {
    ExitProcess(0);  // Fatal error on checksum failure
}
```

ORTE Wire Format

For ORTE/RTPS transport, the structure is wrapped in `iec_item_type` and registered with the ORTE type system:

```
typedef struct {
    unsigned char iec_type;
    struct iec_object iec_obj;
    unsigned char cause;
    unsigned int msg_id;
    unsigned int ioa_control_center;
    unsigned short casdu;
    unsigned char is_neg;
    unsigned char checksum;
} iec_item_type;

// Type registration
iec_item_type_type_register(domain);
```

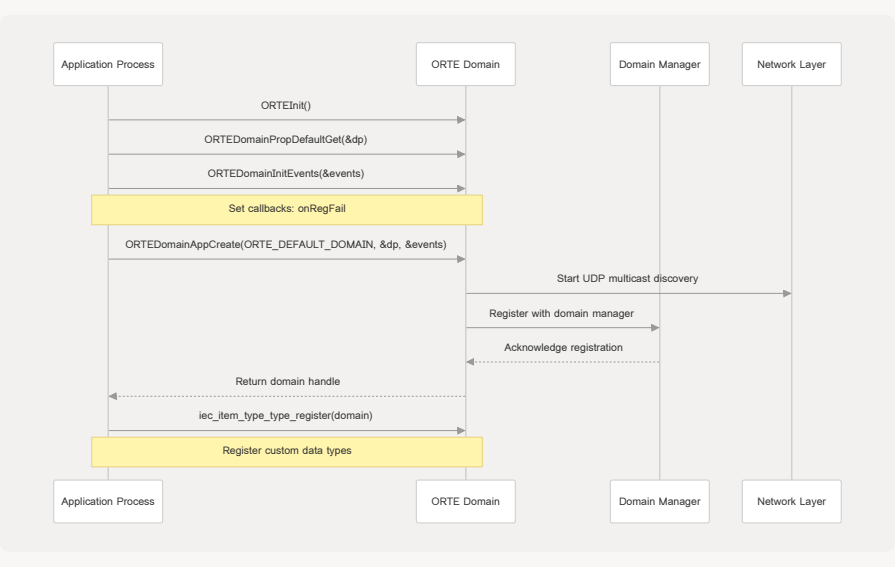
Sources: `src/drivers/iec104/iec104driver_instance.cpp` 52-87 `src/drivers/modbus/modbus_driver_instance.cpp` 52-87 `src/middleware/rtps/include/iec_item_type.h`

ORTE/RTPS Middleware

ORTE (Open Real-Time Ethernet) is the primary middleware, implementing the RTPS (Real-Time Publish-Subscribe) protocol. It provides reliable, ordered delivery of messages with Quality of Service (QoS) guarantees.

Domain Initialization

Each process creates an ORTE domain that participates in the distributed system:


Application Process
ORTE Domain
Domain Manager
Network Layer
ORTEInit()
ORTEDomainPropDefaultGet(&dp)
ORTEDomainInitEvents(&events)
Set callbacks: onRegFail
ORTEDomainAppCreate(ORTE_DEFAULT_DOMAIN, &dp, &events)
Start UDP multicast discovery
Register with domain manager
Acknowledge registration
Return domain handle
iec_item_type_type_register(domain)
Register custom data types
Application Process
ORTE Domain
Domain Manager
Network Layer

Publication Creation

Publishers are created with persistence and strength parameters:

```
// From iec104driver_instance.h constructor
NTPTIME_BUILD(persistence, 5);  // 5 second persistence
int32_t strength = 1;

publisher = ORTEPublicationCreate(
    domain,
    "fifo_control_directionNiec104",  // Topic name
    "iec_item_type",                  // Type name
    &instanceSend,                    // Data instance
    &persistence,
    strength,
    NULL, NULL, NULL);
```

Persistence: How long a published message remains valid for late-joining subscribers.

Strength: Priority when multiple publishers exist for the same topic (higher strength wins).

Subscription Creation

Subscribers register callbacks for message delivery:

```
NTPTIME_BUILD(deadline, 3);                    // 3 second deadline
NTPTIME_BUILD(minimumSeparation, 0);           // No throttling

subscriber = ORTESubscriptionCreate(
    domain,
    IMMEDIATE,                                  // Delivery mode
    BEST_EFFORTS,                              // Reliability
    "fifo_monitor_directionNiec104",           // Topic name
    "iec_item_type",                           // Type name
    &instanceRecv,                             // Data instance
    &deadline,
    &minimumSeparation,
    recvCallBack,                              // Callback function
    this,                                      // User parameter
    IPADDRESS_INVALID);                        // Any source IP
```

Callback Processing

When messages arrive, the ORTE layer invokes the registered callback:

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void *recvCallBackParam) {
    Iec104driver_Instance *cl = (Iec104driver_Instance*)recvCallBackParam;
    iec_item_type *item1 = (iec_item_type*)vinstance;

    switch (info->status) {
        case NEW_DATA:
            if(!quite) {
                struct iec_item item2;
                rebuild_iec_item_message(&item2, item1);
                fifo_put(cl->fifo_monitor_direction, (char *)&item2, sizeof(struct iec_item)
            }
            break;
        case DEADLINE:
            // Deadline miss detected
            break;
    }
}
```

Sources: `src/drivers/iec104/iec104driver_instance.h` 187-241 `src/drivers/iec104/iec104driver_instance.cpp` 89-113

Topic Naming Convention

The system uses a structured naming scheme for ORTE topics that encodes routing information:

Per-Instance Topics

Each driver instance publishes and subscribes to dedicated topics:

Direction	Pattern	Example	Purpose
Monitor (Device→SCADA)	`fifo_monitor_direction{N}{protocol}`	`fifo_monitor_direction1iec104`	Data from driver instance 1
Control (SCADA→Device)	`fifo_control_direction{N}{protocol}`	`fifo_control_direction2modbus`	Commands to driver instance 2

Where `{N}` is the line number (instance ID + 1) and `{protocol}` is the protocol identifier.

Global Topics

For system-wide data aggregation:

Topic Name	Purpose	Subscribers
`fifo_global_monitor_direction`	Aggregated monitoring data	MQTT publisher, OPC server
`fifo_global_control_direction`	Global command channel	Unused in current implementation

Code Example

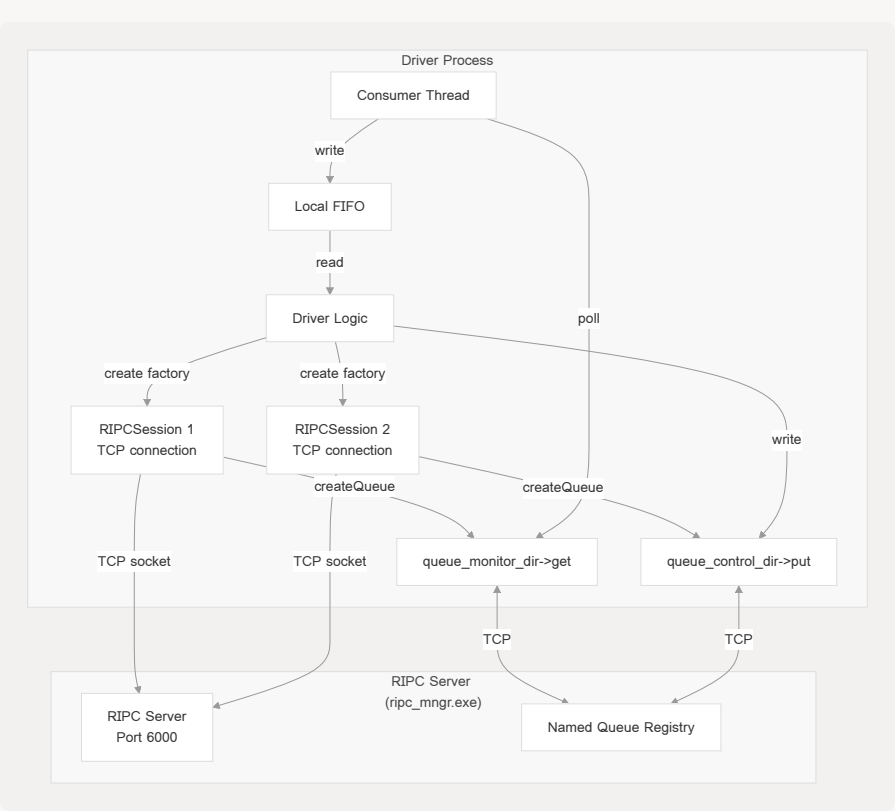
```
// From iec104driver_instance.h constructor
char fifo_control_name[150];
char fifo_monitor_name[150];
char str_instance_id[20];

itoa(instance_id + 1, str_instance_id, 10);

strcpy(fifo_control_name, "fifo_control_direction");
strcat(fifo_control_name, str_instance_id);
strcat(fifo_control_name, "iec104");

strcpy(fifo_monitor_name, "fifo_monitor_direction");
strcat(fifo_monitor_name, str_instance_id);
strcat(fifo_monitor_name, "iec104");
```

Sources: `src/drivers/iec104/iec104driver_instance.h` 159-170 `src/drivers/mqtt/protocol/mqtt_client_app.cpp` 163-172

RIPC Middleware (Optional)

RIPC (Real-time IPC) provides an alternative middleware implementation using TCP sockets and message queues. It is enabled via the `USE_RIPC_MIDDLEWARE` compile-time flag.

Architecture


Driver Process
Consumer Thread
write
Local FIFO
read
Driver Logic
poll
create factory
create factory
RIPCSession 1
TCP connection
RIPCSession 2
TCP connection
write
createQueue
createQueue
TCP socket
TCP socket
queue_monitor_dir->get
queue_control_dir->put
TCP
TCP
RIPC Server
(ripc_mngr.exe)
RIPC Server
Port 6000
Named Queue Registry

Initialization Pattern

```
#ifdef USE_RIPC_MIDDLEWARE
port = 6000;
hostname = "localhost";

factory1 = RIPCClientFactory::getInstance();
factory2 = RIPCClientFactory::getInstance();

session1 = factory1->create(hostname, port);
session2 = factory2->create(hostname, port);

queue_monitor_dir = session1->createQueue(fifo_monitor_name);
queue_control_dir = session2->createQueue(fifo_control_name);

arg.queue_monitor_dir = queue_monitor_dir;
arg.parent = this;

// Start consumer thread to bridge RIPC queue to local FIFO
CreateThread(NULL, 0, LPTHREAD_START_ROUTINE(consumer), (void*)&arg, 0, &threadid);
#endif
```

Consumer Thread Pattern

The consumer thread bridges RIPC queues to local FIFOs for uniform processing:

```
void consumer(void* pParam) {
    struct subs_args* arg = (struct subs_args*)pParam;
    struct iec_item item;
    RIPCObject objDesc(&item, sizeof(struct iec_item));

    while(1) {
        if(exit_consumer) {
            break;
        }

        arg->queue_monitor_dir->get(objDesc);  // Blocking read from RIPC

        fifo_put(arg->fifo_monitor_direction, (char *)&item, sizeof(struct iec_item));
    }
}
```

Sources: `src/drivers/iec104/iec104driver_instance.h` 108-120 `src/drivers/iec104/iec104driver_instance.cpp` 16-39 `src/drivers/iec104/iec104driver_instance.h` 172-184

Named Pipes for Process Control

Named pipes provide bidirectional communication between the monitor process and child driver processes for health monitoring and lifecycle management.

Pipe Naming Convention

Each driver instance creates a uniquely named pipe:

```
Pattern: {driver_name}_namedpipe{instance_id}
Examples:
  - iec104master_namedpipe1
  - modbus_master_namedpipe2
  - opc_client_da_namedpipe3
```

Communication Protocol

Named pipes carry `iec_item` structures for consistency, though they're used primarily for control messages:

Message Type	iec_type	Purpose
Keep-alive	C_EX_IT_1	Child process shutdown notification
Connection lost	C_LO_ST_1	Protocol connection failure
Commands	Various	Control commands forwarded to child

Timeout Detection

The monitor process detects child process failures through message timeout:

```
// From iec104driver_instance.cpp
wait_for_message++;

if(wait_for_message > 60*1000/TICK_MS) {  // 60 seconds timeout
    wait_for_message = 0;

    printf("iec104 master on line %d has been stopped...\n", instanceID + 1);

    QString msg;
    msg.sprintf("iec104 master on line %d has been stopped...", instanceID + 1);
    FailUnit(msg);

    State = STATE_FAIL;
    pConnect->TerminateChild();  // Kill unresponsive child
}
```

Child Process Health Reporting

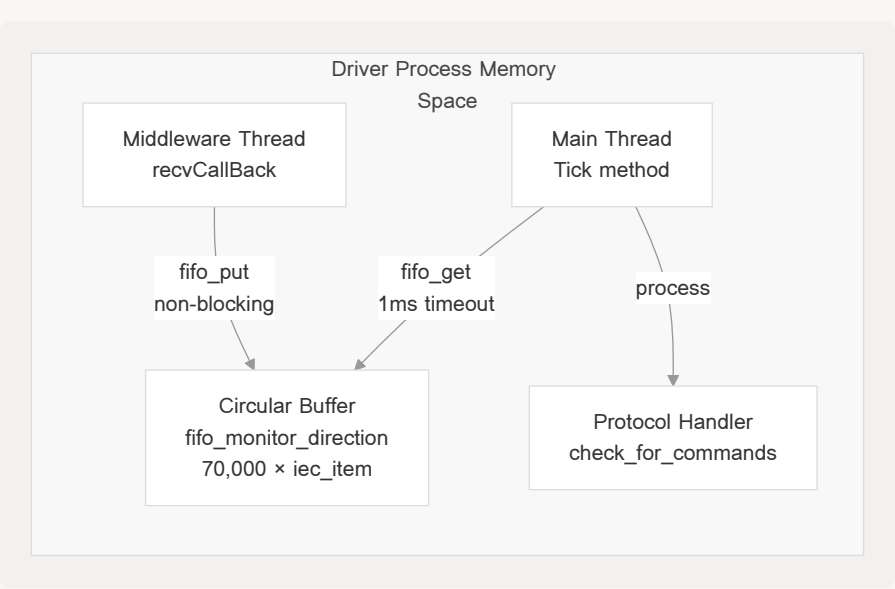
Child processes send periodic updates through the named pipe to prove liveness. The pipe communication is abstracted through the `DriverThread` class hierarchy.

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 1237-1253 `src/drivers/modbus/modbus_driver_instance.cpp` 1054-1067

Local FIFO Buffers

Each driver instance maintains a local FIFO (First-In-First-Out) circular buffer for decoupling middleware reception from protocol processing. This prevents slow protocol operations from blocking middleware threads.

FIFO Architecture


Driver Process Memory Space
Middleware Thread
recvCallBack
Main Thread
Tick method
fifo_put
non-blocking
fifo_get
1ms timeout
process
Circular Buffer
fifo_monitor_direction
70,000 × iec_item
Protocol Handler
check_for_commands

FIFO Configuration

```
// From iec104driver_instance.h
#define MAX_FIFO_SIZE 70000*(sizeof(struct iec_item))

char fifo_monitor_name[150];
strcat(fifo_monitor_name, "_fifo_");  // Unique suffix for local FIFO

fifo_monitor_direction = fifo_open(fifo_monitor_name, MAX_FIFO_SIZE, iec_call_exit_handler);
```

Producer: Middleware Callback

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void *recvCallBackParam) {
    Iec104driver_Instance *cl = (Iec104driver_Instance*)recvCallBackParam;
    iec_item_type *item1 = (iec_item_type*)vinstance;

    switch (info->status) {
        case NEW_DATA:
            if(!quite) {
                struct iec_item item2;
                rebuild_iec_item_message(&item2, item1);
                // Non-blocking write to FIFO
                fifo_put(cl->fifo_monitor_direction, (char *)&item2, sizeof(struct iec_item)
            }
            break;
    }
}
```

Consumer: Main Thread

```
void Iec104driver_Instance::get_items_from_local_fifo(void) {
    unsigned char buf[sizeof(struct iec_item)];
    int len;
    const unsigned wait_limit_ms = 1;
    struct iec_item* p_item;

    // Process up to 50 messages per tick to prevent starvation
    for(int i = 0; (len = fifo_get(fifo_monitor_direction, (char*)buf,
                                   sizeof(struct iec_item), wait_limit_ms)) >= 0; i += 1) {
        p_item = (struct iec_item*)buf;

        // Validate checksum
        unsigned char rc = clearCrc((unsigned char *)buf, sizeof(struct iec_item));
        if(rc != 0) {
            ExitProcess(1);
        }

        // Process message based on iec_type...

        if(i > 50) {
            break;  // Yield to prevent monopolizing CPU
        }
    }
}
```

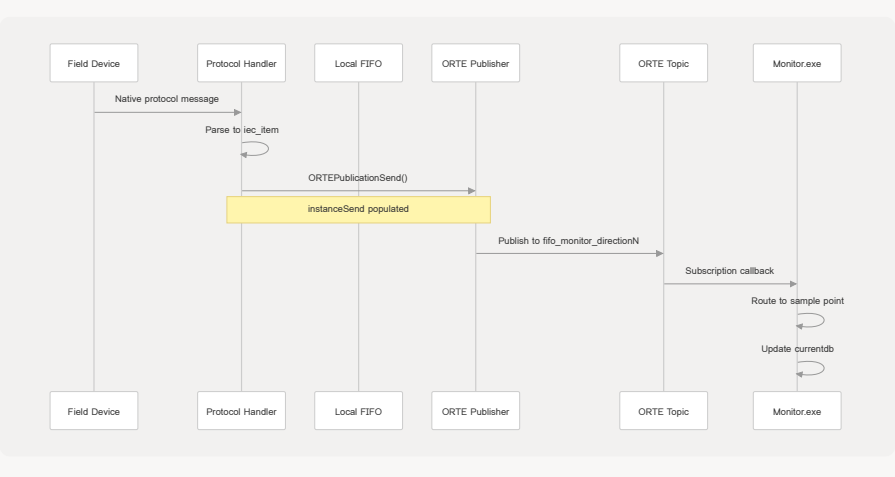
Sources: `src/drivers/iec104/iec104driver_instance.h` 244-248 `src/drivers/iec104/iec104driver_instance.cpp` 753-809 `src/middleware/rtps/include/fifoc.h`

Bidirectional Data Flow

The middleware supports bidirectional communication with separate channels for monitoring (device→SCADA) and control (SCADA→device) directions.

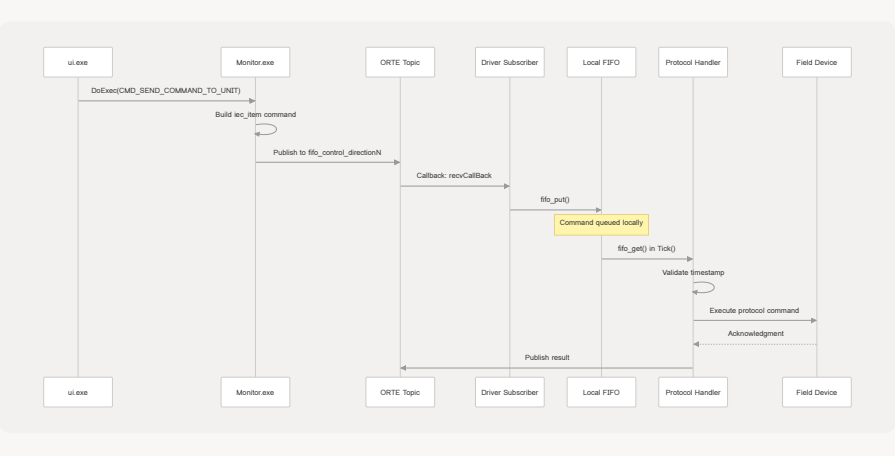
Monitor Direction Flow

Data flows from field devices to the SCADA system:


Field Device
Protocol Handler
Local FIFO
ORTE Publisher
ORTE Topic
Monitor.exe
Native protocol message
Parse to iec_item
ORTEPublicationSend()
instanceSend populated
Publish to fifo_monitor_directionN
Subscription callback
Route to sample point
Update currentdb
Field Device
Protocol Handler
Local FIFO
ORTE Publisher
ORTE Topic
Monitor.exe

Control Direction Flow

Commands flow from the HMI to field devices:


ui.exe
Monitor.exe
ORTE Topic
Driver Subscriber
Local FIFO
Protocol Handler
Field Device
DoExec(CMD_SEND_COMMAND_TO_UNIT)
Build iec_item command
Publish to fifo_control_directionN
Callback: recvCallBack
fifo_put()
Command queued locally
fifo_get() in Tick()
Validate timestamp
Execute protocol command
Acknowledgment
Publish result
ui.exe
Monitor.exe
ORTE Topic
Driver Subscriber
Local FIFO
Protocol Handler
Field Device

Dual-Publisher Pattern

Some drivers publish to both local and global topics:

```
// Send to local monitor direction
memset(&instanceSend, 0x00, sizeof(iec_item_type));
instanceSend.iec_type = p_item->iec_type;
memcpy(&(instanceSend.iec_obj), &(p_item->iec_obj), sizeof(struct iec_object));
instanceSend.msg_id = p_item->msg_id;
instanceSend.checksum = p_item->checksum;
ORTEPublicationSend(publisher);

// Also send to global monitor direction
if(global_publisher) {
    memset(&global_instanceSend, 0x00, sizeof(iec_item_type));
    global_instanceSend.iec_type = p_item->iec_type;
    memcpy(&(global_instanceSend.iec_obj), &(p_item->iec_obj), sizeof(struct iec_object));
    global_instanceSend.cause = p_item->cause;
    global_instanceSend.msg_id = p_item->msg_id;
    ORTEPublicationSend(global_publisher);
}
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 811-828 `src/drivers/modbus/modbus_driver_instance.cpp` 652-670

Memory Management for ORTE

ORTE uses a custom memory manager (`memmgr`) to provide deterministic allocation from a statically-sized pool, critical for real-time performance.

Pool Configuration

```
// From memmgr.h
#define POOL_SIZE 1024 * 1024 * 20  // 20 MB pool
#define MIN_POOL_ALLOC_QUANTAS 16

static byte* pool = NULL;
static ulong pool_free_pos = 0;
```

Initialization

```
void memmgr_init() {
    base.s.next = 0;
    base.s.size = 0;
    freep = 0;
    pool_free_pos = 0;

    mut = ins_mutex_new();  // Thread-safe allocator

    pool = (byte*)malloc(POOL_SIZE);
    memset(pool, 0, POOL_SIZE);

    if(pool == NULL) {
        printf("Failure in allocation of pool memory\n");
    }
}
```

Thread-Safe Allocation

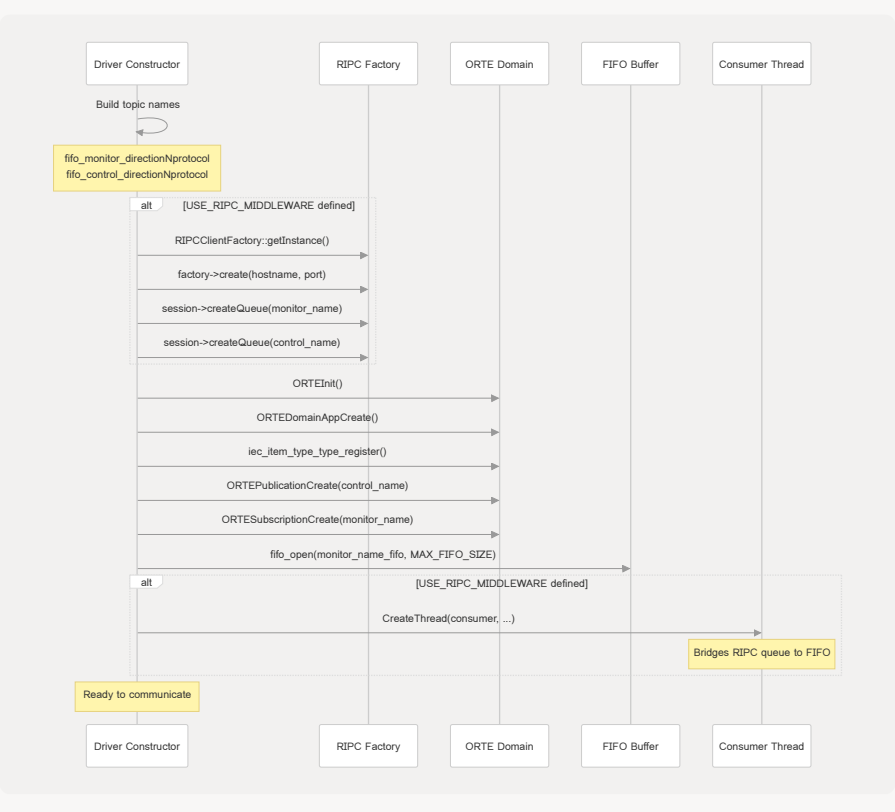
```
void* memmgr_alloc_mt(ulong nbytes) {
    void* p;
    ins_mutex_acquire(mut);  // Lock
    p = memmgr_alloc(nbytes);
    ins_mutex_release(mut);  // Unlock
    return p;
}

void memmgr_free_mt(void* ap) {
    ins_mutex_acquire(mut);  // Lock
    memmgr_free(ap);
    ins_mutex_release(mut);  // Unlock
}
```

The memory manager uses a first-fit allocation strategy with coalescing of adjacent free blocks to minimize fragmentation. Quantized allocation (in units of `mem_header_t` size) reduces external fragmentation.

Sources: `src/middleware/rtps/librtps/memmgr.c` 1-308 `src/middleware/rtps/include/memmgr.h` 1-150

Initialization Sequence

The typical initialization sequence for a driver instance with full middleware stack:


Driver Constructor
RIPC Factory
ORTE Domain
FIFO Buffer
Consumer Thread
Build topic names
fifo_monitor_directionNprotocol
fifo_control_directionNprotocol
alt
[USE_RIPC_MIDDLEWARE defined]
RIPCClientFactory::getInstance()
factory->create(hostname, port)
session->createQueue(monitor_name)
session->createQueue(control_name)
ORTEInit()
ORTEDomainAppCreate()
iec_item_type_type_register()
ORTEPublicationCreate(control_name)
ORTESubscriptionCreate(monitor_name)
fifo_open(monitor_name_fifo, MAX_FIFO_SIZE)
alt
[USE_RIPC_MIDDLEWARE defined]
CreateThread(consumer, ...)
Bridges RIPC queue to FIFO
Ready to communicate
Driver Constructor
RIPC Factory
ORTE Domain
FIFO Buffer
Consumer Thread

Cleanup Sequence

```
~Iec104driver_Instance() {
    #ifdef USE_RIPC_MIDDLEWARE
    exit_consumer = 1;
    fifo_close(fifo_monitor_direction);
    queue_monitor_dir->close();
    queue_control_dir->close();
    session1->close();
    delete session1;
    session2->close();
    delete session2;
    #endif

    ORTEDomainAppDestroy(domain);
    domain = NULL;
}
```

Sources: `src/drivers/iec104/iec104driver_instance.h` 142-260 `src/drivers/iec104/iec104driver_instance.h` 262-289

Message Routing in Monitor Process

The monitor process (`monitor.exe`) subscribes to all driver instance topics and routes messages based on IOA (Information Object Address) to the appropriate sample points.

Routing Table Structure

Each driver instance maintains an IOA-to-name map:

```
// From iec104driver_instance.h
typedef QMap<int, QString> IOANameMap;
IOANameMap ioa_name_map;

// Population from TAGS table
QString cmd = "select * from TAGS where UNIT='"+ Name + "';";
GetConfigureDb()->DoExec(this, cmd, tSetSamplePointNameIOAMap);

// In callback:
int ioa = GetConfigureDb()->GetInt("IOA");
QString SamplePointName = UndoEscapeSQLText(GetConfigureDb()->GetString("NAME"));
ioa_name_map.insert(ioa, SamplePointName);
```

Message Processing

```
void Iec104driver_Instance::get_items_from_local_fifo(void) {
    // ... FIFO read code ...

    IOANameMap::Iterator it;
    it = ioa_name_map.find(p_item->iec_obj.ioa);

    switch(p_item->iec_type) {
        case M_ME_NC_1:  // Measured value, short floating point
            if(it != ioa_name_map.end()) {
                sp_name = it.data();

                iec_type13 var = p_item->iec_obj.o.type13;
                IECValue v(VALUE_TAG, &var, M_ME_NC_1);
                IECValueList l;
                l.insert(l.end(), v);

                PostList(sp_name, l);  // Update sample point in currentdb
            }
            break;
        // ... other message types ...
    }
}
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 831-956 `src/drivers/iec104/iec104driver_instance.cpp` 674-686

Error Handling and Fault Detection

The middleware layer implements multiple levels of error detection and recovery.

Checksum Validation

Every message is validated upon receipt:

```
unsigned char rc = clearCrc((unsigned char *)buf, sizeof(struct iec_item));
if(rc != 0) {
    ExitProcess(1);  // Fatal error - terminate process
}
```

Connection State Detection

Drivers detect connection failures through special message types:

```
case C_LO_ST_1:  // Connection lost
    if(State != STATE_FAIL) {
        printf("iec104 master on line %d has lost connection...\n", instanceID + 1);

        QString msg;
        msg.sprintf("iec104 master on line %d has lost connection...", instanceID + 1);
        FailUnit(msg);  // Notify alarm system

        State = STATE_FAIL;
    }
    break;
```

Automatic Recovery

When connection is restored:

```
if(State == STATE_FAIL) {
    if(p_item->iec_type != C_LO_ST_1) {
        QString msg;
        msg.sprintf("IEC104 master on line %d is now connected to IEC104 server.", instanceI
        UnFailUnit(msg);  // Clear alarm
        State = STATE_ASK_GENERAL_INTERROGATION;  // Resynchronize
    }
}
```

ORTE Registration Failure

```
void onRegFail(void *param) {
    printf("registration to a manager failed\n");
    regfail = 1;
}

ORTEDomainInitEvents(&events);
events.onRegFail = onRegFail;
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 804-809 `src/drivers/iec104/iec104driver_instance.cpp` 766-783 `src/drivers/iec104/iec104driver_instance.cpp` 1210-1223

Thread Safety Considerations

The middleware layer operates in a multi-threaded environment with careful synchronization:

Thread Model

Thread	Owner	Purpose	Synchronization
Main thread	Process	Tick() processing, FIFO reading	Qt event loop
ORTE receive thread	ORTE library	Callback invocation	Internal ORTE locks
ORTE send thread	ORTE library	Message transmission	Internal ORTE locks
Consumer thread	Driver (RIPC mode)	RIPC→FIFO bridge	FIFO thread-safe operations
Memory manager mutex	memmgr	Allocation serialization	Windows Critical Section

Lock-Free FIFO Operations

The FIFO implementation provides lock-free operations suitable for producer-consumer scenarios. The `fifo_put` and `fifo_get` operations are designed to be safe when called from different threads:

```
// Producer (ORTE callback thread)
fifo_put(cl->fifo_monitor_direction, (char *)&item2, sizeof(struct iec_item));

// Consumer (main thread)
len = fifo_get(fifo_monitor_direction, (char*)buf, sizeof(struct iec_item), wait_limit_ms);
```

Memory Manager Synchronization

```
// Thread-safe wrapper using Windows Critical Section
void memmgr_free_mt(void* ap) {
    ins_mutex_acquire(mut);  // EnterCriticalSection
    memmgr_free(ap);
    ins_mutex_release(mut);  // LeaveCriticalSection
}
```

Qt Signal-Slot Thread Safety

The `PostList()` and `PostValue()` methods use Qt's signal-slot mechanism, which is inherently thread-safe for cross-thread communication.

Sources: `src/middleware/rtps/librtps/memmgr.c` 234-242 `src/middleware/rtps/librtps/memmgr.c` 302-307 `src/middleware/rtps/include/memmgr.h` 103-157

Performance Characteristics

Message Throughput

The system is designed to handle high-frequency updates:

- **FIFO capacity:** 70,000 messages (configurable)
- **Per-tick processing limit:** 50 messages to prevent CPU monopolization
- **Typical message size:** `sizeof(struct iec_item)` ≈ 128 bytes
- **Tick interval:** 300ms (TICK_MS) for IEC104 driver

Latency Considerations

Stage	Typical Latency	Notes
Protocol reception	1-100ms	Depends on field device
ORTE publication	<1ms	Local shared memory
ORTE subscription callback	<1ms	Same process space
FIFO write	<1ms	Lock-free operation
FIFO read (polled)	0-1ms	1ms timeout per poll
Sample point update	1-10ms	Database operation

Scalability

The architecture scales horizontally:

- Each driver instance operates independently
- ORTE multicast reduces network load
- No central message broker bottleneck (when using ORTE)
- RIPC mode introduces centralized `ripc_mngr.exe` server (potential bottleneck)

Sources: `src/drivers/iec104/iec104driver_instance.h` 56 `src/drivers/iec104/iec104driver_instance.cpp` 1231-1234

Configuration Examples

Driver Instance Configuration

From `iec104driver_instance.h` constructor:

```
Iec104driver_Instance(Driver *parent, const QString &name, int instance_id) :
    DriverInstance(parent,name),
    fFail(0),
    Countdown(1),
    State(STATE_RESET),
    InTick(0),
    Retry(0),
    Sp(0),
    IecItems(1),
    Values(NULL),
    ParentDriver(parent),
    msg_sent_in_control_direction(0),
    instanceID(instance_id),
    pConnect(NULL),
    wait_for_message(0),
    msg_received_in_monitor_direction(0)
{
    // Create unique topic names using instance_id
    char str_instance_id[20];
    itoa(instance_id + 1, str_instance_id, 10);

    strcpy(fifo_control_name, "fifo_control_direction");
    strcat(fifo_control_name, str_instance_id);
    strcat(fifo_control_name, "iec104");

    // Initialize ORTE domain
    domain = ORTEDomainAppCreate(ORTE_DEFAULT_DOMAIN, &dp, &events, ORTE_FALSE);

    // Create publication for control direction
    publisher = ORTEPublicationCreate(domain, fifo_control_name, "iec_item_type",
                                      &instanceSend, &persistence, strength,
                                      NULL, NULL, NULL);

    // Create subscription for monitor direction
    subscriber = ORTESubscriptionCreate(domain, IMMEDIATE, BEST_EFFORTS,
                                        fifo_monitor_name, "iec_item_type",
                                        &instanceRecv, &deadline, &minimumSeparation,
                                        recvCallBack, this, smIPAddress);

    // Create local FIFO
    fifo_monitor_direction = fifo_open(fifo_monitor_name, MAX_FIFO_SIZE,
                                       iec_call_exit_handler);
}
```

MQTT Global Publisher Configuration

From `mqtt_client_app_publisher.cpp`:

```
MQTT_client_imp_publisher(char* broker_host_name, char* line_number) {
    strcpy(fifo_monitor_name, "fifo_global_monitor_direction");
    strcpy(fifo_control_name, "fifo_global_control_direction");

    domain = ORTEDomainAppCreate(ORTE_DEFAULT_DOMAIN, &dp, &events, ORTE_FALSE);

    publisher = ORTEPublicationCreate(domain, fifo_control_name, "iec_item_type",
                                      &instanceSend, &persistence, strength,
                                      NULL, NULL, NULL);

    subscriber = ORTESubscriptionCreate(domain, IMMEDIATE, BEST_EFFORTS,
                                        fifo_monitor_name, "iec_item_type",
                                        &instanceRecv, &deadline, &minimumSeparation,
                                        recvCallBack, this, smIPAddress);
}
```

Sources: `src/drivers/iec104/iec104driver_instance.h` 142-260 `src/drivers/mqtt/protocol/mqtt_client_app_publisher.cpp` 109-193

Debugging and Diagnostics

Message Tracking

Each message includes a sequence number (`msg_id`) for tracking:

```
item_to_send.msg_id = msg_sent_in_control_direction++;
```

Lost messages can be detected by checking for non-consecutive sequence numbers, though this is currently marked as TODO in the code.

ORTE Domain Inspection

The ORTE domain can be inspected using external tools that connect to the multicast group (typically 239.255.0.1).

FIFO Statistics

The memory manager provides statistics when compiled with `DEBUG_MEMMGR_SUPPORT_STATS`:

```
void memmgr_print_stats() {
    printf("------ Memory manager stats ------\n\n");
    printf("Pool: free_pos = %lu (%lu bytes left)\n\n",
           pool_free_pos, POOL_SIZE - pool_free_pos);
    // ... detailed block listing ...
}
```

State Machine Tracing

Drivers include state machine debugging:

```
switch(State) {
    case STATE_RESET:
        fprintf(stderr, "State = STATE_RESET\n");
        fflush(stderr);
        State = STATE_INIT_DB;
        break;
    // ...
}
```

Sources: `src/middleware/rtps/librtps/memmgr.c` 87-130 `src/drivers/opc_client_da/opc_client_da_instance.cpp` 654-689

Database Architecture

> Relevant source files

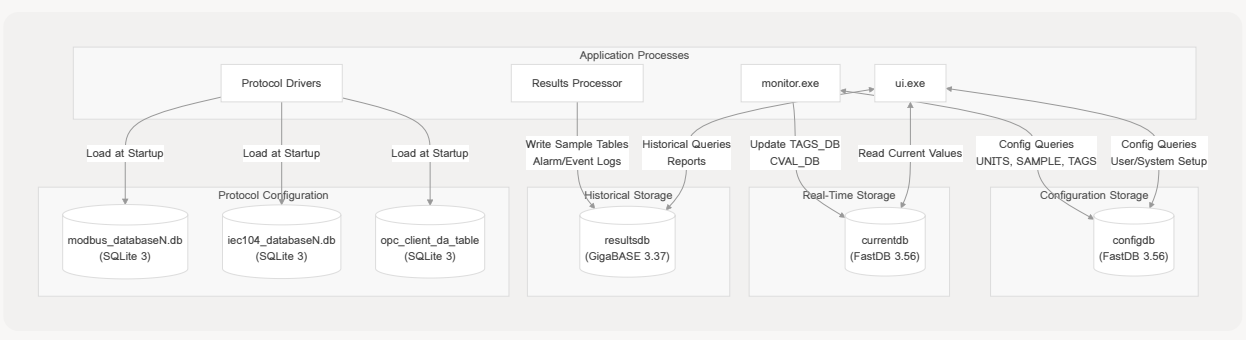
Purpose and Scope

This document describes the database architecture of IndigoSCADA, including the multi-database strategy, schema organization, and data flow patterns. The system uses three different database engines optimized for different workloads: FastDB for configuration and real-time data, GigaBASE for historical storage, and SQLite for protocol-specific configurations.

For information about how data flows from field devices through the databases to the HMI, see Data Flow Pipeline. For details on how databases are accessed from application code, see Monitor Process and Results Processing and Alarming.

Multi-Database Strategy

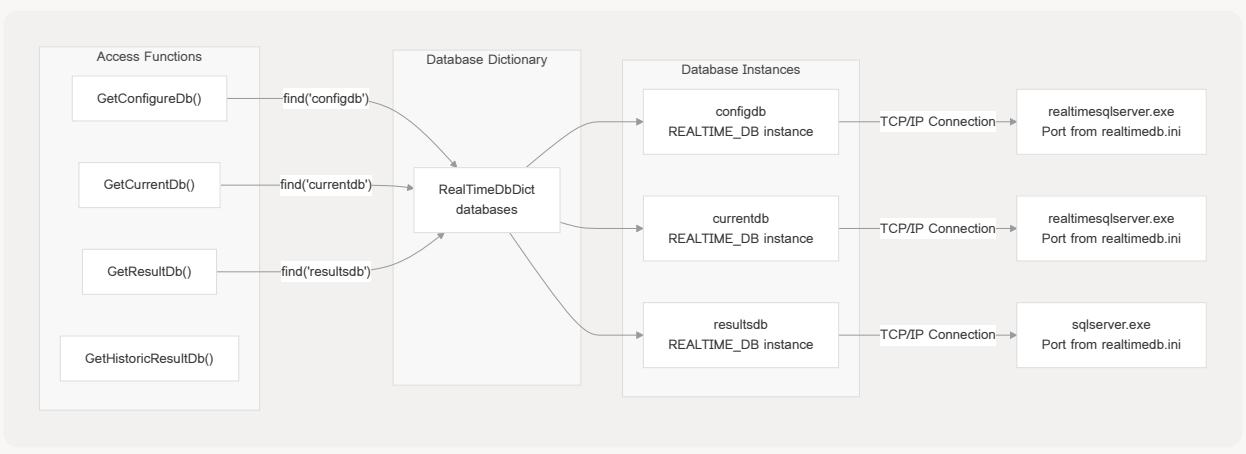
IndigoSCADA employs a sophisticated multi-database architecture where each database engine is selected based on its strengths:


Application Processes
Protocol Drivers
Results Processor
monitor.exe
ui.exe
Load at Startup
Load at Startup
Load at Startup
Write Sample Tables Alarm/Event Logs
Historical Queries Reports
Update TAGS_DB CVAL_DB
Read Current Values
Config Queries UNITS, SAMPLE, TAGS
Config Queries User/System Setup
Protocol Configuration
modbus_databaseN.db (SQLite 3)
iec104_databaseN.db (SQLite 3)
opc_client_da_table (SQLite 3)
Historical Storage
resultsdb (GigaBASE 3.37)
Real-Time Storage
currentdb (FastDB 3.56)
Configuration Storage
configdb (FastDB 3.56)

Sources: `src/database/realtimedb.cpp` 1-300 `src/daq/daq.dsw` 1-100 `Diagram 5 from high-level architecture`

Database Access Layer

IndigoSCADA provides a unified access layer through the `QSDatabase` interface, with accessor functions defined in the real-time database module:


Access Functions
GetConfigureDb()
GetCurrentDb()
GetResultDb()
GetHistoricResultDb()
find('configdb')
find('currentdb')
find('resultsdb')
Database Dictionary
RealTimeDbDict databases
Database Instances
configdb REALTIME_DB instance
currentdb REALTIME_DB instance
resultsdb REALTIME_DB instance
TCP/IP Connection
TCP/IP Connection
TCP/IP Connection
realtimesqlserver.exe Port from realtimedb.ini
realtimesqlserver.exe Port from realtimedb.ini
sqlserver.exe Port from realtimedb.ini

The database dictionary is populated at startup by reading the `realtimedb.ini` configuration file, which specifies connection parameters for each database server:

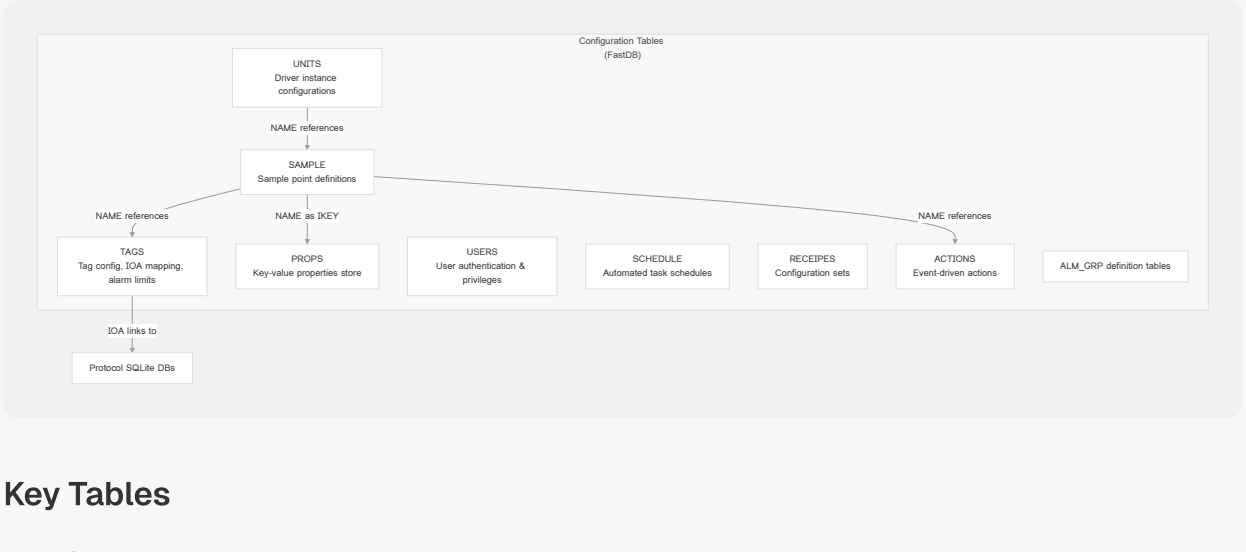
Configuration Key	Purpose
`[databases]` section	Lists online databases
`address`	Database server IP address
`port`	Database server TCP port
`is_low_freq`	Whether database has low update frequency
`user`	Database username (from `[rtsqlserver]` section)
`password`	Database password (from `[rtsqlserver]` section)

Sources: `src/database/realtimedb.cpp` 66-242 `src/database/realtimedb.h` 1-50

Configuration Database (configdb)

The configuration database stores all system configuration using FastDB's in-memory architecture for rapid access. It is served by `realtimesqlserver.exe` and persisted to disk as `configdb.fdb`.

Schema Structure


Configuration Tables (FastDB)
UNITS Driver instance configurations
NAME references
SAMPLE Sample point definitions
NAME references
NAME as IKEY
NAME references
TAGS Tag config, IOA mapping, alarm limits
PROPS Key-value properties store
USERS User authentication & privileges
SCHEDULE Automated task schedules
RECEIPES Configuration sets
ACTIONS Event-driven actions
ALM_GRP definition tables
IOA links to
Protocol SQLite DBs

Key Tables

UNITS Table:

- Defines each driver instance (unit)
- Fields include: `NAME`, `UNITTYPE`, `ENABLED`, configuration properties
- One row per protocol driver instance

SAMPLE Table:

- Defines sample points (data points to monitor/control)
- Fields: `NAME`, `COMMENT`, `UNIT`, `QTYPE` (measurement type), `IPINDEX` (input index), `ENABLED`, `FILEABLE`, `LOGMODE`, `RETRIGGER`, `ALARMTHRESHOLD`
- Each sample point has an associated unit

TAGS Table:

- Central registry mapping sample points to protocol addresses
- Critical fields:
  - `NAME` - Sample point name
  - `TAGNAME` - Tag identifier (e.g., "value", "0.5")
  - `IOA` - Information Object Address (protocol-specific address)
  - `UPPERALARM`, `UPPERWARNING`, `LOWERWARNING`, `LOWERALARM` - Alarm thresholds
  - `ENABLED` - Whether tag is active
  - `PARAMS` - Additional protocol-specific parameters
  - `UNIT` - Associated unit name

PROPS Table:

- Generic key-value store with structure: `(SKEY, IKEY, DVAL)`
- Used for:
  - System properties: `SKEY='System'`
  - Sample point properties: `SKEY='SAMPLEPROPS'`, `IKEY=<sample name>`
  - Hardware semaphores: `SKEY='Semaphore'`

USERS Table:

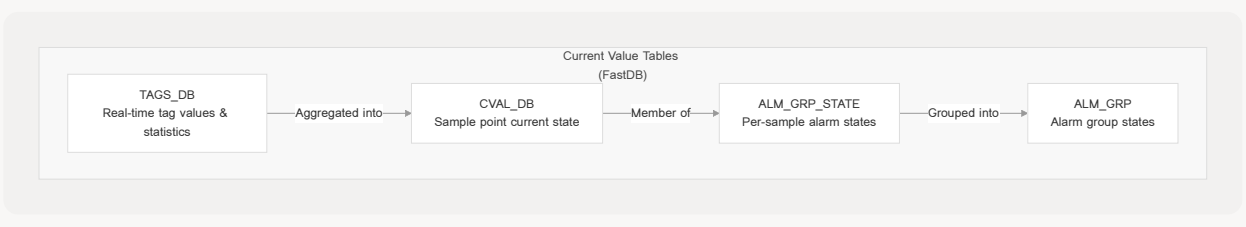
- User authentication and authorization
- Fields include username, password hash, privilege level
- Privilege constants: `PRIVS_ALL`, `PRIVS_ACK_ALARMS`, `PRIVS_FACTORY_ADMIN`, `PRIVS_CAN_EXIT`, `PRIVS_REPORTS`

Sources: `src/ui/sysdialogs/SampleCfg.cpp` 1-200 `src/monitor/monitor.cpp` 300-400 `src/ui/user.cpp` 150-170

Current Values Database (currentdb)

The current values database maintains real-time state using FastDB's high-performance in-memory engine. Updates occur every 100ms during the monitor tick cycle.

Schema Structure


Current Value Tables (FastDB)
TAGS_DB Real-time tag values & statistics
Aggregated into
CVAL_DB Sample point current state
Member of
ALM_GRP_STATE Per-sample alarm states
Grouped into
ALM_GRP Alarm group states

TAGS_DB Table:

- One row per tag per sample point
- Updated when tag values change
- Fields:
  - `NAME` - Sample point name
  - `TAGNAME` - Tag identifier
  - `UPDTIME` - Last update timestamp
  - `VAL` - Current value
  - `MINVAL`, `MAXVAL` - Statistical bounds
  - `SUMVAL`, `SUM2VAL`, `NVAL` - Statistical accumulators
  - `STATE` - Current alarm state
  - `SEQNO` - Sequence number for time ordering

```
-- Example update from monitor tick
update TAGS_DB set
    UPDTIME=<timestamp>,
    VAL=<current_value>,
    MINVAL=<min>,
    MAXVAL=<max>,
    SUMVAL=<sum>,
    SUM2VAL=<sum_of_squares>,
    NVAL=<sample_count>,
    STATE=<alarm_state>,
    SEQNO=<sequence_number>
where NAME='<sample_name>' and TAGNAME='<tag_name>';
```

CVAL_DB Table:

- One row per sample point
- Aggregate state and statistics
- Fields:
  - `NAME` - Sample point name
  - `UPDTIME` - Last update time
  - `ALMTIME` - Last alarm time
  - `NMEASURE`, `NALARM`, `NWARNING` - Event counters
  - `STATE` - Current alarm state (FailureLevel, WarningLevel, AlarmLevel)
  - `ACKFLAG` - Acknowledgment flag
  - `COMMENT` - Operator comment
  - `FAILTIME` - Last failure time
  - `SEQNO` - Sequence number

ALM_GRP and ALM_GRP_STATE Tables:

- Support alarm grouping functionality
- Allow operators to view alarms by logical groupings
- `ALM_GRP` tracks overall group state
- `ALM_GRP_STATE` tracks individual sample point states within groups

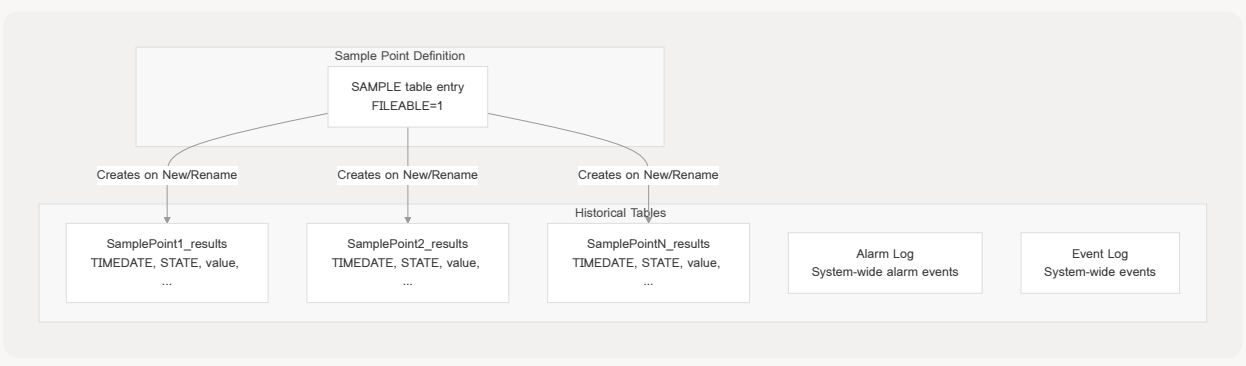
Sources: `src/monitor/monitor.cpp` 349-450 `src/monitor/monitor.cpp` 560-665 `src/database/results.h` 1-100

Results and Historical Database (resultsdb)

The historical database uses GigaBASE for efficient long-term storage and time-series compression. It is served by `sqlserver.exe` and persisted to `resultsdb.dbs`.

Dynamic Table Structure

Each fileable sample point gets its own results table, created dynamically when the sample point is configured:


Sample Point Definition
SAMPLE table entry FILEABLE=1
Creates on New/Rename
Creates on New/Rename
Creates on New/Rename
Historical Tables
SamplePoint1_results TIMEDATE, STATE, value, ...
SamplePoint2_results TIMEDATE, STATE, value, ...
SamplePointN_results TIMEDATE, STATE, value, ...
Alarm Log System-wide alarm events
Event Log System-wide events

Sample Point Results Tables:

Each table is created with this structure:

```
create table SamplePointName (
    TIMEDATE <DATE_TIME_COL_TYPE>,
    STATE int4,
    value real8,
    -- additional tags as defined for sample point
);

create index on SamplePointName.TIMEDATE;
```

The table name is the sample point name, and columns include:

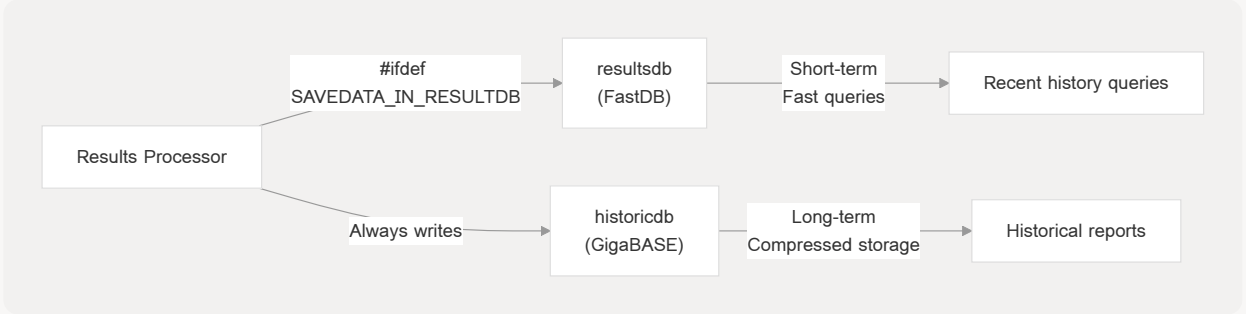
- `TIMEDATE` - Timestamp of the measurement
- `STATE` - Alarm state at time of measurement
- One column per tag defined for that sample point (typically includes "value" tag)

Alarm and Event Logs:

- System-wide logs for audit and reporting
- Written by Results Processor
- Include timestamps, sample point names, alarm levels, and event descriptions

Dual Database Storage:

The system can optionally store data in both FastDB (resultsdb) and GigaBASE (historicdb):

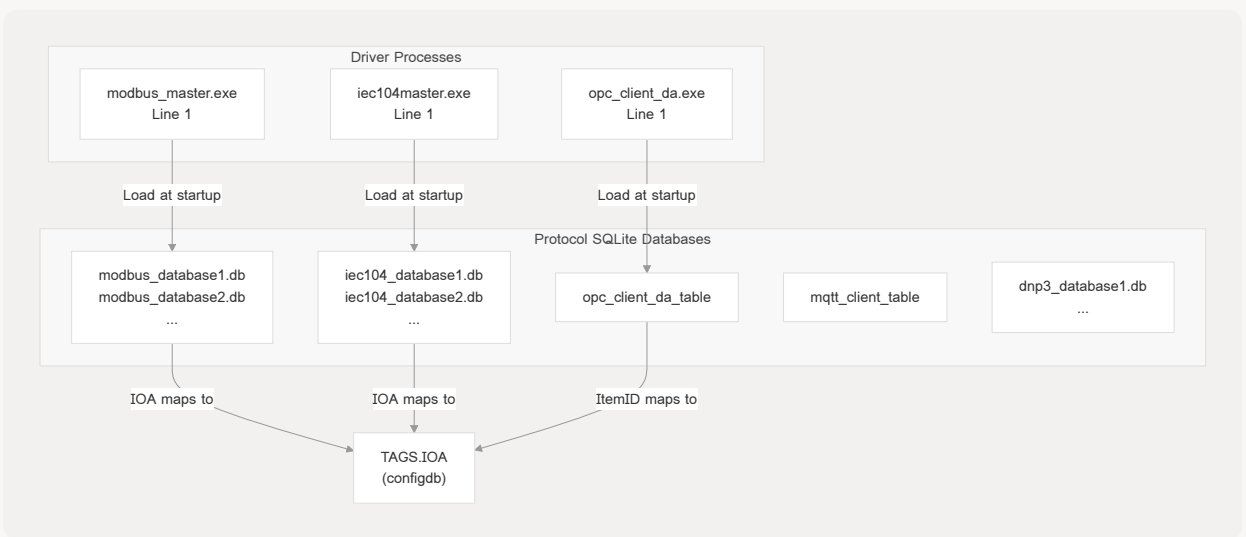

Results Processor
#ifdef SAVEDATA_IN_RESULTDB
Always writes
resultsdb (FastDB)
historicdb (GigaBASE)
Short-term Fast queries
Long-term Compressed storage
Recent history queries
Historical reports

Sources: `src/ui/sysdialogs/SampleCfg.cpp` 195-232 `src/drivers/driver.cpp` 117-157 `src/database/results.h` 313-456

Protocol Configuration Databases (SQLite)

Each protocol driver instance stores its specific configuration in a SQLite database, separate from the main configuration to allow protocol-specific data structures.

SQLite Database Organization


Driver Processes
modbus_master.exe Line 1
iec104master.exe Line 1
opc_client_da.exe Line 1
Load at startup
Load at startup
Load at startup
Protocol SQLite Databases
modbus_database1.db modbus_database2.db ...
iec104_database1.db iec104_database2.db ...
opc_client_da_table
mqtt_client_table
dnp3_database1.db ...
IOA maps to
IOA maps to
ItemID maps to
TAGS.IOA (configdb)

Database Naming Convention

Protocol databases follow the naming pattern: `<protocol>_database<N>.db` where `N` is the line number (driver instance ID + 1).

Common Fields Across Protocol Databases:

- Address mapping (IOA, register address, node address, etc.)
- Protocol-specific parameters (function codes, data types, scaling)
- Polling intervals and priorities
- Connection parameters

Modbus Database Example:

- Register addresses (holding registers, input registers, coils, discrete inputs)
- Function codes (0x03, 0x04, 0x01, 0x02)
- Data type mappings (INT16, UINT16, FLOAT32, etc.)
- Slave ID

IEC 104 Database Example:

- Information Object Addresses (IOA)
- CASDU (Common Address of ASDU)
- Type IDs (M_SP_NA_1, M_ME_NC_1, etc.)
- Spontaneous/polled flags

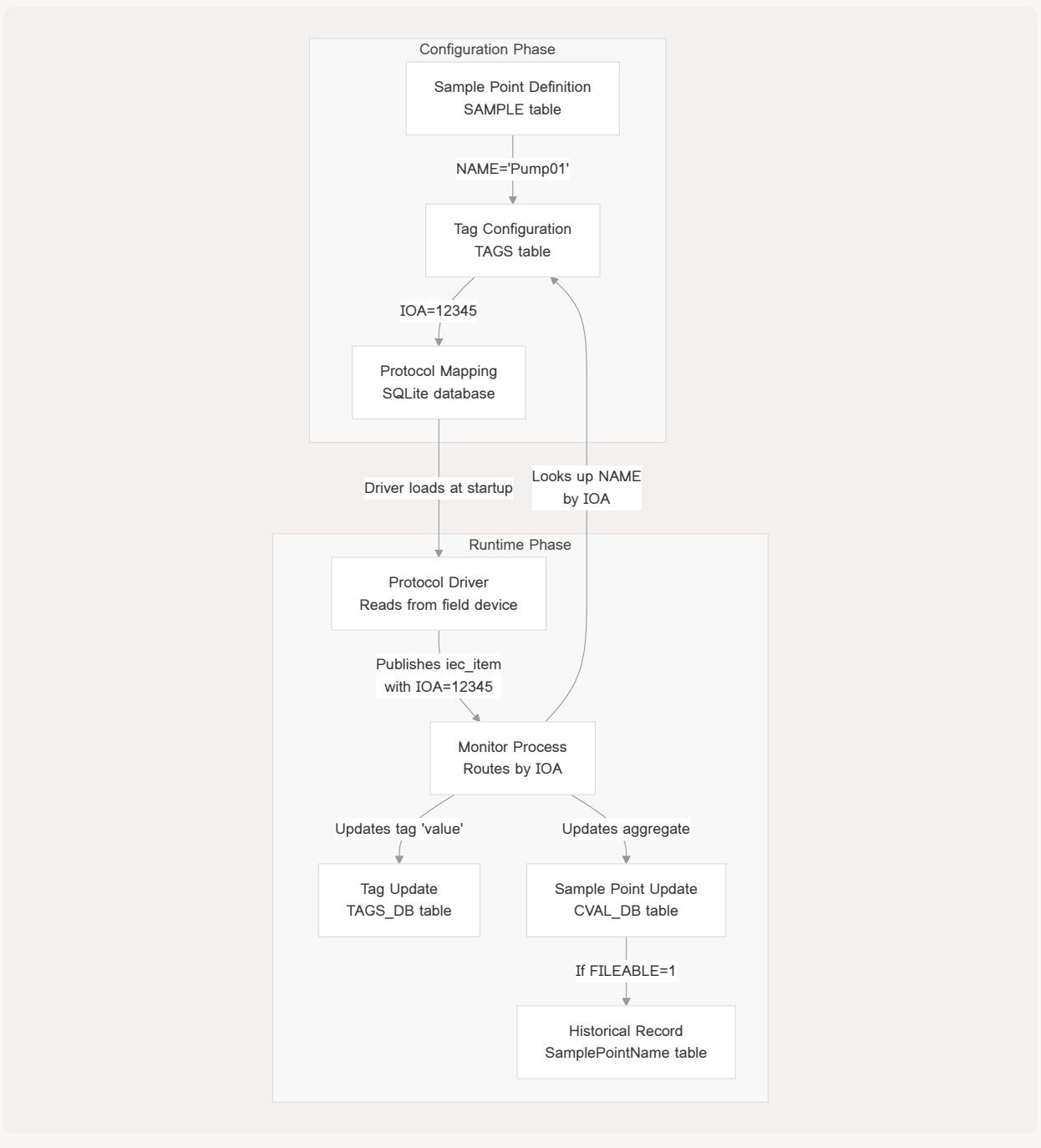
OPC DA Database Example:

- OPC Item IDs
- Server CLSID
- Update rates
- Deadbands

Sources: `src/drivers/modbus/protocol/modbus_master.dsp` 1-147 `src/drivers/iec104/iec104driverthread.cpp` 1-100 `src/ui/sysdialogs/SampleCfg.cpp` 133-140

Sample Point to Database Table Mapping

The following diagram illustrates how a sample point's data flows through the database architecture:


Configuration Phase
Sample Point Definition SAMPLE table
NAME='Pump01'
Tag Configuration TAGS table
IOA=12345
Protocol Mapping SQLite database
Driver loads at startup
Looks up NAME by IOA
Runtime Phase
Protocol Driver Reads from field device
Publishes iec_item with IOA=12345
Monitor Process Routes by IOA
Updates tag 'value'
Updates aggregate
Tag Update TAGS_DB table
Sample Point Update CVAL_DB table
If FILEABLE=1
Historical Record SamplePointName table

Example Flow for Sample Point "Pump01":

1. **Configuration:** Administrator creates sample point "Pump01" with unit "ModbusRTU1", type "M_ME_NC_1" (analog measurement)
2. **Tag Definition:** System creates tag "Pump01.value" with IOA=12345 in TAGS table
3. **Protocol Mapping:** Administrator maps IOA 12345 to Modbus holding register 1000 in `modbus_database1.db`
4. **Driver Load:** `modbus_master.exe` loads SQLite configuration, knows to read register 1000
5. **Data Acquisition:** Driver reads register 1000, publishes `iec_item` with IOA=12345, value=42.5
6. **Monitor Routing:** Monitor receives `iec_item`, queries TAGS by IOA, finds "Pump01.value"
7. **Current Update:** Updates TAGS_DB row for NAME='Pump01', TAGNAME='value' with VAL=42.5
8. **Aggregate Update:** Updates CVAL_DB row for NAME='Pump01' with statistics
9. **Historical Storage:** If FILEABLE=1, inserts row into Pump01_results table

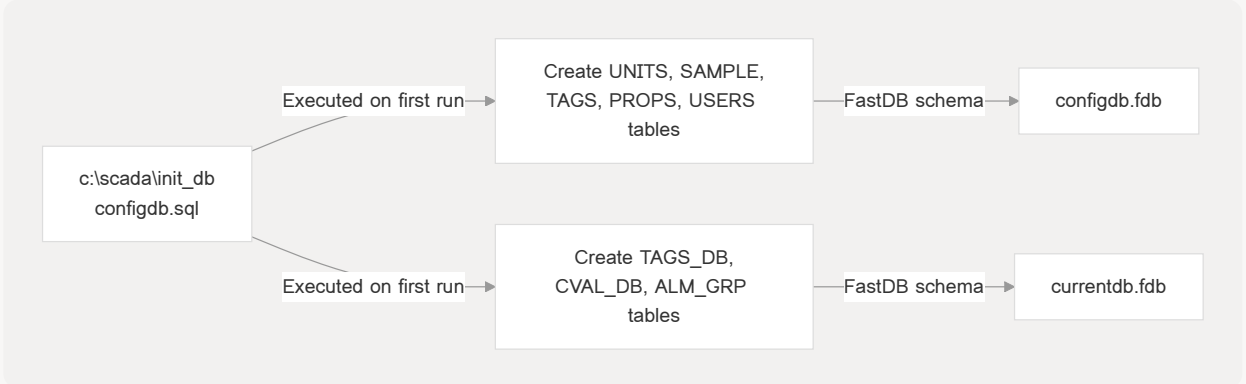
Sources: `src/monitor/monitor.cpp` 682-794 `src/drivers/driver.cpp` 329-341 `src/database/results.h` 313-390

Database Initialization and Schema Creation

Database schemas are initialized through SQL scripts and programmatic table creation:

Initial Schema Creation

The `init_db` directory contains initialization scripts:


c:\scada\init_db configdb.sql
Executed on first run
Executed on first run
Create UNITS, SAMPLE, TAGS, PROPS, USERS tables
Create TAGS_DB, CVAL_DB, ALM_GRP tables
FastDB schema
FastDB schema
configdb.fdb
currentdb.fdb

Dynamic Table Creation

Sample point result tables are created dynamically when sample points are configured:

Creation Trigger: New sample point or rename operation in `SampleCfg` dialog

Table Creation Code Pattern:

```
-- From SampleCfg.cpp, lines 427-455
create table <SamplePointName> (
    TIMEDATE <DATE_TIME_COL_TYPE>,
    STATE int4,
    <tag1> real8,
    <tag2> real8,
    ...
);

create index on <SamplePointName>.TIMEDATE;
```

The system creates the table in both resultsdb (FastDB, optional) and historicdb (GigaBASE, always):

```
#ifdef SAVEDATA_IN_RESULTDB
GetResultDb()->DoExec(0, cmd, 0);  // FastDB
#endif

if(GetHistoricResultDb() != NULL) {
    GetHistoricResultDb()->DoExec(0, cmd, 0);  // GigaBASE
}
```

Database Deletion Operations

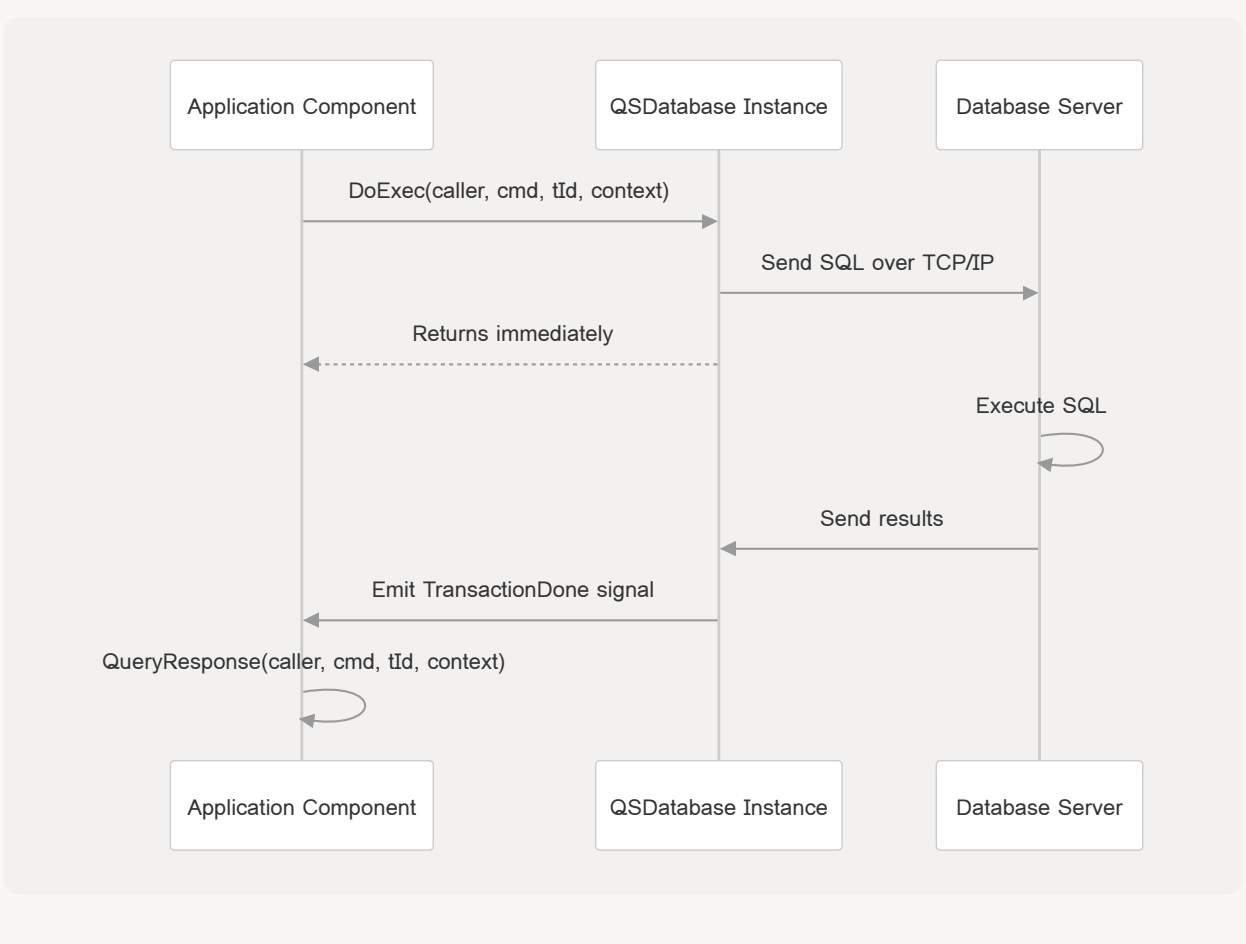
When a sample point is deleted:

1. Remove from SAMPLE table
2. Delete tags from TAGS table
3. Delete properties from PROPS table
4. Remove current values from TAGS_DB and CVAL_DB
5. Drop historical table: `drop table <SamplePointName>;`

Sources: `src/ui/sysdialogs/SampleCfg.cpp` 243-297 `src/ui/sysdialogs/SampleCfg.cpp` 366-465

Database Transaction Management

The database access layer uses an asynchronous transaction model with callbacks:


Application Component
QSDatabase Instance
Database Server
DoExec(caller, cmd, tId, context)
Send SQL over TCP/IP
Returns immediately
Execute SQL
Send results
Emit TransactionDone signal
QueryResponse(caller, cmd, tId, context)
Application Component
QSDatabase Instance
Database Server

Transaction Pattern:

```
// Request
QString cmd = "select * from SAMPLE where NAME='Pump01';";
GetConfigureDb()->DoExec(this, cmd, tItem);

// Callback
void MyClass::QueryResponse(QObject *p, const QString &c, int id, QObject* caller) {
    if(p != this) return;

    switch(id) {
        case tItem:
            // Process results
            QString name = GetConfigureDb()->GetString("NAME");
            break;
    }
}
```

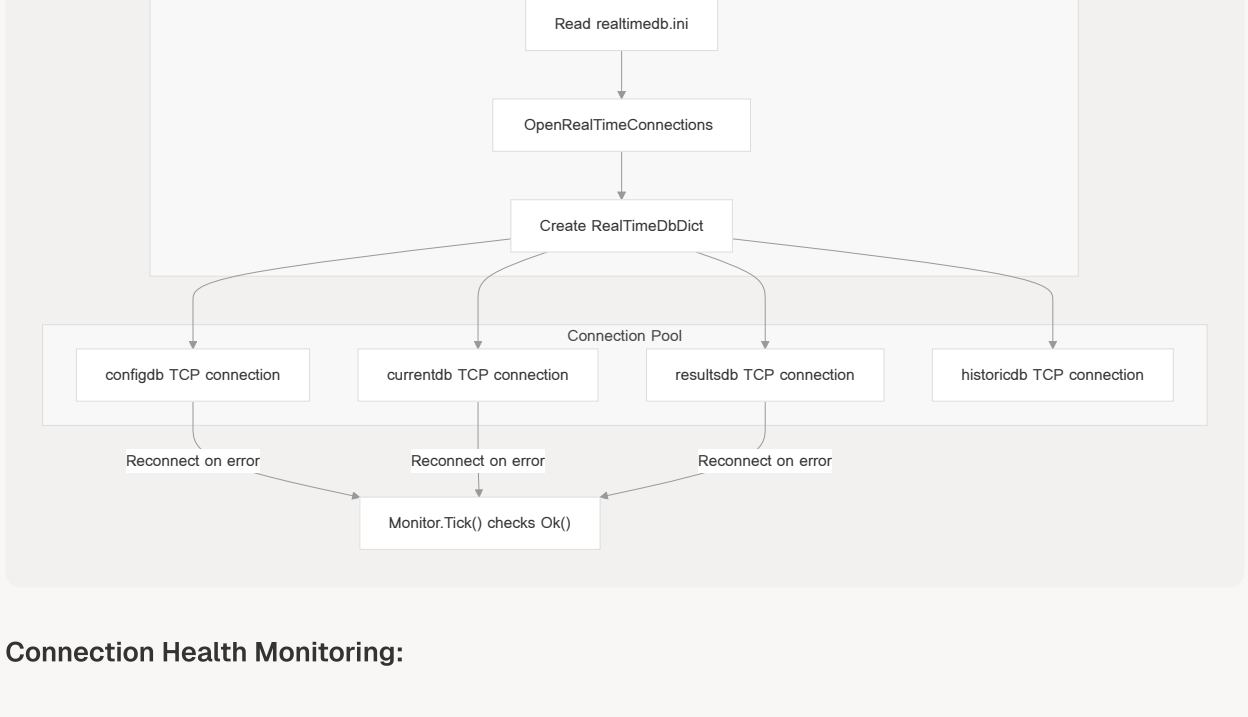
Transaction State Tracking:

The `pending_transactions` counter in `DriverInstance` tracks outstanding database operations to prevent shutdown during active transactions.

Sources: `src/database/realtimedb.h` 20-70 `src/ui/sysdialogs/SampleCfg.cpp` 495-600 `src/drivers/driver.h` 200-250

Database Connection Management

Connections are established at application startup and maintained throughout execution:


Startup Sequence
Read realtimedb.ini
OpenRealTimeConnections
Create RealTimeDbDict
Connection Pool
configdb TCP connection
currentdb TCP connection
resultsdb TCP connection
historicdb TCP connection
Reconnect on error
Reconnect on error
Reconnect on error
Monitor.Tick() checks Ok()

Connection Health Monitoring:

The monitor process checks database health every tick (100ms):

```
// From monitor.cpp, lines 488-547
if(!GetConfigureDb()->Ok() || !GetResultDb()->Ok() || !GetCurrentDb()->Ok()) {
    // Log error
    QSLogEvent("Monitor", "Attempt to restore connection with realtime database server");

    // Disconnect and reconnect
    DisconnectFromRealTimeDatabases();
    ConnectToRealTimeDatabases();

    ++MaxRetryReconnectToRealTimeDb;
}
```

Retry Strategy:

- Automatic reconnection on connection loss
- Exponential backoff not implemented (immediate retry)
- MaxRetryReconnect counters track consecutive failures
- No automatic shutdown on persistent failure (commented out logic suggests this was considered)

Sources: `src/database/realtimedb.cpp` 133-242 `src/monitor/monitor.cpp` 487-558

Summary

IndigoSCADA's database architecture demonstrates a pragmatic multi-database approach:

Database	Engine	Purpose	Access Pattern	Persistence
configdb	FastDB 3.56	Configuration storage	Read-heavy at startup, write on config changes	Disk-backed memory-mapped
currentdb	FastDB 3.56	Real-time values	Write-heavy (100ms updates), read for HMI	Memory with periodic sync
resultsdb	GigaBASE 3.37	Historical archive	Write-heavy, read for reports	Compressed disk storage
Protocol DBs	SQLite 3	Protocol mappings	Read once at driver startup	File-based

This separation provides:

- **Performance:** FastDB's in-memory architecture for hot data paths
- **Scalability:** GigaBASE's efficient time-series storage for unlimited history
- **Simplicity:** SQLite's file-based storage for driver-specific configuration
- **Flexibility:** Independent optimization of each database layer

The TAGS table serves as the central registry, linking sample point names to protocol addresses (IOA) and enabling the monitor process to route data from protocol drivers to the correct sample points and tags.

Sources: src/database/realtimedb.cpp 1-300 src/monitor/monitor.cpp 1-800 src/ui/sysdialogs/SampleCfg.cpp 1-600 src/database/results.h 1-460

Process Management

› Relevant source files

Purpose and Scope

This document describes how `manager.exe` orchestrates system startup, monitors process health, handles automatic restarts, and manages the lifecycle of all IndigoSCADA processes. The manager acts as a supervisory process that launches protocol drivers, the monitor process, UI, and other system components based on configuration files.

For information about individual protocol driver architectures, see Protocol Driver Architecture. For details on inter-process communication mechanisms used between managed processes, see Middleware and IPC.

Overview

The `manager.exe` process serves as the top-level process supervisor in IndigoSCADA. It is responsible for:

- Reading process configuration from `manager.ini`
- Starting up to 127 child processes in sequence
- Monitoring process health through periodic checks
- Automatically restarting failed processes when configured
- Providing a GUI interface for manual system control
- Managing graceful shutdown in reverse start order

The manager can operate in both GUI mode (for manual control and debugging) and non-GUI mode (for automated service operation).

Sources: `src/manager/buttongroups.cpp` 1-624 `src/manager/main.cpp` 1-99

Architecture

Manager Components

Sources: `src/manager/buttongroups.cpp` 157-543 `src/manager/buttongroups.h` 25-55 `src/manager/main.cpp` 24-98

Process Information Storage

The manager maintains process information using Windows-specific data structures:

Field	Type	Purpose
`pProcInfo`	`PROCESS_INFORMATION[127]`	Array storing handle and thread ID for each started process
`pProcInfo[i].hProcess`	`HANDLE`	Process handle for monitoring and termination
`pProcInfo[i].dwThreadId`	`DWORD`	Thread ID for posting messages
`started`	`bool`	Global flag indicating if system is running
`nMaxProcCount`	`const int`	Maximum processes supported (127)

Sources: `src/manager/buttongroups.h` 40-48

Configuration System

manager.ini Structure

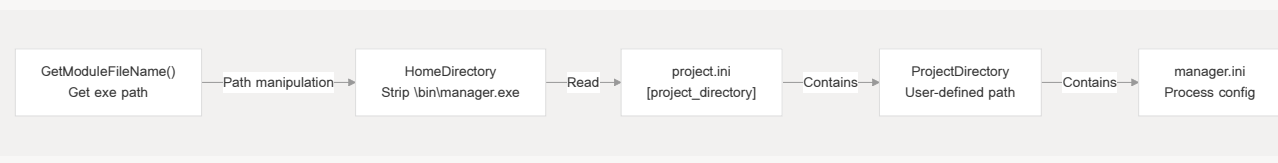
The `manager.ini` file is located in the project directory and contains both global settings and per-process configurations:

```
[Settings]
CheckProcess=10                    ; Check interval in minutes
CheckProcessSeconds=600            ; Alternative: check interval in seconds
AutomaticStart=1                   ; 1=start on launch, 0=manual start

[Process0]
Process=monitor.exe                ; Executable name (relative to bin/)
CommandLine=                       ; Full command line (alternative)
WorkingDir=                        ; Working directory (empty = bin/)
UserInterface=Y                    ; Y=show window, N=hidden
PauseStart=100                     ; Milliseconds to wait after starting
PauseEnd=100                       ; Milliseconds to wait before force kill
Restart=Yes                        ; Yes=auto-restart on failure

[Process1]
Process=iec104master.exe
UserInterface=N
PauseStart=500
Restart=Yes

[Process2]
Process=modbus_master.exe
UserInterface=N
PauseStart=500
Restart=Yes
```

The manager reads up to 10 processes (Process0 through Process9) during normal startup.

Sources: `src/manager/buttongroups.cpp` 162-542

Configuration Resolution


GetModuleFileName()
Get exe path
Path manipulation
HomeDirectory
Strip \bin\manager.exe
Read
project.ini
[project_directory]
Contains
ProjectDirectory
User-defined path
Contains
manager.ini
Process config

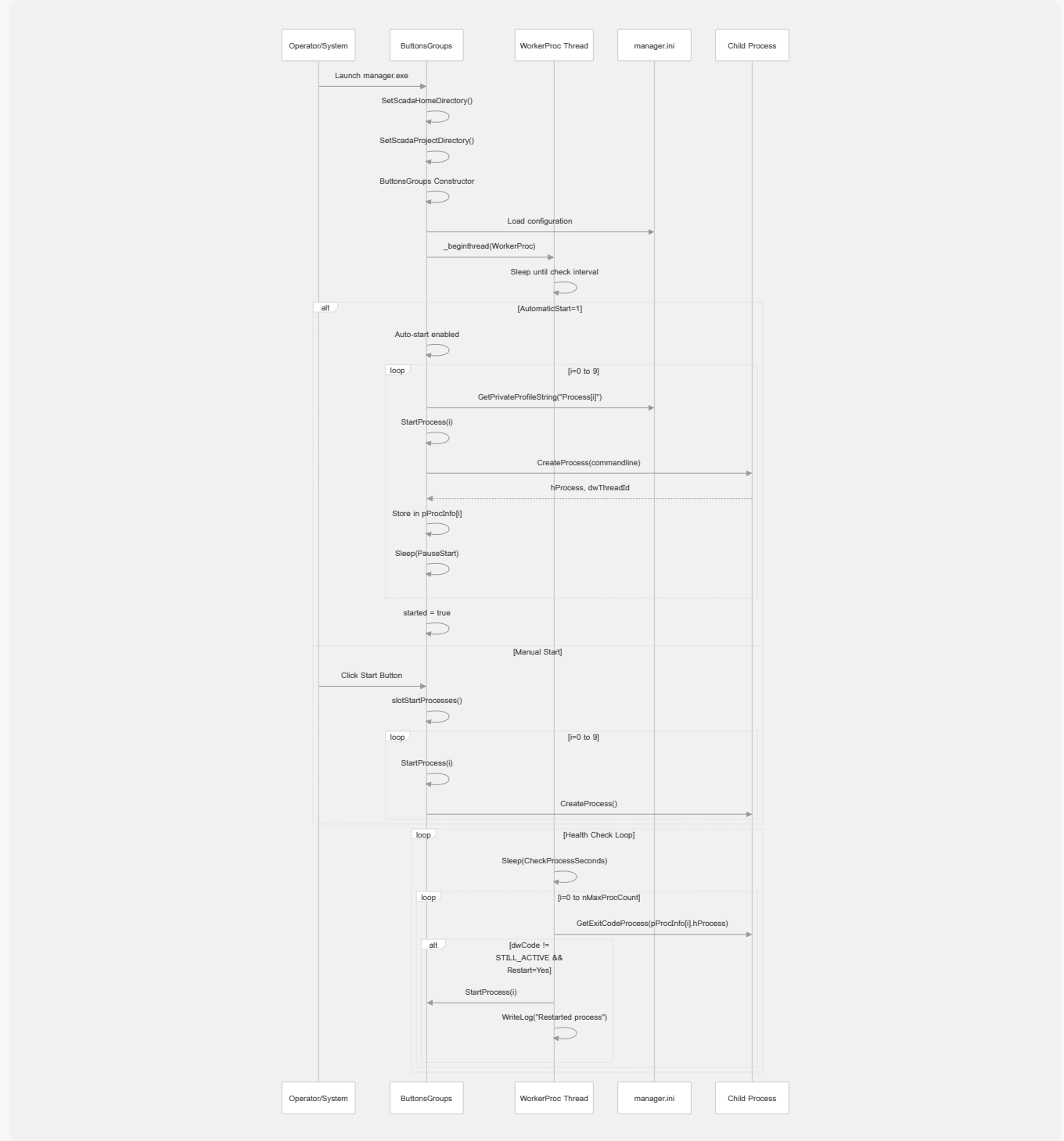
Directory Resolution Functions:

- `SetScadaHomeDirectory()` - Determines installation root by stripping `\bin\manager.exe` from executable path
- `SetScadaProjectDirectory()` - Reads project directory from `project.ini`, creates default if missing
- `GetScadaHomeDirectory()` - Returns cached home directory string
- `GetScadaProjectDirectory()` - Returns cached project directory string

Sources: `src/manager/buttongroups.cpp` 29-138

Process Lifecycle

Startup Sequence


Operator/System
ButtonsGroups
WorkerProc Thread
manager.ini
Child Process
Launch manager.exe
SetScadaHomeDirectory()
SetScadaProjectDirectory()
ButtonsGroups Constructor
Load configuration
_beginthread(WorkerProc)
Sleep until check interval
alt
[AutomaticStart=1]
Auto-start enabled
loop
[i=0 to 9]
GetPrivateProfileString("Process[i]")
StartProcess(i)
CreateProcess(commandline)
hProcess, dwThreadId
Store in pProcInfo[i]
Sleep(PauseStart)
started = true
[Manual Start]
Click Start Button
slotStartProcesses()
loop
[i=0 to 9]
StartProcess(i)
CreateProcess()
loop
[Health Check Loop]
Sleep(CheckProcessSeconds)
loop
[i=0 to nMaxProcCount]
GetExitCodeProcess(pProcInfo[i].hProcess)
alt
[dwCode != STILL_ACTIVE && Restart=Yes]
StartProcess(i)
WriteLog("Restarted process")

Sources: `src/manager/buttongroups.cpp` 387-556

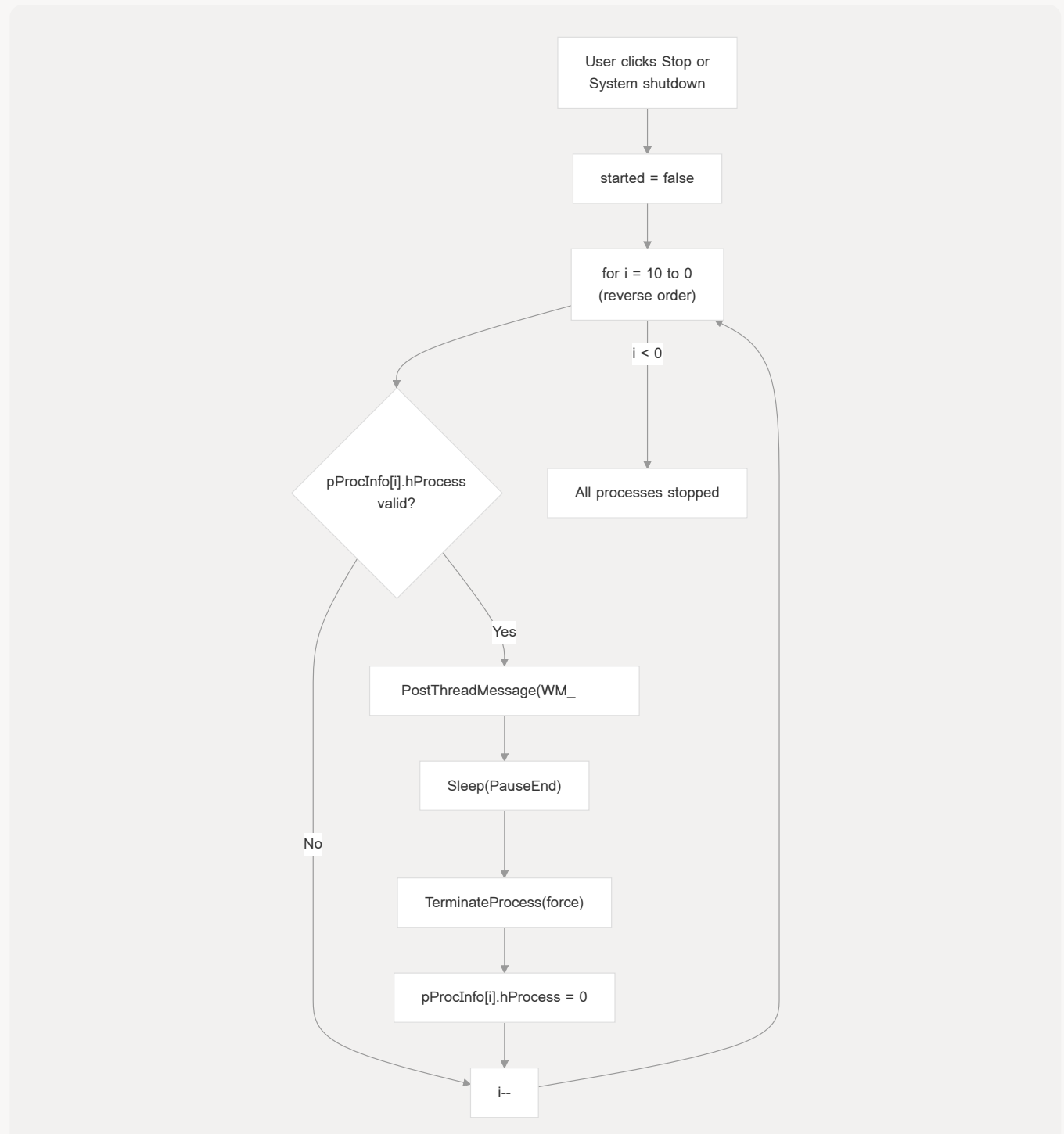
StartProcess Implementation

The `StartProcess()` function follows this logic flow:

1. **Read Process Configuration:**
   - Attempt to read `CommandLine` from `[ProcessN]` section
   - If empty, read `Process` name and prepend with `HomeDirectory\bin\`
2. **Configure Process Window:**
   - If `UserInterface=Y`: Set `SW_SHOWMINIMIZED`, use default desktop
   - If `UserInterface=N`: Set `SW_HIDE`, use current thread's desktop
3. **Create Process:**
   - Call Windows `CreateProcess()` with command line
   - Apply `WorkingDir` if specified, otherwise use NULL (inherits parent)
   - Use `NORMAL_PRIORITY_CLASS` for all processes
   - Set `bInheritHandles=TRUE` to allow child to inherit handles
4. **Post-Start Actions:**
   - Sleep for `PauseStart` milliseconds to stagger startup
   - Store `PROCESS_INFORMATION` in `pProcInfo[nIndex]`
   - Log errors if startup fails

Sources: `src/manager/buttongroups.cpp` 157-286

Shutdown Sequence


User clicks Stop or System shutdown
started = false
for i = 10 to 0 (reverse order)
i < 0
All processes stopped
pProcInfo[i].hProcess valid?
Yes
No
PostThreadMessage(WM_
Sleep(PauseEnd)
TerminateProcess(force)
pProcInfo[i].hProcess = 0
i--

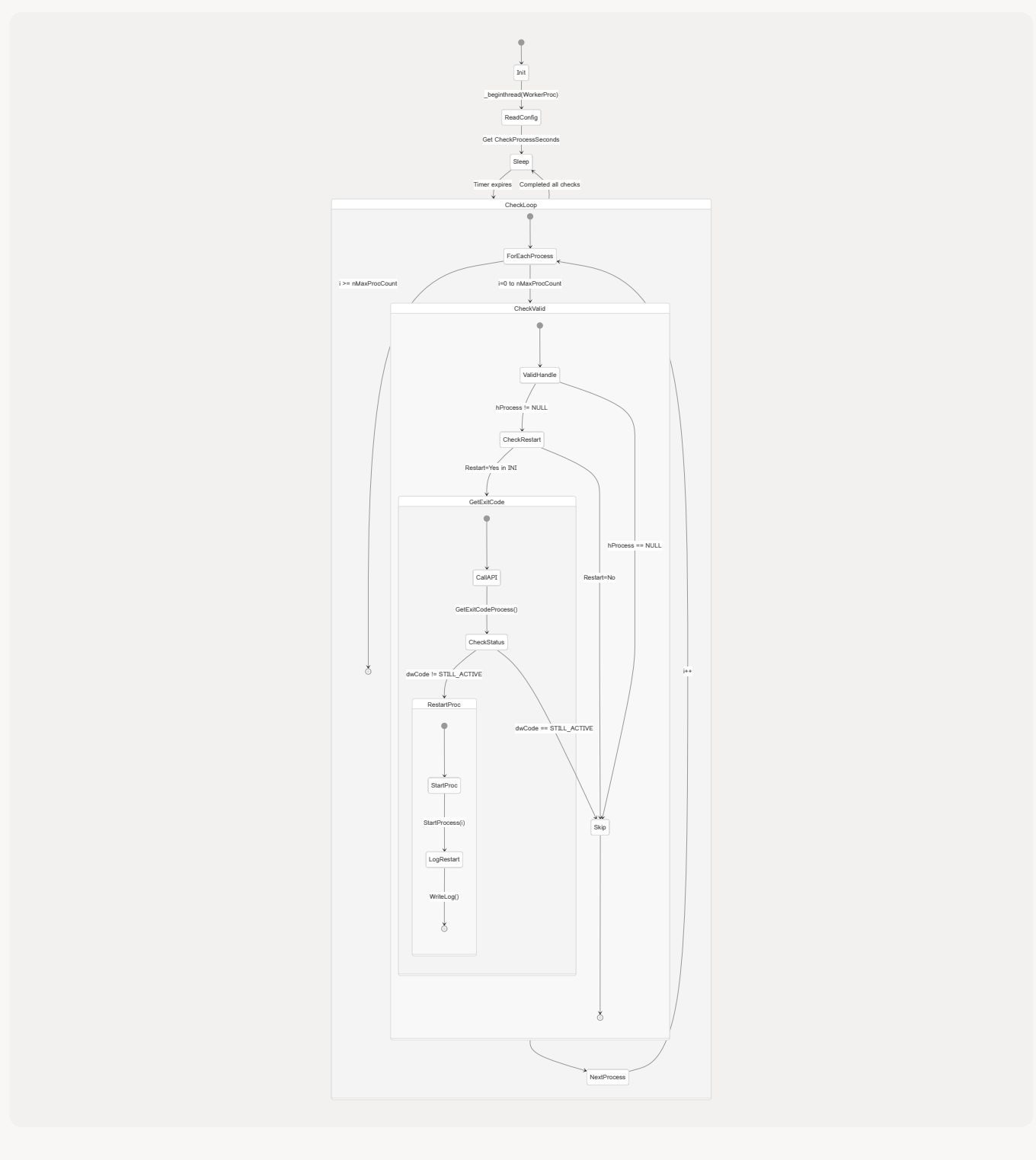
The shutdown proceeds in reverse order (Process9 → Process0) to allow dependent processes to shut down gracefully before their dependencies are terminated.

Sources: `src/manager/buttongroups.cpp` 288-569

Health Monitoring and Restart Logic

ProcessRestarter Worker Thread

The health monitoring system runs in a separate thread spawned during manager initialization:


Init
_beginthread(WorkerProc)
ReadConfig
Get CheckProcessSeconds
Sleep
Timer expires
Completed all checks
CheckLoop
ForEachProcess
i >= nMaxProcCount
i=0 to nMaxProcCount
CheckValid
ValidHandle
hProcess != NULL
CheckRestart
Restart=Yes in INI
GetExitCode
hProcess == NULL
CallAPI
Restart=No
GetExitCodeProcess()
CheckStatus
dwCode != STILL_ACTIVE
i++
RestartProc
dwCode == STILL_ACTIVE
StartProc
StartProcess(i)
Skip
LogRestart
WriteLog()
NextProcess

Worker Thread Configuration:

Setting	Default	Purpose
`CheckProcess`	0	Check interval in minutes (takes precedence)
`CheckProcessSeconds`	600	Check interval in seconds (10 minutes)

The thread calculates the check interval as:

```
if (CheckProcess > 0)
    interval = CheckProcess * 60 seconds
else
    interval = CheckProcessSeconds
```

Sources: `src/manager/buttongroups.cpp` 308-522

Restart Decision Logic

For each process slot `i` from 0 to `nMaxProcCount`:

1. **Check if process exists:** `if (pProcInfo[i].hProcess)`
2. **Check restart configuration:** Read `Restart` value from `[Processi]` section
3. **If Restart=Yes/Y/1:**
   - Call `GetExitCodeProcess(pProcInfo[i].hProcess, &dwCode)`
   - If `dwCode != STILL_ACTIVE`, the process has terminated
   - Call `StartProcess(i)` to restart
   - Log: `"Restarted process N"`
4. **Handle errors:** If `GetExitCodeProcess()` fails, log error code

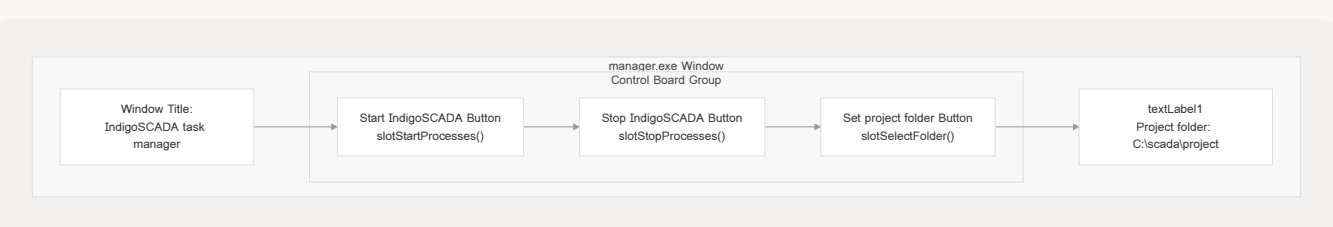
This design allows selective restart behavior on a per-process basis. For example, the UI might have `Restart=No` (manual restart preferred) while protocol drivers have `Restart=Yes` (automatic recovery).

Sources: `src/manager/buttongroups.cpp` 339-377

GUI and Manual Control

ButtonsGroups Widget Layout

The manager provides a Qt-based GUI for manual system control:


manager.exe Window
Control Board Group
Window Title: IndigoSCADA task manager
Start IndigoSCADA Button
slotStartProcesses()
Stop IndigoSCADA Button
slotStopProcesses()
Set project folder Button
slotSelectFolder()
textLabel1
Project folder:
C:\scada\project

Sources: `src/manager/buttongroups.cpp` 387-484

Manual Operation Slots

slotStartProcesses()

```
void ButtonsGroups::slotStartProcesses()
{
    if(started == false)
    {
        started = true;
        for(int i = 0; i < 10; i++)
        {
            StartProcess(i);
        }
    }
}
```

Starts processes 0 through 9 sequentially. Checks `started` flag to prevent duplicate starts.

Sources: `src/manager/buttongroups.cpp` 545-556

slotStopProcesses()

```
void ButtonsGroups::slotStopProcesses()
{
    if(started == true)
    {
        started = false;
        for(int i = 10; i >= 0; i--)
        {
            EndProcess(i);
        }
    }
}
```

Stops processes in reverse order (10 down to 0) to handle dependencies gracefully.

Sources: `src/manager/buttongroups.cpp` 558-569

slotSelectFolder()

Displays a Qt `QFileDialog` in directory-only mode to select a new project folder. Upon selection:

1. Converts forward slashes to backslashes (Windows path normalization)
2. Updates `project.ini` via `WritePrivateProfileString()`
3. Updates GUI label to show new folder
4. Calls `SetScadaProjectDirectory()` to refresh cached path
5. Reloads `pInitFile` path with new project directory

Sources: `src/manager/buttongroups.cpp` 571-624

Logging and Diagnostics

Log File System

The manager maintains a log file at `<HomeDirectory>\logs\manager.log`:

```
void ButtonsGroups::WriteLog(char* pMsg)
{
    SYSTEMTIME oT;
    ::GetLocalTime(&oT);
    FILE* pLog = fopen(pLogFile,"a");
    fprintf(pLog,"%02d/%02d/%04d, %02d:%02d:%02d\n    %s\n",
            oT.wMonth,oT.wDay,oT.wYear,
            oT.wHour,oT.wMinute,oT.wSecond,
            pMsg);
    fclose(pLog);
}
```

Logged Events:

Event	Message Format
Process start failure	`"Failed to start program 'X', error code = N"`
Worker thread failure	`"_beginthread failed, error code = N"`
GetExitCodeProcess failure	`"GetExitCodeProcess failed, error code = N"`
Automatic restart	`"Restarted process N"`

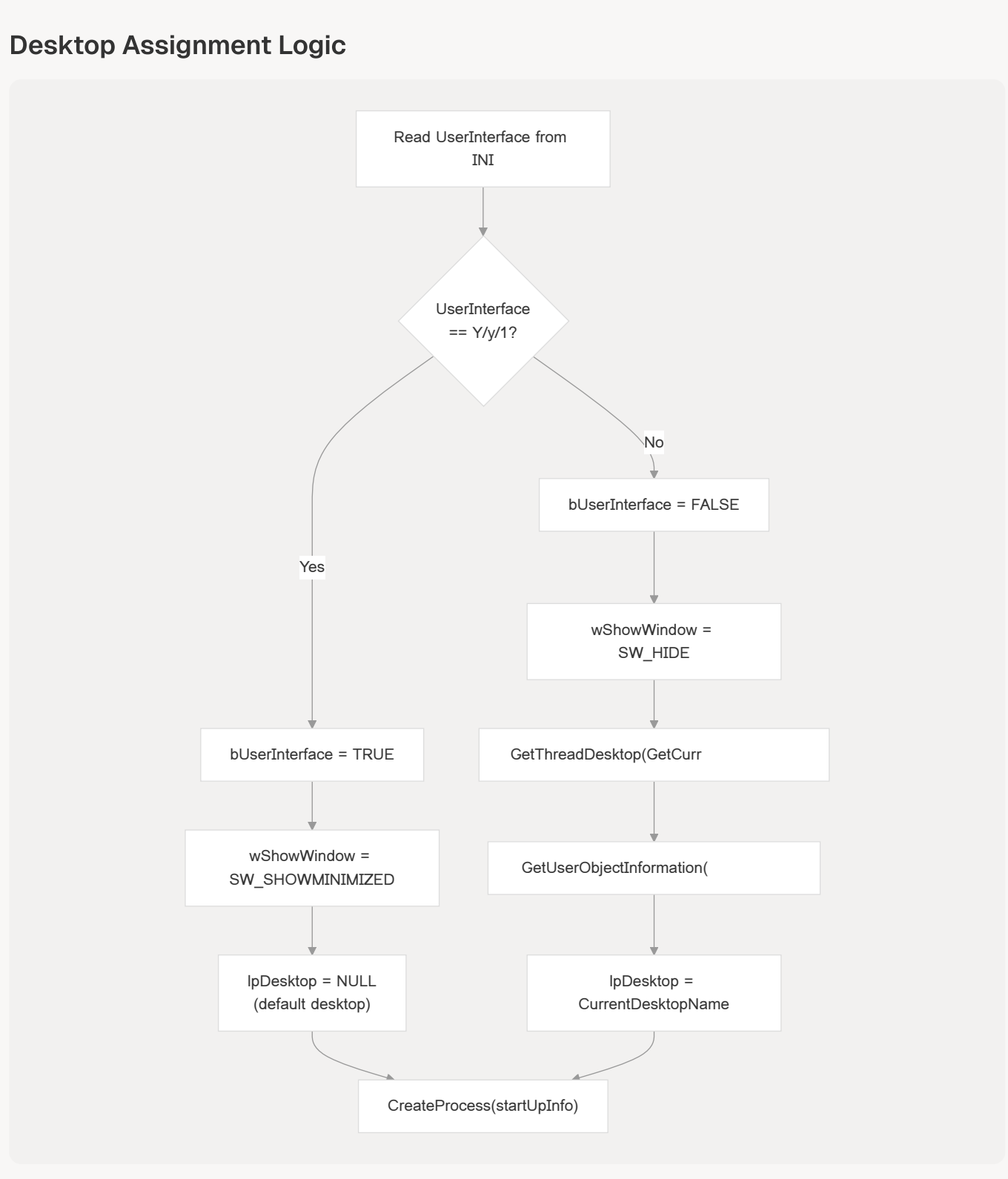
Each log entry includes a timestamp in MM/DD/YYYY, HH:MM:SS format.

Sources: `src/manager/buttongroups.cpp` 140-512

Process Desktop Management

The manager supports running processes on different Windows desktops based on the `UserInterface` flag:

Desktop Assignment Logic


Read UserInterface from INI
UserInterface == Y/y/1?
No
Yes
bUserInterface = FALSE
wShowWindow = SW_HIDE
bUserInterface = TRUE
GetThreadDesktop(GetCurr
wShowWindow = SW_SHOWMINIMIZED
GetUserObjectInformation(
lpDesktop = NULL (default desktop)
lpDesktop = CurrentDesktopName
CreateProcess(startUpInfo)

This allows the manager to run protocol drivers and background services hidden on the same desktop as the manager itself, while allowing the UI to display on the default interactive desktop.

Sources: `src/manager/buttongroups.cpp` 174-257

Integration Points

Relationship to Other Components

The manager interfaces with the broader IndigoSCADA system through:

1. **monitor.exe:** Typically started as Process0, the central data acquisition engine
2. **Protocol Drivers:** Started as Process1+, each with their own configuration in the INI
3. **ui.exe:** User interface process, often configured with `UserInterface=Y`
4. **Project Structure:** All processes read their specific configurations from the project directory
5. **Middleware:** Manager doesn't use ORTE/RTPS; processes coordinate independently after launch

The manager operates at the OS process level and does not participate in IndigoSCADA's internal messaging infrastructure (ORTE/RTPS). It is purely concerned with process lifecycle management.

Sources: `src/manager/buttongroups.cpp` 1-624

Platform Considerations

Windows-Specific Implementation

The current manager implementation is tightly coupled to Windows APIs:

API	Purpose
`CreateProcess()`	Process creation
`GetExitCodeProcess()`	Health checking
`TerminateProcess()`	Forced termination
`PostThreadMessage()`	Graceful shutdown signal
`GetModuleFileName()`	Executable path resolution
`GetPrivateProfileString()`	INI file reading
`GetThreadDesktop()`	Desktop handle retrieval
`GetUserObjectInformation()`	Desktop name retrieval

The code includes `#ifdef WIN32` conditionals, suggesting historical Unix support, but the actual Unix implementations are incomplete or missing.

Sources: `src/manager/buttongroups.cpp` 31-306

Historical Unix Support

Legacy code exists for Unix path handling (`src/manager/buttongroups.cpp` 44-134), but the main process management functions rely entirely on Windows-specific APIs with no Unix equivalents provided. Porting to Linux would require:

- Replacing `CreateProcess()` with `fork()` / `exec()`
- Implementing signal-based process management
- Replacing INI file handling with portable configuration parsing
- Removing desktop management (Windows-specific concept)

Sources: src/manager/buttongroups.cpp 44-134

Data Flow Pipeline

> Relevant source files

This document describes the end-to-end data flow pipeline in IndigoSCADA, tracing how data moves from field devices through protocol drivers, middleware, the monitor process, results processing, and ultimately to the HMI display. It also covers the reverse control direction where operator commands flow from the HMI back to field devices.

For information about the protocol drivers themselves, see Protocol Drivers. For middleware architecture details, see Middleware and IPC. For database storage, see Database Architecture.

Overview of Data Flow Directions

IndigoSCADA implements bidirectional data flow:

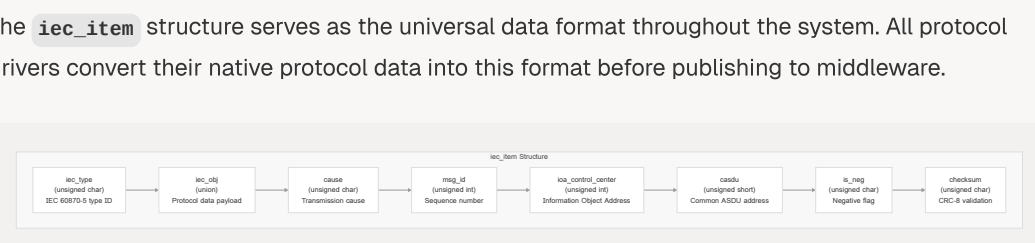
- **Monitor Direction**: Field device data flows from drivers → middleware → monitor.exe → results processor → databases → UI
- **Control Direction**: Commands flow from UI → monitor.exe → middleware → drivers → field devices

All data is normalized to the `iec_item` structure regardless of source protocol, enabling protocol-agnostic processing in the core system.

The iec_item Universal Data Format

Structure Definition

The `iec_item` structure serves as the universal data format throughout the system. All protocol drivers convert their native protocol data into this format before publishing to middleware.


iec_item Structure

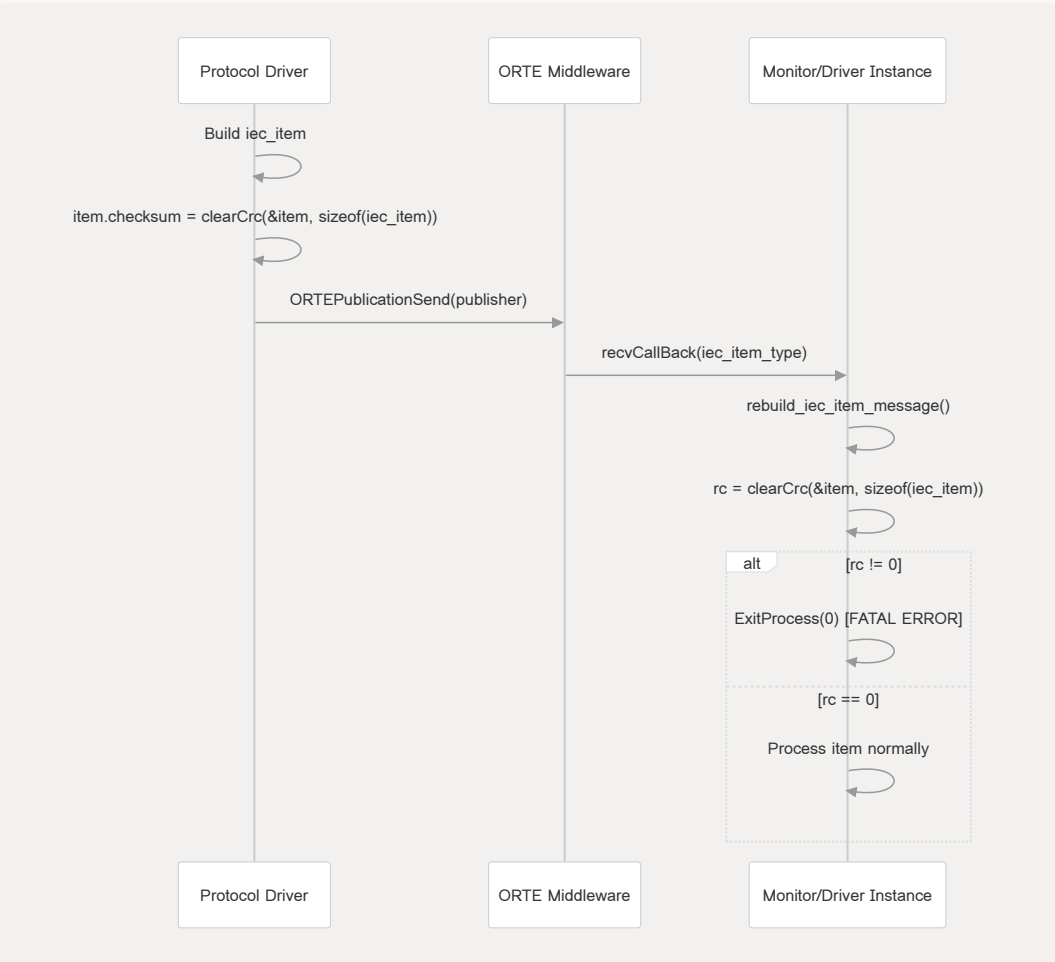
Structure Details:

- **iec_type**: IEC 60870-5-104 type identifier (e.g., M_SP_TB_1 for single point with timestamp, M_ME_TF_1 for float value)
- **iec_obj**: Union containing the actual data payload, typed according to iec_type
- **ioa_control_center**: Unique identifier for the data point (Information Object Address)
- **cause**: Why the data was transmitted (0x03 = spontaneous, 0x14 = general interrogation response)
- **msg_id**: Monotonically increasing sequence number for detecting lost messages
- **checksum**: CRC-8 checksum calculated over the entire structure for integrity validation

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 304-318 `src/drivers/iec104/iec104driver_instance.cpp` 52-87

Checksum Validation

All `iec_item` messages include CRC-8 checksums calculated using the `clearCrc()` function. Receivers validate integrity and exit on checksum failure:

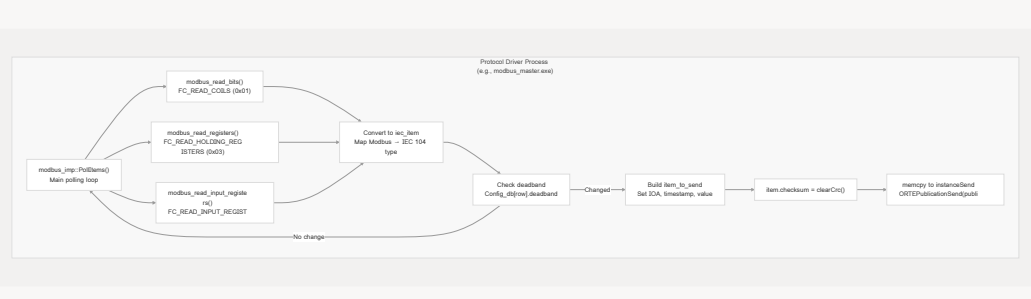

Protocol Driver
ORTE Middleware
Monitor/Driver Instance
Build iec_item
item.checksum = clearCrc(&item, sizeof(iec_item))
ORTEPublicationSend(publisher)
recvCallBack(iec_item_type)
rebuild_iec_item_message()
rc = clearCrc(&item, sizeof(iec_item))
alt
[rc != 0]
ExitProcess(0) [FATAL ERROR]
[rc == 0]
Process item normally
Protocol Driver
ORTE Middleware
Monitor/Driver Instance

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 39-74 `src/monitor/monitor.cpp` 81-114

Monitoring Direction: Field Device to HMI

Stage 1: Field Device to Protocol Driver

Each protocol driver implements its protocol-specific polling logic and converts native protocol data to `iec_item` format.


Protocol Driver Process

Key Functions:

- `PollItems()`: Main loop that iterates through `Config_db` array loaded from SQLite
- Protocol-specific read functions (e.g., `modbus_read_bits()`, `modbus_read_registers()`)
- Deadband filtering prevents flooding on minor changes: `ABS(last_value - new_value) > deadband`
- Type mapping: VT_BOOL → M_SP_TB_1, VT_R4 → M_ME_TF_1, VT_I2 → M_ME_TE_1

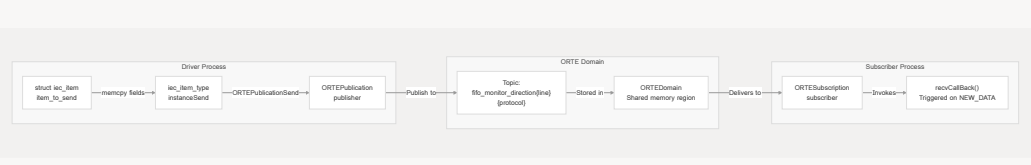
Modbus Example:

```
Reading holding register (0x03) containing float (VT_R4):
1. modbus_read_registers(ctx, address, 2, tab_rp_registers)  // Read 32 bits
2. float real = modbus_get_float(tab_rp_registers)
3. if(ABS(Config_db[row].last_value.f - real) > deadband)
4. item_to_send.iec_type = M_ME_TF_1
5. item_to_send.iec_obj.o.type36.mv = real
6. item_to_send.iec_obj.o.type36.time = actual_time
```

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 562-1229 `src/drivers/modbus/protocol/modbus_imp.cpp` 696-767

Stage 2: Driver to Middleware

Drivers publish via ORTE (Open Real-Time Ethernet) middleware using topic-based publish-subscribe.


Driver Process
ORTE Domain
Subscriber Process

Topic Naming Convention:

- Monitor direction: `"fifo_monitor_direction" + lineNumber + protocolName`
  - Example: `"fifo_monitor_direction1modbus"` for Modbus line 1
  - Example: `"fifo_monitor_direction2iec104"` for IEC 104 line 2
- Control direction: `"fifo_control_direction" + lineNumber + protocolName`

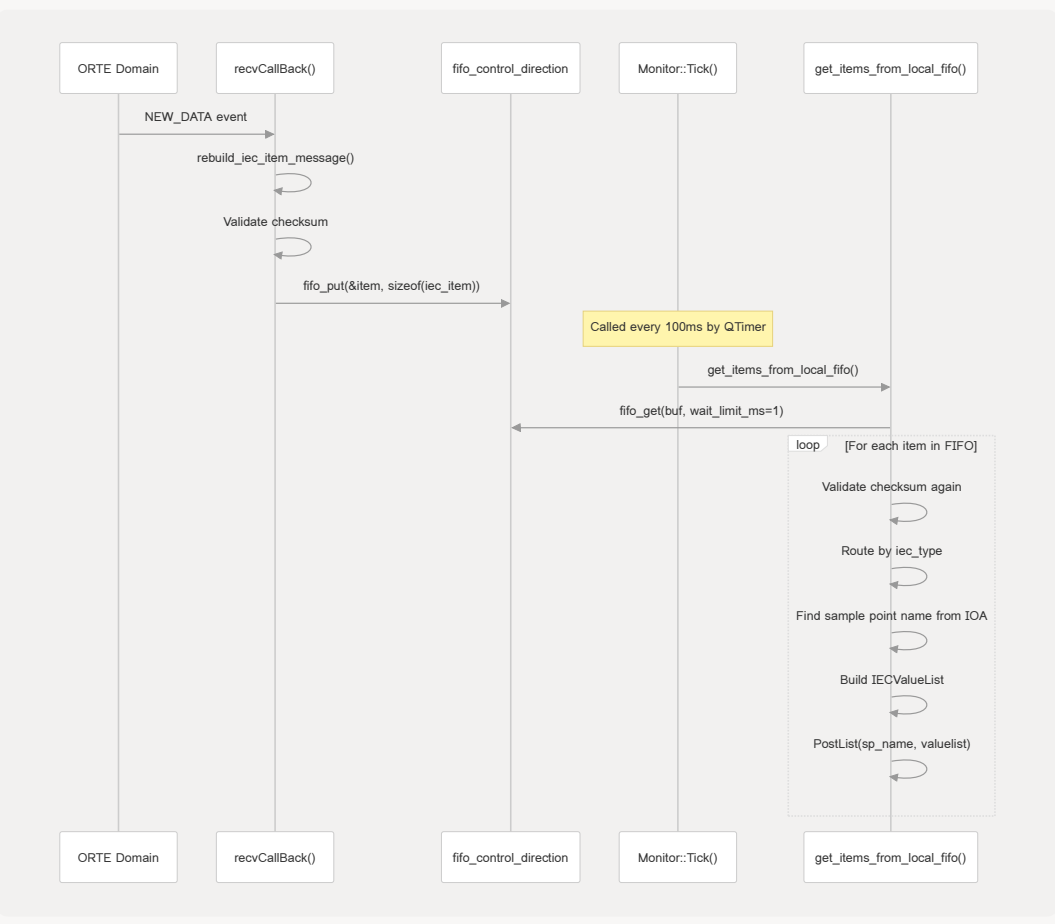
ORTE Configuration:

- Persistence: 5 seconds (messages retained if subscriber temporarily unavailable)
- Strength: 1 (default priority)
- Deadline: 3 seconds (subscriber notified if no data received)
- Quality of Service: BEST_EFFORTS (no guaranteed delivery retries)

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 143-211 `src/drivers/modbus/protocol/modbus_imp.cpp` 1198-1217

Stage 3: Middleware to Monitor Process

The monitor.exe process is the central data acquisition engine that subscribes to all driver topics.


ORTE Domain
recvCallBack()
fifo_control_direction
Monitor::Tick()
get_items_from_local_fifo()
NEW_DATA event
rebuild_iec_item_message()
Validate checksum
fifo_put(&item, sizeof(iec_item))
Called every 100ms by QTimer
get_items_from_local_fifo()
fifo_get(buf, wait_limit_ms=1)
loop
[For each item in FIFO]
Validate checksum again
Route by iec_type
Find sample point name from IOA
Build IECValueList
PostList(sp_name, valuelist)
ORTE Domain
recvCallBack()
fifo_control_direction
Monitor::Tick()
get_items_from_local_fifo()

Callback Registration: The monitor creates a global subscriber for all protocol data:

```
domain = ORTEDomainAppCreate(...)
subscriber = ORTESubscriptionCreate(
    domain,
    IMMEDIATE,
    BEST_EFFORTS,
    "fifo_global_monitor_direction",  // Global topic for all drivers
    "iec_item_type",
    &instanceRecv,
    &deadline,
    &minimumSeparation,
    recvCallBack,
    this,  // Monitor instance passed as callback parameter
    smIPAddress
)
```

FIFO Buffering: The `fifo_control_direction` FIFO provides decoupling between the asynchronous ORTE callback and the synchronous Tick() processing:

- Max size: 65535 bytes
- Thread-safe ring buffer implementation
- Non-blocking reads with configurable timeout

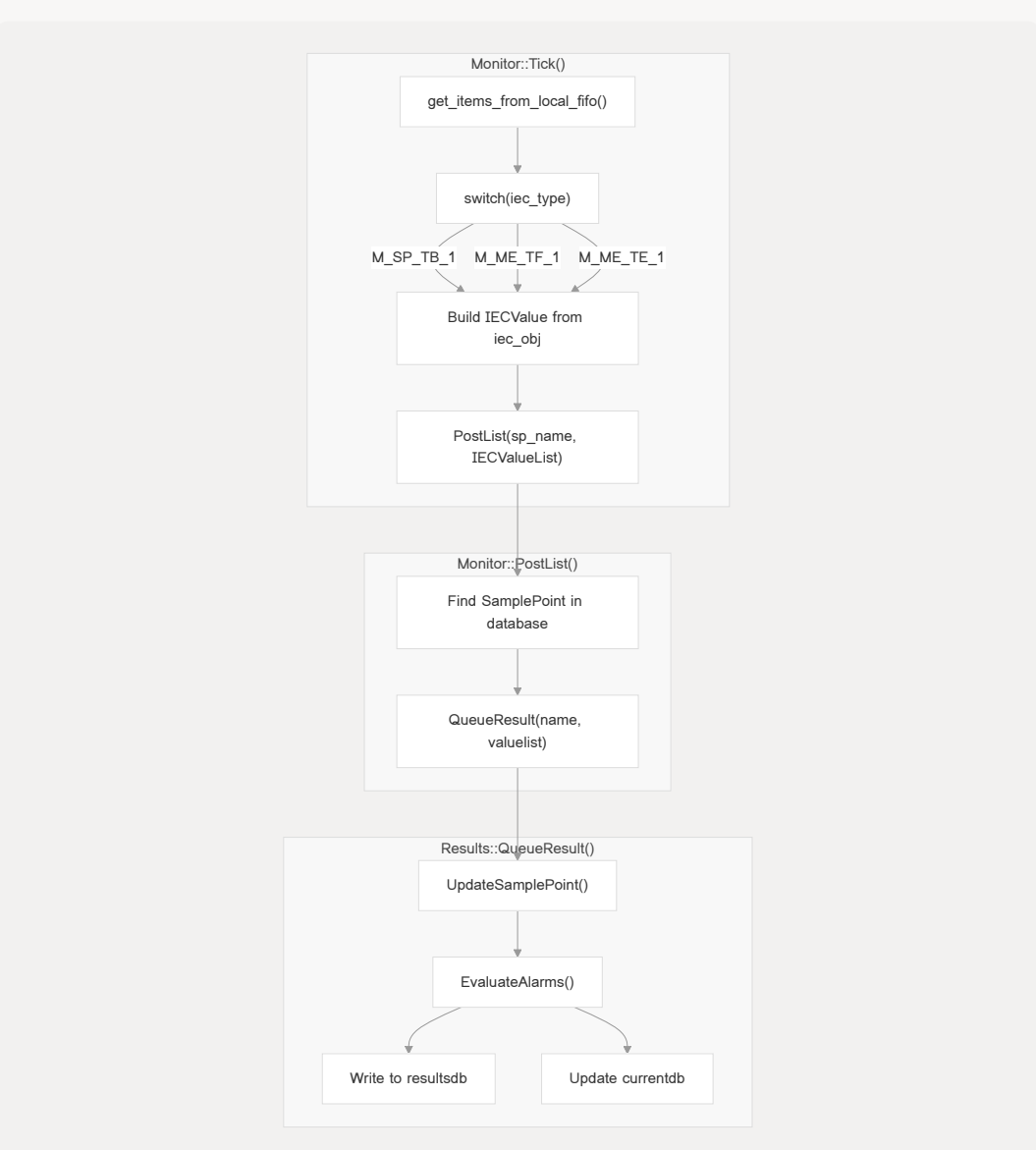
IOA to Sample Point Name Mapping: Monitor maintains `ioa_name_map` dictionary loaded from TAGS table at startup:

```
QString cmd = "select * from TAGS where UNIT='" + Name + "';"
// Populates: ioa_name_map[Config_db[i].ioa_control_center] = SamplePointName
```

Sources: `src/monitor/monitor.cpp` 116-139 `src/drivers/iec104/iec104driver_instance.cpp` 752-829 `src/drivers/modbus/modbus_driver_instance.cpp` 602-670

Stage 4: Monitor to Results Processor

The monitor process forwards validated data to the results processor for alarm evaluation and historical storage.


Monitor::Tick()
get_items_from_local_fifo()
switch(iec_type)
M_SP_TB_1
M_ME_TF_1
M_ME_TE_1
Build IECValue from iec_obj
PostList(sp_name, IECValueList)
Monitor::PostList()
Find SamplePoint in database
QueueResult(name, valuelist)
Results::QueueResult()
UpdateSamplePoint()
EvaluateAlarms()
Write to resultsdb
Update currentdb

IECValue Creation: Different IEC types map to different value structures:

```
case M_SP_TB_1:  // Single point with timestamp
    iec_type30 var = p_item->iec_obj.o.type30
    IECValue v(VALUE_TAG, &var, M_SP_TB_1)

case M_ME_TF_1:  // Float measurement with timestamp
    iec_type36 var = p_item->iec_obj.o.type36
    IECValue v(VALUE_TAG, &var, M_ME_TF_1)

case M_ME_TE_1:  // Scaled measurement with timestamp
    iec_type35 var = p_item->iec_obj.o.type35
    IECValue v(VALUE_TAG, &var, M_ME_TE_1)
```

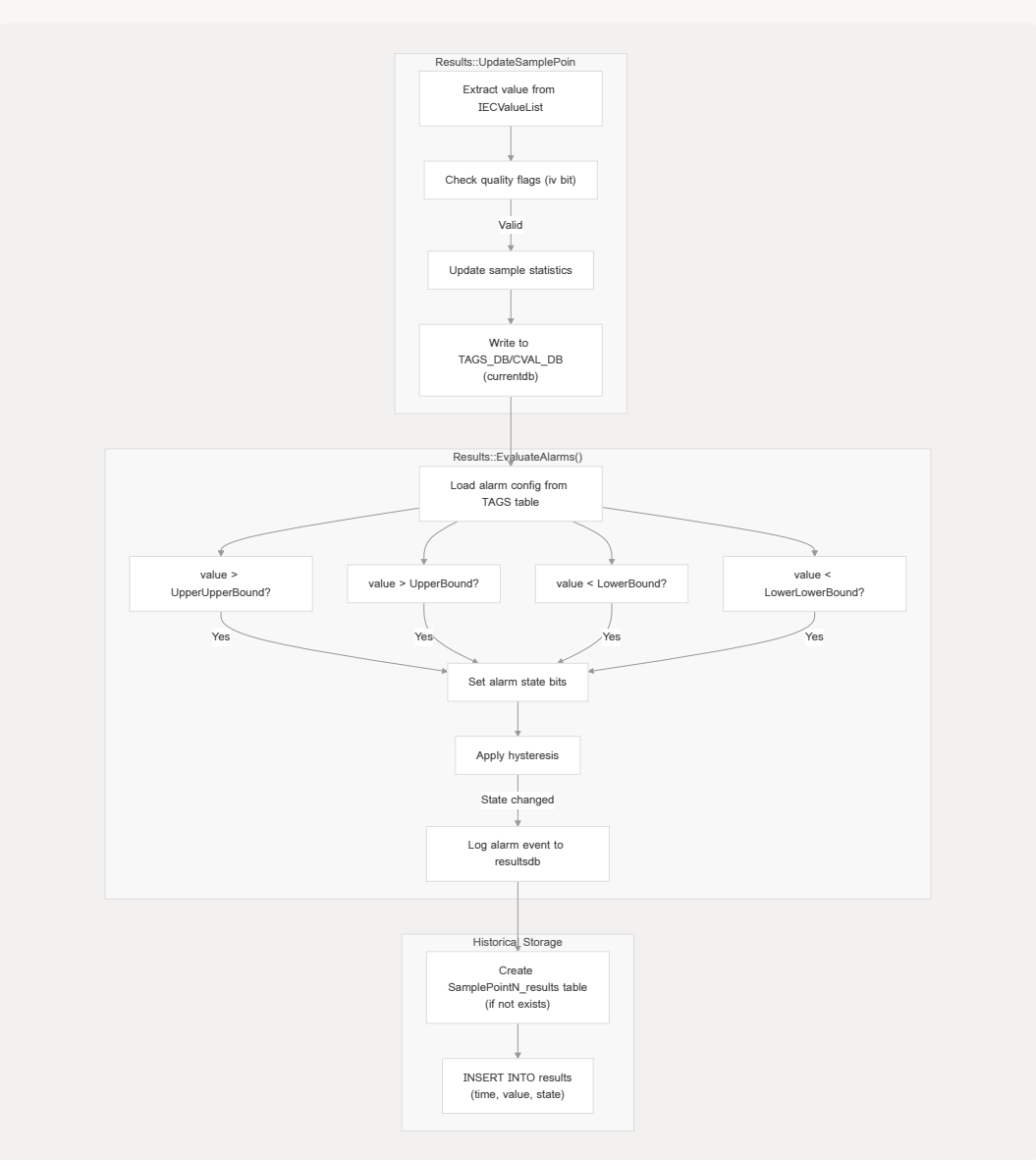
Sample Point Lookup:

```
// Monitor maintains dictionary of SamplePoint objects
SamplePointDict::iterator it = SampleDict.find(sp_name)
if(it != SampleDict.end()) {
    SamplePoint *sp = (*it).second
    // Forward to results processor
}
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 837-1109 `src/drivers/modbus/modbus_driver_instance.cpp` 678-1108

Stage 5: Results Processor to Databases

The results processor evaluates alarms and writes data to both currentdb (real-time) and resultsdb (historical).


Results::UpdateSamplePoin
Extract value from IECValueList
Check quality flags (iv bit)
Valid
Update sample statistics
Write to TAGS_DB/CVAL_DB (currentdb)
Results::EvaluateAlarms()
Load alarm config from TAGS table
value > UpperUpperBound?
value > UpperBound?
value < LowerBound?
value < LowerLowerBound?
Yes
Yes
Yes
Yes
Set alarm state bits
Apply hysteresis
State changed
Log alarm event to resultsdb
Historical Storage
Create SamplePointN_results table (if not exists)
INSERT INTO results (time, value, state)

Database Tables:

Table	Database	Purpose	Key Fields
TAGS_DB	currentdb (FastDB)	Real-time tag values	NAME, VALUE, UPDTIME, STATE
CVAL_DB	currentdb (FastDB)	Current sample point values	NAME, VAL, RECEIPE, UPDTIME
SamplePointN_results	resultsdb (GigaBASE)	Historical time-series data	TIMEDATE, VAL, STATE, STATE_BIT
ALM_GRP_STATE	resultsdb	Alarm group aggregation	GRP, STATE_BIT

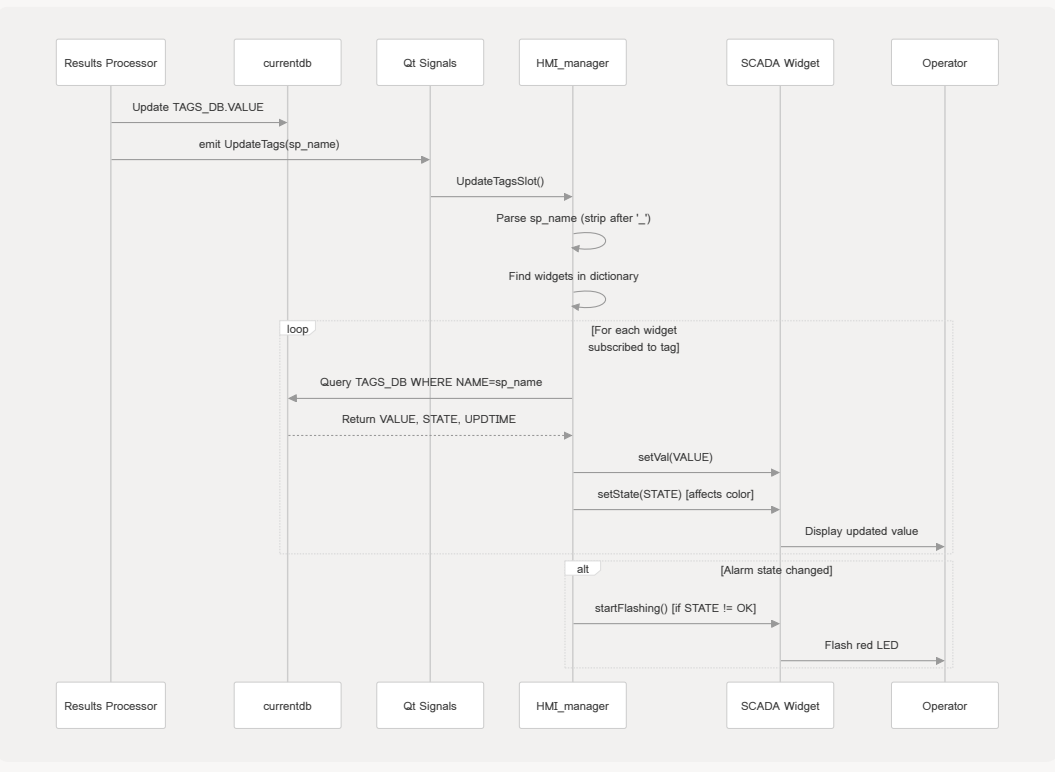
Alarm State Bits:

- Bit 0: Value out of range (OOR)
- Bit 1: Lower-Lower alarm
- Bit 2: Lower alarm
- Bit 3: Upper alarm
- Bit 4: Upper-Upper alarm
- Bit 5: Alarm acknowledged
- Bit 6-7: Reserved

Sources: Code pattern inferred from `src/drivers/iec104/iec104driver_instance.cpp` 837-1109 and database schema references

Stage 6: Databases to UI

The UI queries databases and updates HMI widgets in response to UpdateTags signals.


Results Processor
currentdb
Qt Signals
HMI_manager
SCADA Widget
Operator
Update TAGS_DB.VALUE
emit UpdateTags(sp_name)
UpdateTagsSlot()
Parse sp_name (strip after '_')
Find widgets in dictionary
loop
[For each widget subscribed to tag]
Query TAGS_DB WHERE NAME=sp_name
Return VALUE, STATE, UPDTIME
setVal(VALUE)
setState(STATE) [affects color]
Display updated value
alt
[Alarm state changed]
startFlashing() [if STATE != OK]
Flash red LED
Results Processor
currentdb
Qt Signals
HMI_manager
SCADA Widget
Operator

Widget Dictionary Structure: The `HMI_manager` maintains dictionaries of widgets keyed by truncated sample point name:

```
// Widget naming convention: strip everything after underscore
QString widget_name = sample_point_name.left(sample_point_name.find('_'))

// Dictionaries for different widget types
QwtThermo_dictionary[widget_name] = list<QwtThermo*>
LED_dictionary[widget_name] = list<SinglePointLed*>
Switch_dictionary[widget_name] = list<PSwitch*>
Plot_dictionary[widget_name] = list<QwtPlot*>
```

Value Update Mechanism:

```
void HMI_manager::UpdateTagsSlot() {
    // Query currentdb
    QString cmd = "select VALUE,STATE from TAGS_DB where NAME='" + tag_name + "'"

    // Update all widgets subscribed to this tag
    for(auto widget : LED_dictionary[truncated_name]) {
        widget->setVal(value)
        widget->setState(state)  // Controls LED color/flashing
    }
}
```

Flashing LED Alarm Indication:

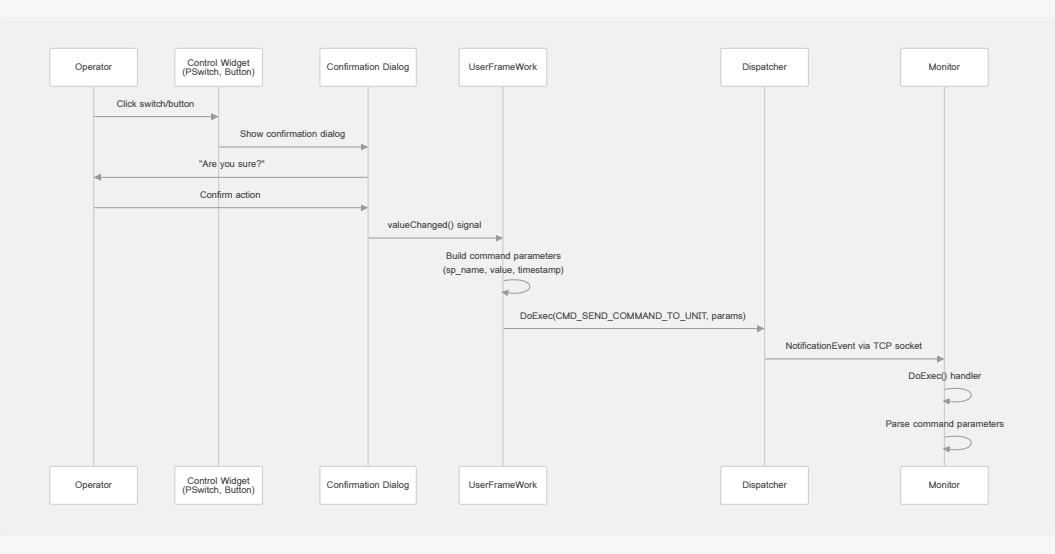
- Normal state: Green LED, solid
- Alarm unacknowledged: Red LED, flashing at 1 Hz
- Alarm acknowledged: Amber LED, solid
- Communication error: Gray LED

Sources: Pattern inferred from `src/ui/user.cpp` 367-486 and HMI widget architecture

Control Direction: HMI to Field Device

Stage 1: UI to Monitor

Operator actions in the UI trigger command generation and transmission to the monitor process.


Operator
Control Widget (PSwitch, Button)
Confirmation Dialog
UserFrameWork
Dispatcher
Monitor
Click switch/button
Show confirmation dialog
"Are you sure?"
Confirm action
valueChanged() signal
Build command parameters (sp_name, value, timestamp)
DoExec(CMD_SEND_COMMAND_TO_UNIT, params)
NotificationEvent via TCP socket
DoExec() handler
Parse command parameters
Operator
Control Widget (PSwitch, Button)
Confirmation Dialog
UserFrameWork
Dispatcher
Monitor

Command Parameters: Commands include the following data passed through the Dispatcher:

Field	Source	Example
Sample point name	Widget identifier	"PUMP_001_CONTROL"
Command value	Widget state	1.0 (ON) or 0.0 (OFF)
Timestamp	System time as epoch_ms	"1609459200000"
Command type	Widget type mapping	C_SC_TA_1 (single command with timestamp)

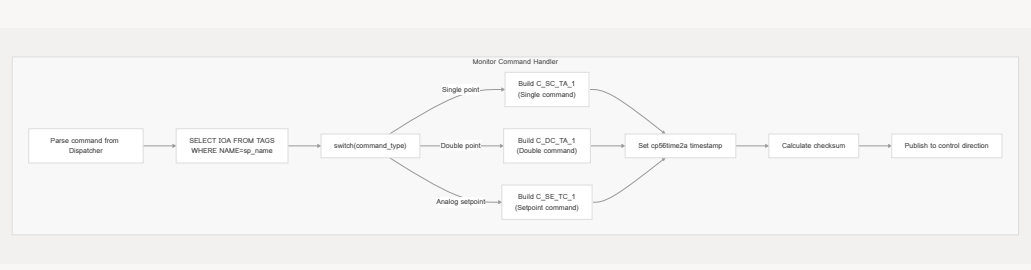
Privilege Checking: Before executing commands, the UI checks user privileges:

```
if(!(GetUserDetails().privs & PRIVS_OPERATOR)) {
    QMessageBox::critical("Insufficient privileges")
    return
}
```

Sources: `src/ui/user.cpp` 523-640

Stage 2: Monitor IOA Lookup and Command Construction

The monitor queries the database to resolve sample point name to IOA and constructs the appropriate IEC 104 command.


Monitor Command Handler

Command Type Mapping:

Widget Type	Value Type	IEC 104 Command
Single point switch	Boolean	C_SC_TA_1 (Type 58)
Double point switch	On/Off/Invalid	C_DC_TA_1 (Type 59)
Setpoint (normalized)	int16 [-32768, 32767]	C_SE_TA_1 (Type 61)
Setpoint (scaled)	int16 [-32768, 32767]	C_SE_TB_1 (Type 62)
Setpoint (float)	float	C_SE_TC_1 (Type 63)

Database Query:

```
QString cmd = "select IOA from TAGS where NAME='" + sample_point_name + "' and UNIT='" + unit_name + "'
GetConfigureDb()->DoExec(this, cmd, tGetIOAfromSamplePointName, command_params)
```

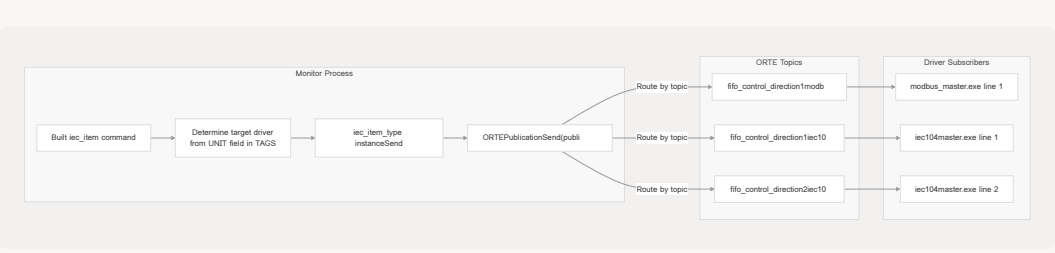
Command Structure Example:

```
// C_SE_TC_1: Set point command, short floating point with time tag
struct iec_item item_to_send
memset(&item_to_send, 0x00, sizeof(iec_item))
item_to_send.iec_type = C_SE_TC_1
item_to_send.iec_obj.ioa = IOACommand  // From database lookup
item_to_send.iec_obj.o.type63.sv = (float)command_value
item_to_send.iec_obj.o.type63.time = iec_cmd_time  // Current timestamp
item_to_send.msg_id = msg_sent_in_control_direction++
item_to_send.checksum = clearCrc(&item_to_send, sizeof(iec_item))
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 314-512 `src/drivers/modbus/modbus_driver_instance.cpp` 334-390

Stage 3: Monitor to Middleware

Monitor publishes the command to the appropriate control direction topic.


Monitor Process
ORTE Topics
Driver Subscribers

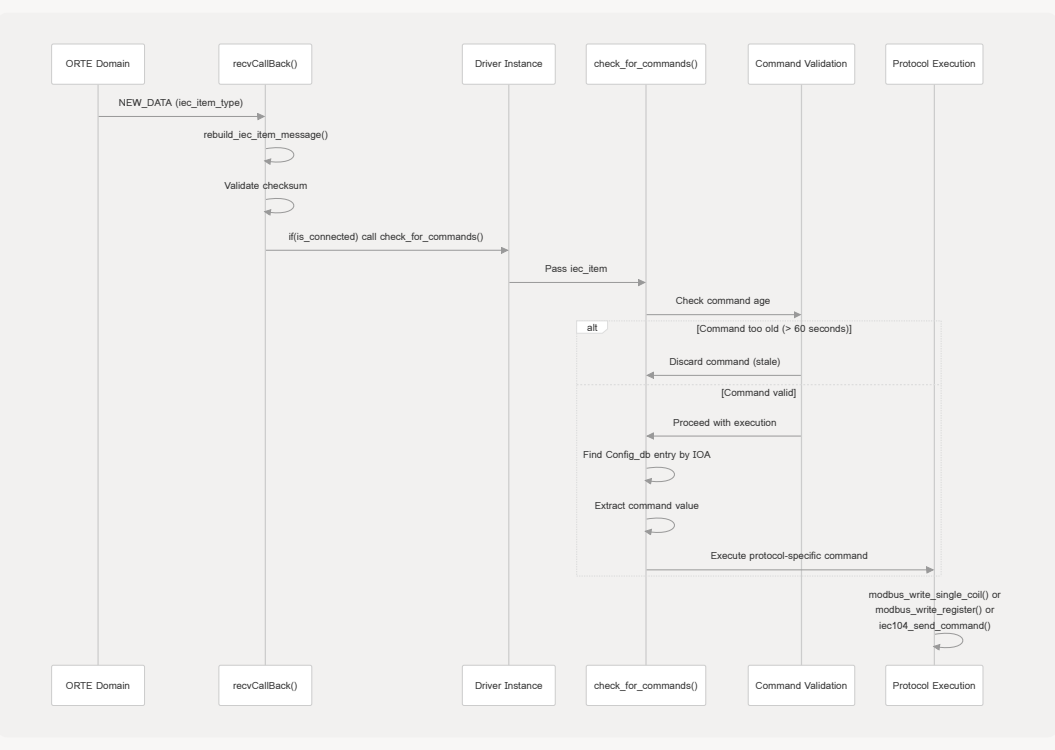
Topic Selection: Each driver instance subscribes to its own control direction topic:

- Topic format: `"fifo_control_direction" + lineNumber + protocolName`
- Monitor determines which topic to publish to based on the UNIT field in the TAGS table
- UNIT field specifies which driver instance manages the tag (e.g., "MODBUS_LINE1", "IEC104_LINE2")

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 514-531 `src/monitor/monitor.cpp` 173-210

Stage 4: Middleware to Driver

Drivers receive commands via ORTE callback and validate them before execution.


ORTE Domain
recvCallBack()
Driver Instance
check_for_commands()
Command Validation
Protocol Execution
NEW_DATA (iec_item_type)
rebuild_iec_item_message()
Validate checksum
if(is_connected) call check_for_commands()
Pass iec_item
Check command age
alt
[Command too old (> 60 seconds)]
Discard command (stale)
[Command valid]
Proceed with execution
Find Config_db entry by IOA
Extract command value
Execute protocol-specific command
modbus_write_single_coil() or modbus_write_register() or iec104_send_command()
ORTE Domain
recvCallBack()
Driver Instance
check_for_commands()
Command Validation
Protocol Execution

Command Age Validation: Drivers discard commands older than MAX_COMMAND_SEND_TIME (typically 60 seconds) to prevent stale commands from executing:

```
#define MAX_COMMAND_SEND_TIME 60

void check_for_commands(struct iec_item *queued_item) {
    int64_t command_epoch = epoch_from_cp56time2a(&queued_item->iec_obj.o.type58.time)
    int64_t current_epoch = time(NULL)

    if((current_epoch - command_epoch) > MAX_COMMAND_SEND_TIME) {
        fprintf(stderr, "Command too old, discarding\n")
        return  // Discard stale command
    }

    // Proceed with command execution...
}
```

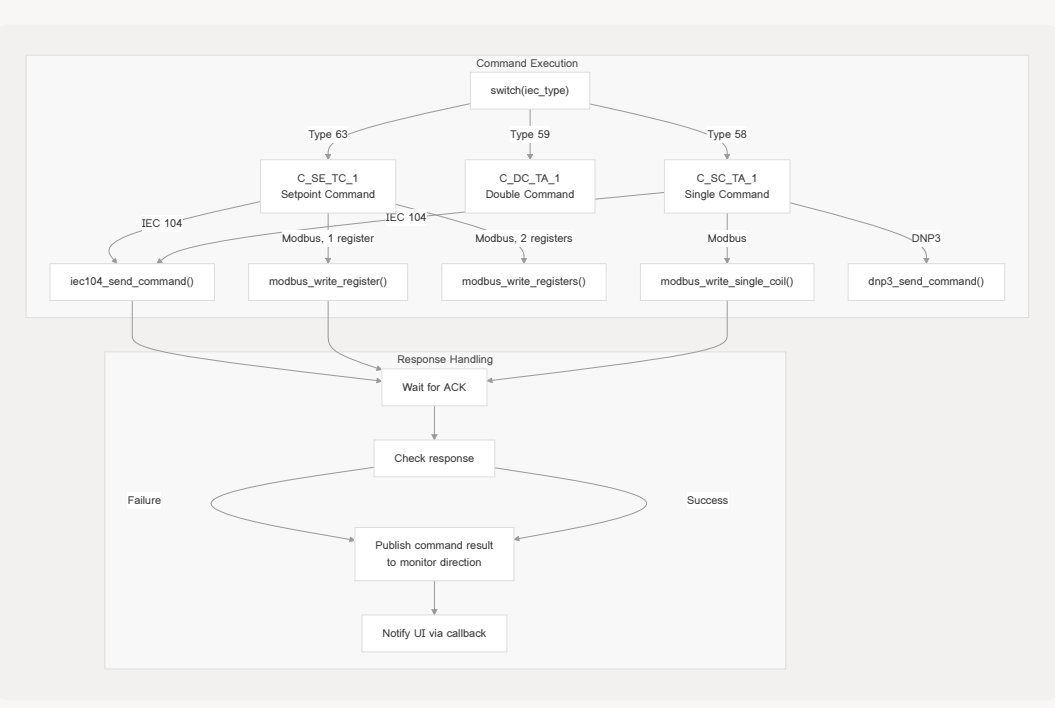
IOA to Database Row Mapping:

```
// Find the configuration entry for this command
for(int j = 0; j < db_n_rows; j++) {
    if(Config_db[j].ioa_control_center == queued_item->iec_obj.ioa) {
        // Execute command using Config_db[j] parameters
        break
    }
}
```

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 1343-1608

Stage 5: Driver to Field Device

The driver executes the protocol-specific command to the field device.


Command Execution
switch(iec_type)
Type 63
Type 59
Type 58
C_SE_TC_1 Setpoint Command
C_DC_TA_1 Double Command
C_SC_TA_1 Single Command
IEC 104
Modbus, 1 register
IEC 104
Modbus, 2 registers
Modbus
DNP3
iec104_send_command()
modbus_write_register()
modbus_write_registers()
modbus_write_single_coil()
dnp3_send_command()
Response Handling
Wait for ACK
Check response
Failure
Success
Publish command result to monitor direction
Notify UI via callback

Modbus Command Execution:

```
case C_SC_TA_1:  // Single command (coil write)
    int coil_value = queued_item->iec_obj.o.type58.scs
    int address = Config_db[j].modbus_address

    modbus_set_slave(ctx, Config_db[j].slave_id)
    rc = modbus_write_bit(ctx, address, coil_value)

    if(rc != 1) {
        fprintf(stderr, "Write coil failed: %s\n", modbus_strerror(errno))
    }
    break

case C_SE_TC_1:  // Setpoint command (register write)
    float value = queued_item->iec_obj.o.type63.sv
    int address = Config_db[j].modbus_address

    // Convert float to 2 registers
    modbus_set_float(value, tab_rp_registers)

    modbus_set_slave(ctx, Config_db[j].slave_id)
    rc = modbus_write_registers(ctx, address, 2, tab_rp_registers)

    if(rc != 2) {
        fprintf(stderr, "Write registers failed\n")
    }
    break
```

Command Result Publishing: Drivers optionally publish command execution results back to the monitor direction topic for UI notification:

```
// Build command result item
struct iec_item result_item
result_item.iec_type = queued_item->iec_type  // Echo command type
result_item.iec_obj.ioa = queued_item->iec_obj.ioa
result_item.cause = (rc == SUCCESS) ? 0x07 : 0x0A  // Activation/Deactivation confirmation/d
result_item.is_neg = (rc != SUCCESS) ? 1 : 0  // Negative flag on error

ORTEPublicationSend(publisher)  // Publish result
```

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 1343-1608

Data Flow Performance Characteristics

Latency Budget

End-to-end latency from field device value change to HMI display update:

Stage	Typical Latency	Source
Field device poll cycle	100-2000 ms	Driver polling_time configuration
Driver processing + publish	1-5 ms	In-process conversion and memcpy
ORTE middleware transport	< 1 ms	Shared memory IPC
Monitor Tick() processing	0-100 ms	Waits for next 100ms Tick cycle
FIFO get + validation	< 1 ms	Ring buffer operation
Results processing + DB write	5-20 ms	FastDB in-memory write
UI UpdateTags signal	1-10 ms	Qt signal/slot mechanism
Widget repaint	10-50 ms	Qt rendering pipeline
Total monitoring direction	**117-2187 ms**	Dominated by poll cycle

Control direction (HMI command to field device):

- Command generation: 1-5 ms
- Dispatcher IPC: 5-20 ms
- Monitor processing + IOA lookup: 10-50 ms
- ORTE publish + driver receive: < 1 ms
- Driver command execution: 50-500 ms (protocol dependent)
- **Total control direction: 66-576 ms**

Throughput Characteristics

Maximum theoretical message rates:

Component	Max Rate	Limiting Factor
Single driver	1000 items/sec	ORTE publication rate
Monitor FIFO	65535 items buffered	FIFO size / sizeof(iec_item)
Monitor processing	500-1000 items/sec	Tick() cycle (100ms) + DB queries
Results processor	100-500 items/sec	GigaBASE write latency
UI updates	60 items/sec	Qt repaint rate (60 Hz)

Practical limits: System typically operates at 10-50 items/sec per driver due to polling intervals.

Deadband Filtering Impact

Deadband reduces message traffic by filtering out minor value changes:

```
Without deadband:  Polling every 1s → 86,400 messages/day per point
With 1% deadband:  ~100-1000 messages/day per point (99% reduction)
```

Example: Temperature sensor with ±0.1°C noise and 0.5°C deadband produces ~20 messages/day vs. 86,400 without filtering.

Sources: Performance characteristics inferred from `src/monitor/monitor.cpp` 35-40 `src/drivers/modbus/protocol/modbus_imp.cpp` 253-388

Error Handling and Recovery

Checksum Failure Handling

All message transfers include CRC-8 checksums. On validation failure, processes exit immediately:

```
unsigned char rc = clearCrc((unsigned char *)buf, sizeof(struct iec_item))
if(rc != 0) {
    ExitProcess(1)  // FATAL: Corrupted message detected
}
```

This aggressive strategy prevents corrupted data from propagating. The manager.exe supervisor detects process exit and restarts the driver.

Communication Loss Detection

Driver to Field Device:

```
if(comm_error_counter >= db_n_rows) {
    // All polling attempts failed - connection lost
    item_to_send.iec_type = C_LO_ST_1  // Lost status
    item_to_send.cause = 0x03  // Spontaneous
    ORTEPublicationSend(publisher)  // Notify monitor

    is_connected = false
    modbus_close(ctx)

    // Enter retry loop
    while(!is_connected) {
        if(modbus_connect(ctx) == -1) {
            Sleep(5000)  // Wait 5s before retry
        } else {
            is_connected = true
            general_interrogation = true  // Re-poll all points
        }
    }
}
```

Monitor to Driver:

```
#define USE_KEEP_ALIVE_WATCH_DOG
gl_timeout_connection_with_parent++

if(gl_timeout_connection_with_parent > 1000*60/pollingTime) {
    break  // Exit loop - parent presumed dead
}
// Named pipe keep-alive resets this counter periodically
```

ORTE Middleware Failures

ORTE provides automatic retry and buffering:

- **Persistence**: Messages buffered for 5 seconds if subscriber unavailable
- **Deadline**: Subscriber notified via DEADLINE event if no data for 3 seconds
- **Strength**: Publisher priority for conflict resolution (default = 1)

Monitor detects communication failures via state tracking:

```
if(State == STATE_FAIL) {
    if(p_item->iec_type != C_LO_ST_1) {
        // Driver reconnected - exit fail state
        UnFailUnit(msg)
        State = STATE_ASK_GENERAL_INTERROGATION
    }
}
```

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 253-388 `src/drivers/modbus/protocol/main_master.cpp` 28-54

Summary

The IndigoSCADA data flow pipeline implements a layered architecture with clear separation of concerns:

1. **Protocol drivers** normalize diverse field protocols to the universal `iec_item` format
2. **ORTE middleware** provides low-latency pub/sub transport with automatic buffering
3. **Monitor process** centralizes routing, validation, and database coordination
4. **Results processor** evaluates alarms and manages historical storage
5. **HMI** subscribes to database changes and updates widgets reactively

Key design patterns:

- **Universal data format** (`iec_item`) enables protocol-agnostic core
- **Checksum validation** at every boundary prevents corruption propagation
- **Deadband filtering** reduces message traffic by 90-99%
- **Topic-based routing** enables dynamic addition of protocol drivers
- **Aggressive error handling** (ExitProcess on corruption) combined with supervisor restart ensures data integrity

The bidirectional pipeline supports both monitoring (device → HMI) and control (HMI → device) with latencies suitable for SCADA applications (100ms - 2s).

Sources: Overall architecture from `src/monitor/monitor.cpp` `src/drivers/modbus/protocol/modbus_imp.cpp` `src/drivers/iec104/iec104driver_instance.cpp` `src/ui/user.cpp`

IEC 60870-5-104 Driver

› Relevant source files

Purpose and Scope

This document describes the IEC 60870-5-104 protocol driver implementation in IndigoSCADA. The IEC 104 driver enables communication with IEC 60870-5-104 compliant devices (RTUs, PLCs, substations) over TCP/IP networks. It implements a master/client role, polling field devices and executing control commands.

For general protocol driver architecture patterns, see Protocol Driver Architecture. For other protocol implementations, see Protocol Drivers.

Architecture Overview

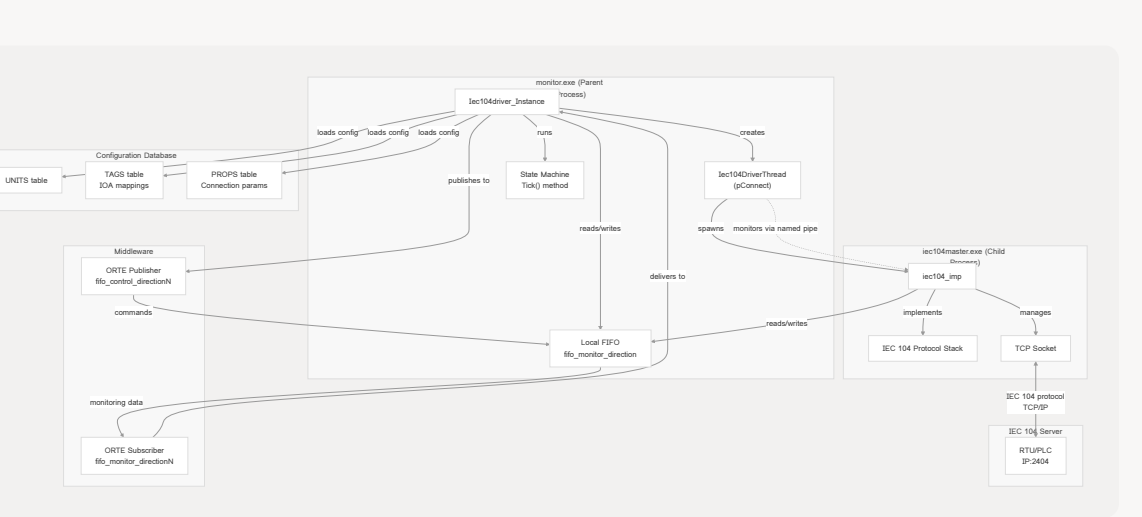
The IEC 104 driver follows IndigoSCADA's common driver architecture pattern with a parent-child process model:

Parent Process Context: `Iec104driver_Instance` class running in `monitor.exe` manages configuration, state machine, and middleware integration.

Child Process: `iec104master.exe` containing `iec104_imp` class handles TCP socket communication and IEC 104 protocol stack.

Communication Channels:

- Local FIFO buffers for parent-child IPC
- ORTE/RTPS publish-subscribe for system-wide data distribution
- Named pipes for process health monitoring


monitor.exe (Parent Process)
Iec104driver_Instance
loads config
loads config
loads config
runs
creates
Configuration Database
UNITS table
TAGS table
IOA mappings
PROPS table
Connection params
publishes to
State Machine
Tick() method
Iec104DriverThread
(pConnect)
reads/writes
spawns
monitors via named pipe
Middleware
ORTE Publisher
fifo_control_directionN
commands
delivers to
iec104master.exe (Child Process)
iec104_imp
implements
manages
IEC 104 Protocol Stack
TCP Socket
Local FIFO
fifo_monitor_direction
reads/writes
IEC 104 protocol
TCP/IP
IEC 104 Server
RTU/PLC
IP:2404
monitoring data
ORTE Subscriber
fifo_monitor_directionN

Sources: `src/drivers/iec104/iec104driver_instance.h` 1-336 `src/drivers/iec104/iec104driver_instance.cpp` 122-144

Key Components

Iec104driver_Instance Class

Primary driver instance class inheriting from `DriverInstance`. Manages one communication line to an IEC 104 server.

Key Members:

- `pConnect` - Pointer to `Iec104DriverThread` managing child process
- `State` - Current state machine state
- `Values[]` - Array of `Track` structures for sample point tracking
- `Cfg` - `InstanceCfg` structure with connection parameters
- `ioa_name_map` - IOANameMap mapping IOA addresses to sample point names
- `fifo_monitor_direction` - Local FIFO handle
- `publisher/subscriber` - ORTE middleware handles

Key Methods:

- `Start()` - Initialize and load configuration
- `Stop()` - Shutdown driver and child process
- `Tick()` - State machine execution (called every 300ms)
- `Connect()` - Spawn child process via `Iec104DriverThread`
- `get_items_from_local_fifo()` - Process incoming monitoring data

Sources: `src/drivers/iec104/iec104driver_instance.h` 62-336 `src/drivers/iec104/iec104driver_instance.cpp` 142-289

Iec104DriverThread Class

Manages the child process lifecycle using the `DriverThread` pattern. Creates and monitors `iec104master.exe`.

Sources: `src/drivers/iec104/iec104driverthread.h` (referenced but not provided)

iec104_imp Class

Protocol implementation running in child process `iec104master.exe`. Handles TCP connection, IEC 104 protocol framing, and command/data exchange.

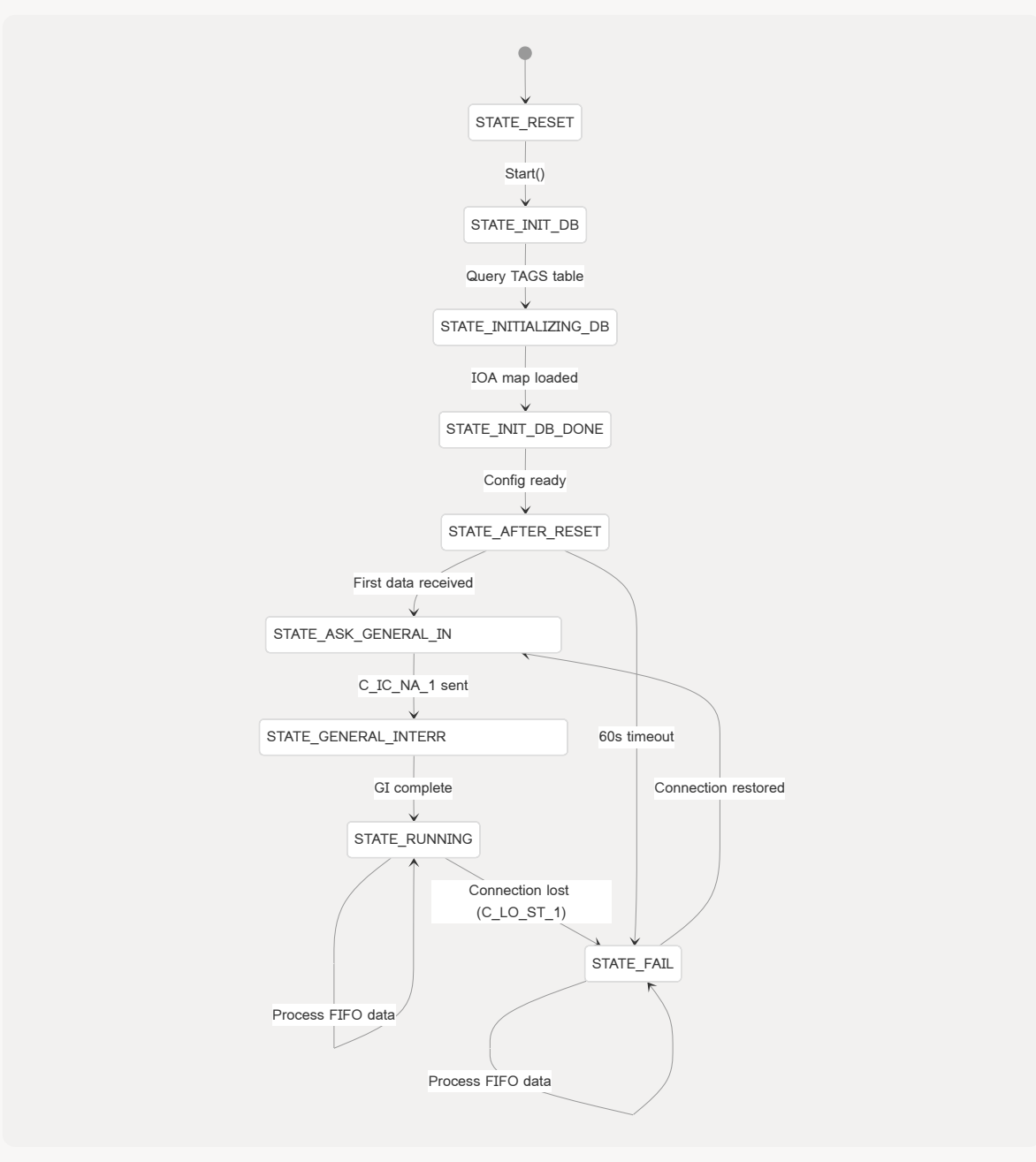
Key Responsibilities:

- TCP socket management and reconnection
- IEC 104 APDU encoding/decoding
- General interrogation (GI) execution
- Periodic data polling
- Command execution with timestamp validation

Sources: `src/drivers/iec104/protocol/iec104_imp.cpp` 1-1200

State Machine

The driver implements a state machine in `Iec104driver_Instance::Tick()` called every 300ms (TICK_MS).


STATE_RESET
Start()
STATE_INIT_DB
Query TAGS table
STATE_INITIALIZING_DB
IOA map loaded
STATE_INIT_DB_DONE
Config ready
STATE_AFTER_RESET
First data received
STATE_ASK_GENERAL_IN
C_IC_NA_1 sent
STATE_GENERAL_INTERR
60s timeout
Connection restored
GI complete
STATE_RUNNING
Connection lost
(C_LO_ST_1)
STATE_FAIL
Process FIFO data
Process FIFO data

State Descriptions:

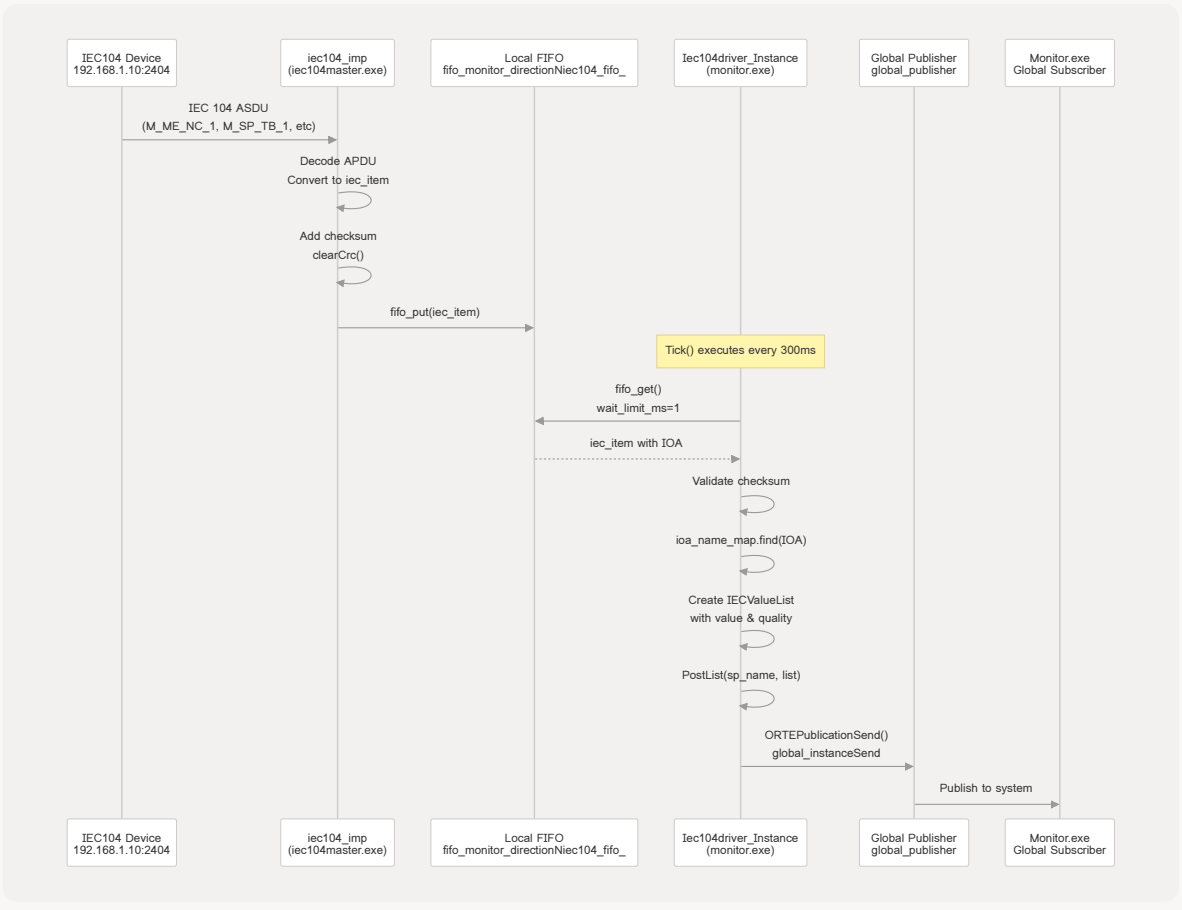
State	Description
`STATE_RESET`	Initial state after driver start
`STATE_INIT_DB`	Begin loading IOA-to-name mappings from TAGS table
`STATE_INITIALIZING_DB`	Database query in progress
`STATE_INIT_DB_DONE`	IOA mappings loaded successfully
`STATE_AFTER_RESET`	Waiting for first message from child process
`STATE_ASK_GENERAL_INTERROGATION`	Send C_IC_NA_1 (General Interrogation)
`STATE_GENERAL_INTERROGATION_DONE`	Waiting for GI completion
`STATE_RUNNING`	Normal operation - processing data
`STATE_FAIL`	Connection lost or timeout

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 661-750 `src/drivers/iec104/iec104driver_instance.h` 122-134

Data Flow Architecture

Monitoring Direction (Device → SCADA)

Data flows from IEC 104 devices through the driver to the IndigoSCADA system:


IEC104 Device
192.168.1.10:2404
iec104_imp
(iec104master.exe)
Local FIFO
fifo_monitor_directionNiec104_fifo_
Iec104driver_Instance
(monitor.exe)
Global Publisher
global_publisher
Monitor.exe
Global Subscriber
IEC 104 ASDU
(M_ME_NC_1, M_SP_TB_1, etc)
Decode APDU
Convert to iec_item
Add checksum
clearCrc()
fifo_put(iec_item)
Tick() executes every 300ms
fifo_get()
wait_limit_ms=1
iec_item with IOA
Validate checksum
ioa_name_map.find(IOA)
Create IECValueList
with value & quality
PostList(sp_name, list)
ORTEPublicationSend()
global_instanceSend
Publish to system

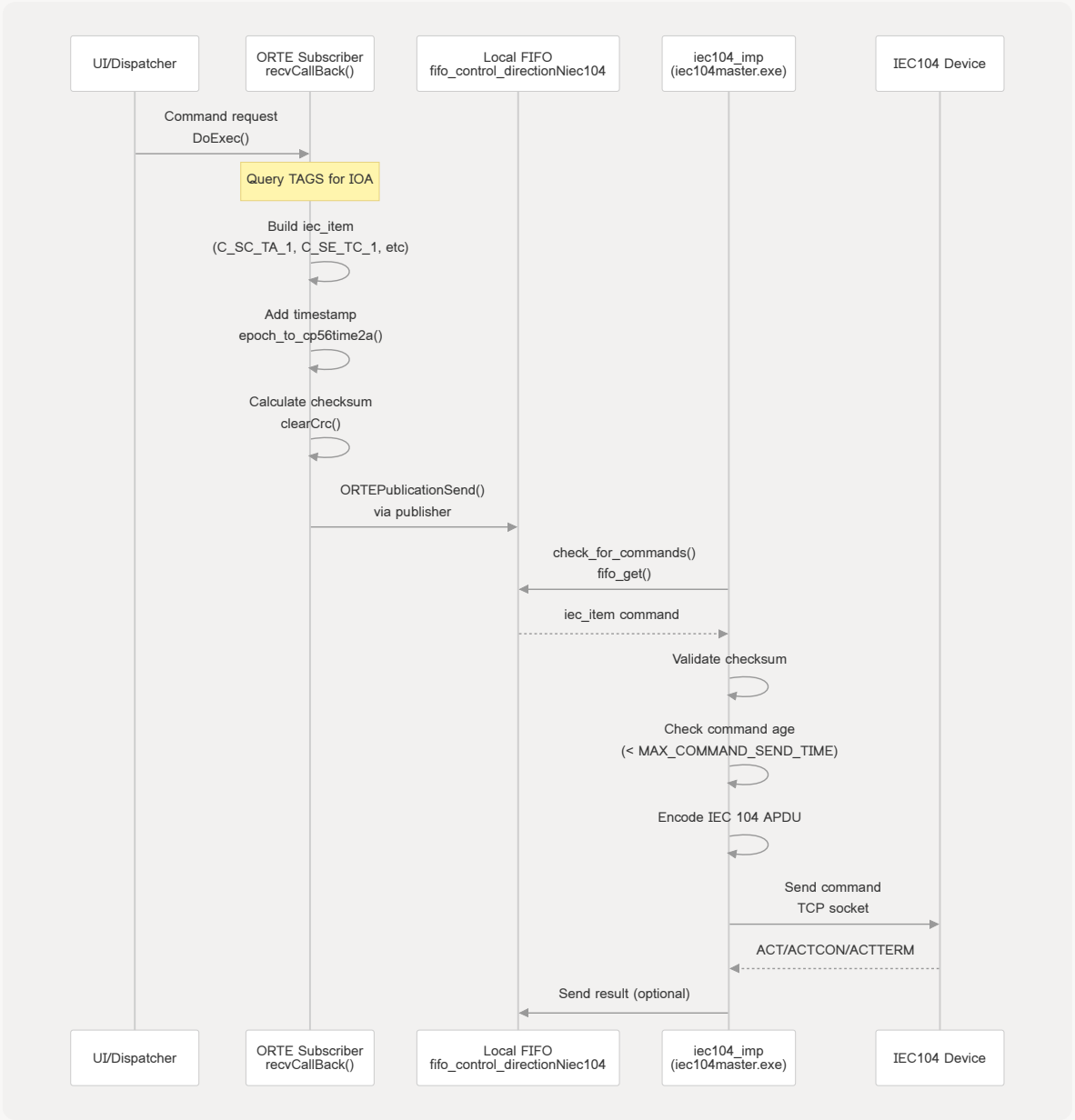
Key Functions:

- `get_items_from_local_fifo()` - Reads from FIFO with 1ms timeout
  `src/drivers/iec104/iec104driver_instance.cpp` 753-1253
- IOA lookup using `ioa_name_map` QMap
  `src/drivers/iec104/iec104driver_instance.cpp` 833-835
- Type-specific handling for each ASDU type
  `src/drivers/iec104/iec104driver_instance.cpp` 837-1229

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 753-1253

Control Direction (SCADA → Device)

Commands flow from UI/monitor through middleware to the device:


UI/Dispatcher
ORTE Subscriber
recvCallBack()
Local FIFO
fifo_control_directionNiec104
iec104_imp
(iec104master.exe)
IEC104 Device
Command request
DoExec()
Query TAGS for IOA
Build iec_item
(C_SC_TA_1, C_SE_TC_1, etc)
Add timestamp
epoch_to_cp56time2a()
Calculate checksum
clearCrc()
ORTEPublicationSend()
via publisher
check_for_commands()
fifo_get()
iec_item command
Validate checksum
Check command age
(< MAX_COMMAND_SEND_TIME)
Encode IEC 104 APDU
Send command
TCP socket
ACT/ACTCON/ACTTERM
Send result (optional)

Command Types Supported:

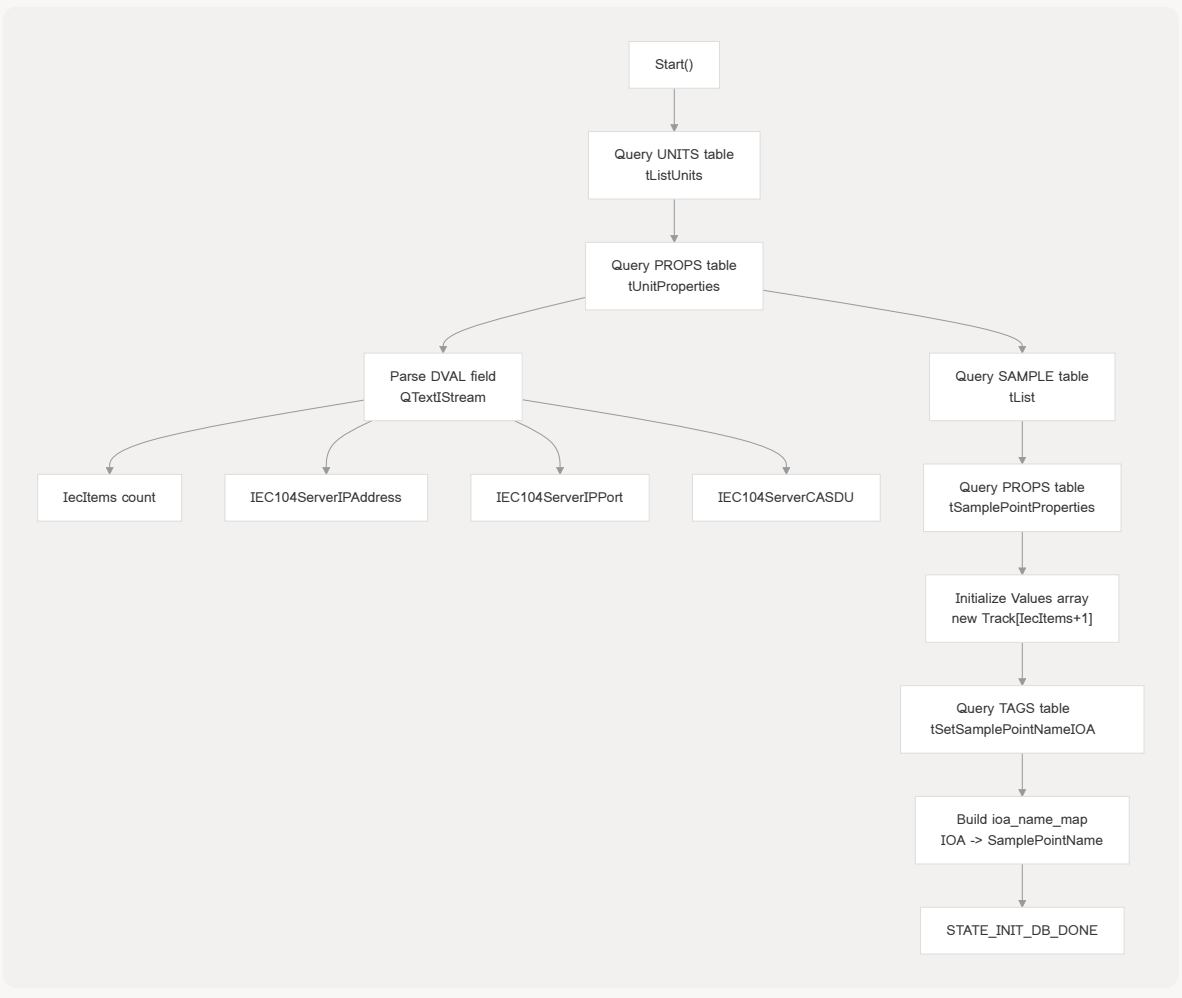
- `C_IC_NA_1` - General interrogation
  `src/drivers/iec104/iec104driver_instance.cpp` 346-356
- `C_SC_NA_1` / `C_SC_TA_1` - Single command
  `src/drivers/iec104/iec104driver_instance.cpp` 358-397
- `C_DC_NA_1` / `C_DC_TA_1` - Double command
  `src/drivers/iec104/iec104driver_instance.cpp` 371-412
- `C_SE_NA_1` / `C_SE_TA_1` - Set point normalized
  `src/drivers/iec104/iec104driver_instance.cpp` 414-466
- `C_SE_NB_1` / `C_SE_TB_1` - Set point scaled
  `src/drivers/iec104/iec104driver_instance.cpp` 427-480
- `C_SE_NC_1` / `C_SE_TC_1` - Set point short float
  `src/drivers/iec104/iec104driver_instance.cpp` 440-496
- `C_CS_NA_1` - Clock synchronization
  `src/drivers/iec104/iec104driver_instance.cpp` 498-508

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 314-532 `src/drivers/iec104/protocol/iec104_imp.cpp` 94-250

Configuration

Unit Properties Loading

Configuration is loaded from the `configdb` FastDB database through a query cascade:


Start()
Query UNITS table
tListUnits
Query PROPS table
tUnitProperties
Parse DVAL field
QTextIStream
Query SAMPLE table
tList
IecItems count
IEC104ServerIPAddress
IEC104ServerIPPort
IEC104ServerCASDU
Query PROPS table
tSamplePointProperties
Initialize Values array
new Track[IecItems+1]
Query TAGS table
tSetSamplePointNameIOA
Build ioa_name_map
IOA -> SamplePointName
STATE_INIT_DB_DONE

Configuration Parameters (from PROPS DVAL field):

Parameter	Type	Description
`IecItems`	int	Number of data points
`IEC104ServerIPAddress`	QString	Server IP (e.g., "192.168.1.10")
`IEC104ServerIPPort`	int	TCP port (default 2404)
`IEC104ServerCASDU`	int	Common Address of ASDU

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 157-288

IOA Mapping

The driver maps IEC 104 Information Object Addresses (IOA) to IndigoSCADA sample point names using the `TAGS` table:

```
select * from TAGS where UNIT='[unit_name]';
```

Results populate `ioa_name_map` (QMap<int, QString>):

```
IOA 100 → "Voltage_L1"
IOA 101 → "Voltage_L2"
IOA 200 → "Breaker_Status"
```

When data arrives with IOA=100, the driver looks up "Voltage_L1" and calls `PostList("Voltage_L1", values)` to update the sample point.

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 534-545 `src/drivers/iec104/iec104driver_instance.cpp` 833-1229

Protocol Implementation Details

Supported ASDU Types (Monitoring Direction)

The driver processes the following IEC 104 ASDU types:

ASDU Type	Description	Handler Location
`M_SP_NA_1`	Single-point information	`iec104driver_instance.cpp` 839-860
`M_DP_NA_1`	Double-point information	`iec104driver_instance.cpp` 862-884
`M_ME_NA_1`	Measured value, normalized	`iec104driver_instance.cpp` 886-908
`M_ME_NB_1`	Measured value, scaled	`iec104driver_instance.cpp` 910-932
`M_ME_NC_1`	Measured value, short float	`iec104driver_instance.cpp` 934-956
`M_SP_TB_1`	Single-point with CP56Time2a	`iec104driver_instance.cpp` 958-980
`M_DP_TB_1`	Double-point with CP56Time2a	`iec104driver_instance.cpp` 982-1004
`M_ME_TD_1`	Measured normalized with time	`iec104driver_instance.cpp` 1011-1033
`M_ME_TE_1`	Measured scaled with time	`iec104driver_instance.cpp` 1035-1057
`M_ME_TF_1`	Measured short float with time	`iec104driver_instance.cpp` 1058-1080
`M_ME_TN_1`	Measured normalized (extended)	`iec104driver_instance.cpp` 1081-1103
`M_IT_NA_1`	Integrated totals	`iec104driver_instance.cpp` 1105-1127
`M_IT_TB_1`	Integrated totals with time	`iec104driver_instance.cpp` 1129-1151
`M_ME_TO_1`	Measured normalized (extended)	`iec104driver_instance.cpp` 1153-1177
`M_ME_TQ_1`	Measured scaled (extended)	`iec104driver_instance.cpp` 1179-1203

Special Message Types:

- `C_EX_IT_1` - Child process exiting notification `iec104driver_instance.cpp` 1205-1209
- `C_LO_ST_1` - Connection lost, triggers STATE_FAIL `iec104driver_instance.cpp` 1210-1223

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 837-1229

Data Type Conversion

Each ASDU type undergoes conversion to `IECValue` for internal processing:

```
// Example: M_ME_NC_1 (short float)
iec_type13 var = p_item->iec_obj.o.type13;
IECValue v(VALUE_TAG, &var, M_ME_NC_1);
IECValueList l;
l.insert(l.end(), v);
PostList(sp_name, l);
```

The `IECValue` class encapsulates:

- Raw value (integer, float, status bits)
- Quality descriptor (IV, NT, SB, BL)
- Timestamp (if present)
- ASDU type identifier

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 934-948

General Interrogation

Upon entering `STATE_ASK_GENERAL_INTERROGATION`, the driver sends a C_IC_NA_1:

```
struct iec_item item_to_send;
memset(&item_to_send, 0x00, sizeof(struct iec_item));
item_to_send.iec_type = C_IC_NA_1;
item_to_send.iec_obj.ioa = 0;           // Broadcast
item_to_send.iec_obj.o.type100.qoi = 1; // Station interrogation
item_to_send.msg_id = msg_sent_in_control_direction++;
item_to_send.checksum = clearCrc((unsigned char *)&item_to_send, sizeof(struct iec_item));

ORTEPublicationSend(publisher);
```

This requests a full data refresh from the server.

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 693-729

Middleware Integration

ORTE Domain Configuration

The driver initializes ORTE (Open Real-Time Ethernet) middleware on construction:

```
ORTEDomainProp dp;
ORTEDomainPropDefaultGet(&dp);
domain = ORTEDomainAppCreate(ORTE_DEFAULT_DOMAIN, &dp, &events, ORTE_FALSE);
iec_item_type_type_register(domain);
```

Publisher (`fifo_control_directionNiec104`):

- Publishes commands from parent to child process
- Persistence: 5 seconds
- Type: `iec_item_type`

Subscriber (`fifo_monitor_directionNiec104`):

- Receives monitoring data from child process
- Deadline: 3 seconds
- Callback: `recvCallBack()`

Global Publisher (`global_publisher`):

- Broadcasts data to entire IndigoSCADA system
- Other drivers and monitor.exe subscribe to this

Sources: `src/drivers/iec104/iec104driver_instance.h` 188-241 `src/drivers/iec104/iec104driver_instance.cpp` 1327-1383

Local FIFO Buffering

A local FIFO provides buffering between ORTE middleware and application code:

```
const size_t max_fifo_queue_size = MAX_FIFO_SIZE; // 70000 * sizeof(iec_item)
fifo_monitor_direction = fifo_open(fifo_monitor_name, max_fifo_queue_size, iec_call_exit_han
```

The ORTE subscriber callback writes to the FIFO:

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void *recvCallBackParam) {
    Iec104driver_Instance *cl = (Iec104driver_Instance*)recvCallBackParam;
    iec_item_type *item1 = (iec_item_type*)vinstance;

    struct iec_item item2;
    rebuild_iec_item_message(&item2, item1);
    fifo_put(cl->fifo_monitor_direction, (char *)&item2, sizeof(struct iec_item));
}
```

Sources: `src/drivers/iec104/iec104driver_instance.h` 243-248 `src/drivers/iec104/iec104driver_instance.cpp` 89-113

Error Handling and Recovery

Connection Loss Detection

The driver detects connection loss through two mechanisms:

1. Explicit Loss Message (`C_LO_ST_1`):

```
case C_LO_ST_1:
{
    if(State != STATE_FAIL) {
        printf("iec104 master on line %d has lost connection...\n", instanceID + 1);
        QString msg;
        msg.sprintf("iec104 master on line %d has lost connection...", instanceID + 1);
        FailUnit(msg);
        State = STATE_FAIL;
    }
}
```

2. Message Timeout (60 seconds):

```
if(wait_for_message > 60*1000/TICK_MS) { // 60 seconds timeout
    wait_for_message = 0;
    printf("iec104 master on line %d has been stopped...\n", instanceID + 1);
    QString msg;
    msg.sprintf("iec104 master on line %d has been stopped...", instanceID + 1);
    FailUnit(msg);
    State = STATE_FAIL;
    pConnect->TerminateChild();
}
```

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 1210-1253

Automatic Reconnection

When in `STATE_FAIL`, the driver continues processing FIFO data. Upon receiving any non-`C_LO_ST_1` message:

```
if(State == STATE_FAIL) {
    if(p_item->iec_type != C_LO_ST_1) {
        QString msg;
        msg.sprintf("IEC104 master on line %d is now connected to IEC104 server.", instanceI
        UnFailUnit(msg);
        State = STATE_ASK_GENERAL_INTERROGATION;
        printf("IEC104 master on line %d is now connected to IEC104 server.\n", instanceID +
    }
}
```

This triggers a new general interrogation to resynchronize data.

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 766-783

Checksum Validation

All `iec_item` messages include an 8-bit CRC checksum:

```
unsigned char rc = clearCrc((unsigned char *)buf, sizeof(struct iec_item));
if(rc != 0) {
    ExitProcess(1); // Fatal error - corrupted message
}
```

The `clearCrc()` function validates data integrity between processes and across middleware.

Sources: `src/drivers/iec104/iec104driver_instance.cpp` 804-809 `src/drivers/iec104/iec104driver_instance.cpp` 52-87

Multiple Instance Support

IndigoSCADA supports multiple IEC 104 driver instances (communication lines) simultaneously. Each instance:

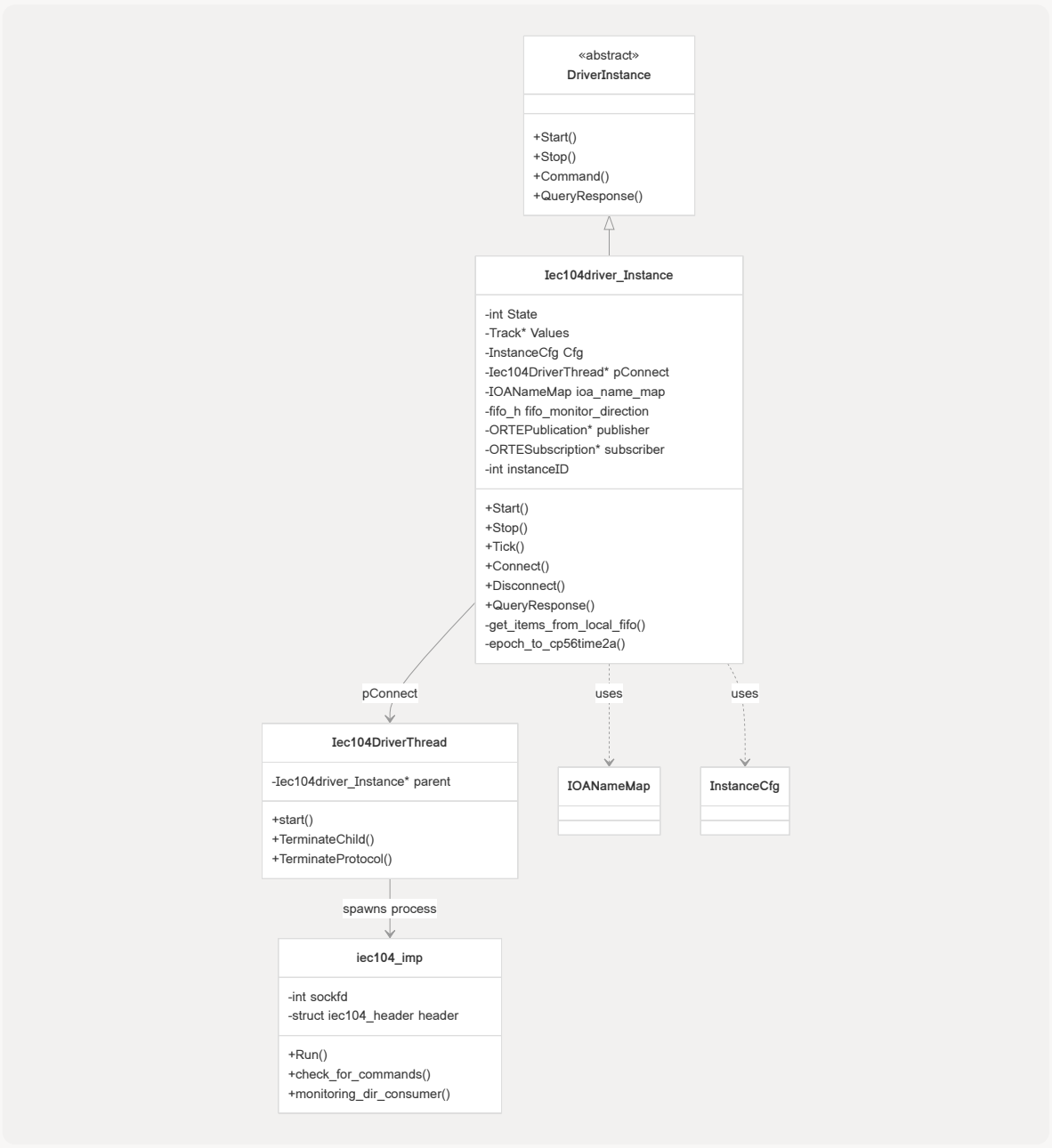
- Has a unique `instanceID` (0-based index)
- Creates unique FIFO names: `fifo_monitor_directionNiec104` where N = instanceID + 1
- Spawns separate `iec104master.exe` child process
- Maintains independent state machine and configuration

Configuration example from `manager.ini`:

```
[iec104driver_1]
[iec104driver_2]
[iec104driver_3]
```

Sources: `src/drivers/iec104/iec104driver_instance.h` 142-146 `src/drivers/iec104/iec104driver_instance.h` 159-170

Class Hierarchy


«abstract»
DriverInstance
+Start()
+Stop()
+Command()
+QueryResponse()
Iec104driver_Instance
-int State
-Track* Values
-InstanceCfg Cfg
-Iec104DriverThread* pConnect
-IOANameMap ioa_name_map
-fifo_h fifo_monitor_direction
-ORTEPublication* publisher
-ORTESubscription* subscriber
-int instanceID
+Start()
+Stop()
+Tick()
+Connect()
+Disconnect()
+QueryResponse()
-get_items_from_local_fifo()
-epoch_to_cp56time2a()
pConnect
uses
uses
Iec104DriverThread
-Iec104driver_Instance* parent
+start()
+TerminateChild()
+TerminateProtocol()
IOANameMap
InstanceCfg
spawns process
iec104_imp
-int sockfd
-struct iec104_header header
+Run()
+check_for_commands()
+monitoring_dir_consumer()

Sources: `src/drivers/iec104/iec104driver_instance.h` 62-336 `src/drivers/iec104/protocol/iec104_imp.cpp` 94-1200

Build and Deployment

Executables:

- `iec104driver.dll` - Driver instance implementation loaded by `monitor.exe`
- `iec104master.exe` - Child process containing protocol implementation

Dependencies:

- `common.lib` - Shared utilities (clearCrc, fifo, etc.)
- `IndentedTrace.lib` - Debug tracing
- ORTE/RTPS middleware libraries
- Qt 3.3.8 (QString, QMap, QTimer)

Configuration Files:

- `C:\scada\project\protocol_databaseN.db` - SQLite configuration (deprecated path)
- FastDB `configdb` - UNITS, TAGS, PROPS, SAMPLE tables

Sources: `src/drivers/iec104/iec104driver_instance.h 1-20` Diagram 1 in high-level architecture

Modbus Driver

› Relevant source files

Purpose and Scope

This page documents the Modbus protocol driver in IndigoSCADA, which provides connectivity to Modbus TCP and Modbus RTU devices. The driver supports both **master** (client) and **slave** (server) operational modes, enabling IndigoSCADA to poll Modbus devices for data acquisition and respond to external Modbus masters as a data provider.

For master polling implementation details, see 3.2.1. For slave server implementation, see 3.2.2. For general protocol driver architecture patterns, see 2.1.

Overview

The Modbus driver consists of two separate executable implementations:

- **modbus_master.exe** - Polls Modbus devices (master/client mode)
- **modbus_slave.exe** - Responds to Modbus requests (slave/server mode)

Both executables integrate with IndigoSCADA through:

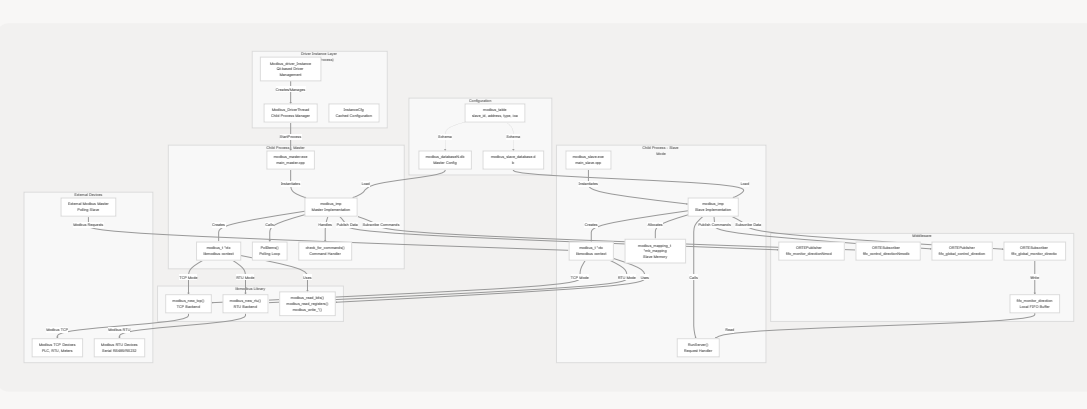
- **ORTE middleware** for publishing/subscribing to `iec_item_type` messages
- **SQLite configuration databases** defining register mappings
- **Named pipes** for keep-alive communication with the parent monitor process (master only)

The driver uses the **libmodbus** library for protocol implementation, providing transport abstraction between TCP and RTU (serial) backends.

Sources: `src/drivers/modbus/protocol/modbus_imp.h` 1-132
`src/drivers/modbus/protocol/modbus_slave_imp.h` 1-136
`src/drivers/modbus/protocol/main_master.cpp` 1-355
`src/drivers/modbus/protocol/main_slave.cpp` 1-232

Architecture

Component Structure



Sources: `src/drivers/modbus/modbus_driver_instance.h` 59-332
`src/drivers/modbus/protocol/modbus_imp.h` 77-132
`src/drivers/modbus/protocol/modbus_slave_imp.h` 78-134

Configuration Database Schema

Both master and slave modes use SQLite databases to map Modbus registers to IndigoSCADA Information Object Addresses (IOAs).

Database Structure

Column	Type	Description
`slave_id`	int	Modbus slave/unit ID (1-247)
`modbus_function_read`	int	Function code for reading (0x01-0x04)
`modbus_function_write`	int	Function code for writing (0x05, 0x06, 0x0F, 0x10)
`modbus_address`	int	Register or coil address (0-65535)
`modbus_type`	varchar	Data type: VT_BOOL, VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R4SWAP
`ioa_control_center`	int	IEC 60870-5 Information Object Address for IndigoSCADA
`deadband`	float	Deadband threshold for analog value change detection

Data Type Mapping

The driver uses OPC-style type constants defined in `src/drivers/modbus/protocol/modbus_db.h` 18-32 :

```
VT_BOOL   = 11  // 1-bit boolean (coils/discrete inputs)
VT_I2     = 2   // 16-bit signed integer
VT_UI2    = 18  // 16-bit unsigned integer
VT_I4     = 3   // 32-bit signed integer (2 registers)
VT_UI4    = 19  // 32-bit unsigned integer (2 registers)
VT_R4     = 4   // 32-bit IEEE float (2 registers, MSB first)
VT_R4SWAP = 22  // 32-bit IEEE float (2 registers, LSB first)
```

Configuration File Locations

- **Master:** `<project_dir>\modbus_database<line_number>.db`
- **Slave:** `<project_dir>\modbus_slave_database.db`

The project directory is read from `project.ini` configuration file.

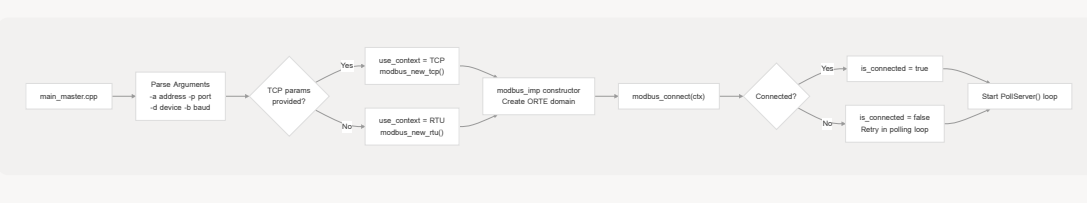
Sources: `src/drivers/modbus/protocol/modbus_db.h` 14-51
`src/drivers/modbus/protocol/load_database_modbus.cpp` 173-254
`src/drivers/modbus/protocol/load_database_modbus_slave.cpp` 173-250

Master Implementation

Initialization and Context Creation

The master driver supports both TCP and RTU transports, selected at runtime via command-line arguments.

Context Selection


main_master.cpp
Parse Arguments
-a address -p port
-d device -b baud
TCP params provided?
Yes
No
use_context = TCP
modbus_new_tcp()
use_context = RTU
modbus_new_rtu()
modbus_imp constructor
Create ORTE domain
modbus_connect(ctx)
Connected?
Yes
No
is_connected = true
is_connected = false
Retry in polling loop
Start PollServer() loop

Command-line parameters:

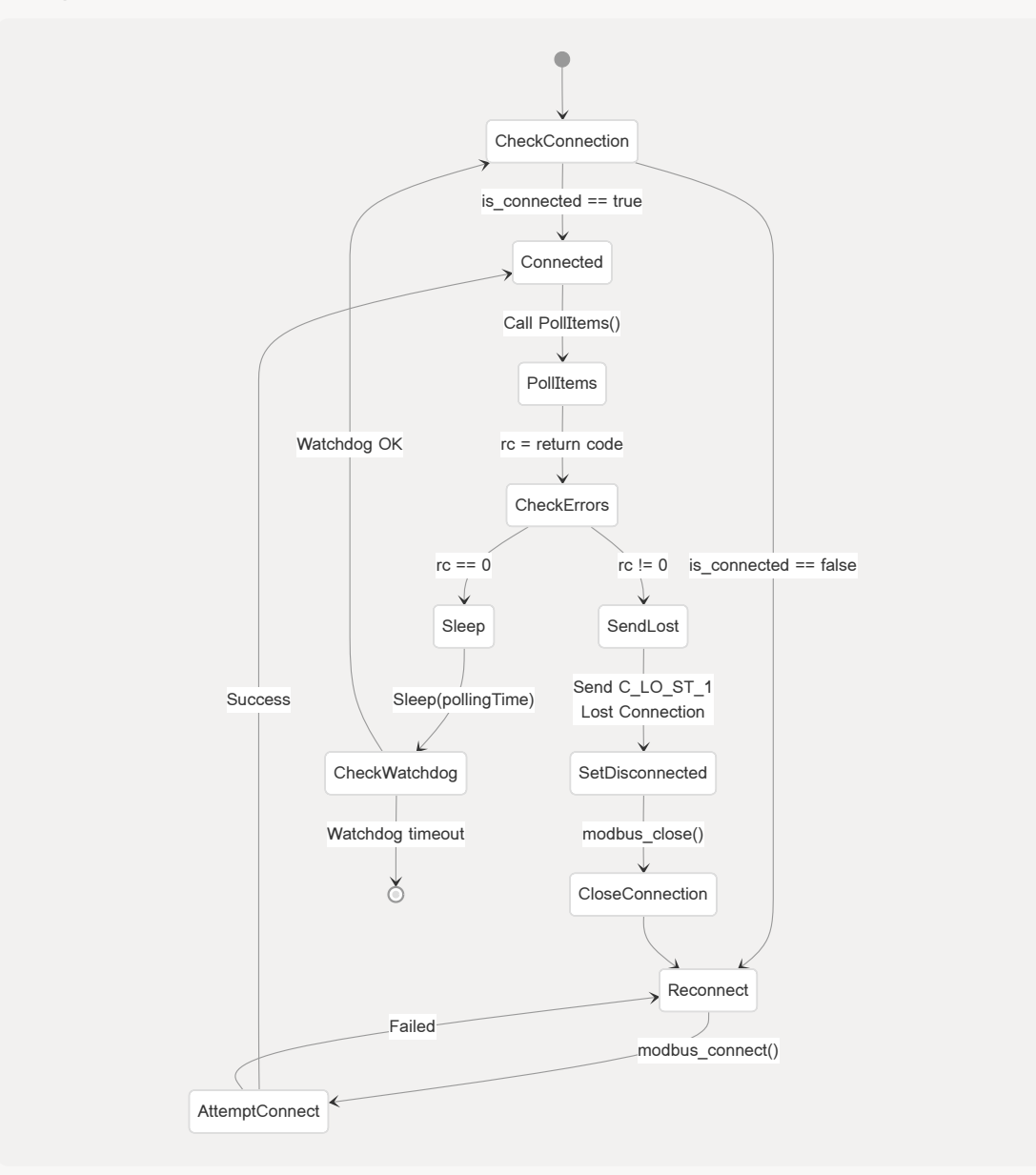
- TCP: `-a <ip> -p <port> -l <line_num> -t <poll_ms>`
- RTU: `-d <device> -b <baud> -c <databits> -e <stopbit> -g <parity> -l <line_num> -t <poll_ms> -u <rts_on_ms> -v <rts_off_ms>`

Sources: `src/drivers/modbus/protocol/main_master.cpp` 72-355
`src/drivers/modbus/protocol/modbus_imp.cpp` 110-236

Polling Loop and Data Acquisition

The master's `PollServer()` method implements the polling loop with connection management and watchdog monitoring.

Polling State Machine


CheckConnection
is_connected == true
Connected
Call PollItems()
PollItems
Watchdog OK
rc = return code
CheckErrors
rc == 0
rc != 0
is_connected == false
Sleep
SendLost
Success
Sleep(pollingTime)
Send C_LO_ST_1
Lost Connection
CheckWatchdog
SetDisconnected
Watchdog timeout
modbus_close()
CloseConnection
Reconnect
Failed
modbus_connect()
AttemptConnect

The `PollItems()` method iterates through all configured database rows and executes the appropriate Modbus read function based on `modbus_function_read`.

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 253-388
`src/drivers/modbus/protocol/modbus_imp.cpp` 562-1228

Function Code Support

The master supports the following Modbus read functions:

Function Code	Name	Description	Address Space
0x01	`FC_READ_COILS`	Read output coils	0x0000-0xFFFF
0x02	`FC_READ_DISCRETE_INPUTS`	Read input contacts	0x0000-0xFFFF
0x03	`FC_READ_HOLDING_REGISTERS`	Read holding registers	0x0000-0xFFFF
0x04	`FC_READ_INPUT_REGISTERS`	Read input registers	0x0000-0xFFFF

Read Operation Example (Holding Registers)

For a configured row with:

- `modbus_function_read = 0x03` (FC_READ_HOLDING_REGISTERS)
- `modbus_type = VT_R4` (32-bit float)
- `modbus_address = 1000`
- `slave_id = 1`

The driver executes at `src/drivers/modbus/protocol/modbus_imp.cpp` 696-767 :

```
modbus_set_slave(ctx, 1);
modbus_read_registers(ctx, 1000, 2, tab_rp_registers);
float value = modbus_get_float(tab_rp_registers);
```

Then converts to IEC 60870-5-104 format:

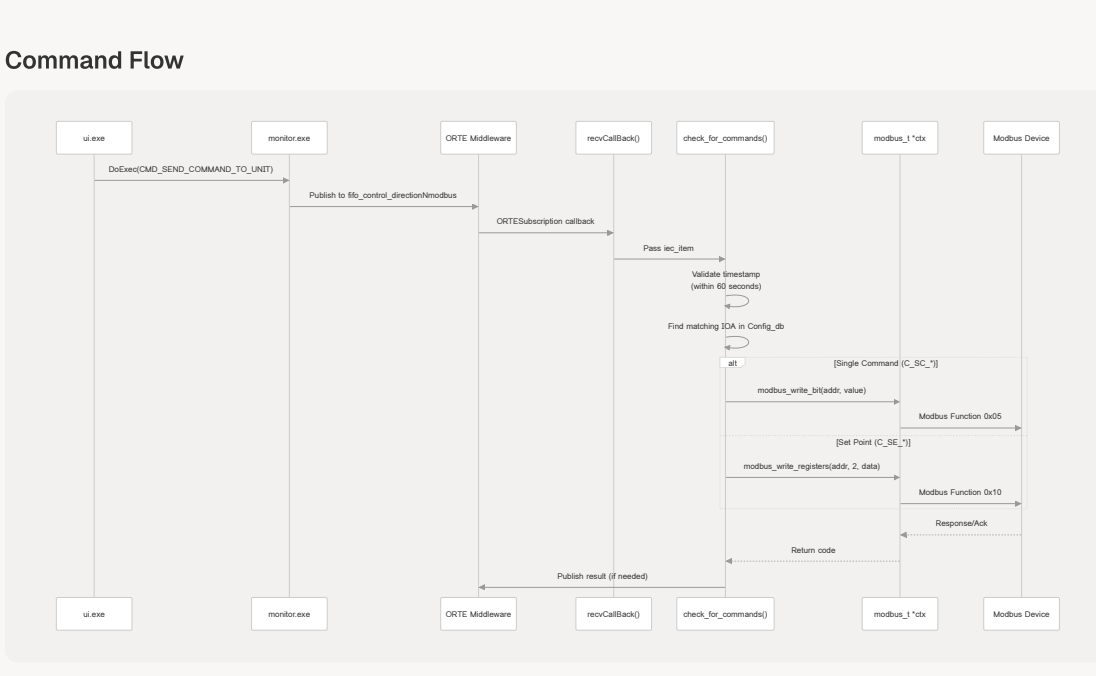
- `iec_type = M_ME_TF_1` (Short floating point with time tag)
- `iec_obj.ioa = ioa_control_center`
- `iec_obj.o.type36.mv = value`

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 586-1155

Command Handling (Control Direction)

Commands from the HMI flow through ORTE middleware to `check_for_commands()`, which maps IEC 104 command types to Modbus write operations.

Command Flow


ui.exe
monitor.exe
ORTE Middleware
recvCallBack()
check_for_commands()
modbus_t *ctx
Modbus Device
DoExec(CMD_SEND_COMMAND_TO_UNIT)
Publish to fifo_control_directionNmodbus
ORTESubscription callback
Pass iec_item
Validate timestamp
(within 60 seconds)
Find matching IOA in Config_db
alt
[Single Command (C_SC_*)]
modbus_write_bit(addr, value)
Modbus Function 0x05
[Set Point (C_SE_*)]
modbus_write_registers(addr, 2, data)
Modbus Function 0x10
Response/Ack
Return code
Publish result (if needed)

Supported command types:

- `C_SC_TA_1` / `C_SC_NA_1` → `modbus_write_bit()` (FC 0x05)
- `C_SE_TA_1` / `C_SE_NA_1` → `modbus_write_registers()` (FC 0x10) for 16-bit values
- `C_SE_TB_1` / `C_SE_NB_1` → `modbus_write_registers()` (FC 0x10) for 32-bit values
- `C_SE_TC_1` / `C_SE_NC_1` → `modbus_write_registers()` (FC 0x10) for float values

Age validation: Commands older than 60 seconds (MAX_COMMAND_SEND_TIME) are rejected.

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 1343-1700

Deadband Filtering

To reduce unnecessary updates, the master implements deadband filtering for analog values. Values are only published if the change exceeds the configured deadband threshold.

```
// Example at modbus_imp.cpp:745-754
if(ABS(Config_db[rowNumber].last_value.f - real) > Config_db[rowNumber].deadband)
{
    Config_db[rowNumber].last_value.f = real;
    send_item = true;
}
else
{
    send_item = false;
}
```

Boolean values use simple change detection without deadband.

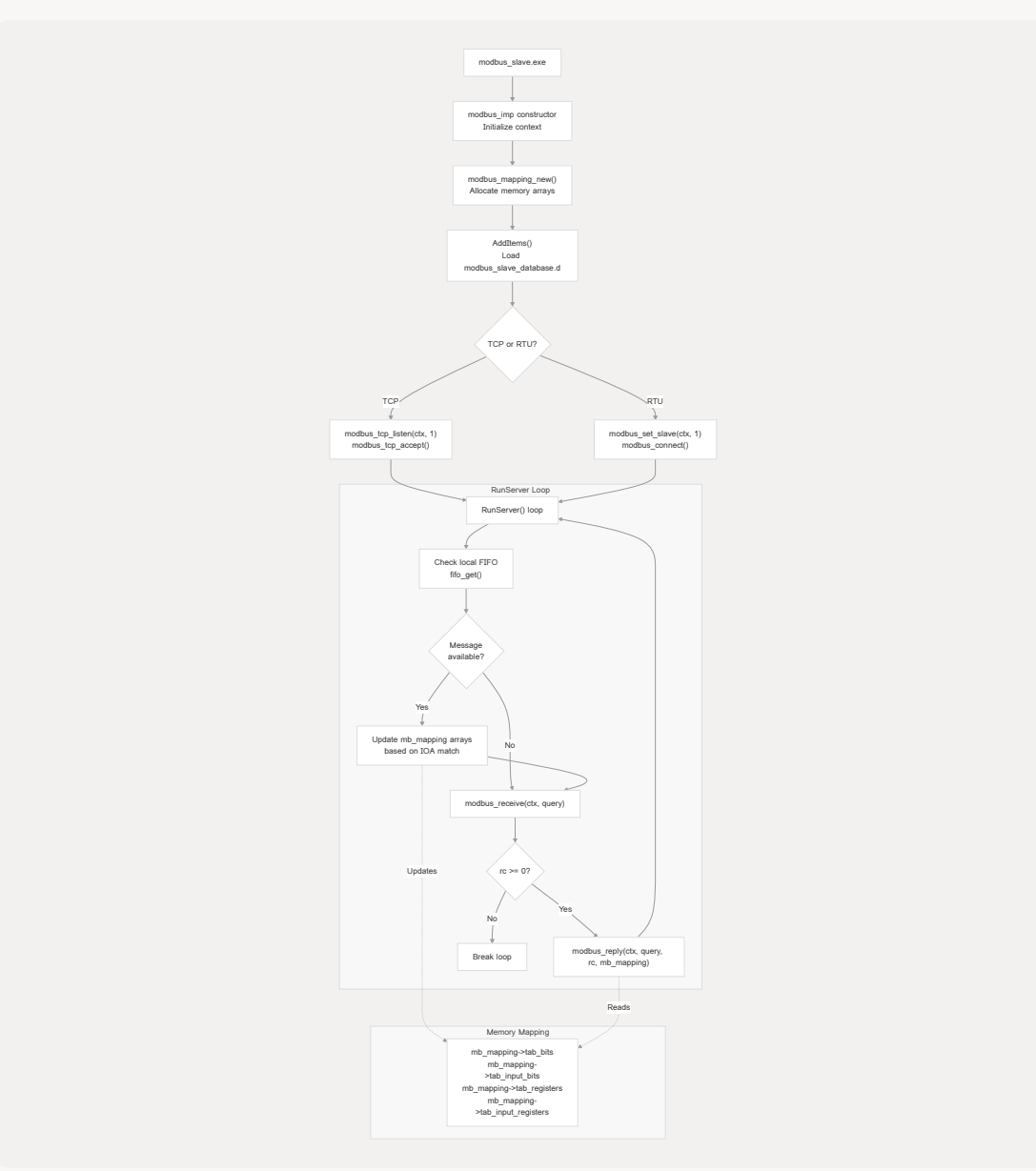
Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 611-620
`src/drivers/modbus/protocol/modbus_imp.cpp` 745-754

Slave Implementation

Server Operation

The slave driver operates as a Modbus server, listening for incoming requests and responding from its internal memory mapping.

Slave Architecture


modbus_slave.exe
modbus_imp constructor
Initialize context
modbus_mapping_new()
Allocate memory arrays
AddItems()
Load
modbus_slave_database.d
TCP or RTU?
TCP
RTU
modbus_tcp_listen(ctx, 1)
modbus_tcp_accept()
modbus_set_slave(ctx, 1)
modbus_connect()
RunServer Loop
RunServer() loop
Check local FIFO
fifo_get()
Message available?
Yes
No
Update mb_mapping arrays
based on IOA match
modbus_receive(ctx, query)
rc >= 0?
No
Yes
Break loop
modbus_reply(ctx, query,
rc, mb_mapping)
Updates
Reads
Memory Mapping
mb_mapping->tab_bits
mb_mapping->tab_input_bits
mb_mapping->tab_registers
mb_mapping->tab_input_registers

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 106-227
`src/drivers/modbus/protocol/modbus_slave_imp.cpp` 244-466
`src/drivers/modbus/protocol/modbus_slave_imp.h` 78-134

Memory Mapping Structure

The slave allocates memory using `modbus_mapping_new()` at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 216-224 :

```
mb_mapping = modbus_mapping_new(
    MODBUS_MAX_READ_BITS,        // nb_bits (coils)
    MODBUS_MAX_READ_BITS,        // nb_input_bits
    MODBUS_MAX_READ_REGISTERS,   // nb_registers (holding)
    MODBUS_MAX_READ_REGISTERS    // nb_input_registers
);
```

This creates four arrays:

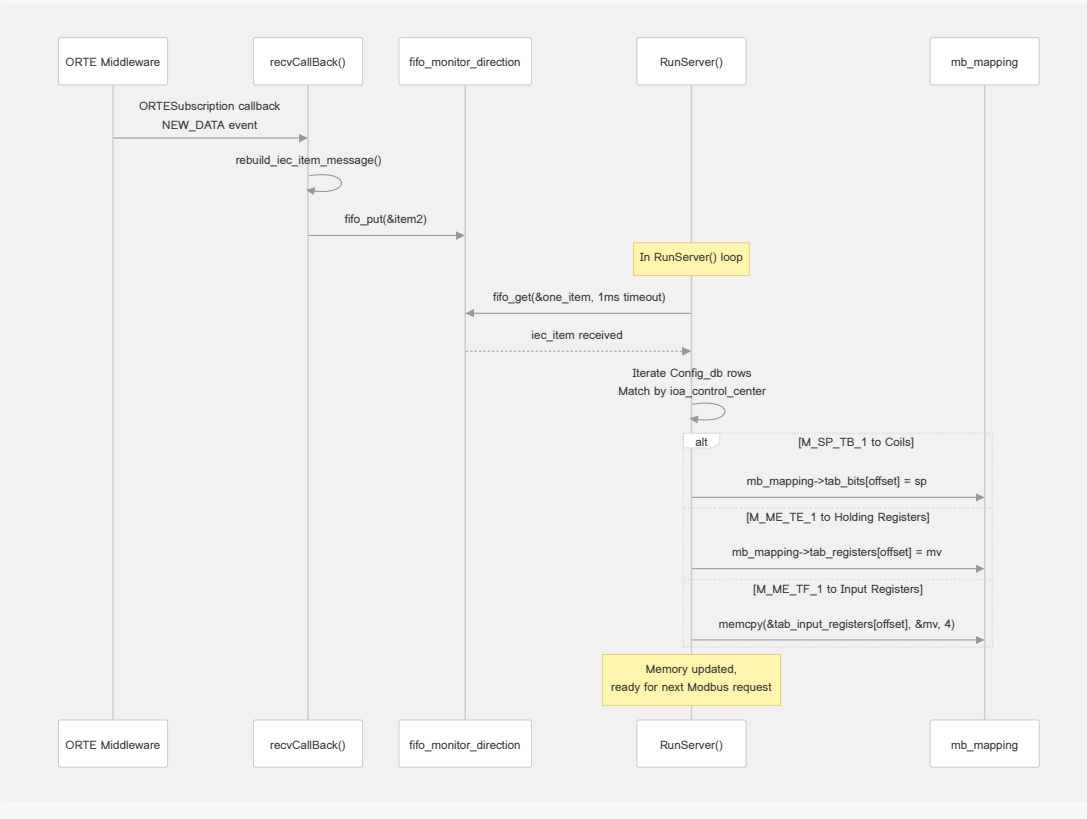
- `tab_bits[]` - Coils (FC 0x01, writable)
- `tab_input_bits[]` - Discrete inputs (FC 0x02, read-only)
- `tab_registers[]` - Holding registers (FC 0x03, writable)
- `tab_input_registers[]` - Input registers (FC 0x04, read-only)

External Modbus masters can read from any array and write to coils/holding registers.

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 216-224
`src/drivers/modbus/protocol/modbus_slave_imp.cpp` 290-448

Updating Slave Memory from Middleware

The slave subscribes to `fifo_global_monitor_direction` and updates its memory mapping when messages arrive matching configured IOAs.


ORTE Middleware
recvCallBack()
fifo_monitor_direction
RunServer()
mb_mapping
ORTESubscription callback
NEW_DATA event
rebuild_iec_item_message()
fifo_put(&item2)
In RunServer() loop
fifo_get(&one_item, 1ms timeout)
iec_item received
Iterate Config_db rows
Match by ioa_control_center
alt
[M_SP_TB_1 to Coils]
mb_mapping->tab_bits[offset] = sp
[M_ME_TE_1 to Holding Registers]
mb_mapping->tab_registers[offset] = mv
[M_ME_TF_1 to Input Registers]
memcpy(&tab_input_registers[offset], &mv, 4)
Memory updated,
ready for next Modbus request

Example mapping logic at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 292-374 :

- If `iec_obj.ioa == Config_db[rowNumber].ioa_control_center`
- And `modbus_function_read == FC_READ_HOLDING_REGISTERS`
- And `iec_type == M_ME_TF_1` (float value)
- Then copy float to `tab_registers[modbus_address]`

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 74-97
`src/drivers/modbus/protocol/modbus_slave_imp.cpp` 255-461

Transport Abstraction (TCP vs RTU)

The driver uses libmodbus to abstract transport differences. Both master and slave select the context at initialization based on command-line parameters.

Context Creation

Transport	Function	Parameters
TCP	`modbus_new_tcp(ip, port)`	IP address, TCP port (default 502)
RTU	`modbus_new_rtu(device, baud, parity, data_bit, stop_bit, rtsOnTime, rtsOffTime)`	Serial device path, baud rate, parity ('N'/'E'/'O'), data/stop bits, RTS timing

RTU-specific Features

For RS485 half-duplex communication, the driver supports **RTS control timing**:

- `rtsOnTime` - Leading time (ms) to assert RTS before transmission
- `rtsOffTime` - Trailing time (ms) to keep RTS asserted after transmission

These parameters are passed at `src/drivers/modbus/protocol/modbus_imp.cpp` 138 and implemented in libmodbus RTU backend.

Common operations (transport-agnostic):

```
modbus_set_slave(ctx, slave_id);           // Set target slave ID
modbus_read_registers(ctx, addr, nb, dest);// Read registers
modbus_write_registers(ctx, addr, nb, src);// Write registers
modbus_connect(ctx);                       // Establish connection
modbus_close(ctx);                         // Close connection
```

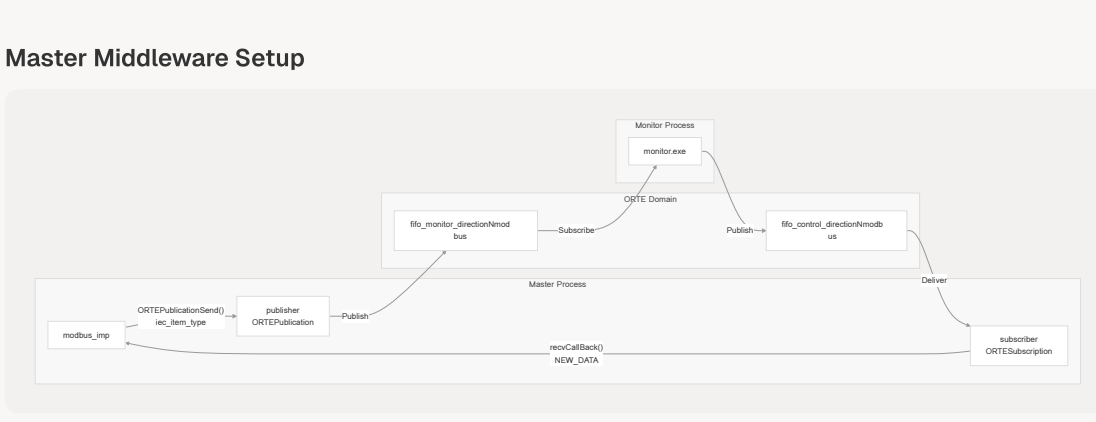
Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 132-139
`src/drivers/modbus/protocol/modbus_imp.h` 32-55
`src/drivers/modbus/protocol/modbus_slave_imp.cpp` 127-134

Middleware Integration

ORTE Publisher/Subscriber Pattern

Both master and slave use ORTE for real-time data exchange with the IndigoSCADA system.

Master Middleware Setup



Topic naming convention:

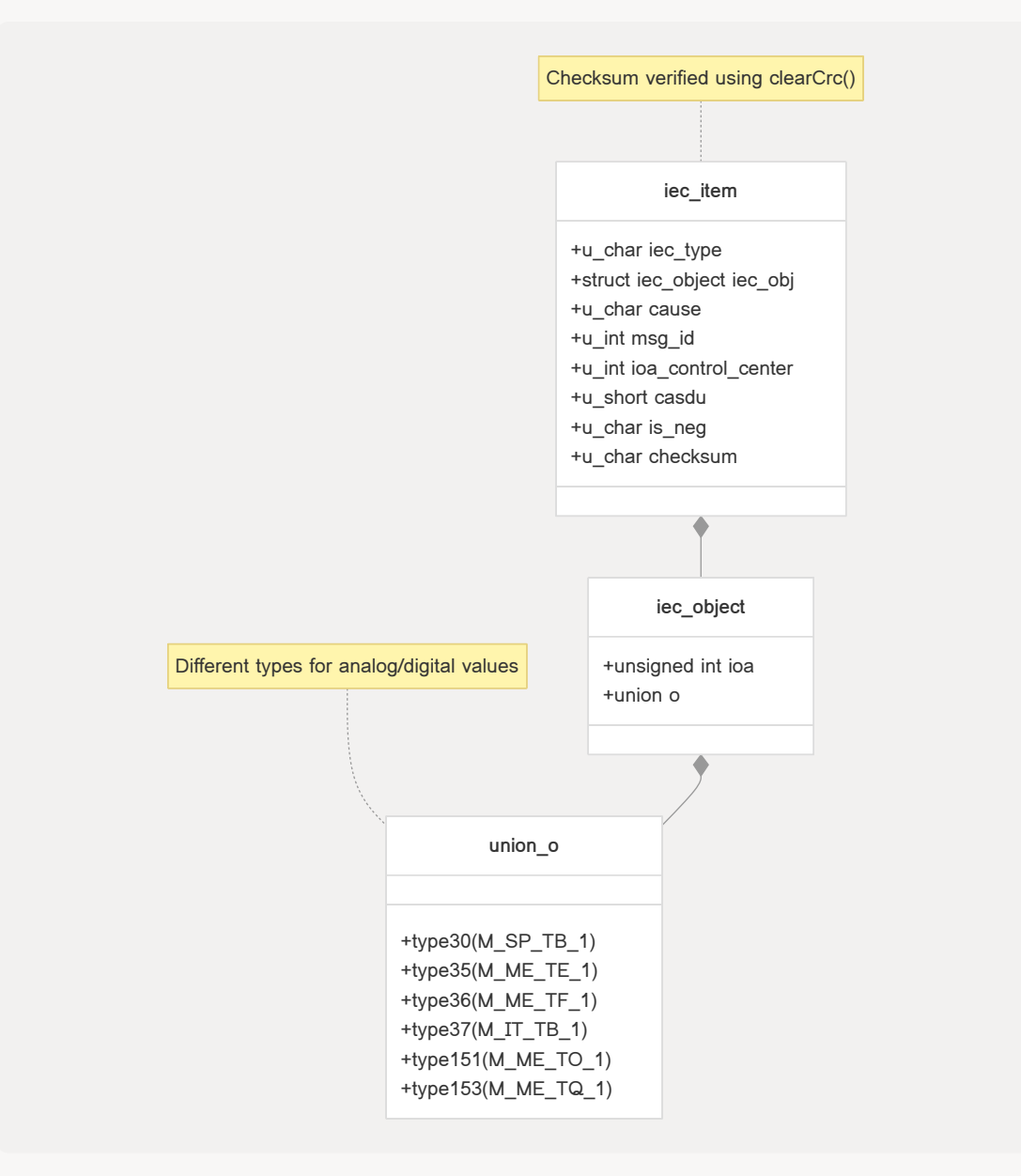
- Monitor direction: `fifo_monitor_direction<line_number>modbus`
- Control direction: `fifo_control_direction<line_number>modbus`

Where `<line_number>` is the driver instance ID (1, 2, 3...).

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 143-211
`src/drivers/modbus/modbus_driver_instance.h` 154-237

IEC 60870-5-104 Message Format

All data exchanged through middleware uses the `iec_item` structure, converted to `iec_item_type` for ORTE transmission.


Checksum verified using clearCrc()
iec_item
+u_char iec_type
+struct iec_object iec_obj
+u_char cause
+u_int msg_id
+u_int ioa_control_center
+u_short casdu
+u_char is_neg
+u_char checksum
iec_object
+unsigned int ioa
+union o
Different types for analog/digital values
union_o
+type30(M_SP_TB_1)
+type35(M_ME_TE_1)
+type36(M_ME_TF_1)
+type37(M_IT_TB_1)
+type151(M_ME_TO_1)
+type153(M_ME_TQ_1)

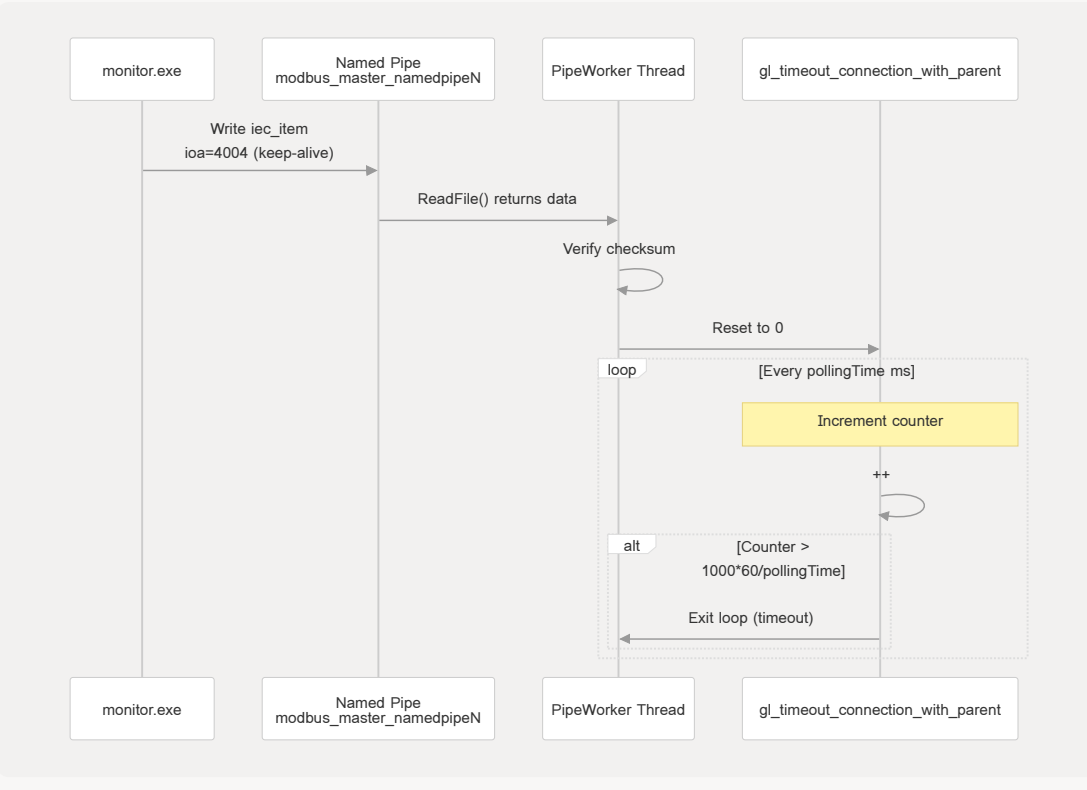
Checksum validation: Every message includes a CRC-8 checksum verified at `src/drivers/modbus/protocol/modbus_imp.cpp` 53-62

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 39-74
`src/drivers/modbus/protocol/modbus_imp.cpp` 1179-1217

Connection Management and Watchdog

Keep-Alive Mechanism (Master Only)

The master maintains a **named pipe** connection with the parent `monitor.exe` process for health monitoring.


monitor.exe
Named Pipe
modbus_master_namedpipeN
PipeWorker Thread
gl_timeout_connection_with_parent
Write iec_item
ioa=4004 (keep-alive)
ReadFile() returns data
Verify checksum
Reset to 0
loop
[Every pollingTime ms]
Increment counter
++
alt
[Counter >
1000*60/pollingTime]
Exit loop (timeout)

Timeout calculation: If no keep-alive received for approximately 60 seconds, the master terminates.

Pipe naming: `\\.\pipe\modbus_master_namedpipe<line_number>`

Sources: `src/drivers/modbus/protocol/main_master.cpp` 361-502
`src/drivers/modbus/protocol/modbus_imp.cpp` 372-381

Reconnection Logic

When communication errors occur, the master attempts automatic reconnection:

```
// At modbus_imp.cpp:348-364
if(!is_connected)
{
    if (modbus_connect(ctx) == -1)
    {
        fprintf(stderr, "Connection failed: %s\n", modbus_strerror(errno));
    }
    else
    {
        is_connected = true;
        general_interrogation = true;  // Re-poll all points
        loops = 0;
    }
}
```

After reconnection, a **general interrogation** is triggered to refresh all data points.

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 343-364
`src/drivers/modbus/protocol/modbus_imp.cpp` 278-294

Error Handling

Communication Error Tracking

The master tracks communication errors per polling cycle using `comm_error_counter`:

```
// At modbus_imp.cpp:1220-1224
if(comm_error_counter >= db_n_rows)
{
    return 1;  // Total failure - all items failed
}
```

When all configured items fail to poll, the driver:

1. Publishes a `C_LO_ST_1` (Lost connection) message
2. Sets `is_connected = false`
3. Closes the Modbus connection
4. Enters reconnection loop

Individual read failures increment `comm_error_counter` but don't immediately terminate the connection, allowing partial failures.

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 573-604
`src/drivers/modbus/protocol/modbus_imp.cpp` 1220-1227

Build Configuration

Master Executable

The Modbus master is built using Visual Studio 6.0 project file `src/drivers/modbus/protocol/modbus_master.dsp`

Key dependencies:

- `utilities.lib` - CRC, time utilities
- `rtps.lib` - ORTE middleware
- `fifo.lib` - Local FIFO buffers
- `sqlite3.c` - Database access
- `ws2_32.lib`, `winmm.lib` - Windows sockets and timing

Output: `c:\scada\bin\modbus_master.exe`

Slave Executable

The Modbus slave uses `src/drivers/modbus/protocol/modbus_slave.dsp` 1-145

Additional requirement: Includes local FIFO implementation for buffer management.

Output: `c:\scada\bin\modbus_slave.exe`

Sources: `src/drivers/modbus/protocol/modbus_slave.dsp` 1-145

Usage Examples

Master Configuration Example

SQLite database entry for reading a 32-bit float holding register:

```
INSERT INTO modbus_table VALUES(
    1,              -- slave_id
    3,              -- modbus_function_read (FC_READ_HOLDING_REGISTERS)
    16,             -- modbus_function_write (FC_WRITE_MULTIPLE_REGISTERS)
    1000,           -- modbus_address
    'VT_R4',        -- modbus_type
    5001,           -- ioa_control_center
    0.1             -- deadband
);
```

This configuration:

- Polls slave ID 1
- Reads 2 holding registers starting at address 1000
- Interprets as 32-bit IEEE float
- Maps to IOA 5001 in IndigoSCADA
- Only publishes if value changes by more than 0.1

Slave Configuration Example

SQLite database entry for exposing a boolean coil:

```
INSERT INTO modbus_table VALUES(
    1,              -- slave_id
    1,              -- modbus_function_read (FC_READ_COILS)
    5,              -- modbus_function_write (FC_WRITE_SINGLE_COIL)
    500,            -- modbus_address
    'VT_BOOL',      -- modbus_type
    6001,           -- ioa_control_center
    0.0             -- deadband (ignored for boolean)
);
```

When IndigoSCADA publishes a `M_SP_TB_1` message with IOA 6001, the slave updates `mb_mapping->tab_bits[500]`, making it available to external Modbus masters.

Sources: `src/drivers/modbus/protocol/load_database_modbus.cpp` 32-169
`src/drivers/modbus/protocol/load_database_modbus_slave.cpp` 32-169

Data Type Conversion Details

Float Handling (VT_R4 vs VT_R4SWAP)

Modbus uses two 16-bit registers to store 32-bit floats. Byte order varies by device:

Standard order (VT_R4):

```
Register[0]: MSW (Most Significant Word)
Register[1]: LSW (Least Significant Word)
```

Swapped order (VT_R4SWAP): At `src/drivers/modbus/protocol/modbus_imp.cpp` 732-741

```
uint16_t tmp1 = tab_rp_registers[0];
uint16_t tmp2 = tab_rp_registers[1];
tab_rp_registers[0] = tmp2;  // Swap
tab_rp_registers[1] = tmp1;
real = modbus_get_float(&tab_rp_registers[0]);
```

This accommodates devices using LSW-first byte order.

Integer Type Conversion

Modbus Type	Registers	C Type	IEC 104 Type	Field
VT_BOOL	1 bit	`uint8_t`	M_SP_TB_1	`type30.sp`
VT_I2	1	`short`	M_ME_TE_1	`type35.mv`
VT_UI2	1	`unsigned short`	M_ME_TQ_1	`type153.mv`
VT_I4	2	`int`	M_IT_TB_1	`type37.counter`
VT_UI4	2	`unsigned int`	M_ME_TO_1	`type151.mv`

Helper functions used:

- `modbus_get_int()` - Extract 32-bit signed from 2 registers
- [illegible] - Extract 32-bit unsigned from 2 registers
- `modbus_get_float()` - Extract IEEE 754 float from 2 registers

Sources: src/drivers/modbus/protocol/modbus_imp.cpp 698-827
src/drivers/modbus/protocol/modbus_imp.cpp 829-920

Modbus Master Implementation

> Relevant source files

Purpose and Scope

This page documents the Modbus master protocol implementation, which enables IndigoSCADA to poll and control Modbus devices. The implementation supports both Modbus TCP and Modbus RTU transports and uses the libmodbus library for low-level protocol handling.

This page covers the core `modbus_imp` class, configuration loading, polling logic, command execution, and data type mapping. For the parent driver coordination and process management, see Protocol Driver Architecture. For the slave implementation, see Modbus Slave Implementation.

Architecture Overview

The Modbus master implementation follows the child process pattern, where `modbus_master.exe` runs as a separate process managed by the parent `Modbus_driver_Instance`. The implementation abstracts TCP and RTU backends through libmodbus and publishes data via ORTE middleware.

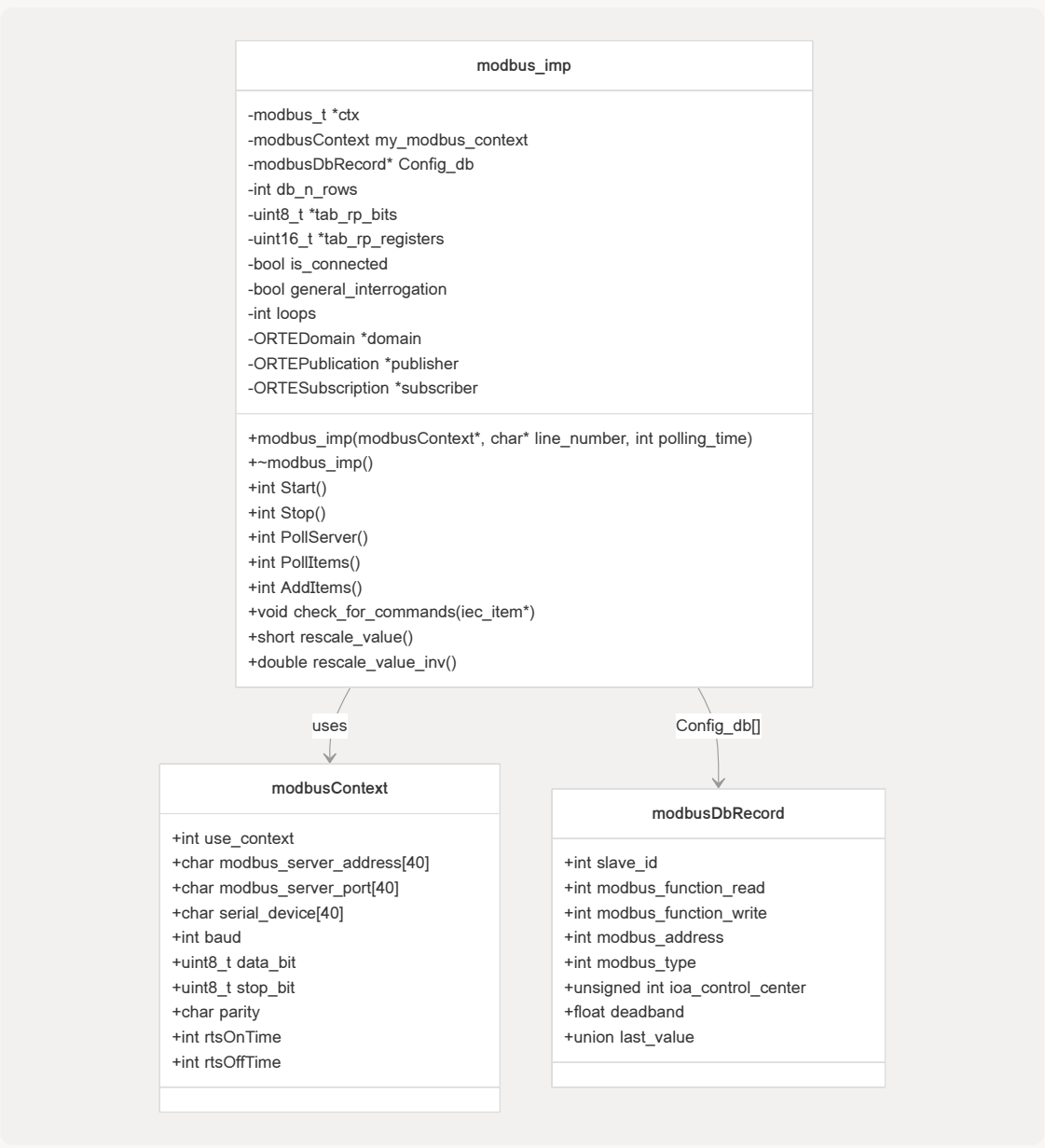
Sources:

- `src/drivers/modbus/protocol/modbus_imp.h` 77-132
- `src/drivers/modbus/protocol/modbus_imp.cpp` 110-236
- `src/drivers/modbus/protocol/main_master.cpp` 72-355

Core Classes and Structures

modbus_imp Class

The `modbus_imp` class is the central implementation that manages the Modbus connection, polling, and command execution.


modbus_imp
-modbus_t *ctx
-modbusContext my_modbus_context
-modbusDbRecord* Config_db
-int db_n_rows
-uint8_t *tab_rp_bits
-uint16_t *tab_rp_registers
-bool is_connected
-bool general_interrogation
-int loops
-ORTEDomain *domain
-ORTEPublication *publisher
-ORTESubscription *subscriber
+modbus_imp(modbusContext*, char* line_number, int polling_time)
+~modbus_imp()
+int Start()
+int Stop()
+int PollServer()
+int PollItems()
+int AddItems()
+void check_for_commands(iec_item*)
+short rescale_value()
+double rescale_value_inv()
uses
Config_db[]
modbusContext
+int use_context
+char modbus_server_address[40]
+char modbus_server_port[40]
+char serial_device[40]
+int baud
+uint8_t data_bit
+uint8_t stop_bit
+char parity
+int rtsOnTime
+int rtsOffTime
modbusDbRecord
+int slave_id
+int modbus_function_read
+int modbus_function_write
+int modbus_address
+int modbus_type
+unsigned int ioa_control_center
+float deadband
+union last_value

Sources:

- `src/drivers/modbus/protocol/modbus_imp.h` 77-132
- `src/drivers/modbus/protocol/modbus_imp.h` 32-55
- `src/drivers/modbus/protocol/modbus_db.h` 34-50

Configuration Loading

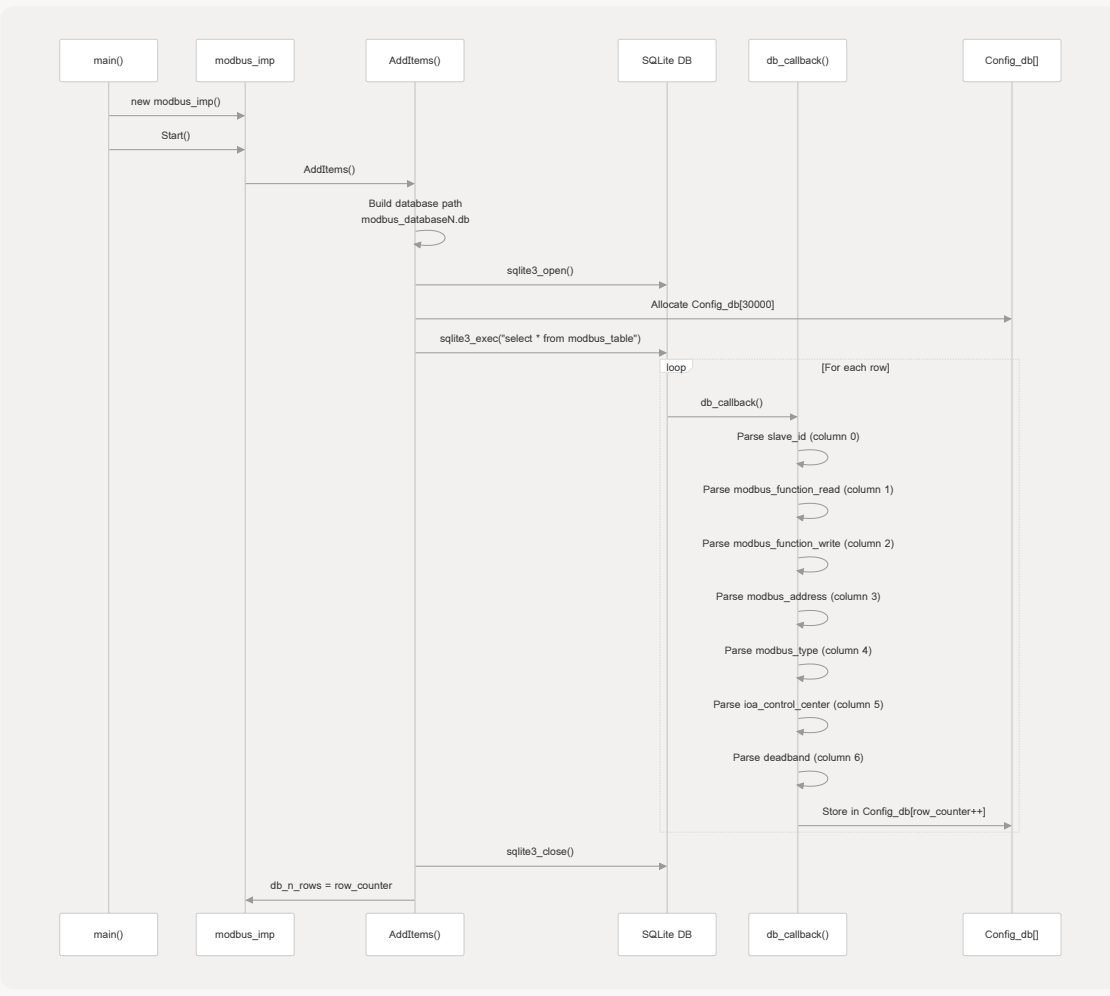
Configuration is loaded from a SQLite database file named `modbus_databaseN.db`, where N is the line number. The `AddItems()` function reads the `modbus_table` and populates the `Config_db` array.

Database Schema

The `modbus_table` contains the following columns:

Column	Type	Description
`slave_id`	INTEGER	Modbus slave/unit ID (1-247)
`modbus_function_read`	INTEGER	Function code for reading (0x01-0x04)
`modbus_function_write`	INTEGER	Function code for writing (0x05, 0x06, 0x0F, 0x10)
`modbus_address`	INTEGER	Register or coil address
`modbus_type`	TEXT	Data type (VT_BOOL, VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R4SWAP)
`ioa_control_center`	INTEGER	Information Object Address for SCADA system
`deadband`	REAL	Deadband threshold for analog values

Configuration Loading Flow


main()
modbus_imp
AddItems()
SQLite DB
db_callback()
Config_db[]
new modbus_imp()
Start()
AddItems()
Build database path
modbus_databaseN.db
sqlite3_open()
Allocate Config_db[30000]
sqlite3_exec("select * from modbus_table")
loop
[For each row]
db_callback()
Parse slave_id (column 0)
Parse modbus_function_read (column 1)
Parse modbus_function_write (column 2)
Parse modbus_address (column 3)
Parse modbus_type (column 4)
Parse ioa_control_center (column 5)
Parse deadband (column 6)
Store in Config_db[row_counter++]
sqlite3_close()
db_n_rows = row_counter

Sources:

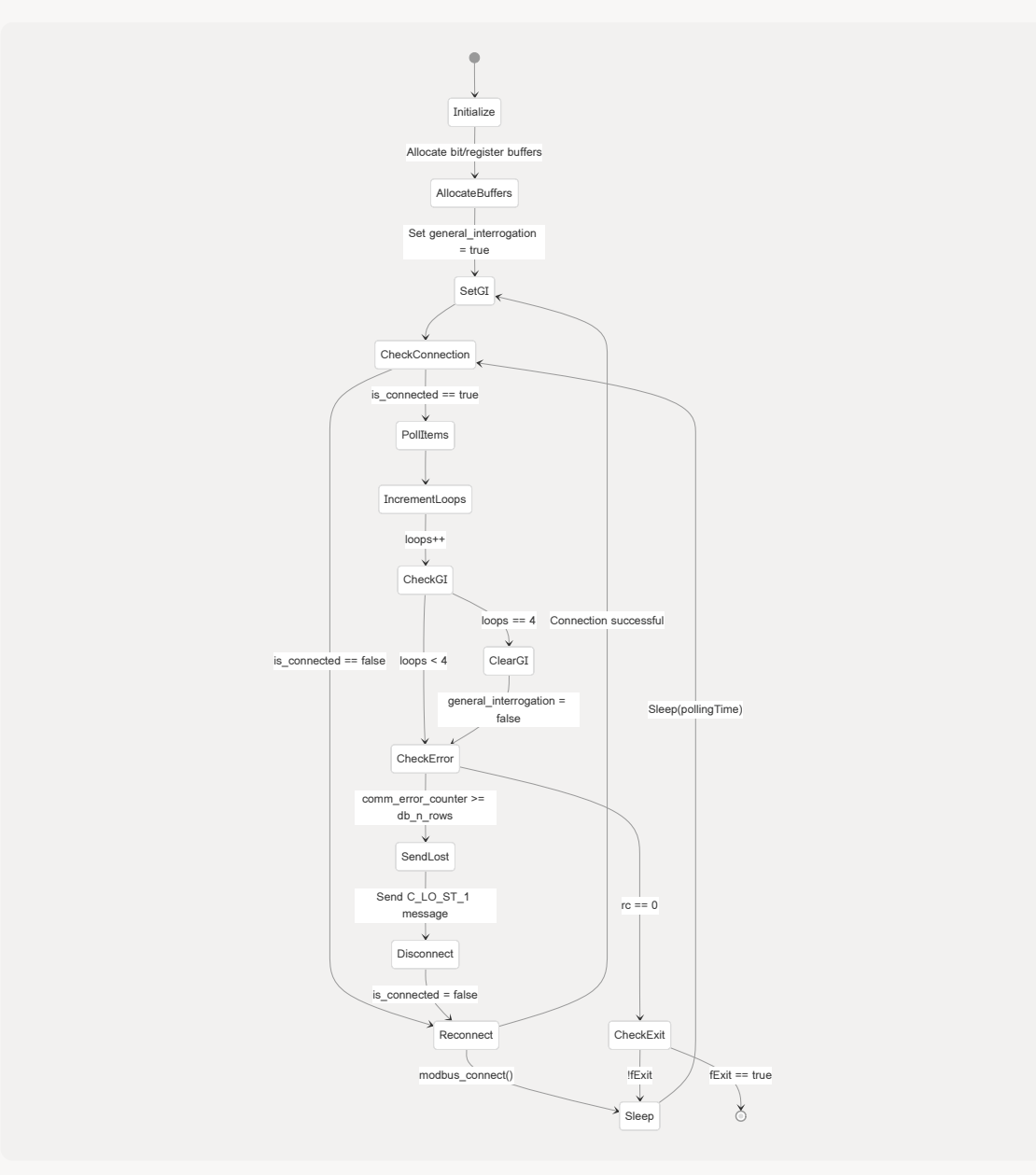
- `src/drivers/modbus/protocol/load_database_modbus.cpp` 173-254
- `src/drivers/modbus/protocol/load_database_modbus.cpp` 32-169

Polling Logic

The polling mechanism iterates through all configured data points at a regular interval, reading values from Modbus devices and publishing changes via middleware.

PollServer Main Loop

The `PollServer()` method implements the main polling loop with connection management and general interrogation:


Initialize
Allocate bit/register buffers
AllocateBuffers
Set general_interrogation = true
SetGI
CheckConnection
is_connected == true
PollItems
IncrementLoops
loops++
CheckGI
loops == 4
Connection successful
is_connected == false
loops < 4
ClearGI
general_interrogation = false
Sleep(pollingTime)
CheckError
comm_error_counter >= db_n_rows
SendLost
Send C_LO_ST_1 message
Disconnect
rc == 0
is_connected = false
Reconnect
CheckExit
modbus_connect()
!fExit
fExit == true
Sleep

Key Features:

- **General Interrogation**: First 4 loops publish all values regardless of change
- **Connection Monitoring**: Tracks communication errors per poll cycle
- **Automatic Reconnection**: Attempts reconnect if connection lost
- **Watchdog Timer**: Terminates if no parent communication for extended period

Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 253-388

PollItems Function Code Dispatch

The `PollItems()` method iterates through the `Config_db` array and dispatches to appropriate libmodbus functions based on the configured function code:

Function Code	Modbus Function	libmodbus API	Supported Types
0x01	Read Coils	`modbus_read_bits()`	VT_BOOL
0x02	Read Discrete Inputs	`modbus_read_input_bits()`	VT_BOOL
0x03	Read Holding Registers	`modbus_read_registers()`	VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R4SWAP
0x04	Read Input Registers	`modbus_read_input_registers()`	VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R4SWAP

Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 562-1229

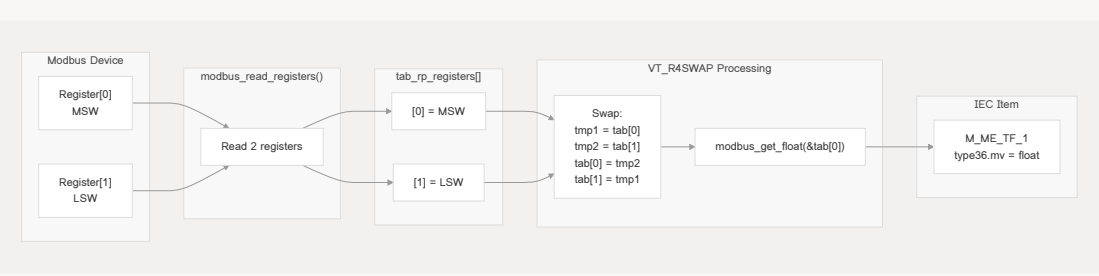
Data Type Mapping and Conversion

The implementation supports multiple data types and converts them to IEC 60870-5-104 types for middleware publication.

Type Mapping Table

Modbus Type	Size	IEC 104 Type	libmodbus Conversion	Description
VT_BOOL	1 bit	M_SP_TB_1	Direct	Single point (coil/discrete input)
VT_I2	16 bits	M_ME_TE_1	Direct cast	Signed 16-bit integer
VT_UI2	16 bits	M_ME_TQ_1	Direct cast	Unsigned 16-bit integer
VT_I4	32 bits	M_IT_TB_1	`modbus_get_int()`	Signed 32-bit integer (2 registers)
VT_UI4	32 bits	M_ME_TO_1	`modbus_get_uint()`	Unsigned 32-bit integer (2 registers)
VT_R4	32 bits	M_ME_TF_1	`modbus_get_float()`	32-bit float (2 registers)
VT_R4SWAP	32 bits	M_ME_TF_1	Swap registers + `modbus_get_float()`	32-bit float with swapped word order

Data Type Processing Example (VT_R4SWAP)

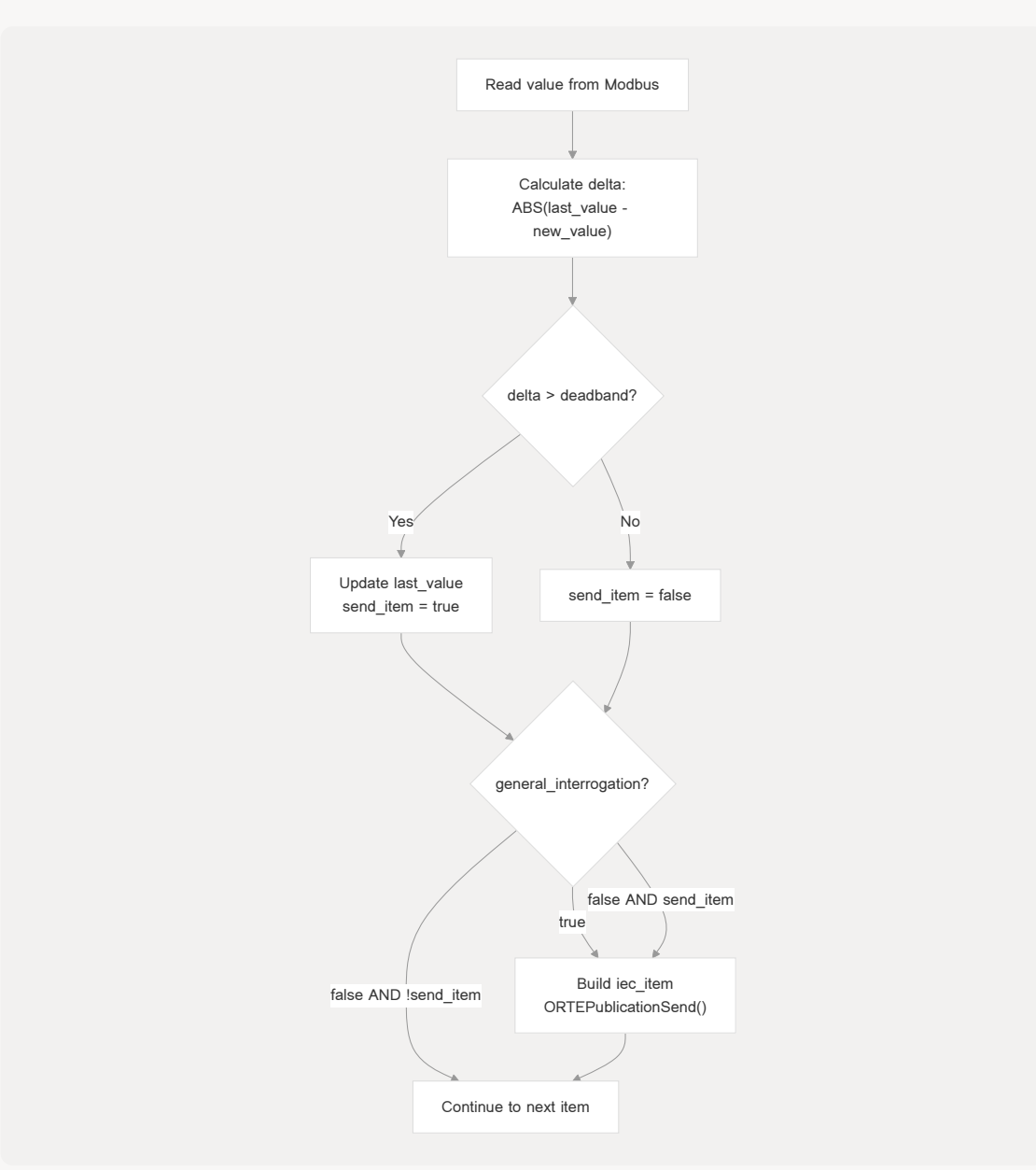

Modbus Device
Register[0] MSW
Register[1] LSW
modbus_read_registers()
Read 2 registers
tab_rp_registers[]
[0] = MSW
[1] = LSW
VT_R4SWAP Processing
Swap:
tmp1 = tab[0]
tmp2 = tab[1]
tab[0] = tmp2
tab[1] = tmp1
modbus_get_float(&tab[0])
IEC Item
M_ME_TF_1
type36.mv = float

Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 722-741
- `src/drivers/modbus/protocol/modbus_imp.cpp` 962-971
- `src/drivers/modbus/protocol/modbus_db.h` 18-32

Deadband Filtering

Analog values (VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R4SWAP) support deadband filtering to reduce unnecessary updates:


Read value from Modbus
Calculate delta: ABS(last_value - new_value)
delta > deadband?
Yes
No
Update last_value send_item = true
send_item = false
general_interrogation?
false AND send_item
true
false AND !send_item
Build iec_item ORTEPublicationSend()
Continue to next item

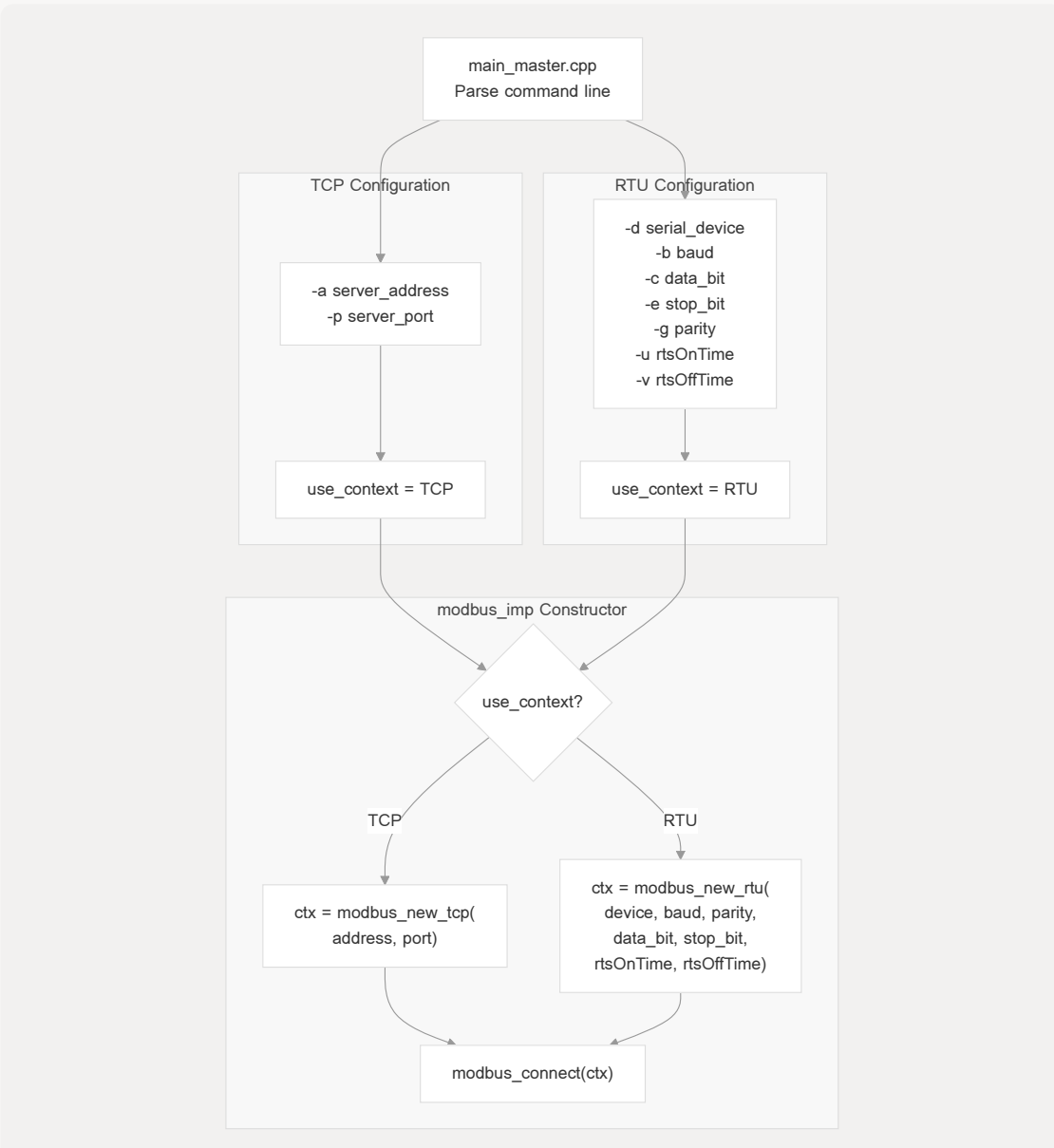
Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 745-754
- `src/drivers/modbus/protocol/modbus_imp.cpp` 775-784

Backend Abstraction (TCP vs RTU)

The implementation abstracts TCP and RTU transports through the libmodbus context API. The backend is selected at initialization based on the `modbusContext.use_context` field.

Backend Selection


main_master.cpp
Parse command line
TCP Configuration
-a server_address
-p server_port
use_context = TCP
RTU Configuration
-d serial_device
-b baud
-c data_bit
-e stop_bit
-g parity
-u rtsOnTime
-v rtsOffTime
use_context = RTU
modbus_imp Constructor
use_context?
TCP
RTU
ctx = modbus_new_tcp(address, port)
ctx = modbus_new_rtu(device, baud, parity, data_bit, stop_bit, rtsOnTime, rtsOffTime)
modbus_connect(ctx)

TCP Example Command Line:

```
modbus_master.exe -a 192.168.1.100 -p 502 -l 1 -t 1000
```

RTU Example Command Line:

```
modbus_master.exe -d "\\\\.\\COM1" -b 9600 -c 8 -e 1 -g N -l 1 -t 1000 -u 10 -v 10
```

Sources:

- `src/drivers/modbus/protocol/main_master.cpp` 126-192
- `src/drivers/modbus/protocol/main_master.cpp` 287-324
- `src/drivers/modbus/protocol/modbus_imp.cpp` 132-139

RTU RTS Control

For RS-485 half-duplex communication, the RTU backend supports configurable RTS (Request To Send) timing:

Parameter	Description	Default
`rtsOnTime`	Leading time in milliseconds before transmission	0
`rtsOffTime`	Trailing time in milliseconds after transmission	0

These parameters are passed to `modbus_new_rtu()` and control the RTS signal timing for RS-485 transceivers.

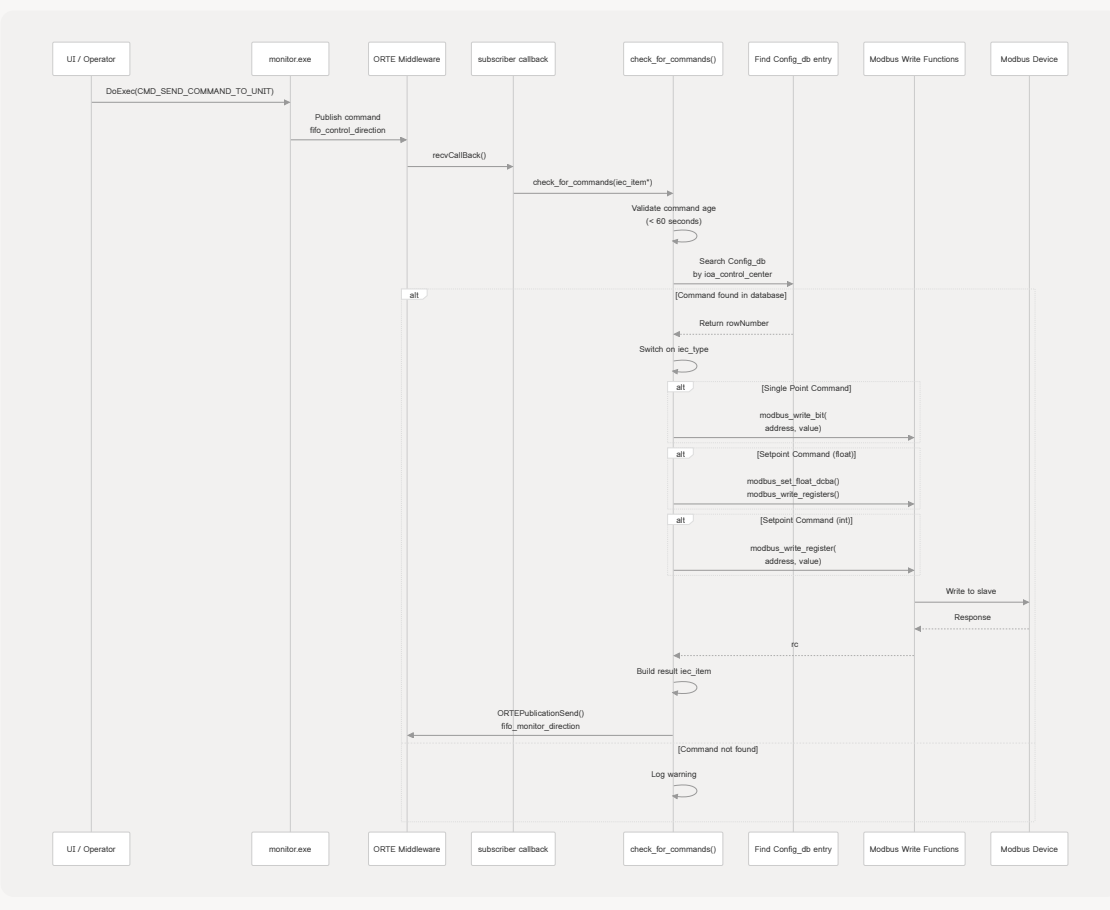
Sources:

- `src/drivers/modbus/protocol/modbus_imp.h` 53-54
- `src/drivers/modbus/protocol/modbus_imp.cpp` 128-129
- `src/drivers/modbus/protocol/modbus_imp.cpp` 138

Command Execution (Write Operations)

The `check_for_commands()` method handles write commands received from the control direction. Commands arrive as IEC 60870-5-104 types and are converted to Modbus write operations.

Command Processing Flow


UI / Operator
monitor.exe
ORTE Middleware
subscriber callback
check_for_commands()
Find Config_db entry
Modbus Write Functions
Modbus Device
DoExec(CMD_SEND_COMMAND_TO_UNIT)
Publish command fifo_control_direction
recvCallBack()
check_for_commands(iec_item*)
Validate command age (< 60 seconds)
Search Config_db by ioa_control_center
alt
[Command found in database]
Return rowNumber
Switch on iec_type
alt
[Single Point Command]
modbus_write_bit(address, value)
alt
[Setpoint Command (float)]
modbus_set_float_dcba() modbus_write_registers()
alt
[Setpoint Command (int)]
modbus_write_register(address, value)
Write to slave
Response
rc
Build result iec_item
ORTEPublicationSend() fifo_monitor_direction
[Command not found]
Log warning

Supported Command Types

IEC 104 Command Type	Description	Modbus Function	libmodbus API
C_SC_TA_1, C_SC_NA_1	Single Command	FC_WRITE_SINGLE_COIL (0x05)	`modbus_write_bit()`
C_DC_TA_1, C_DC_NA_1	Double Command	FC_WRITE_SINGLE_COIL (0x05)	`modbus_write_bit()`
C_SE_TA_1, C_SE_NA_1	Setpoint (normalized)	FC_WRITE_SINGLE_REGISTER (0x06)	`modbus_write_register()`
C_SE_TB_1, C_SE_NB_1	Setpoint (scaled)	FC_WRITE_SINGLE_REGISTER (0x06)	`modbus_write_register()`
C_SE_TC_1, C_SE_NC_1	Setpoint (float)	FC_WRITE_MULTIPLE_REGISTERS (0x10)	`modbus_set_float_dcba()` + `modbus_write_registers()`

Command Age Validation

Commands include a timestamp (`cp56time2a` structure) that is validated against the current time to prevent stale commands from being executed:

```
#define MAX_COMMAND_SEND_TIME 60

int64_t command_epoch = epoch_from_cp56time2a(&(queued_item->iec_obj.o.type58.time));
struct cp56time2a actual_time;
get_utc_host_time(&actual_time);
int64_t time_now_epoch = epoch_from_cp56time2a(&actual_time);

if((time_now_epoch - command_epoch) < MAX_COMMAND_SEND_TIME)
{
    // Command is fresh, execute it
}
```

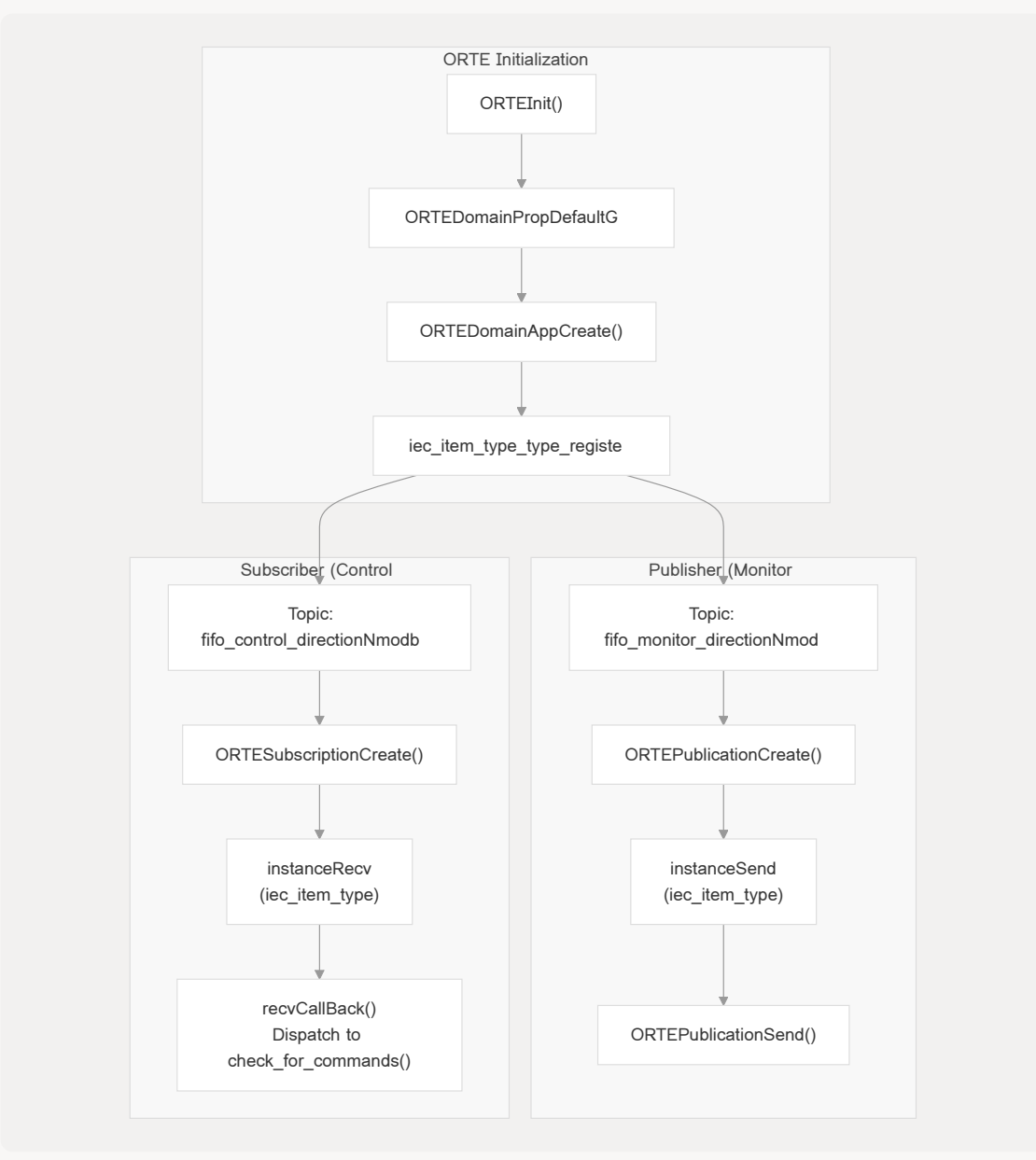
Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 1343-1887
- `src/drivers/modbus/protocol/modbus_imp.cpp` 22
- `src/drivers/modbus/protocol/modbus_imp.cpp` 532-557

Middleware Integration

The Modbus master publishes data to the monitor direction and subscribes to commands from the control direction using ORTE middleware.

ORTE Domain Setup


ORTE Initialization
ORTEInit()
ORTEDomainPropDefaultG
ORTEDomainAppCreate()
iec_item_type_type_registe
Subscriber (Control
Topic: fifo_control_directionNmodb
ORTESubscriptionCreate()
instanceRecv (iec_item_type)
recvCallBack() Dispatch to check_for_commands()
Publisher (Monitor
Topic: fifo_monitor_directionNmod
ORTEPublicationCreate()
instanceSend (iec_item_type)
ORTEPublicationSend()

Topic Naming Convention:

- Monitor direction: `fifo_monitor_direction<N>modbus` where N is the line number
- Control direction: `fifo_control_direction<N>modbus` where N is the line number

Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 143-211
- `src/drivers/modbus/protocol/modbus_imp.cpp` 171-185
- `src/drivers/modbus/protocol/modbus_imp.cpp` 189-208

IEC Item Construction

Data values are packaged into `iec_item` structures before publication:

```
struct iec_item item_to_send;
struct cp56time2a actual_time;

memset(&item_to_send, 0x00, sizeof(struct iec_item));

item_to_send.iec_obj.ioa = Config_db[rowNumber].ioa_control_center;
item_to_send.cause = 0x03; // Spontaneous
item_to_send.iec_type = M_ME_TF_1; // Measured value, short floating point
get_local_host_time(&actual_time);
item_to_send.iec_obj.o.type36.mv = float_value;
item_to_send.iec_obj.o.type36.time = actual_time;
item_to_send.iec_obj.o.type36.iv = 0; // Valid
item_to_send.msg_id = n_msg_sent;
item_to_send.checksum = clearCrc((unsigned char *)&item_to_send, sizeof(struct iec_item));

// Copy to ORTE instance
memcpy(&instanceSend.iec_obj, &item_to_send.iec_obj, sizeof(iec_object));
instanceSend.iec_type = item_to_send.iec_type;
instanceSend.cause = item_to_send.cause;
instanceSend.msg_id = item_to_send.msg_id;
instanceSend.checksum = item_to_send.checksum;

ORTEPublicationSend(publisher);
```

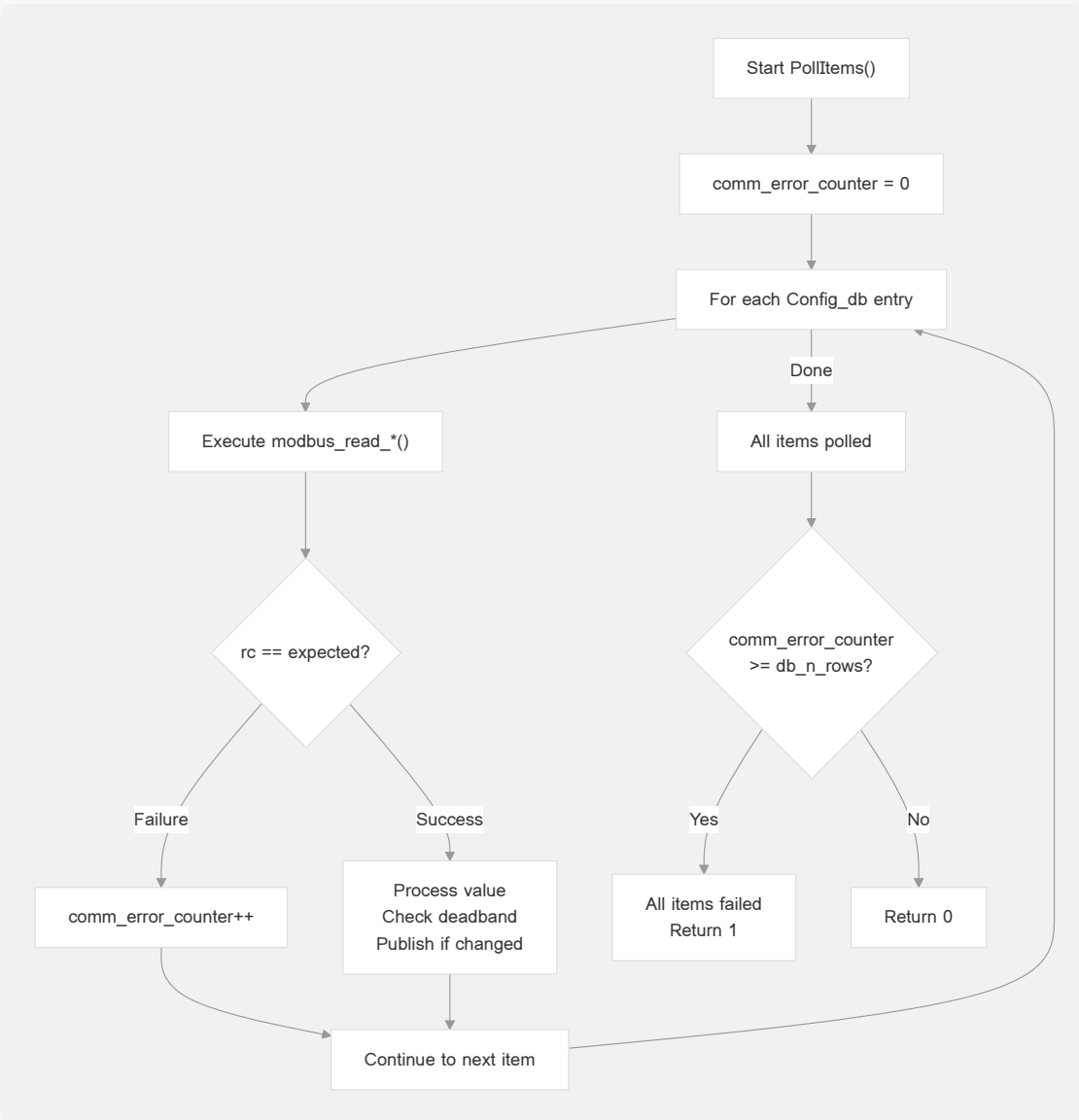
Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 1177-1217
- `src/drivers/modbus/protocol/modbus_imp.cpp` 756-766

Connection Loss Detection and Recovery

The implementation monitors connection health and implements automatic reconnection when communication errors occur.

Error Detection


Start PollItems()
comm_error_counter = 0
For each Config_db entry
Done
Execute modbus_read_*()
All items polled
rc == expected?
comm_error_counter >= db_n_rows?
Failure
Success
Yes
No
comm_error_counter++
Process value Check deadband Publish if changed
All items failed Return 1
Return 0
Continue to next item

Reconnection Logic

When `PollItems()` returns an error (all items failed), the main loop:

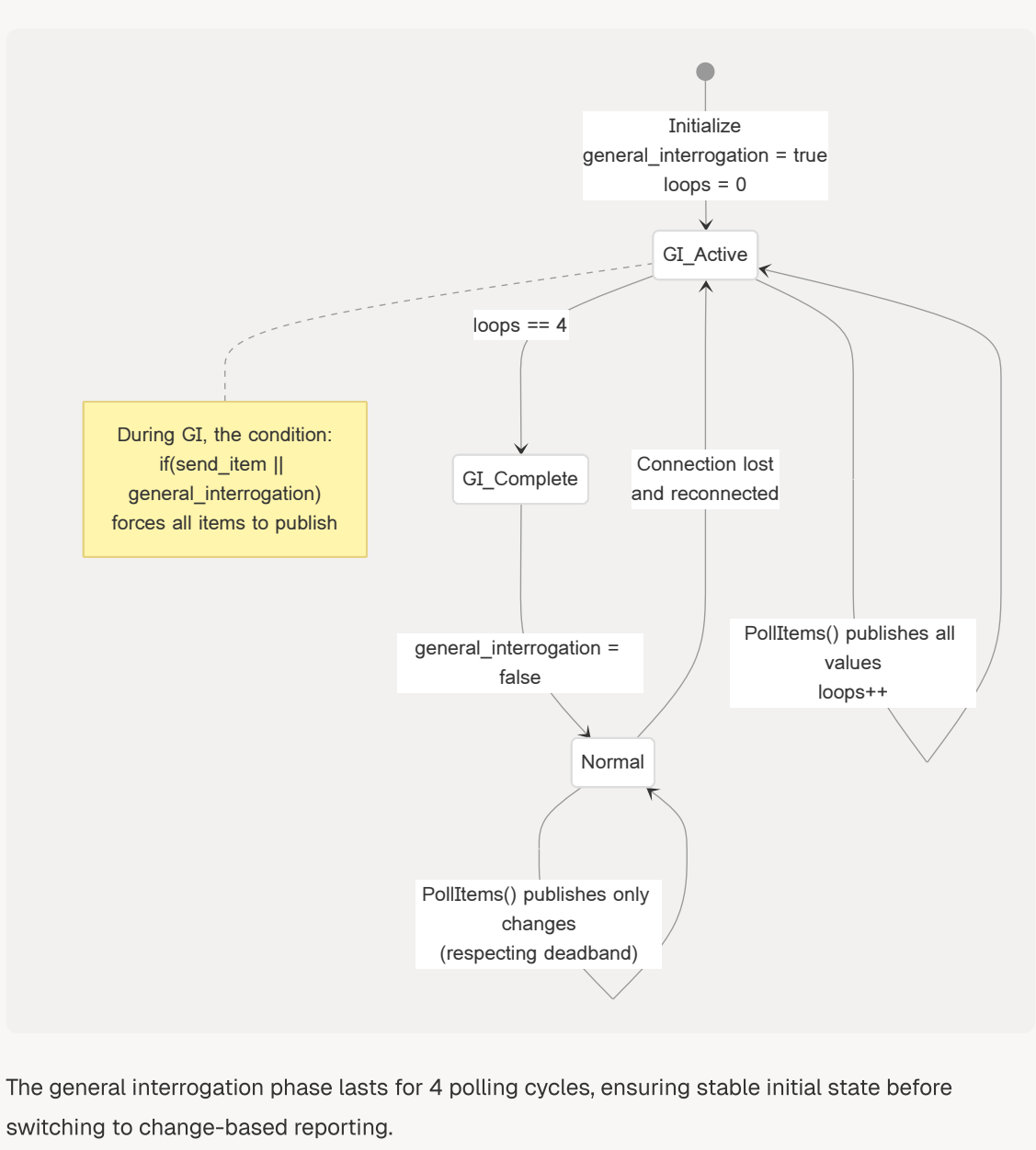
1. Sends a `C_LO_ST_1` (Connection Lost) message to the parent
2. Sets `is_connected = false`
3. Closes the connection with `modbus_close(ctx)`
4. Enters reconnection loop that attempts `modbus_connect(ctx)` each cycle
5. On successful reconnection, resets `general_interrogation = true` to refresh all values

Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 573
- `src/drivers/modbus/protocol/modbus_imp.cpp` 602-605
- `src/drivers/modbus/protocol/modbus_imp.cpp` 1220-1224
- `src/drivers/modbus/protocol/modbus_imp.cpp` 296-363

General Interrogation Mechanism

General interrogation ensures all data points are published when the driver first starts or reconnects, regardless of deadband filtering or value changes.


Initialize
general_interrogation = true
loops = 0
GI_Active
loops == 4
During GI, the condition:
if(send_item ||
general_interrogation)
forces all items to publish
GI_Complete
Connection lost and reconnected
general_interrogation = false
PollItems() publishes all values loops++
Normal
PollItems() publishes only changes (respecting deadband)

The general interrogation phase lasts for 4 polling cycles, ensuring stable initial state before switching to change-based reporting.

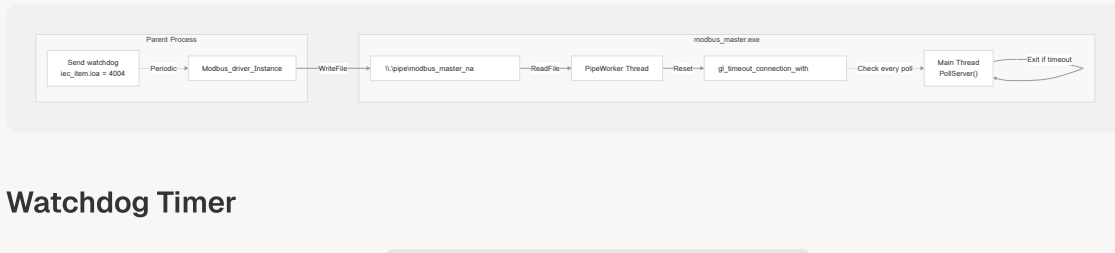
Sources:

- `src/drivers/modbus/protocol/modbus_imp.cpp` 278-294
- `src/drivers/modbus/protocol/modbus_imp.cpp` 359-362
- `src/drivers/modbus/protocol/modbus_imp.cpp` 1177-1178

Keep-Alive Communication

The master process maintains a keep-alive connection with its parent process through a named pipe. This ensures the parent can detect if the child process hangs or crashes.

Named Pipe Architecture


Parent Process
Send watchdog iec_item.ioa = 4004
Periodic
Modbus_driver_Instance
WriteFile
modbus_master.exe
\\.\pipe\modbus_master_na
ReadFile
PipeWorker Thread
Reset
gl_timeout_connection_with
Check every poll
Main Thread PollServer()
Exit if timeout

Watchdog Timer

The main polling loop increments `gl_timeout_connection_with_parent` on each cycle. The pipe worker resets it to 0 when a keep-alive message (IOA 4004) is received. If the counter exceeds a threshold (1000 * 60 / pollingTime iterations), the process terminates.

Sources:

- `src/drivers/modbus/protocol/main_master.cpp` 358-502
- `src/drivers/modbus/protocol/modbus_imp.cpp` 374-381
- `src/drivers/modbus/protocol/main_master.cpp` 492-498

Summary

The Modbus master implementation provides a robust, configurable interface to Modbus devices with the following key characteristics:

- **Backend Abstraction**: Supports both TCP and RTU transports via libmodbus
- **Database-Driven Configuration**: All data points defined in SQLite `modbus_table`
- **Type System**: Maps 8 Modbus data types to IEC 60870-5-104 types
- **Deadband Filtering**: Reduces unnecessary updates for analog values
- **Command Execution**: Handles single-point and setpoint commands with age validation
- **Connection Management**: Automatic reconnection with general interrogation
- **Middleware Integration**: ORTE publish/subscribe for data and commands
- **Process Supervision**: Keep-alive pipe communication with parent process

The implementation is located primarily in:

- `src/drivers/modbus/protocol/modbus_imp.cpp`
- `src/drivers/modbus/protocol/modbus_imp.h`
- `src/drivers/modbus/protocol/load_database_modbus.cpp`
- `src/drivers/modbus/protocol/main_master.cpp`

Modbus Slave Implementation

> Relevant source files

Purpose and Scope

This page documents the Modbus slave (server) implementation in IndigoSCADA, which allows the system to act as a Modbus slave device that responds to requests from external Modbus master clients. The slave maintains memory areas (coils, discrete inputs, holding registers, input registers) that are populated from IndigoSCADA's monitoring direction and served to external Modbus masters on request.

For information about the Modbus master implementation that polls external devices, see Modbus Master Implementation. For general Modbus driver architecture, see Modbus Driver.

Overview

The Modbus slave implementation enables IndigoSCADA to expose its internal data points to external systems via the Modbus protocol. This is the inverse of the master driver: instead of polling remote devices, the slave responds to queries from external Modbus masters. The slave executable `modbus_slave.exe` runs as a standalone process that:

1. Subscribes to the global monitor direction FIFO to receive updated values from IndigoSCADA
2. Maps these values to Modbus memory areas based on SQLite configuration
3. Listens for incoming Modbus requests (TCP or RTU)
4. Responds with current values from its memory mapping

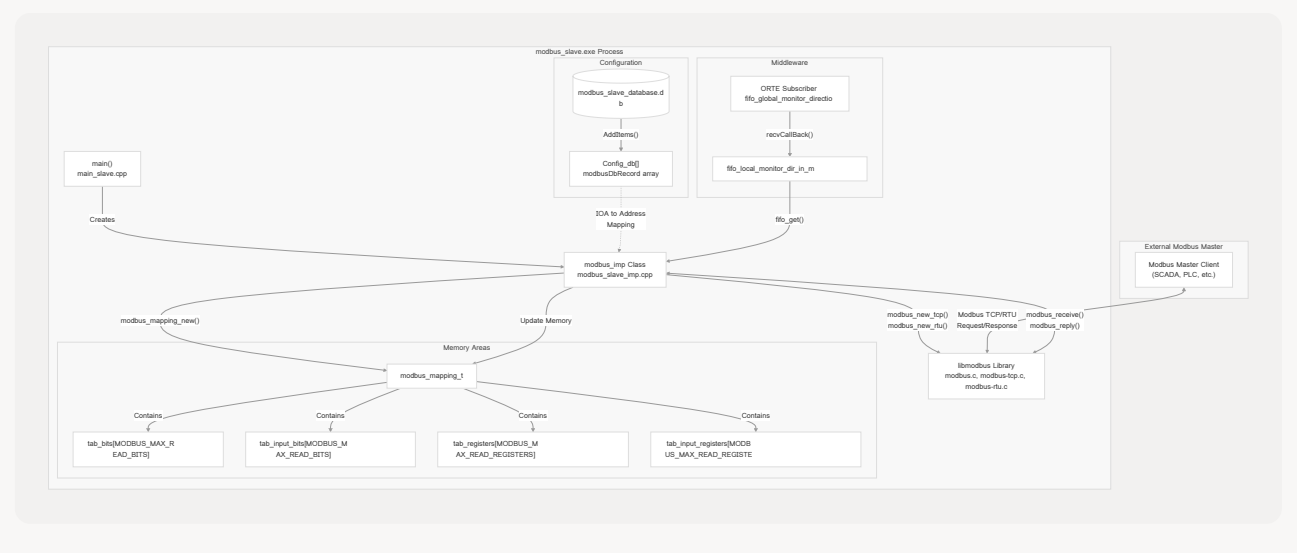
Key Components:

- **Executable**: `modbus_slave.exe`
- **Main Implementation Class**: `modbus_imp`
- **Configuration Database**: `modbus_slave_database.db` (SQLite)
- **Memory Structure**: `modbus_mapping_t` from libmodbus
- **Supported Modes**: Modbus TCP and Modbus RTU

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.h` 1-136 `src/drivers/modbus/protocol/main_slave.cpp` 1-231

Architecture and Class Structure

The slave implementation follows a simpler architecture than the master driver because it does not use the DriverInstance/DriverThread pattern. Instead, it runs as a single-process server.


modbus_slave.exe Process
Configuration
modbus_slave_database.db
AddItems()
Config_db[] modbusDbRecord array
Middleware
ORTE Subscriber fifo_global_monitor_directio
recvCallBack()
fifo_local_monitor_dir_in_m
main() main_slave.cpp
Creates
IOA to Address Mapping
fifo_get()
modbus_imp Class modbus_slave_imp.cpp
External Modbus Master
Modbus Master Client (SCADA, PLC, etc.)
modbus_mapping_new()
Update Memory
modbus_new_tcp() modbus_new_rtu()
Modbus TCP/RTU Request/Response
modbus_receive() modbus_reply()
Memory Areas
modbus_mapping_t
libmodbus Library modbus.c, modbus-tcp.c, modbus-rtu.c
Contains
Contains
Contains
Contains
tab_bits[MODBUS_MAX_READ_BITS]
tab_input_bits[MODBUS_MAX_READ_BITS]
tab_registers[MODBUS_MAX_READ_REGISTERS]
tab_input_registers[MODBUS_MAX_READ_REGISTE

Class: `modbus_imp`

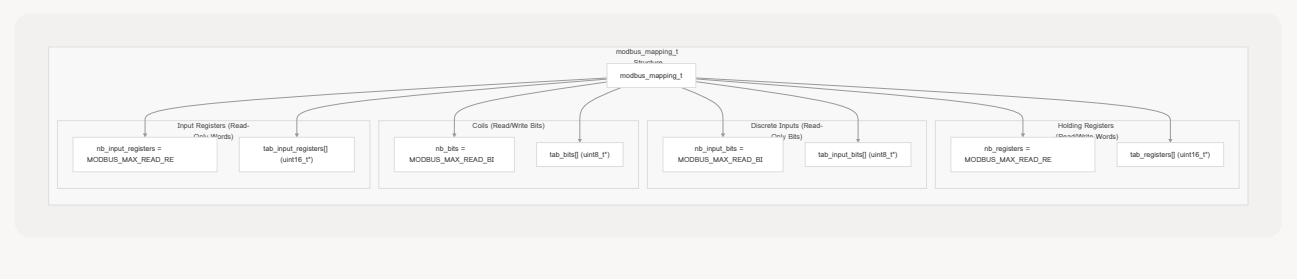
The `modbus_imp` class encapsulates all slave functionality:

Member	Type	Purpose
`ctx`	`modbus_t*`	libmodbus context (TCP or RTU)
`mb_mapping`	`modbus_mapping_t*`	Modbus memory areas
`Config_db`	`modbusDbRecord*`	Configuration array loaded from SQLite
`domain`	`ORTEDomain*`	ORTE domain for middleware
`subscriber`	`ORTESubscription*`	Subscribes to monitor direction
`fifo_monitor_direction`	`fifo_h`	Local FIFO buffer
`my_modbus_context`	`modbusContext`	TCP/RTU configuration

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.h` 78-134 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 106-227

Memory Mapping Structure

The Modbus slave uses the `modbus_mapping_t` structure from libmodbus to maintain its memory areas. This structure directly corresponds to the four main Modbus memory types:


modbus_mapping_t Structure
modbus_mapping_t
Input Registers (Read-Only Words)
Coils (Read/Write Bits)
Discrete Inputs (Read-Only Bits)
Holding Registers (Read/Write Words)

Memory Allocation

The memory mapping is allocated in the constructor at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 216-224 :

```
mb_mapping = modbus_mapping_new(MODBUS_MAX_READ_BITS, MODBUS_MAX_READ_BITS,
                                MODBUS_MAX_READ_REGISTERS, MODBUS_MAX_READ_REGISTERS);
```

This creates all four memory areas with maximum size. The actual addresses used are determined by the configuration database.

Address Mapping

Each configured data point in `modbus_slave_database.db` specifies:

- `modbus_function_read` - Which function code (0x01-0x04) determines which memory area
- `modbus_address` - The offset within that memory area
- `modbus_type` - Data type (VT_BOOL, VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R4SWAP)
- `ioa_control_center` - The IndigoSCADA IOA that maps to this Modbus address

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 216-224 `src/drivers/modbus/protocol/modbus_slave_db.h` 1-51

Configuration Database

The slave loads its configuration from an SQLite database named `modbus_slave_database.db`, located in the project directory. The database contains a single table `modbus_table` with the following schema:

Database Record Structure: `modbusDbRecord`

Field	Type	Description
`slave_id`	`int`	Slave ID for RTU mode
`modbus_function_read`	`int`	Function code (0x01-0x04) determining memory area
`modbus_function_write`	`int`	Function code for write operations (0x05, 0x06, 0x0F, 0x10)
`modbus_address`	`int`	Address/offset within the memory area
`modbus_type`	`int`	Data type (VT_BOOL, VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, etc.)
`ioa_control_center`	`unsigned int`	IOA in IndigoSCADA that maps to this address
`deadband`	`float`	Deadband for analog values (not currently used in slave)

Function Code to Memory Area Mapping:

Function Code	Constant	Memory Area	Access
0x01	`FC_READ_COILS`	`tab_bits[]`	Read/Write bits
0x02	`FC_READ_DISCRETE_INPUTS`	`tab_input_bits[]`	Read-only bits
0x03	`FC_READ_HOLDING_REGISTERS`	`tab_registers[]`	Read/Write 16-bit registers
0x04	`FC_READ_INPUT_REGISTERS`	`tab_input_registers[]`	Read-only 16-bit registers

Database Loading

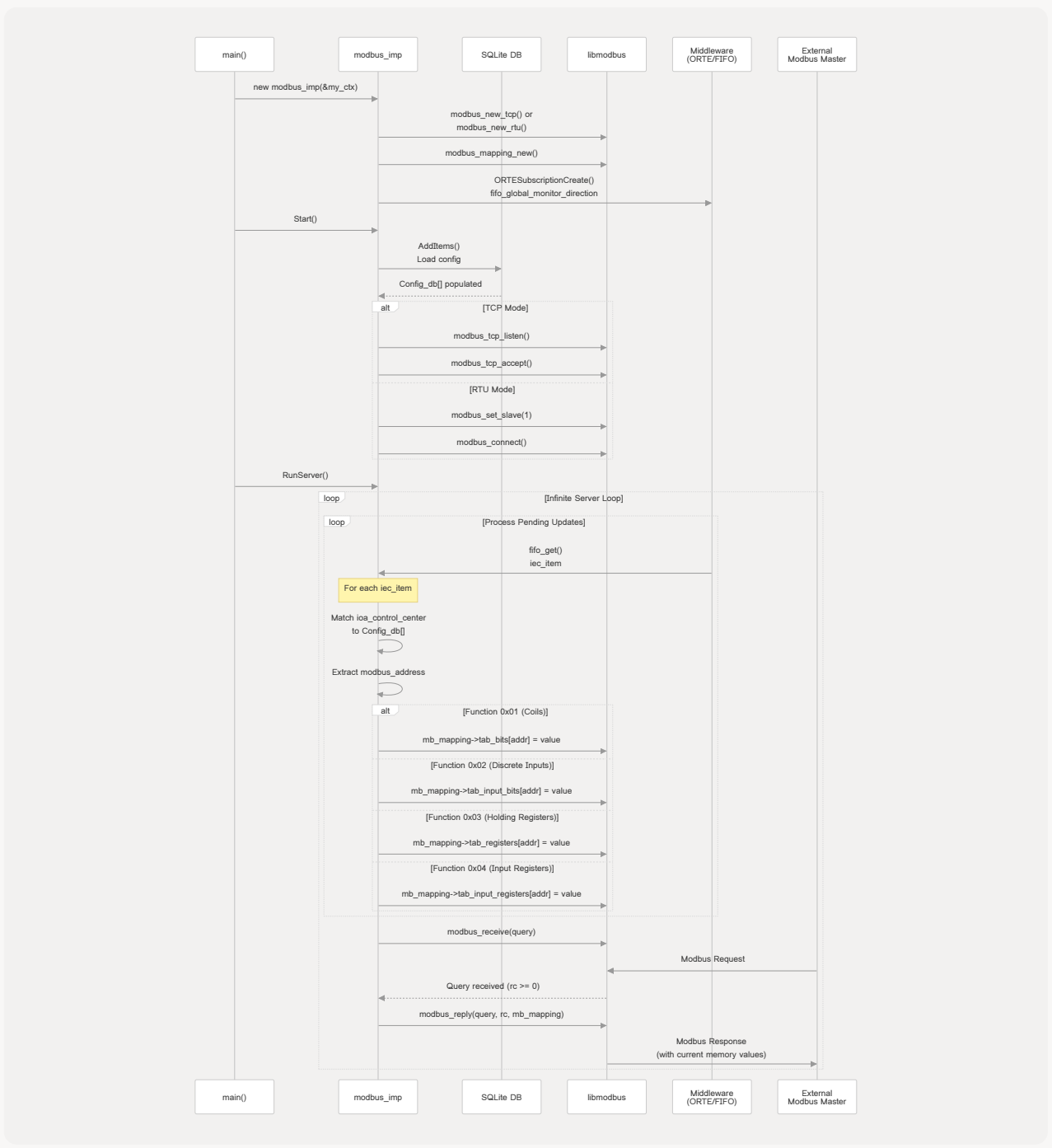
The `AddItems()` method at `src/drivers/modbus/protocol/load_database_modbus_slave.cpp` 173-250 loads the configuration:

1. Opens `modbus_slave_database.db` from the project directory
2. Executes `SELECT * FROM modbus_table`
3. Parses each row via the callback function `db_callback()`
4. Populates the `Config_db[]` array with `modbusDbRecord` structures

Sources: `src/drivers/modbus/protocol/modbus_slave_db.h` 34-50 `src/drivers/modbus/protocol/load_database_modbus_slave.cpp` 32-169 `src/drivers/modbus/protocol/load_database_modbus_slave.cpp` 173-250

Server Operation Flow

The slave operates in an infinite loop, continuously updating its memory from IndigoSCADA and serving Modbus requests.


main()
modbus_imp
SQLite DB
libmodbus
Middleware (ORTE/FIFO)
External Modbus Master
new modbus_imp(&my_ctx)
modbus_new_tcp() or modbus_new_rtu()
modbus_mapping_new()
ORTESubscriptionCreate() fifo_global_monitor_direction
Start()
AddItems() Load config
Config_db[] populated
alt
[TCP Mode]
modbus_tcp_listen()
modbus_tcp_accept()
[RTU Mode]
modbus_set_slave(1)
modbus_connect()
RunServer()
loop
[Infinite Server Loop]
loop
[Process Pending Updates]
fifo_get() iec_item
For each iec_item
Match ioa_control_center to Config_db[]
Extract modbus_address
alt
[Function 0x01 (Coils)]
mb_mapping->tab_bits[addr] = value
[Function 0x02 (Discrete Inputs)]
mb_mapping->tab_input_bits[addr] = value
[Function 0x03 (Holding Registers)]
mb_mapping->tab_registers[addr] = value
[Function 0x04 (Input Registers)]
mb_mapping->tab_input_registers[addr] = value
modbus_receive(query)
Modbus Request
Query received (rc >= 0)
modbus_reply(query, rc, mb_mapping)
Modbus Response (with current memory values)
main()
modbus_imp
SQLite DB
libmodbus
Middleware (ORTE/FIFO)
External Modbus Master

Server Loop Details

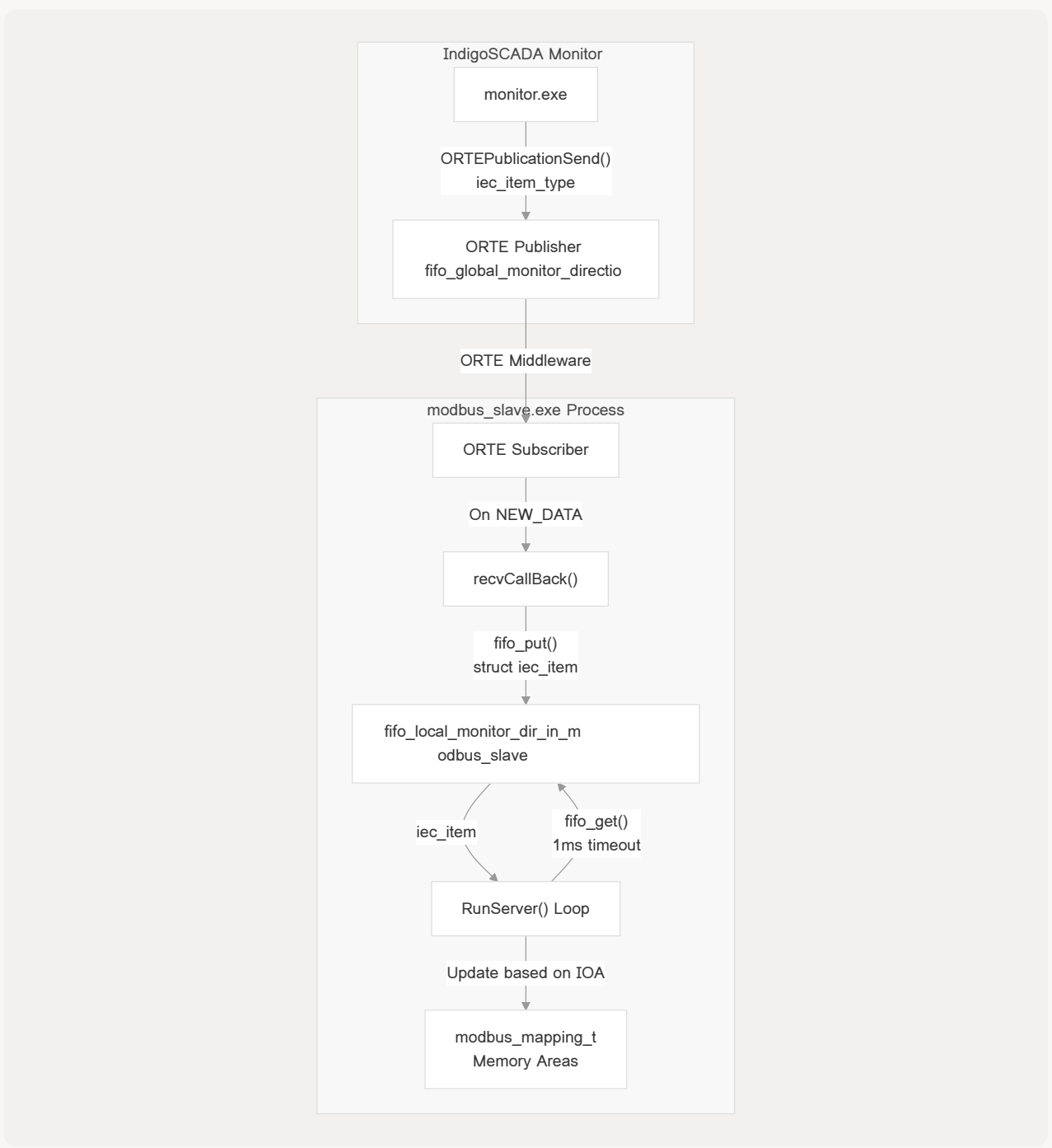
The `RunServer()` method at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 244-466 implements the main server loop:

1. **Update Phase** (lines 258-449):
    - Calls `fifo_get()` with 1ms timeout to check for pending iec_items
    - For each received item, searches `Config_db[]` for matching `ioa_control_center`
    - Based on `modbus_function_read`, updates the corresponding memory area
    - Handles type conversion from IEC 60870-5 types to Modbus types
2. **Service Phase** (lines 451-461):
    - Calls `modbus_receive(ctx, query)` to wait for incoming request
    - If request received (`rc >= 0`), calls `modbus_reply()` with current memory state
    - If connection closed (`rc < 0`), breaks loop and exits

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 244-466

Middleware Integration

The slave integrates with IndigoSCADA's middleware to receive real-time data updates. This allows values from other protocol drivers (IEC 104, DNP3, OPC, etc.) to be exposed via Modbus.


IndigoSCADA Monitor
monitor.exe
ORTEPublicationSend() iec_item_type
ORTE Publisher fifo_global_monitor_directio
ORTE Middleware
modbus_slave.exe Process
ORTE Subscriber
On NEW_DATA
recvCallBack()
fifo_put() struct iec_item
fifo_local_monitor_dir_in_modbus_slave
iec_item
fifo_get() 1ms timeout
RunServer() Loop
Update based on IOA
modbus_mapping_t Memory Areas

ORTE Subscription Setup

The middleware is initialized in the constructor at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 141-205 :

1. Creates ORTE domain
2. Creates local FIFO: `fifo_local_monitor_dir_in_modbus_slave`
3. Subscribes to global monitor direction: `fifo_global_monitor_direction`
4. Registers callback function `recvCallBack()`

Callback Function: `recvCallBack()`

The callback at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 74-97 executes when data arrives:

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void *recvCallBackParam)
{
    modbus_imp * cl = (modbus_imp*)recvCallBackParam;
    iec_item_type *item1 = (iec_item_type*)vinstance;

    switch (info->status)
    {
        case NEW_DATA:
        {
            struct iec_item item2;
            rebuild_iec_item_message(&item2, item1);  // Convert and validate
            fifo_put(cl->fifo_monitor_direction, (char*)&item2, sizeof(struct iec_item));
        }
        break;
    }
}
```

Data Type Mapping

The update logic at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 290-449 handles conversion from IEC 60870-5 types to Modbus memory:

IEC Type	IEC Constant	Modbus Memory	Mapping
Single Point	`M_SP_TB_1`	Coils or Discrete Inputs	`type30.sp` → `tab_bits[offset]`
Scaled Value	`M_ME_TE_1`	Holding or Input Registers	`type35.mv` → `tab_registers[offset]`
Normalized Value	`M_ME_TQ_1`	Holding or Input Registers	`type153.mv` → `tab_registers[offset]`
Short Floating Point	`M_ME_TF_1`	Holding or Input Registers	`type36.mv` → `memcpy(&tab_registers[offset])`
Binary Counter	`M_IT_TB_1`	Holding or Input Registers	`type37.counter` → `tab_registers[offset]`

Sources: `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 74-97 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 141-205 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 290-449

TCP vs RTU Mode Selection

The slave supports both Modbus TCP and Modbus RTU, selected via command-line arguments.

Command-Line Arguments

From `src/drivers/modbus/protocol/main_slave.cpp` 56-132 :

Argument	Purpose	Example
`-a`	TCP server IP address	`-a 192.168.1.100`
`-p`	TCP server port	`-p 502`
`-d`	RTU serial device	`-d COM1` or `-d /dev/ttyS0`
`-b`	RTU baud rate	`-b 9600`
`-c`	RTU data bits	`-c 8`
`-e`	RTU stop bits	`-e 1`
`-f`	RTU parity	`-f N` (None, Odd, Even)

Mode Selection Logic

At `src/drivers/modbus/protocol/main_slave.cpp` 140-166 :

```
if(strlen(modbusServerAddress) > 0 && strlen(modbusServerPort) > 0)
{
    use_context = TCP;
}
else
{
    use_context = RTU;
}
```

Context Creation

In the constructor at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 127-134 :

```
if(my_modbus_context.use_context == TCP)
{
    ctx = modbus_new_tcp(my_modbus_context.modbus_server_address,
                         atoi(my_modbus_context.modbus_server_port));
}
else
{
    ctx = modbus_new_rtu(my_modbus_context.serial_device,
                         my_modbus_context.baud,
                         my_modbus_context.parity,
                         my_modbus_context.data_bit,
                         my_modbus_context.stop_bit,
                         my_modbus_context.rtsOnTime,
                         my_modbus_context.rtsOffTime);
}
```

Connection Establishment

In `Start()` at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 486-495 :

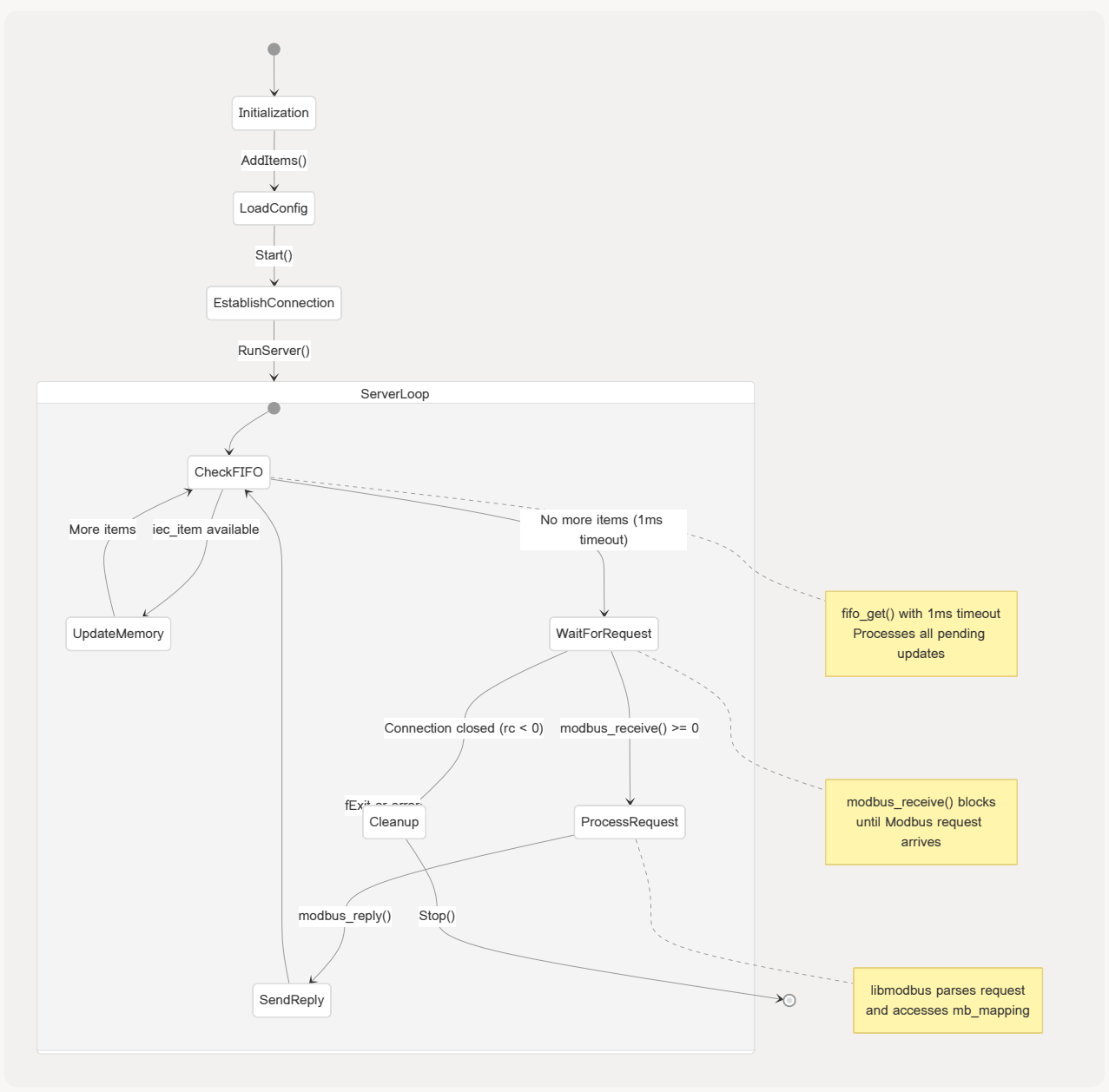
```
if(my_modbus_context.use_context == TCP)
{
    socket = modbus_tcp_listen(ctx, 1);      // Listen for 1 connection
    modbus_tcp_accept(ctx, &socket);         // Accept connection
}
else
{
    modbus_set_slave(ctx, 1);                // Set slave ID
    modbus_connect(ctx);                     // Open serial port
}
```

TCP Mode: Creates a TCP socket, listens on the specified port, and accepts one client connection at a time.

RTU Mode: Opens the serial port and sets the slave ID. The slave responds to requests addressed to its ID.

Sources: `src/drivers/modbus/protocol/main_slave.cpp` 56-166 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 127-134 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 486-495

Process Lifecycle


Initialization
AddItems()
LoadConfig
Start()
EstablishConnection
RunServer()
ServerLoop
CheckFIFO
More items
iec_item available
UpdateMemory
No more items (1ms timeout)
WaitForRequest
Connection closed (rc < 0)
modbus_receive() >= 0
Cleanup
ProcessRequest
modbus_reply()
Stop()
SendReply
fifo_get() with 1ms timeout Processes all pending updates
modbus_receive() blocks until Modbus request arrives
libmodbus parses request and accesses mb_mapping

Initialization Phase

1. `main()` parses command-line arguments
2. Creates `modbus_imp` instance
    - Initializes libmodbus context (TCP or RTU)
    - Allocates `modbus_mapping_t` memory
    - Sets up ORTE subscription
3. Calls `Start()`
    - Loads configuration from SQLite via `AddItems()`
    - Establishes TCP listen socket or opens serial port

Server Loop Phase

The `RunServer()` loop runs continuously:

1. Processes all pending FIFO items (updates memory)
2. Blocks on `modbus_receive()` waiting for requests
3. When request arrives, `modbus_reply()` sends response
4. Loop continues until connection closed or error

Cleanup Phase

When loop exits, `Stop()` at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 501-532 performs cleanup:

- Closes FIFO
- Frees `modbus_mapping_t`
- Closes socket (TCP) or serial port (RTU)
- Frees libmodbus context

Sources: `src/drivers/modbus/protocol/main_slave.cpp` 64-231 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 244-466 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 468-499 `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 501-532

Build Configuration

The slave is built as a separate console application.

Visual Studio Project

The project file at `src/drivers/modbus/protocol/modbus_slave.dsp` 1-145 defines:

Executable: `modbus_slave.exe` output to `c:\scada\bin\`

Source Files:

- `modbus_slave_imp.cpp` - Main implementation
- `main_slave.cpp` - Entry point
- `load_database_modbus_slave.cpp` - Configuration loader
- `modbus.c`, `modbus-tcp.c`, `modbus-rtu.c`, `modbus-data.c` - libmodbus library
- `iec_item_type.c` - Middleware types
- `clear_crc_eight.c` - Checksum validation
- `time64.c` - Time utilities
- `sqlite3.c` - SQLite embedded database

Include Directories:

- `$(QTDIR)\include`
- `..\..\..\common` - Common headers
- `..\..\..\database` - Database interfaces
- `..\..\..\utilities` - Utility functions
- `..\..\..\configurator\sqlite` - SQLite
- `..\..\..\fifo` - FIFO implementation
- `..\..\..\middleware\rtps\include` - RTPS/ORTE middleware

Link Libraries:

- `rtps.lib` - RTPS middleware
- `utilities.lib` - Utility functions
- `fifo.lib` - FIFO library
- Standard Windows libraries

Sources: `src/drivers/modbus/protocol/modbus_slave.dsp` 1-145

Error Handling and Logging

Exit Handler

The function `iec_call_exit_handler()` at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 781-835 logs errors to `logs\modbus_slave.log` before raising SIGABRT.

Checksum Validation

All `iec_item` messages received from FIFO are validated:

```
rc = clearCrc((unsigned char *)&one_item, sizeof(struct iec_item));
if(rc != 0)
{
    sprintf(err_msg, "Message checksum error");
    iec_call_exit_handler(__LINE__,__FILE__,err_msg);
}
```

This is at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 272-277

Connection Errors

When `modbus_receive()` returns a negative value, it indicates connection closed or error. The loop breaks and cleanup begins.

Debug Mode

Debug mode is enabled at `src/drivers/modbus/protocol/modbus_slave_imp.cpp` 214 :

```
modbus_set_debug(ctx, TRUE);
```

This causes libmodbus to print detailed protocol traces to stderr.

Sources: src/drivers/modbus/protocol/modbus_slave_imp.cpp 214
src/drivers/modbus/protocol/modbus_slave_imp.cpp 272-277
src/drivers/modbus/protocol/modbus_slave_imp.cpp 781-835

DNP3 Driver

› Relevant source files

Purpose and Scope

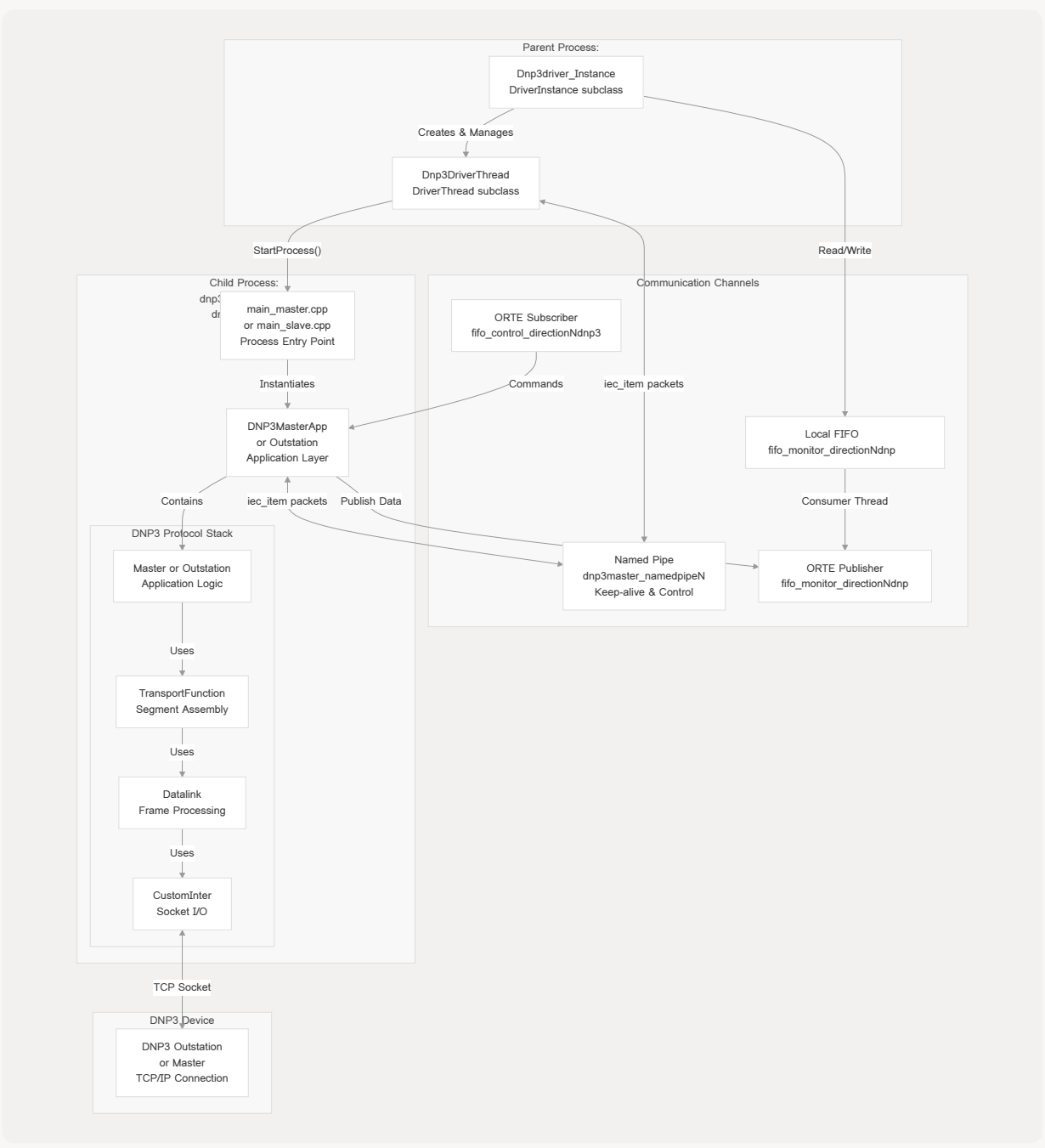
This page documents the DNP3 protocol driver implementation in IndigoSCADA. The DNP3 driver supports the DNP3 (Distributed Network Protocol version 3) industrial communication protocol, commonly used in electric utilities and water/wastewater systems. The driver operates as a child process (`dnp3master.exe` or `dnp3slave.exe`) that communicates with DNP3 devices over TCP/IP and exchanges data with the monitor process via ORTE middleware and named pipes.

For information about the common driver architecture pattern shared across all protocol drivers, see Protocol Driver Architecture. For database configuration specifics, see Protocol Configuration Tools.

Architecture Overview

The DNP3 driver follows IndigoSCADA's standard child process pattern with protocol-specific implementation. The driver implements a multi-layered DNP3 protocol stack including datalink, transport, and application layers.

Process Structure

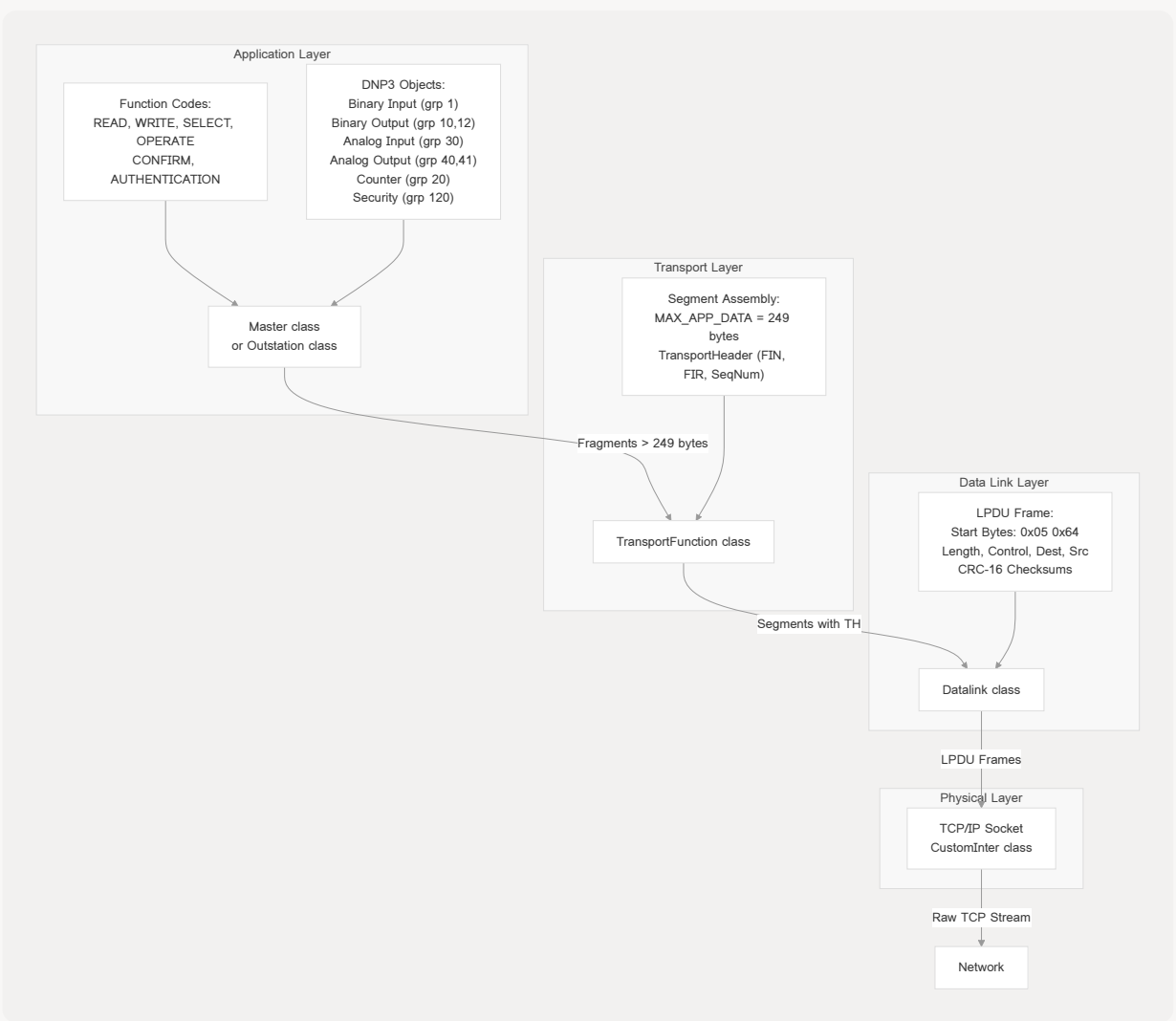

Parent Process:
Dnp3driver_Instance DriverInstance subclass
Creates & Manages
Dnp3DriverThread DriverThread subclass
StartProcess()
Read/Write
Child Process:
main_master.cpp or main_slave.cpp Process Entry Point
Communication Channels
ORTE Subscriber fifo_control_directionNdnp3
Instantiates
Commands
iec_item packets
DNP3MasterApp or Outstation Application Layer
Local FIFO fifo_monitor_directionNdnp
Contains
iec_item packets
Publish Data
Consumer Thread
DNP3 Protocol Stack
Master or Outstation Application Logic
Named Pipe dnp3master_namedpipeN Keep-alive & Control
ORTE Publisher fifo_monitor_directionNdnp
Uses
TransportFunction Segment Assembly
Uses
Datalink Frame Processing
Uses
CustomInter Socket I/O
TCP Socket
DNP3 Device
DNP3 Outstation or Master TCP/IP Connection

Sources: `src/drivers/dnp3/dnp3driver_instance.h` 1-249
`src/drivers/dnp3/protocol/main_master.cpp` 1-448
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 1-632

DNP3 Protocol Stack

The DNP3 driver implements a three-layer protocol stack conforming to the DNP3 specification. Each layer provides distinct services and operates independently.

Protocol Stack Layers


Application Layer
Function Codes: READ, WRITE, SELECT, OPERATE CONFIRM, AUTHENTICATION
DNP3 Objects: Binary Input (grp 1) Binary Output (grp 10,12) Analog Input (grp 30) Analog Output (grp 40,41) Counter (grp 20) Security (grp 120)
Master class or Outstation class
Transport Layer
Segment Assembly: MAX_APP_DATA = 249 bytes TransportHeader (FIN, FIR, SeqNum)
Fragments > 249 bytes
TransportFunction class
Data Link Layer
LPDU Frame: Start Bytes: 0x05 0x64 Length, Control, Dest, Src CRC-16 Checksums
Segments with TH
Datalink class
LPDU Frames
Physical Layer
TCP/IP Socket CustomInter class
Raw TCP Stream
Network

Sources: `src/drivers/dnp3/protocol/transport.cpp` 29-261
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 18-26

Transport Layer Segmentation

The transport function splits application layer fragments larger than 249 bytes into smaller segments and reassembles them on reception. Each segment includes a transport header with FIN (final), FIR (first), and sequence number bits.

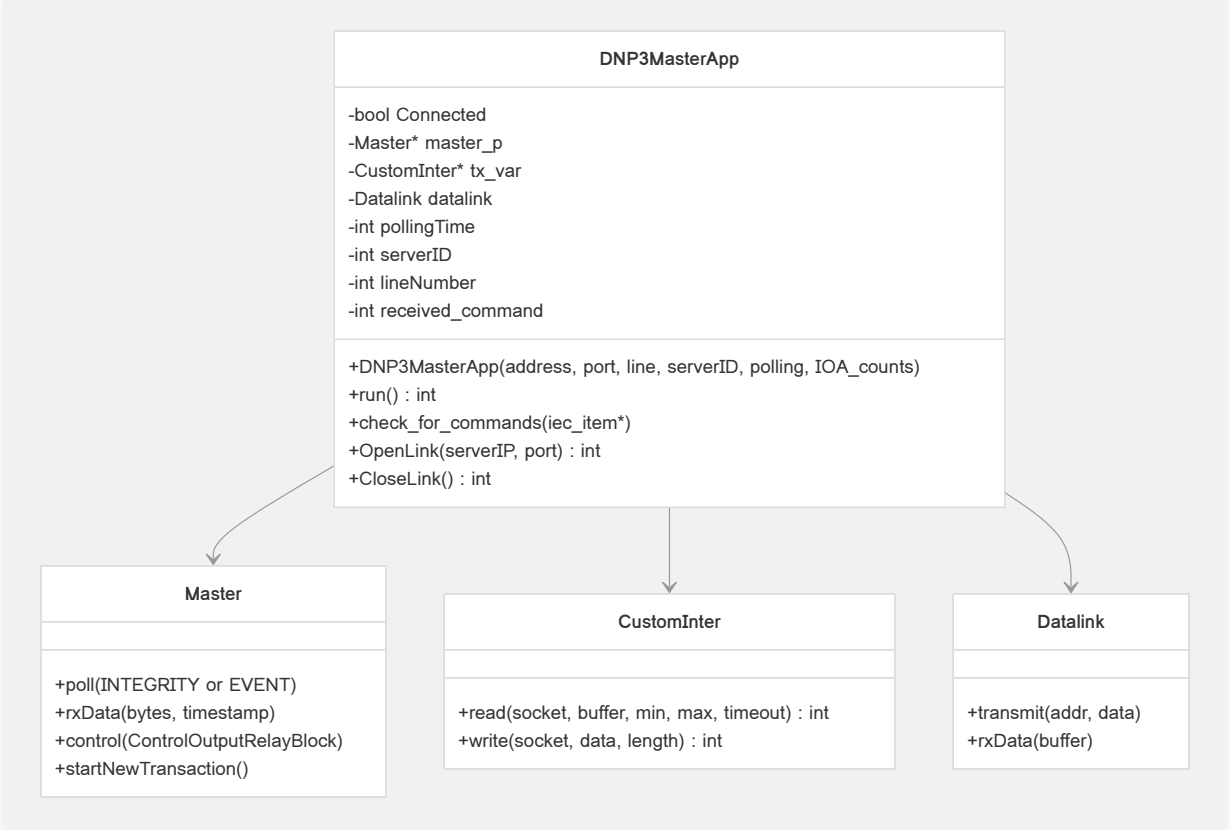
Transport Header Field	Bits	Description
FIN (Final)	1 bit (0x80)	Set when this is the last segment in sequence
FIR (First)	1 bit (0x40)	Set when this is the first segment in sequence
Sequence Number	6 bits (0x3F)	Wraps 0-63, increments per segment within fragment

Sources: `src/drivers/dnp3/protocol/transport.cpp` 41-82
`src/drivers/dnp3/protocol/transport.cpp` 95-142

Master Implementation

The DNP3 master implementation initiates communications with DNP3 outstations (slaves), performs polling for data, and executes control commands.

DNP3MasterApp Class


DNP3MasterApp
-bool Connected
-Master* master_p
-CustomInter* tx_var
-Datalink datalink
-int pollingTime
-int serverID
-int lineNumber
-int received_command
+DNP3MasterApp(address, port, line, serverID, polling, IOA_counts)
+run() : int
+check_for_commands(iec_item*)
+OpenLink(serverIP, port) : int
+CloseLink() : int
Master
+poll(INTEGRITY or EVENT)
+rxData(bytes, timestamp)
+control(ControlOutputRelayBlock)
+startNewTransaction()
CustomInter
+read(socket, buffer, min, max, timeout) : int
+write(socket, data, length) : int
Datalink
+transmit(addr, data)
+rxData(buffer)

Sources: `src/drivers/dnp3/protocol/dnp3_master_app.cpp` 117-266

Polling and General Interrogation

The master performs periodic integrity polls (general interrogation) to request all data from the outstation. The polling mechanism operates in the main run loop:

Integrity Poll Sequence:

1. Check if general interrogation command received or initial startup (`received_command == C_IC_NA_1` or `-1`)
2. Call `master_p->poll(Master::INTEGRITY)`
3. Read response data via `tx_var->read()`
4. Parse received bytes into DNP3 objects
5. Convert to `iec_item` structures and publish to middleware

Configuration Parameters:

- `pollingTime`: Interval in milliseconds between transactions (command-line `-t`)
- `integrityPollInterval`: Seconds between integrity polls (hardcoded to 10)
- `serverID`: DNP3 address of the outstation (command-line `-s`)
- `lineNumber`: Master address for this instance (command-line `-l`)

Sources: `src/drivers/dnp3/protocol/dnp3_master_app.cpp` 424-608
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 220-245

Command Line Arguments

The `dnp3master.exe` process accepts the following command-line arguments:

Argument	Description	Example
`-a`	DNP3 server IP address	`192.168.1.100`
`-p`	DNP3 server TCP port	`20000`
`-l`	Line number (master address)	`1`
`-s`	Server ID (outstation address)	`2`
`-t`	Polling time in milliseconds	`1000`
`-c`	Starting IOA for Analog Outputs	`200`
`-d`	Starting IOA for Binary Outputs	`150`
`-e`	Starting IOA for Counters	`100`
`-f`	Starting IOA for Binary Inputs	`50`
`-g`	Starting IOA for Analog Inputs	`1`

Sources: `src/drivers/dnp3/protocol/main_master.cpp` 62-211

Control Command Execution

Control commands received from the monitor process via ORTE subscription are processed in the `check_for_commands()` method:

Supported IEC 104 Command Types:

- `C_SC_TA_1`, `C_SC_NA_1`: Single Command (Binary Output)
- `C_DC_TA_1`, `C_DC_NA_1`: Double Command
- `C_SE_TA_1`, `C_SE_NA_1`: Set Point Command Short Float
- `C_SE_TB_1`, `C_SE_NB_1`: Set Point Command Scaled Value
- `C_SE_TC_1`, `C_SE_NC_1`: Set Point Command Normalized Value
- `C_BO_TA_1`, `C_BO_NA_1`: Bitstring Command
- `C_IC_NA_1`: General Interrogation
- `C_EX_IT_1`: Exit Process

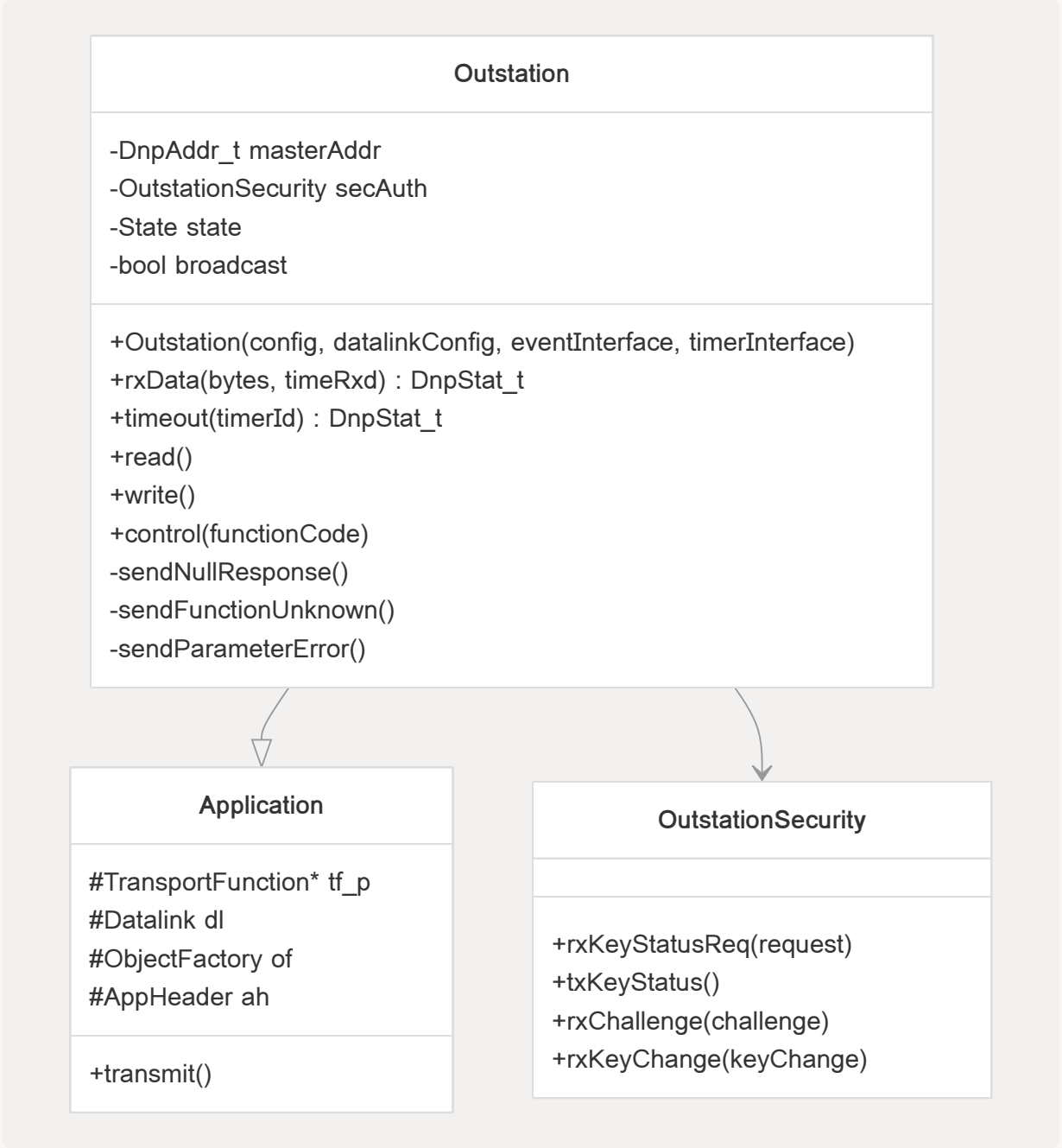
When a control command is received, it sets `received_command` which is then processed in the main loop by calling `master_p->control()` with a `ControlOutputRelayBlock` object.

Sources: `src/drivers/dnp3/protocol/dnp3_master_app.cpp` 369-412
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 530-550

Outstation (Slave) Implementation

The DNP3 outstation responds to master requests and can send unsolicited responses. The outstation implementation is minimal in the current codebase, primarily serving as a test endpoint.

Outstation Class


Outstation
-DnpAddr_t masterAddr
-OutstationSecurity secAuth
-State state
-bool broadcast
+Outstation(config, datalinkConfig, eventInterface, timerInterface)
+rxData(bytes, timeRxd) : DnpStat_t
+timeout(timerId) : DnpStat_t
+read()
+write()
+control(functionCode)
-sendNullResponse()
-sendFunctionUnknown()
-sendParameterError()
Application
#TransportFunction* tf_p
#Datalink dl
#ObjectFactory of
#AppHeader ah
+transmit()
OutstationSecurity
+rxKeyStatusReq(request)
+txKeyStatus()
+rxChallenge(challenge)
+rxKeyChange(keyChange)

Sources: `src/drivers/dnp3/protocol/outstation.cpp` 46-136
`src/drivers/dnp3/protocol/main_slave.cpp` 27-59

Request Processing

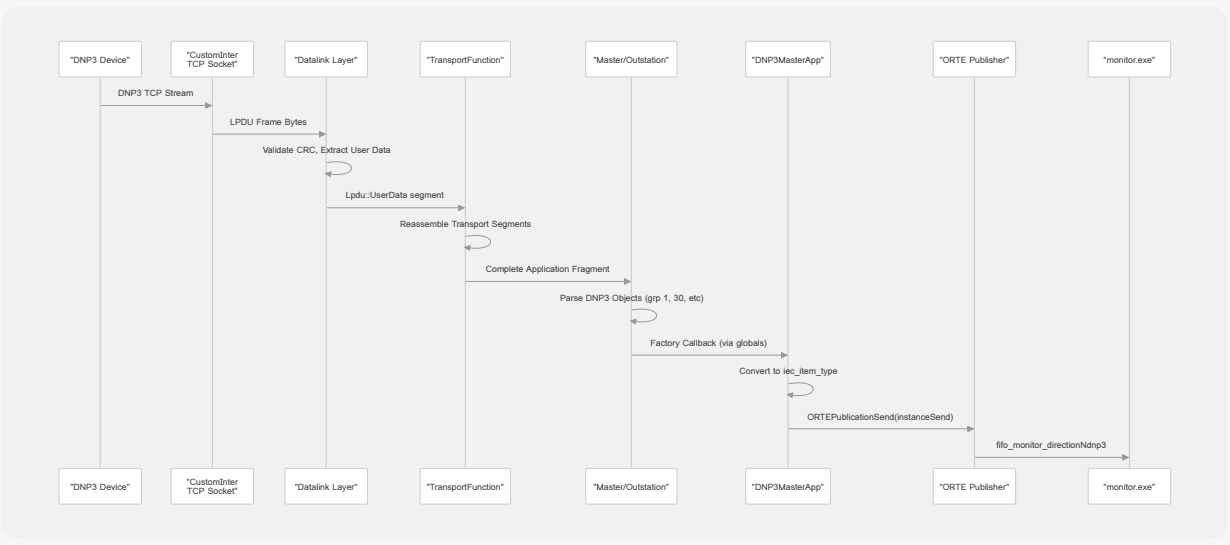
The outstation processes incoming requests based on function codes:

Function Code	Handler Method	Response
`READ`	`read()`	Static data or events based on Class 0/1/2/3 objects
`WRITE`	`write()`	Null response or parameter error
`SELECT`	`control(SELECT)`	Echo control request with status
`OPERATE`	`control(OPERATE)`	Echo control request with status
`CONFIRM`	(inline)	Process confirmation of unsolicited response
`AUTHENTICATION_REPLY`	(via secAuth)	Secure authentication handshake

Sources: `src/drivers/dnp3/protocol/outstation.cpp` 210-273
`src/drivers/dnp3/protocol/outstation.cpp` 302-381

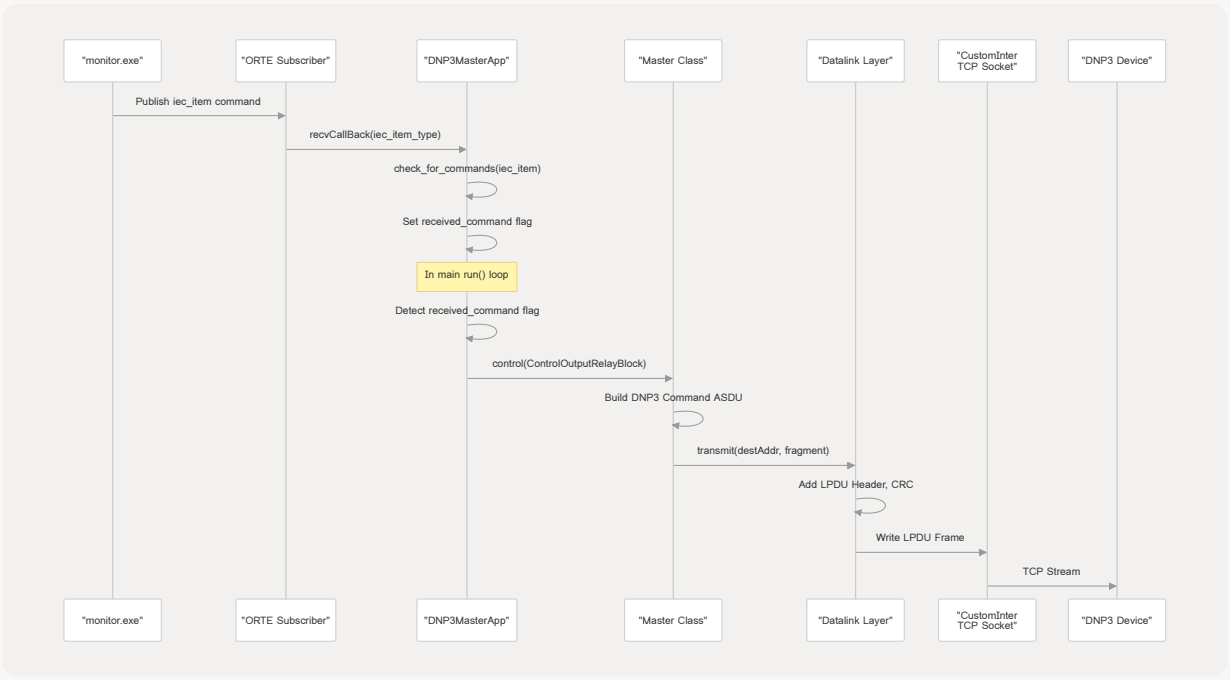
Data Flow and Middleware Integration

Monitor Direction (Device → IndigoSCADA)


"DNP3 Device"
"CustomInter TCP Socket"
"Datalink Layer"
"TransportFunction"
"Master/Outstation"
"DNP3MasterApp"
"ORTE Publisher"
"monitor.exe"
DNP3 TCP Stream
LPDU Frame Bytes
Validate CRC, Extract User Data
Lpdu UserData segment
Reassemble Transport Segments
Complete Application Fragment
Parse DNP3 Objects (grp 1, 30, etc)
Factory Callback (via globals)
Convert to iec_item_type
ORTEPublicationSend(instanceSend)
fifo_monitor_directionNdnp3

Sources: `src/drivers/dnp3/protocol/dnp3_master_app.cpp` 442-470
`src/drivers/dnp3/protocol/transport.cpp` 146-258

Control Direction (IndigoSCADA → Device)


"monitor.exe"
"ORTE Subscriber"
"DNP3MasterApp"
"Master Class"
"Datalink Layer"
"CustomInter TCP Socket"
"DNP3 Device"
Publish iec_item command
recvCallBack(iec_item_type)
check_for_commands(iec_item)
Set received_command flag
In main run() loop
Detect received_command flag
control(ControlOutputRelayBlock)
Build DNP3 Command ASDU
transmit(destAddr, fragment)
Add LPDU Header, CRC
Write LPDU Frame
TCP Stream

Sources: `src/drivers/dnp3/protocol/dnp3_master_app.cpp` 76-101
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 530-550

IOA Mapping Strategy

The DNP3 driver uses a simple offset-based IOA (Information Object Address) mapping scheme. Starting IOA values are provided as command-line arguments:

DNP3 Object Type	Starting IOA	Default	IEC 104 Type
Analog Inputs (grp 30)	`-g`	1	M_ME_* (Measured values)
Binary Inputs (grp 1)	`-f`	50	M_SP_* (Single point)
Counters (grp 20)	`-e`	100	M_IT_* (Integrated totals)
Binary Outputs (grp 10,12)	`-d`	150	M_BO_* (Bitstring outputs)
Analog Outputs (grp 40,41)	`-c`	200	M_ME_* (Measured values)

The actual IOA for a DNP3 point is calculated as: `IOA = starting_IOA + DNP3_point_index`

This mapping is stored in the `CustomDb` structure accessed by the factory callbacks during data conversion.

Sources: `src/drivers/dnp3/protocol/main_master.cpp` 78-96
`src/drivers/dnp3/protocol/main_master.cpp` 206-211
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 135-139

Keep-Alive and Process Management

Named Pipe Watchdog

The DNP3 master process implements a watchdog mechanism using Windows named pipes to detect loss of communication with the parent monitor process:

Pipe Worker Thread:

- Creates named pipe: `\\.\pipe\dnp3master_namedpipeN` (where N is line number)
- Listens for keep-alive messages with IOA `4004`
- Resets `gl_timeout_connection_with_parent` counter on each keep-alive
- Exits process if counter exceeds threshold (`1000*60/pollingTime`)

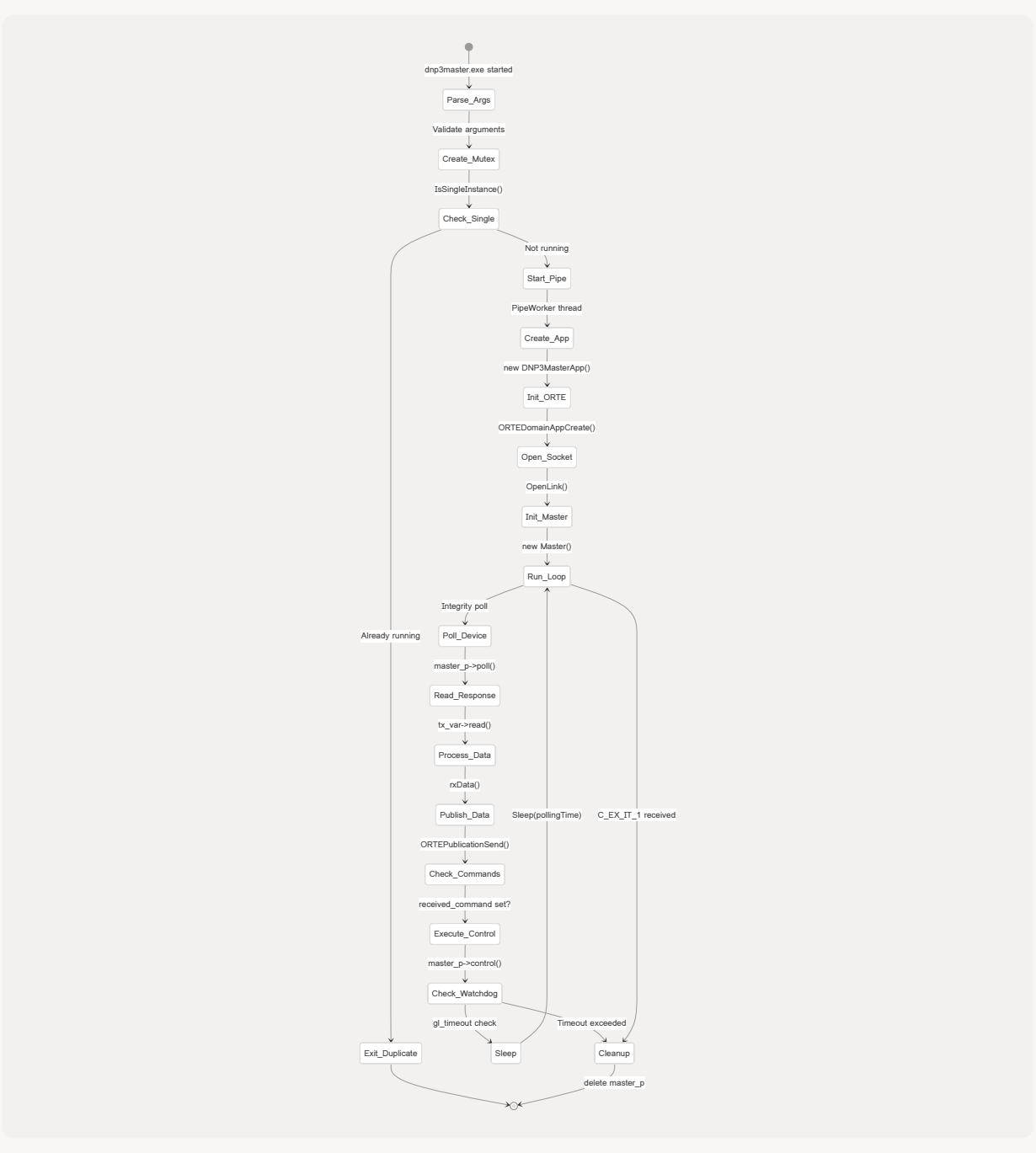
Main Loop Timeout Check:

```
#ifdef USE_KEEP_ALIVE_WATCH_DOG
gl_timeout_connection_with_parent++;
if(gl_timeout_connection_with_parent > 1000*60/pollingTime)
{
    return 1; //exit loops for timeout of connection with parent
}
#endif
```

This ensures that orphaned DNP3 processes terminate automatically if the parent monitor crashes.

Sources: `src/drivers/dnp3/protocol/main_master.cpp` 306-447
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 552-561

Process Lifecycle


dnp3master.exe started
Parse_Args
Validate arguments
Create_Mutex
IsSingleInstance()
Check_Single
Not running
Start_Pipe
PipeWorker thread
Create_App
new DNP3MasterApp()
Init_ORTE
ORTEDomainAppCreate()
Open_Socket
OpenLink()
Init_Master
new Master()
Run_Loop
Integrity poll
Poll_Device
Already running
master_p->poll()
Read_Response
tx_var->read()
Process_Data
rxData()
Publish_Data
Sleep(pollingTime)
C_EX_IT_1 received
ORTEPublicationSend()
Check_Commands
received_command set?
Execute_Control
master_p->control()
Check_Watchdog
gl_timeout check
Timeout exceeded
Exit_Duplicate
Sleep
Cleanup
delete master_p

Sources: `src/drivers/dnp3/protocol/main_master.cpp` 71-299
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 424-608

Secure Authentication

The DNP3 driver includes support for DNP3 Secure Authentication (SA) as specified in IEEE 1815-2012. This feature is primarily implemented but depends on cryptographic libraries (`USE_CRIPTO` define).

Security State Machine

The secure authentication implementation uses a state machine with separate states for master and outstation:

Master States:

- `INIT`: Initial state before key exchange
- `IDLE`: Normal operation with valid session keys
- `WAIT_FOR_RESPONSE`: Waiting for authentication response
- `WAIT_FOR_KEY_STATUS`: Waiting for key status from outstation
- `WAIT_FOR_KEY_CONFIRMATION`: Waiting for confirmation after key change

Outstation States:

- `IDLE`: Normal operation
- `WAIT_FOR_RESPONSE`: Waiting for authentication response
- `WAIT_FOR_KEY_CHANGE`: Waiting for master to send new session keys

Security Objects (Group 120)

Variation	Object Type	Description
120.1	Challenge	Master sends challenge to outstation
120.2	Reply	Response to challenge with HMAC
120.4	Key Status Request	Master requests key status
120.5	Key Status	Outstation reports key status
120.6	Key Change	Master sends new wrapped session keys
120.7	Error	Authentication error indication

HMAC Calculation:

- Algorithm: SHA-1 HMAC (Challenge::SHA_1)
- Key: Pre-shared update key (configurable, defaults to `0,1,2,3,4,5,...`)
- Input: Challenge ASDU + (optionally) Challenged ASDU for critical messages
- Output: Leftmost 8 octets of 20-byte SHA-1 output

Sources: `src/drivers/dnp3/protocol/security.cpp` 45-856
`src/drivers/dnp3/protocol/outstation.cpp` 138-141

Configuration Database

Unlike some other protocol drivers, the DNP3 driver does NOT use a SQLite database for configuration. Instead, all configuration is passed via command-line arguments to the `dnp3master.exe` process. The `Dnp3DriverThread` class in the parent process is responsible for constructing the command line with appropriate parameters from the UNITS table.

Configuration Flow:

1. Monitor process reads UNITS table from configdb (FastDB)
2. `Dnp3DriverThread::Start()` constructs command line with `-a`, `-p`, `-l`, `-s`, `-t`, etc.
3. `CreateProcess()` launches `dnp3master.exe` with arguments
4. Child process parses arguments in `main()` using `getopt()`
5. Arguments passed to `DNP3MasterApp` constructor

This differs from IEC 104 and Modbus drivers which use SQLite databases (`iec104_databaseN.db`, `modbus_databaseN.db`) for IOA mappings.

Sources: `src/drivers/dnp3/protocol/main_master.cpp` 62-211
`src/drivers/dnp3/dnp3driver_instance.h` 99-104

Build Configuration

Project Files

The DNP3 driver consists of two executables built from separate Visual Studio DSP projects:

dnp3master.dsp:

- Output: `c:\scada\bin\dnp3master.exe`
- Links against: `ws2_32.lib`, `rtps.lib`
- Include paths: Protocol stack headers, common utilities, middleware
- Source files: Master application, protocol stack layers (datalink, transport, master, station, factory, objects)

dnp3slave.dsp:

- Output: `c:\scada\bin\dnp3slave.exe`
- Links against: `ws2_32.lib`, `rtps.lib`
- Similar structure but uses `main_slave.cpp` and `Outstation` class

Sources: `src/drivers/dnp3/protocol/dnp3master.dsp` 1-185
`src/drivers/dnp3/protocol/dnp3slave.dsp` 1-169

Key Dependencies

Component	Purpose	Files
Protocol Stack	DNP3 layers	`datalink.cpp`, `transport.cpp`, `master.cpp`, `outstation.cpp`, `station.cpp`
Object Handling	DNP3 data objects	`object.cpp`, `factory.cpp`, `asdu.cpp`
Communication	Socket I/O	`custom.cpp`, `transmit_interface.cpp`
Middleware	ORTE integration	`iec_item_type.c` from common
Utilities	CRC, getopt	`clear_crc_eight.c`, `getopt.c`
Security	Authentication	`security.cpp`, `wrap.h` (optional), `sha1.h` (optional)

Sources: `src/drivers/dnp3/protocol/dnp3master.dsp` 91-182

Limitations and Future Enhancements

Current Limitations:

1. **Slave Implementation Incomplete**: The `dnp3slave.exe` outstation implementation is minimal and primarily for testing
2. **No Serial Support**: Only TCP/IP transport is implemented (no serial RS-232/485)
3. **Hard-coded Configuration**: Integrity poll interval and other parameters are hard-coded rather than configurable
4. **Global Variables**: Uses `gl_instanceSend` and `gl_publisher` globals for factory callbacks (noted with "remove ASAP" comments)
5. **Limited Data Type Support**: Only basic DNP3 object types are fully implemented
6. **Crypto Optional**: Secure authentication requires `USE_CRIPTO` compile flag and external crypto libraries

Potential Enhancements:

- SQLite database configuration similar to other drivers
- Support for DNP3 over serial (RFC 1006 or direct)
- Full outstation implementation for protocol gateway scenarios
- Event scanning and unsolicited response generation
- File transfer objects (Group 70)
- Time synchronization objects
- Dead band and data filtering

Sources: `src/drivers/dnp3/protocol/dnp3_master_app.cpp` 109-112
`src/drivers/dnp3/protocol/dnp3_master_app.cpp` 239-243

src/drivers/dnp3/protocol/security.cpp 38-41

OPC DA Client

> Relevant source files

Purpose and Scope

This page documents the OPC Data Access (DA) client driver implementation in IndigoSCADA. The OPC DA client connects to external OPC DA servers to acquire real-time process data and publish it to the IndigoSCADA middleware. This driver follows the common child process architecture pattern used by other IndigoSCADA protocol drivers.

For information about OPC Alarms & Events client functionality, see OPC AE Client. For OPC Historical Data Access, see OPC HDA Client. For an overview of all OPC Classic clients, see OPC Classic Clients. For general driver architecture patterns, see Protocol Driver Architecture.

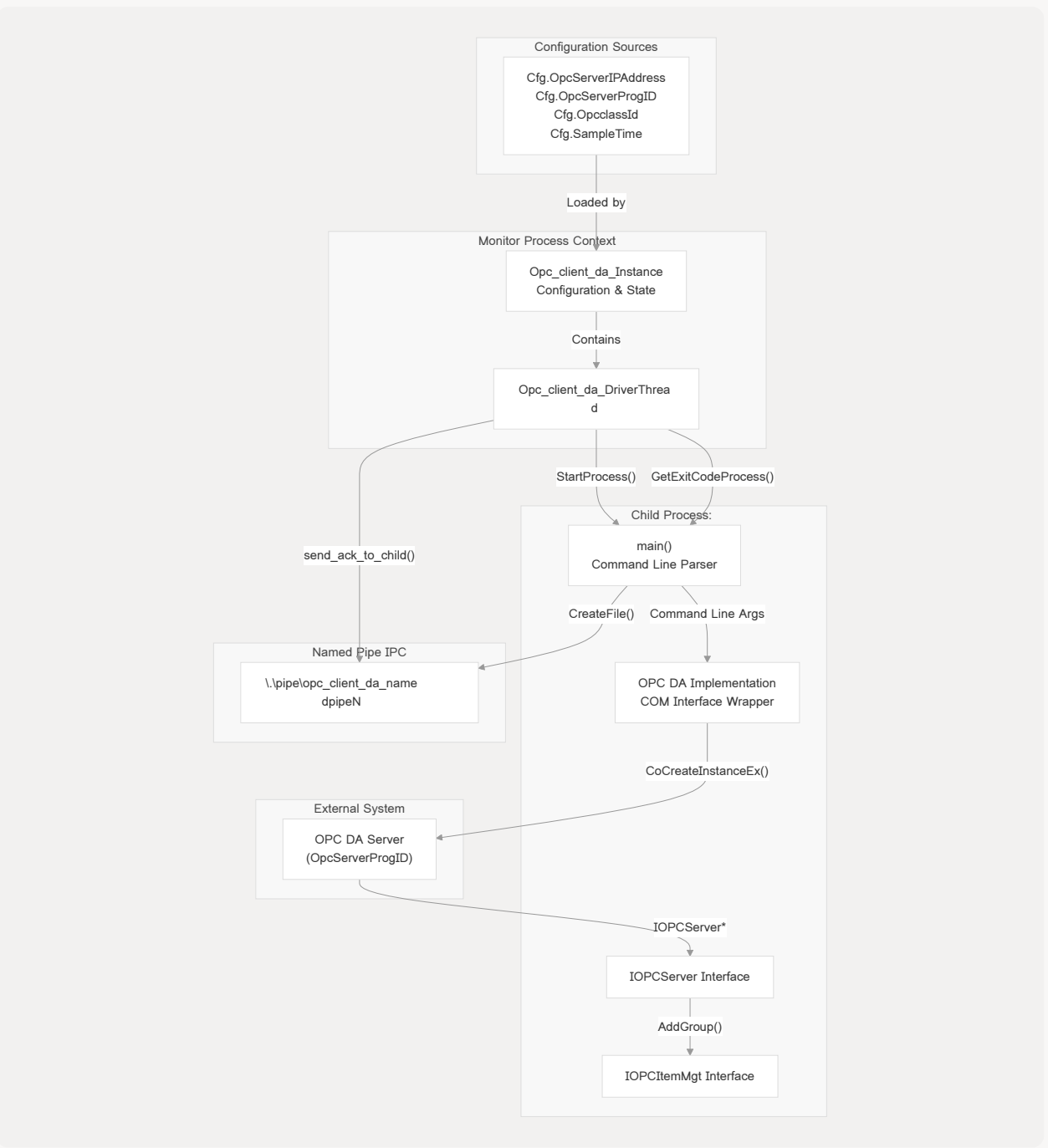
Architecture Overview

The OPC DA client driver consists of two processes:

1. **Parent Process (DriverThread)**: Runs within monitor.exe context, manages child process lifecycle, monitors health
2. **Child Process (opc_client_da.exe)**: Implements the actual OPC DA protocol communication and data acquisition

This separation provides process isolation, allowing the OPC client to crash and restart without affecting the main SCADA system.

Component Architecture Diagram


Configuration Sources
Cfg.OpcServerIPAddress
Cfg.OpcServerProgID
Cfg.OpcclassId
Cfg.SampleTime
Loaded by
Monitor Process Context
Opc_client_da_Instance
Configuration & State
Contains
Opc_client_da_DriverThread
StartProcess()
GetExitCodeProcess()
send_ack_to_child()
Child Process:
main()
Command Line Parser
CreateFile()
Command Line Args
Named Pipe IPC
\\.\pipe\opc_client_da_namedpipeN
OPC DA Implementation
COM Interface Wrapper
CoCreateInstanceEx()
External System
OPC DA Server
(OpcServerProgID)
IOPCServer*
IOPCServer Interface
AddGroup()
IOPCItemMgt Interface

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.h`
`src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 22-135

Driver Thread Class: Opc_client_da_DriverThread

The `Opc_client_da_DriverThread` class inherits from `DriverThread` and manages the lifecycle of the OPC DA child process.

Class Structure

Member Variable	Type	Purpose
`pProcInfo[nMaxProcCount]`	`PROCESS_INFORMATION`	Array to store process handles (max 127)
`pCommandLine[nBufferSize+1]`	`char[]`	Command line string for child process
`pWorkingDir[nBufferSize+1]`	`char[]`	Working directory for child process
`nIndex`	`int`	Current process index (always 1 for single instance)
`pipe_name[150]`	`char[]`	Named pipe path for IPC
`h_pipe`	`HANDLE`	Pipe handle for communication
`pipe_sends_cont`	`int`	Counter for periodic keep-alive messages
`restart_count`	`int`	Counter for restart timing checks
`msg_id`	`int`	Sequential message ID for iec_item packets
`instanceID`	`int`	Driver instance number (from parent)
`Done`	`bool`	Flag indicating thread completion

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.h` 22-74

Key Methods

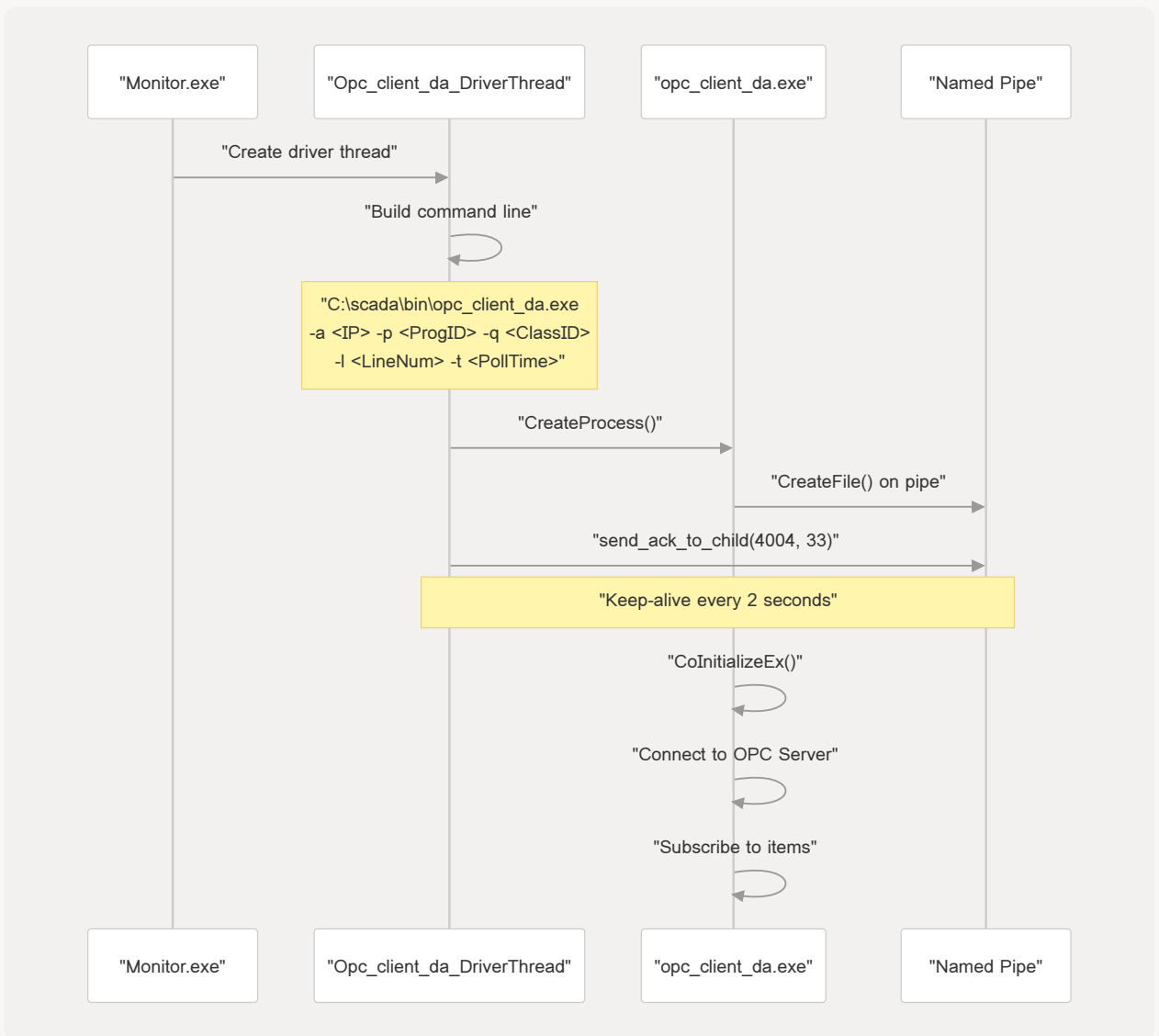
Method	Purpose
`run()`	Main thread loop - starts child process, monitors health, sends keep-alive
`StartProcess(char*, char*)`	Creates child process using Win32 CreateProcess API
`EndProcess(int)`	Terminates child process gracefully then forcefully
`send_ack_to_child(int, int, char*)`	Sends iec_item packet to child via named pipe
`pipe_put(char*, char*, int)`	Low-level pipe write operation
`TerminateProtocol()`	Shutdown handler - stops child and sets halt flag
`TerminateChild()`	Emergency shutdown - immediately terminates child

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.h` 62-73
`src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 271-294

Process Lifecycle

Child Process Startup

The child process is launched during the `run()` method of `Opc_client_da_DriverThread`:


"Monitor.exe"
"Opc_client_da_DriverThread"
"opc_client_da.exe"
"Named Pipe"
"Create driver thread"
"Build command line"
"C:\scada\bin\opc_client_da.exe -a <IP> -p <ProgID> -q <ClassID> -l <LineNum> -t <PollTime>"
"CreateProcess()"
"CreateFile() on pipe"
"send_ack_to_child(4004, 33)"
"Keep-alive every 2 seconds"
"CoInitializeEx()"
"Connect to OPC Server"
"Subscribe to items"
"Monitor.exe"
"Opc_client_da_DriverThread"
"opc_client_da.exe"
"Named Pipe"

Command Line Construction `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 28-51 :

The driver thread constructs the command line with these parameters:

- `-a` : OPC Server IP address (`Cfg.OpcServerIPAddress`)
- `-p` : OPC Server ProgID (`Cfg.OpcServerProgID`)
- `-q` : OPC Server ClassID GUID (`Cfg.OpcclassId`)
- `-l` : Line number (instanceID + 1)
- `-t` : Polling time in milliseconds (`Cfg.SampleTime`)

Example command line:

```
C:\scada\bin\opc_client_da.exe -a 192.168.1.100 -p OPCServer.V1 -q {12345678-1234-1234-1234-12345678901
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 28-59

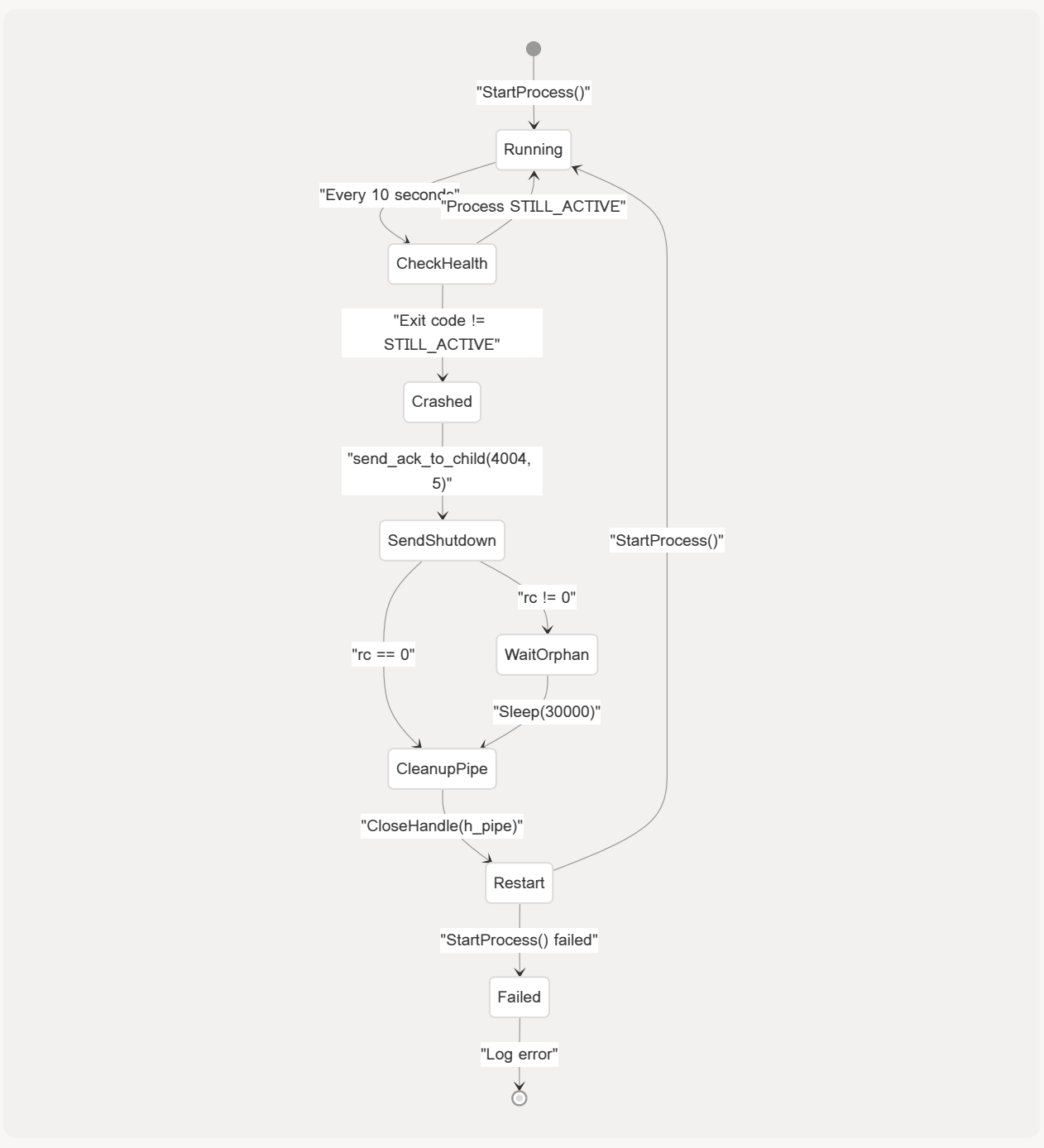
Named Pipe Creation

The named pipe path is constructed as: `\\.\pipe\opc_client_da_namedpipe<N>` where N is the line number (instanceID + 1).

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 32-36

Process Monitoring and Restart Logic

The driver thread continuously monitors the child process health and automatically restarts it on failure:


"StartProcess()"
Running
"Every 10 seconds"
"Process STILL_ACTIVE"
CheckHealth
"Exit code != STILL_ACTIVE"
Crashed
"send_ack_to_child(4004, 5)"
SendShutdown
"StartProcess()"
"rc != 0"
"rc == 0"
WaitOrphan
"Sleep(30000)"
CleanupPipe
"CloseHandle(h_pipe)"
Restart
"StartProcess() failed"
Failed
"Log error"

Health Check Implementation `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 74-121 :

```
restart_count++;

if(restart_count%10 == 0) //Check every 10 seconds
{
   if(pProcInfo[nIndex].hProcess)
   {
      DWORD dwCode;
      if(::GetExitCodeProcess(pProcInfo[nIndex].hProcess, &dwCode))
      {
         if(dwCode != STILL_ACTIVE)
         {
            // Process has crashed - restart it
            pProcInfo[nIndex].hProcess = 0;
            rc = send_ack_to_child(4004, 5, pipe_name);
            // ... cleanup and restart
         }
      }
   }
}
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 74-121

Keep-Alive Mechanism

The parent process sends periodic keep-alive messages to the child process every 2 seconds:

Keep-Alive Packet Structure

The keep-alive message uses the standard `iec_item` structure with type 37:

Field	Value	Purpose
`iec_type`	37	Internal keep-alive message type
`iec_obj.ioa`	4004	Special IOA for protocol messages
`iec_obj.o.type37.counter`	33	Keep-alive indicator value
`msg_id`	Sequential	Incrementing message counter
`checksum`	CRC	Message integrity check

Implementation `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 123-128 :

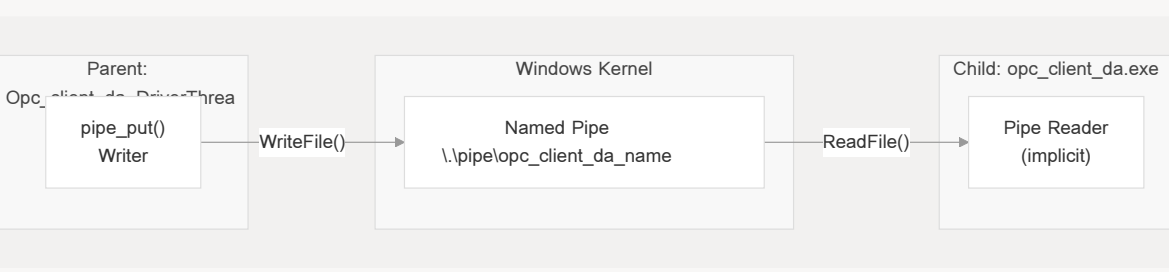
```
pipe_sends_cont++;

if(pipe_sends_cont%2 == 0)
{
   rc = send_ack_to_child(4004, 33, pipe_name); //Send a packet every 2 seconds
}
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 123-128
`src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 220-232

Named Pipe Communication

The driver uses Windows named pipes for bidirectional communication with the child process.

Pipe Communication Flow


Parent: Opc_client_da_DriverThread
pipe_put()
Writer
WriteFile()
Windows Kernel
Named Pipe
\\.\pipe\opc_client_da_name
ReadFile()
Child: opc_client_da.exe
Pipe Reader
(implicit)

pipe_put() Implementation

The `pipe_put()` method handles lazy pipe initialization and write operations `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 234-269 :

Lazy Initialization:

1. Opens pipe with `CreateFile(pipe_name, GENERIC_WRITE, ...)`
2. Waits for pipe availability with `WaitNamedPipe(pipe_name, 20)`
3. Sets message mode with `SetNamedPipeHandleState(h_pipe, &mode, ...)`

Write Operation:

- Uses `WriteFile(h_pipe, buf, len, &written, NULL)`
- Returns success/failure status for error handling

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 234-269

Process Creation Details

StartProcess() Method

The `StartProcess()` method creates the child process with specific Windows configuration `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 138-195 :

Key Configuration:

- **Show Window**: `SW_SHOWMINIMIZED` (minimized console window)
- **Creation Flags**: `CREATE_NEW_CONSOLE` (separate console)
- **Desktop**: NULL (default desktop)
- **Working Directory**: `C:\scada\bin`

STARTUPINFO Structure:

```
STARTUPINFO startUpInfo = {
    sizeof(STARTUPINFO), NULL, "", NULL, 0, 0, 0, 0, 0, 0, 0,
    STARTF_USESHOWWINDOW, 0, 0, NULL, 0, 0, 0
};
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 138-195

EndProcess() Method

Graceful process termination `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 204-218 :

1. Posts `WM_QUIT` message to main thread
2. Sleeps 100ms to allow graceful shutdown
3. Forcefully terminates with `TerminateProcess()`

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 204-218

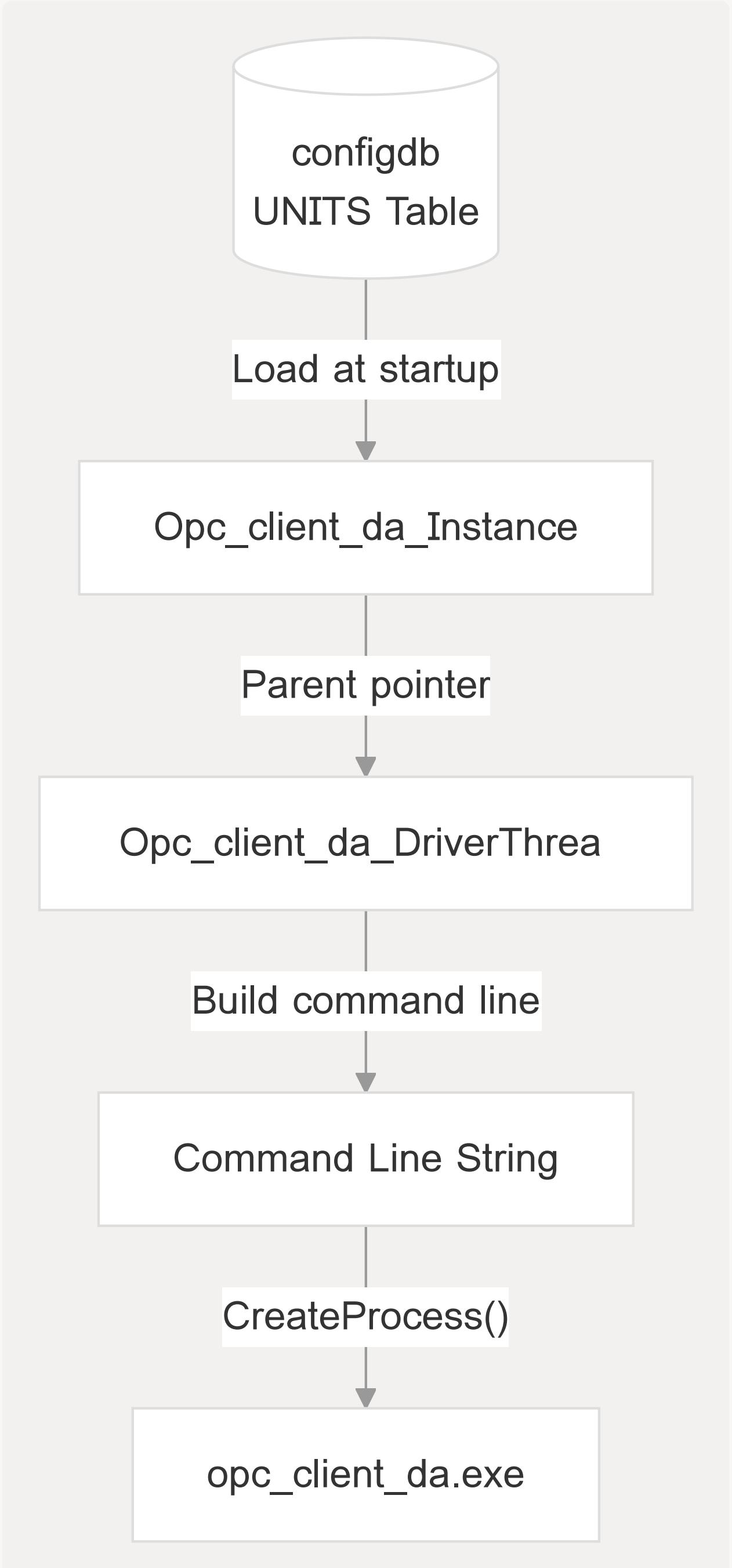
Configuration Integration

The OPC DA driver receives configuration from the `Opc_client_da_Instance` parent class:

Configuration Parameters

Parameter	Source Field	Purpose
OPC Server IP	`Cfg.OpcServerIPAddress`	Network address of OPC server
ProgID	`Cfg.OpcServerProgID`	OPC server programmatic identifier
ClassID	`Cfg.OpcclassId`	GUID for COM class instantiation
Polling Time	`Cfg.SampleTime`	Data acquisition interval (ms)
Instance ID	`instanceID`	Unique driver instance number

Configuration Flow:


configdb
UNITS Table
Load at startup
Opc_client_da_Instance
Parent pointer
Opc_client_da_DriverThrea
Build command line
Command Line String
CreateProcess()
opc_client_da.exe

Access Pattern `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 38-50 :

```
((Opc_client_da_Instance*)Parent)->Cfg.OpcServerIPAddress
((Opc_client_da_Instance*)Parent)->Cfg.OpcServerProgID
((Opc_client_da_Instance*)Parent)->Cfg.OpcclassId
((Opc_client_da_Instance*)Parent)->Cfg.SampleTime
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 38-50

Thread Lifecycle Management

Thread Initialization

Constructor `src/drivers/opc_client_da/opc_client_dadriverthread.h` 40-54 :

- Initializes pipe handle to NULL
- Sets process index to 1
- Zeros all process info structures
- Retrieves instance ID from parent

Thread Termination

TerminateProtocol() `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 271-278 :

- Calls `EndProcess(nIndex)` to stop child
- Sets `fRequestHalt = true` to exit main loop

TerminateChild() `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 280-293 :

- Emergency termination without cleanup
- Posts WM_QUIT and forcefully terminates

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 271-293

Comparison with Other OPC Drivers

The OPC DA, AE, and HDA drivers share identical process management architecture:

Feature	OPC DA	OPC AE	OPC HDA
Driver Thread Class	`Opc_client_da_DriverThread`	`Opc_client_ae_DriverThread`	`Opc_client_hda_DriverThread`
Child Executable	`opc_client_da.exe`	`opc_client_ae.exe`	`opc_client_hda.exe`
Pipe Name Pattern	`opc_client_da_namedpipeN`	`opc_client_ae_namedpipeN`	`opc_client_hda_namedpipeN`
Keep-Alive Interval	2 seconds	2 seconds	2 seconds
Health Check Interval	10 seconds	10 seconds	10 seconds
Command Line Args	`-a -p -q -l -t`	`-a -p -q -l -t`	`-a -p -q -l -t`

The primary difference lies in the child process implementation, which interfaces with different OPC specifications (Data Access, Alarms & Events, Historical Data Access).

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.h`
`src/drivers/opc_client_ae/opc_client_aedriverthread.cpp`
`src/drivers/opc_client_hda/opc_client_hdadriverthread.h`
`src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp`

Integration with Driver Architecture Pattern

The OPC DA driver follows the common IndigoSCADA driver pattern documented in Protocol Driver Architecture:

Pattern Conformance

Pattern Element	OPC DA Implementation
DriverInstance	`Opc_client_da_Instance` (implied, not in provided files)
DriverThread	`Opc_client_da_DriverThread`
Child Process	`opc_client_da.exe`
IPC Mechanism	Windows named pipe
Configuration Storage	SQLite (opc_client_da_table, implied)
Health Monitoring	GetExitCodeProcess() every 10 seconds
Keep-Alive	iec_item type 37, every 2 seconds
Process Management	CreateProcess() / TerminateProcess()

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.h`
`src/drivers/opc_client_da/opc_client_dadriverthread.cpp`

Error Handling

Process Startup Failures

If `CreateProcess()` fails `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 184-188 :

```
fprintf(stderr,"Failed to start program '%s', error code = %d\n",
        pCommandLine, GetLastError());
```

Process Crash Detection

The health check detects crashes via exit code and logs restart attempts `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 103-112 :

```
if(StartProcess(pCommandLine, pWorkingDir))
{
    fprintf(stderr, "Restarted process %s\n", pCommandLine);
}
else
{
    fprintf(stderr, "Failed to restart process %s\n", pCommandLine);
}
```

Orphan Process Handling

If the shutdown acknowledgment fails, the system waits 30 seconds for orphan cleanup `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 88-101 :

```
if(rc != 0)
{
    //wait the orphan child process to stop
```

```
        CloseHandle(h_pipe);
        h_pipe = NULL;
        Sleep(30000);
    }
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 84-121

Constants and Limits

Constant	Value	Purpose
`nMaxProcCount`	127	Maximum number of process slots
`nBufferSize`	500	Command line and working dir buffer size
`pipe_name` size	150	Named pipe path buffer size
Keep-alive IOA	4004	Special address for protocol messages
Keep-alive value	33	Keep-alive indicator
Shutdown message	5	Shutdown command value
Health check period	10 seconds	Process health check interval
Keep-alive period	2 seconds	Keep-alive message interval
Graceful shutdown delay	100ms	Time for graceful termination
Orphan wait time	30 seconds	Wait time for orphan cleanup

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.h` 19-20 `src/drivers/opc_client_da/opc_client_dadriverthread.cpp`

OPC AE Client

> Relevant source files

Purpose and Scope

This document describes the IndigoSCADA OPC Alarms & Events (AE) client driver implementation. The OPC AE Client subscribes to alarm and event notifications from OPC AE servers, converting them to IndigoSCADA's internal `iec_item` format for processing by the monitor and results processor. This driver follows the standard IndigoSCADA protocol driver pattern with a parent driver thread managing a child process that performs the actual OPC AE communication.

For information about OPC Data Access (real-time data polling), see OPC DA Client. For OPC Historical Data Access, see OPC HDA Client. For general protocol driver architecture patterns, see Protocol Driver Architecture.

Architecture Overview

The OPC AE Client uses a two-process architecture where the `Opc_client_ae_DriverThread` parent thread manages a child process (`opc_client_ae.exe`) that handles the actual OPC AE protocol communication.

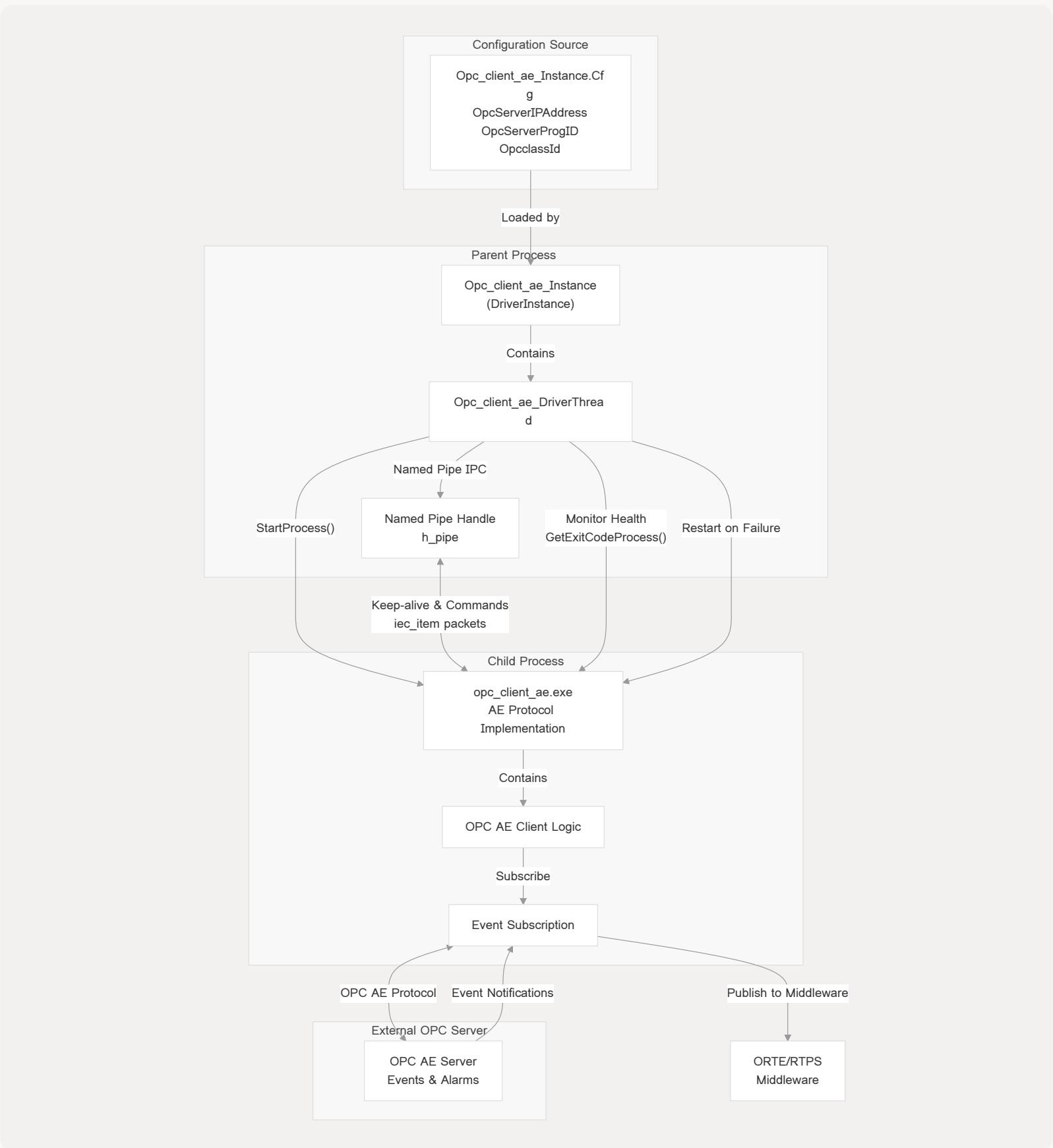

Configuration Source
Opc_client_ae_Instance.Cfg
OpcServerIPAddress
OpcServerProgID
OpcclassId
Loaded by
Parent Process
Opc_client_ae_Instance
(DriverInstance)
Contains
Opc_client_ae_DriverThread
Named Pipe IPC
StartProcess()
Named Pipe Handle
h_pipe
Monitor Health
GetExitCodeProcess()
Restart on Failure
Keep-alive & Commands
iec_item packets
Child Process
opc_client_ae.exe
AE Protocol
Implementation
Contains
OPC AE Client Logic
Subscribe
Event Subscription
OPC AE Protocol
Event Notifications
Publish to Middleware
External OPC Server
OPC AE Server
Events & Alarms
ORTE/RTPS
Middleware

Diagram: OPC AE Client Two-Process Architecture

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.h` 1-78
`src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 1-280

Driver Thread Class Structure

The `Opc_client_ae_DriverThread` class manages the child process lifecycle and inter-process communication.

Class Definition

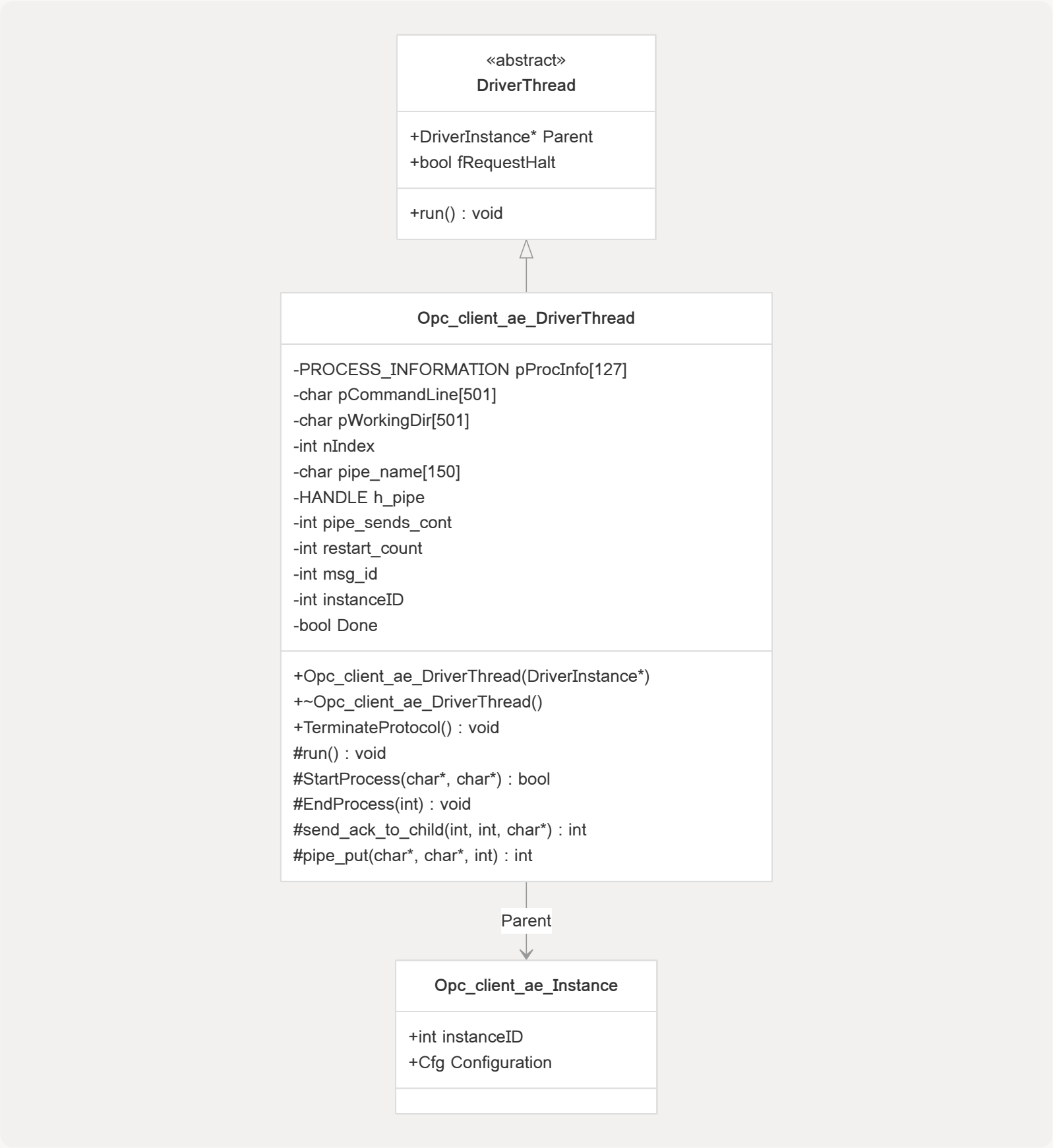

«abstract»
DriverThread
+DriverInstance* Parent
+bool fRequestHalt
+run() : void
Opc_client_ae_DriverThread
-PROCESS_INFORMATION pProcInfo[127]
-char pCommandLine[501]
-char pWorkingDir[501]
-int nIndex
-char pipe_name[150]
-HANDLE h_pipe
-int pipe_sends_cont
-int restart_count
-int msg_id
-int instanceID
-bool Done
+Opc_client_ae_DriverThread(DriverInstance*)
+~Opc_client_ae_DriverThread()
+TerminateProtocol() : void
#run() : void
#StartProcess(char*, char*) : bool
#EndProcess(int) : void
#send_ack_to_child(int, int, char*) : int
#pipe_put(char*, char*, int) : int
Parent
Opc_client_ae_Instance
+int instanceID
+Cfg Configuration

Diagram: Opc_client_ae_DriverThread Class Hierarchy

Key Member Variables

Member	Type	Purpose
`pProcInfo`	`PROCESS_INFORMATION[127]`	Process information for child processes (max 127 instances)
`pCommandLine`	`char[501]`	Command line string for launching child process
`pipe_name`	`char[150]`	Named pipe path: `\\\\.\\pipe\\opc_client_ae_namedpipe{N}`
`h_pipe`	`HANDLE`	Windows handle to the named pipe for IPC
`pipe_sends_cont`	`int`	Counter for periodic keep-alive messages
`restart_count`	`int`	Counter for child process health checks
`instanceID`	`int`	Instance identifier from parent configuration
`Done`	`bool`	Thread completion flag

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.h` 24-76

Child Process Lifecycle Management

Startup Sequence

The driver thread constructs a command line and starts the child process during its `run()` method execution.

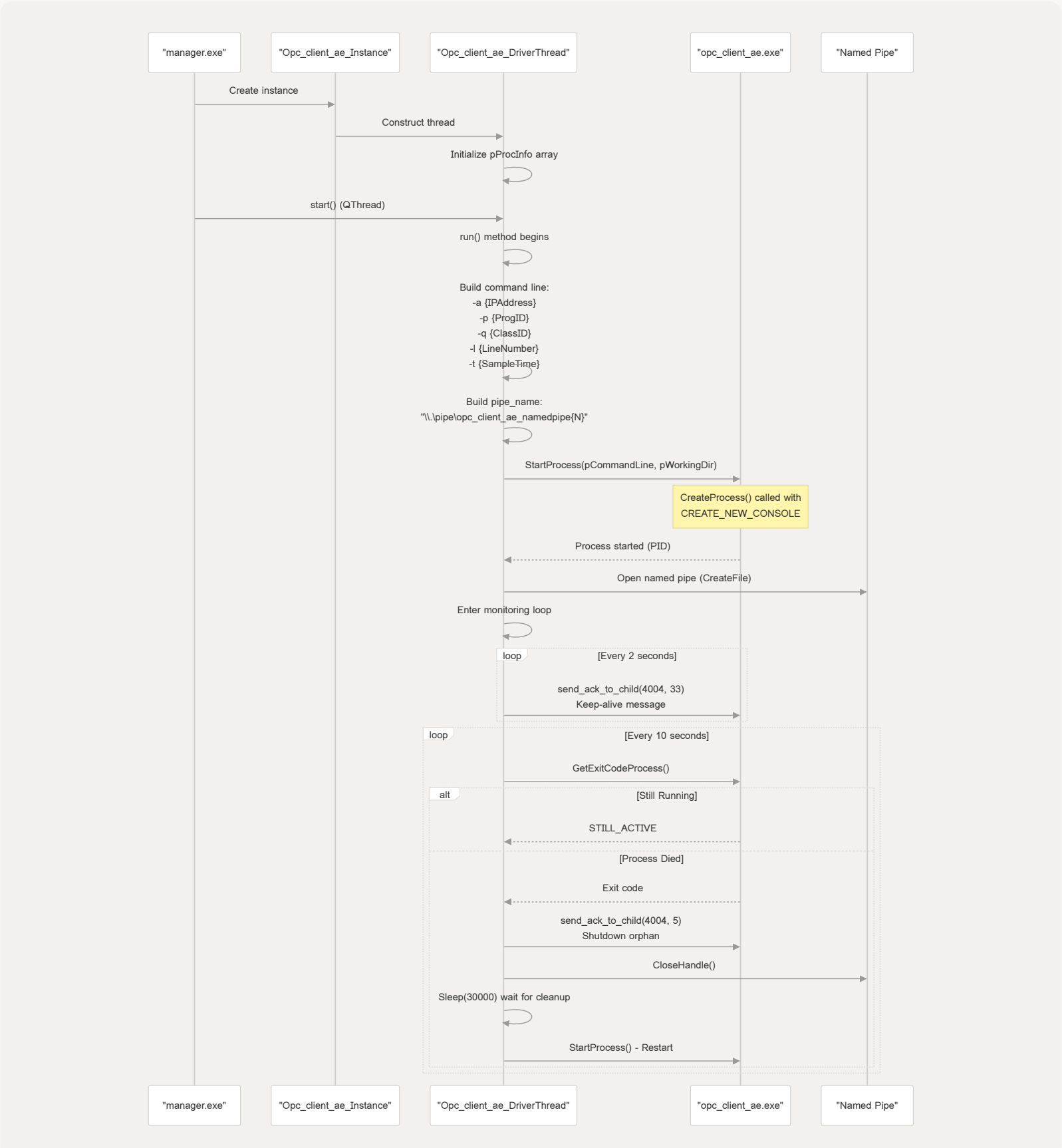

"manager.exe"
"Opc_client_ae_Instance"
"Opc_client_ae_DriverThread"
"opc_client_ae.exe"
"Named Pipe"
Create instance
Construct thread
Initialize pProcInfo array
start() (QThread)
run() method begins
Build command line:
-a {IPAddress}
-p {ProgID}
-q {ClassID}
-l {LineNumber}
-t {SampleTime}
Build pipe_name:
"\\.\pipe\opc_client_ae_namedpipe{N}"
StartProcess(pCommandLine, pWorkingDir)
CreateProcess() called with
CREATE_NEW_CONSOLE
Process started (PID)
Open named pipe (CreateFile)
Enter monitoring loop
loop
[Every 2 seconds]
send_ack_to_child(4004, 33)
Keep-alive message
loop
[Every 10 seconds]
GetExitCodeProcess()
alt
[Still Running]
STILL_ACTIVE
[Process Died]
Exit code
send_ack_to_child(4004, 5)
Shutdown orphan
CloseHandle()
Sleep(30000) wait for cleanup
StartProcess() - Restart

Diagram: Child Process Startup and Monitoring Sequence

Command Line Construction

The driver constructs the child process command line as follows:

```
C:\scada\bin\opc_client_ae.exe -a {OpcServerIPAddress} -p {OpcServerProgID}
    -q {OpcclassId} -l {lineNumber} -t {SampleTime}
```

Parameters:

- `-a`: OPC Server IP address or hostname
- `-p`: OPC Server ProgID (programmatic identifier)
- `-q`: OPC Server CLSID (class identifier)
- `-l`: Line number (instance ID + 1)
- `-t`: Sample/polling time in milliseconds

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 28-53

Process Monitoring and Restart Logic

The driver thread monitors child process health every 10 seconds using `GetExitCodeProcess()`. If the child exits unexpectedly:

1. Attempt graceful shutdown by sending termination message (address 4004, data 5)
2. Close the named pipe handle
3. Wait 30 seconds for orphan process cleanup
4. Restart the child process with `StartProcess()`

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 74-121

Inter-Process Communication

Named Pipe Mechanism

Communication between parent and child uses Windows named pipes with message mode.

Pipe Naming Convention:

```
\\.\pipe\opc_client_ae_namedpipe{instanceID+1}
```

For instance ID 0, the pipe name is `\\.\pipe\opc_client_ae_namedpipe1`.

Keep-Alive Protocol

The parent sends periodic keep-alive messages to the child process every 2 seconds to maintain the connection.

Keep-Alive Message Format:

```
struct iec_item {
    iec_type = 37;            // Message type
    iec_obj.ioa = 4004;       // Address
    iec_obj.o.type37.counter = 33;  // Keep-alive data
    msg_id = <incremental>;  // Message sequence number
    checksum = <calculated>; // CRC checksum
}
```

The termination message uses the same structure with `counter = 5` instead of 33.

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 220-232

Pipe Communication Functions

`pipe_put()`

Writes data to the named pipe with lazy connection establishment:

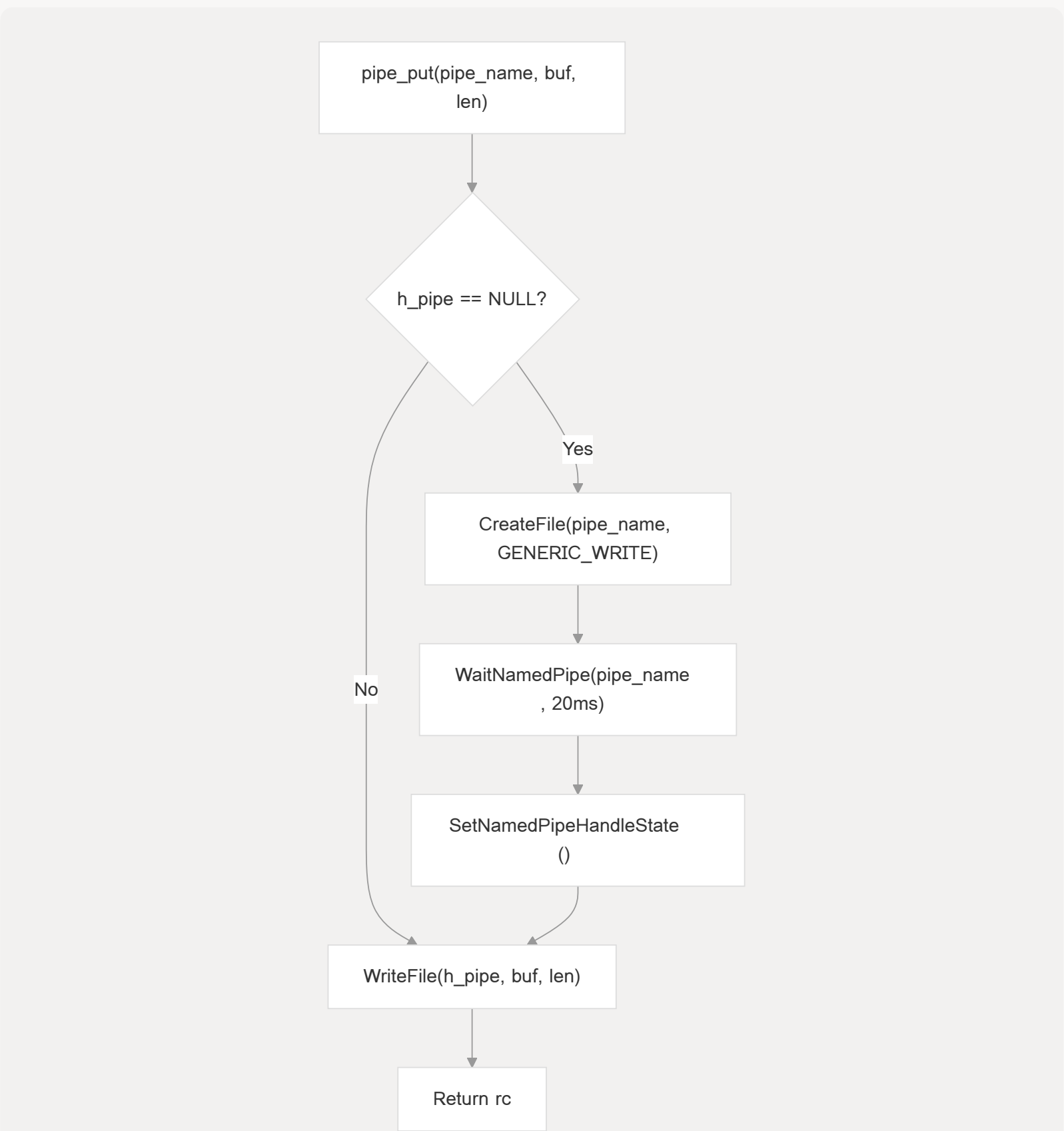

pipe_put(pipe_name, buf, len)
h_pipe == NULL?
Yes
No
CreateFile(pipe_name, GENERIC_WRITE)
WaitNamedPipe(pipe_name, 20ms)
SetNamedPipeHandleState()
WriteFile(h_pipe, buf, len)
Return rc

Diagram: Named Pipe Write Operation Flow

The pipe is opened on first use and kept open for subsequent writes. If the child process restarts, the pipe handle is closed and reopened on the next write attempt.

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 234-269

Process Termination

Graceful Shutdown

When the driver thread is requested to halt:

1. `TerminateProtocol()` is called
2. `EndProcess(nIndex)` terminates the child:
    - Posts `WM_QUIT` to child's main thread
    - Sleeps 100ms to allow graceful exit
    - Calls `TerminateProcess()` to force termination if still running
3. Sets `fRequestHalt = true` to exit monitoring loop

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 271-278
`src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 204-218

Configuration and Integration

Configuration Structure

The driver obtains configuration from the parent `Opc_client_ae_Instance` object:

Configuration Field	Purpose
`OpcServerIPAddress`	IP address or hostname of OPC AE server
`OpcServerProgID`	OPC server programmatic identifier
`OpcclassId`	OPC server CLSID (class identifier)
`SampleTime`	Event subscription update rate (milliseconds)

These parameters are loaded from the configuration database during instance creation and passed to the child process via command line arguments.

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 38-50

System Integration Points

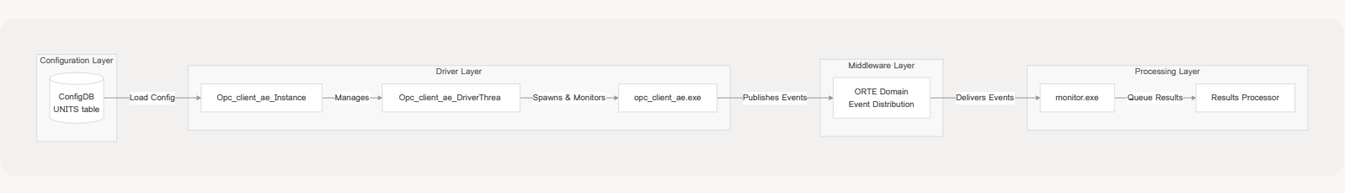

Configuration Layer
ConfigDB
UNITS table
Load Config
Driver Layer
Opc_client_ae_Instance
Manages
Opc_client_ae_DriverThread
Spawns & Monitors
opc_client_ae.exe
Publishes Events
Middleware Layer
ORTE Domain
Event Distribution
Delivers Events
Processing Layer
monitor.exe
Queue Results
Results Processor

Diagram: OPC AE Client System Integration

The child process (`opc_client_ae.exe`) subscribes to events from the OPC AE server and publishes them to the ORTE middleware using the `fifo_monitor_direction` publication channel. The monitor process subscribes to these events, routes them to appropriate sample points, and forwards them to the results processor for alarm evaluation and historical storage.

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.h` 1-78
`src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 1-135

Comparison with OPC DA and HDA Clients

All three OPC Classic client drivers (DA, AE, HDA) share the same architectural pattern:

Aspect	OPC DA	OPC AE	OPC HDA
Child Executable	`opc_client_da.exe`	`opc_client_ae.exe`	`opc_client_hda.exe`
Named Pipe Pattern	`opc_client_da_namedpipe{N}`	`opc_client_ae_namedpipe{N}`	`opc_client_hda_namedpipe{N}`
DriverThread Class	`Opc_client_da_DriverThread`	`Opc_client_ae_DriverThread`	`Opc_client_hda_DriverThread`
Primary Function	Poll real-time data	Subscribe to events/alarms	Query historical data
Update Mechanism	Periodic polling	Event-driven subscription	On-demand queries

All three use identical child process management, keep-alive protocols, and restart logic. The only differences are the protocol-specific implementations in their respective child processes.

Sources: `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 1-280 `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 1-296 `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 1-279

OPC HDA Client

› Relevant source files

Purpose and Scope

This document describes the OPC HDA (Historical Data Access) Client driver implementation in IndigoSCADA. The OPC HDA Client retrieves historical data from OPC HDA servers, supporting time-based queries and aggregation functions. This driver follows the standard IndigoSCADA child process architecture pattern where a driver thread manages a separate child process that performs the actual OPC HDA protocol operations.

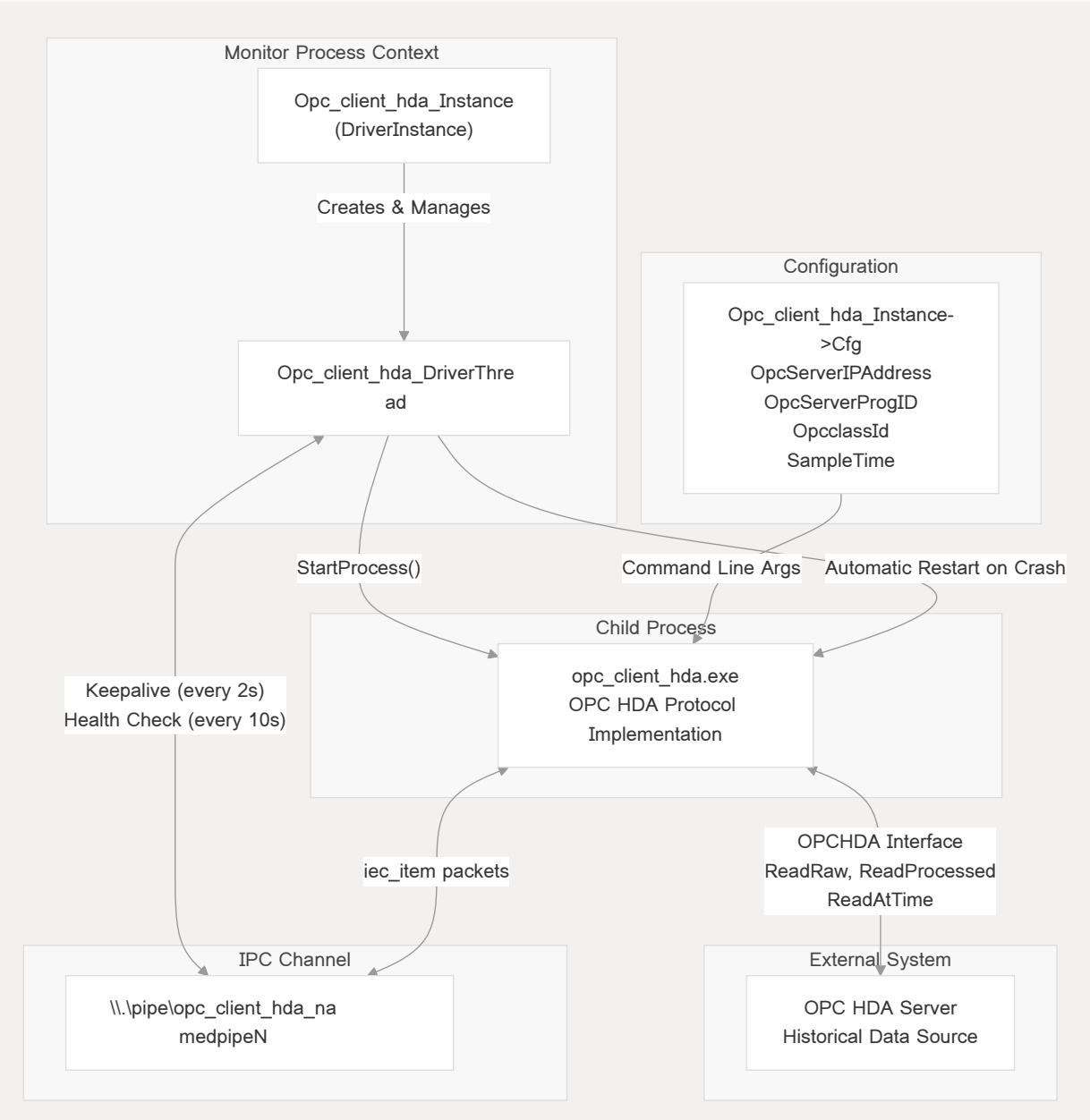
For real-time data acquisition from OPC servers, see OPC DA Client. For alarm and event subscription, see OPC AE Client. For the broader context of all OPC Classic clients, see OPC Classic Clients.

Architecture Overview

The OPC HDA Client driver implements a two-process architecture consistent with other IndigoSCADA protocol drivers:

1. **Parent Process (Driver Thread)**: Manages child process lifecycle, health monitoring, and keepalive communication
2. **Child Process (opc_client_hda.exe)**: Performs actual OPC HDA client operations, historical data retrieval, and middleware communication

Component Relationships


Monitor Process Context
Opc_client_hda_Instance (DriverInstance)
Creates & Manages
Opc_client_hda_DriverThread
Configuration
Opc_client_hda_Instance->Cfg OpcServerIPAddress OpcServerProgID OpcclassId SampleTime
StartProcess()
Command Line Args
Automatic Restart on Crash
Child Process
opc_client_hda.exe OPC HDA Protocol Implementation
Keepalive (every 2s) Health Check (every 10s)
iec_item packets
OPCHDA Interface ReadRaw, ReadProcessed ReadAtTime
IPC Channel
\\.\pipe\opc_client_hda_namedpipeN
External System
OPC HDA Server Historical Data Source

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.h` 1-75 `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 1-279

Driver Thread Class Structure

Class Definition

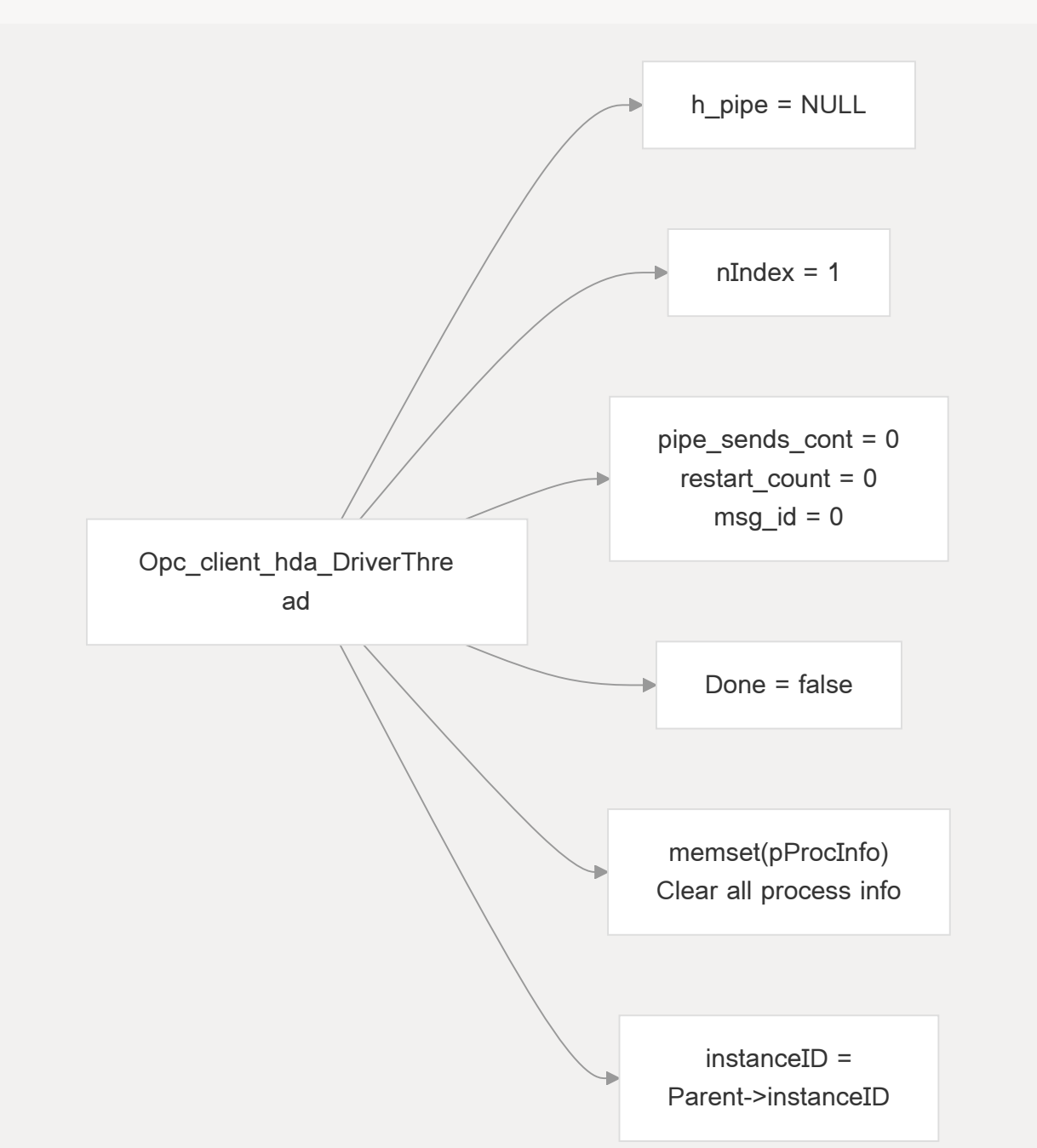
The `Opc_client_hda_DriverThread` class inherits from `DriverThread` and manages the lifecycle of the OPC HDA child process.

Key Member Variables:

Member	Type	Purpose
`pProcInfo`	`PROCESS_INFORMATION[nMaxProcCount]`	Windows process information for child processes
`pCommandLine`	`char[nBufferSize+1]`	Command line string for launching child process
`pWorkingDir`	`char[nBufferSize+1]`	Working directory for child process
`nIndex`	`int`	Index into pProcInfo array (always 1 for single process)
`pipe_name`	`char[150]`	Named pipe path for IPC
`h_pipe`	`HANDLE`	Windows handle to named pipe
`pipe_sends_cont`	`int`	Counter for periodic keepalive messages
`restart_count`	`int`	Counter for periodic health checks
`msg_id`	`int`	Message ID for iec_item packets
`instanceID`	`int`	Driver instance identifier
`Done`	`bool`	Thread completion flag

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.h` 23-55

Constructor and Initialization


Opc_client_hda_DriverThread
h_pipe = NULL
nIndex = 1
pipe_sends_cont = 0 restart_count = 0 msg_id = 0
Done = false
memset(pProcInfo) Clear all process info
instanceID = Parent->instanceID

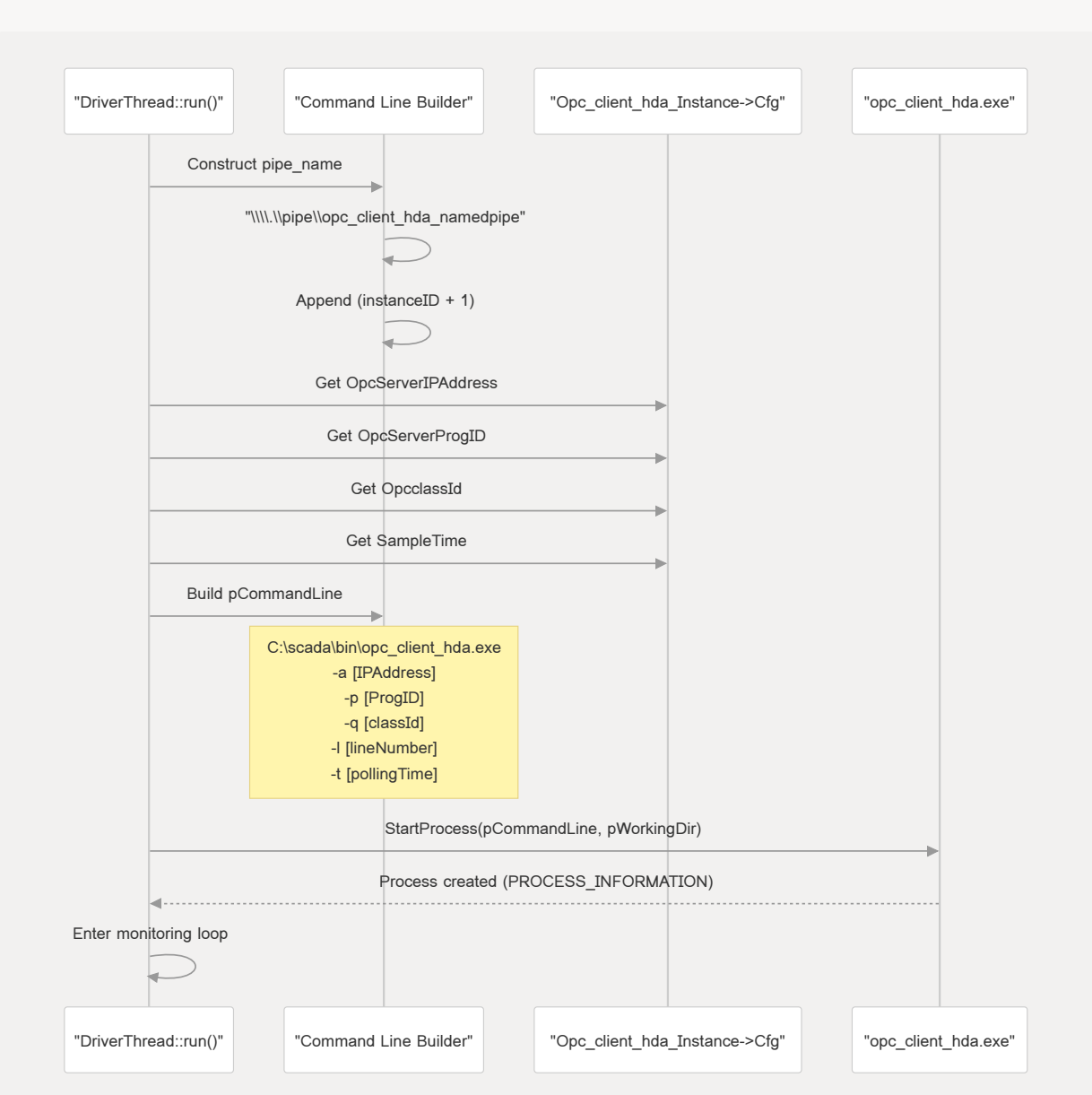
Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.h` 41-55

Child Process Management

Process Startup Sequence

The `run()` method orchestrates the child process lifecycle. The startup sequence constructs command line arguments from configuration and launches the OPC HDA executable.

Command Line Construction:


"DriverThread::run()"
"Command Line Builder"
"Opc_client_hda_Instance->Cfg"
"opc_client_hda.exe"
Construct pipe_name
"\\\\.\\pipe\\opc_client_hda_namedpipe"
Append (instanceID + 1)
Get OpcServerIPAddress
Get OpcServerProgID
Get OpcclassId
Get SampleTime
Build pCommandLine
C:\scada\bin\opc_client_hda.exe -a [IPAddress] -p [ProgID] -q [classId] -l [lineNumber] -t [pollingTime]
StartProcess(pCommandLine, pWorkingDir)
Process created (PROCESS_INFORMATION)
Enter monitoring loop

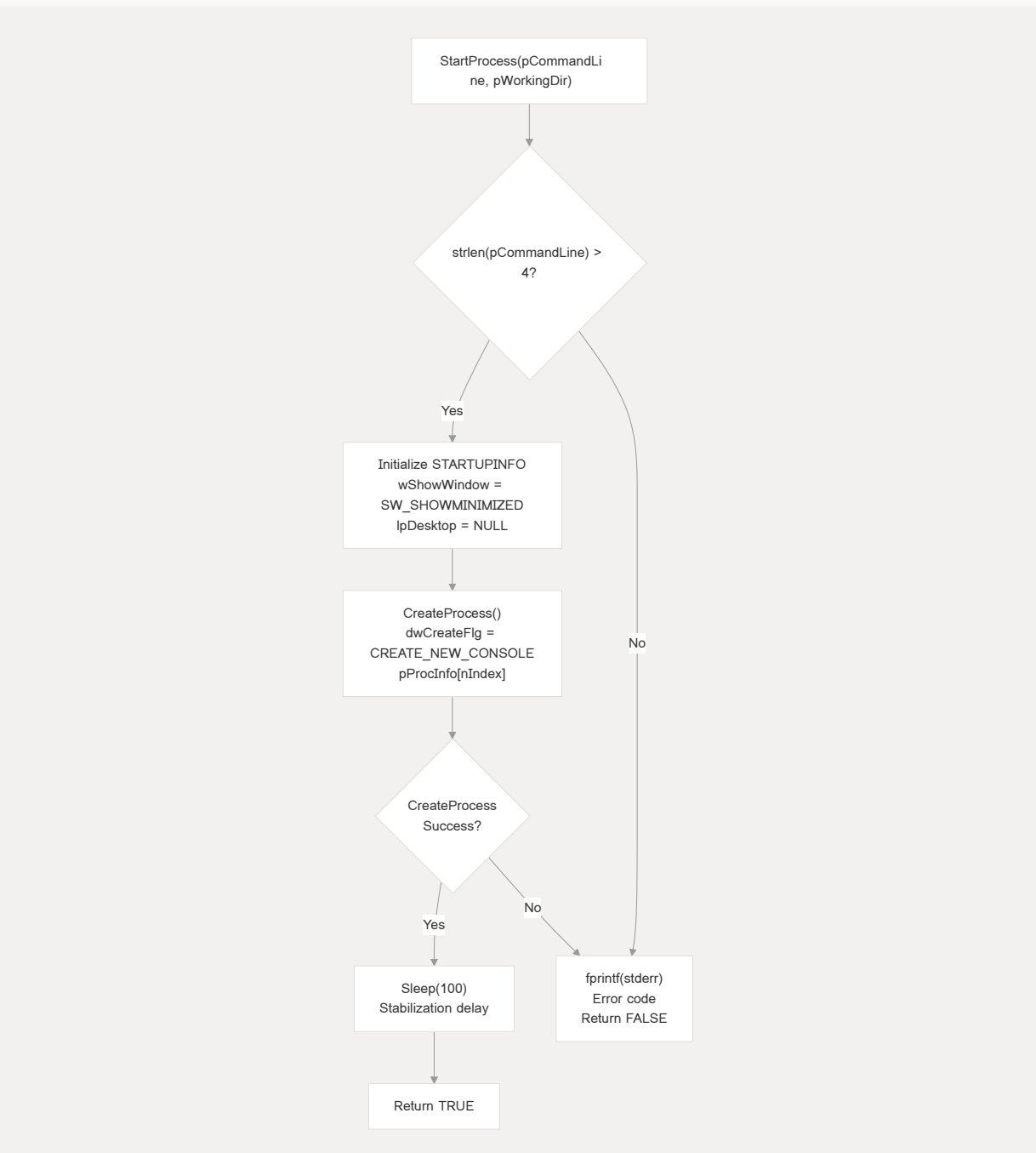
Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 22-135

StartProcess Implementation

The `StartProcess()` method creates the child process using the Windows `CreateProcess()` API with specific configuration for SCADA operation.

Key Configuration:

Setting	Value	Purpose
Window Mode	`SW_SHOWMINIMIZED`	Shows minimized console window for user interface
Desktop	`NULL` (default desktop)	Runs on current desktop
Creation Flags	`CREATE_NEW_CONSOLE`	Separate console for independent operation
Working Directory	`C:\scada\bin`	Standard IndigoSCADA binary directory


StartProcess(pCommandLine, pWorkingDir)
strlen(pCommandLine) > 4?
Yes
Initialize STARTUPINFO wShowWindow = SW_SHOWMINIMIZED lpDesktop = NULL
CreateProcess() dwCreateFlg = CREATE_NEW_CONSOLE pProcInfo[nIndex]
No
CreateProcess Success?
Yes
No
Sleep(100) Stabilization delay
fprintf(stderr) Error code Return FALSE
Return TRUE

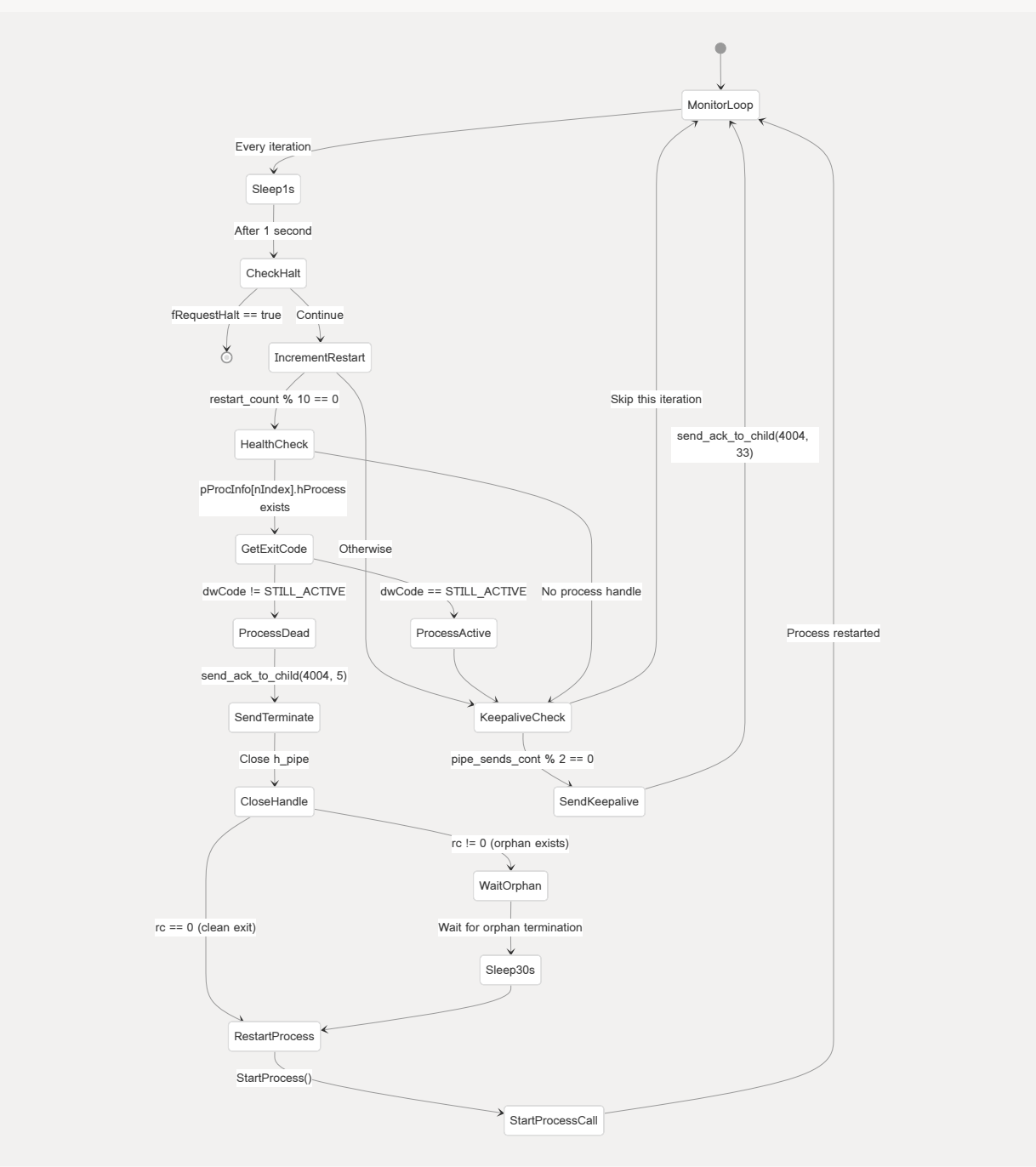
Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 138-195

Health Monitoring and Keepalive

Monitoring Loop

The driver thread continuously monitors child process health and sends periodic keepalive messages to maintain communication.

Monitoring Activities:


MonitorLoop
Every iteration
Sleep1s
After 1 second
CheckHalt
fRequestHalt == true
Continue
IncrementRestart
restart_count % 10 == 0
HealthCheck
pProcInfo[nIndex].hProcess exists
GetExitCode
Otherwise
dwCode != STILL_ACTIVE
dwCode == STILL_ACTIVE
No process handle
Skip this iteration
send_ack_to_child(4004, 33)
Process restarted
ProcessDead
ProcessActive
send_ack_to_child(4004, 5)
SendTerminate
KeepaliveCheck
Close h_pipe
pipe_sends_cont % 2 == 0
CloseHandle
SendKeepalive
rc != 0 (orphan exists)
WaitOrphan
rc == 0 (clean exit)
Wait for orphan termination
Sleep30s
RestartProcess
StartProcess()
StartProcessCall

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 65-130

Keepalive Mechanism

Keepalive messages are sent every 2 seconds (every other 1-second loop iteration) to verify communication channel integrity.

Message Format:

The keepalive uses the standard `iec_item` structure with type 37:

Field	Value	Purpose
`iec_type`	37	Keep-alive message type
`iec_obj.ioa`	4004	Fixed address for control messages
`iec_obj.o.type37.counter`	33 (keepalive) or 5 (terminate)	Command code
`msg_id`	Incrementing counter	Message sequence tracking
`checksum`	CRC	Data integrity verification

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 220-232

Named Pipe Communication

Pipe Establishment

The driver thread writes to a named pipe that the child process creates and listens on. The pipe name is constructed using the instance ID to support multiple concurrent driver instances.

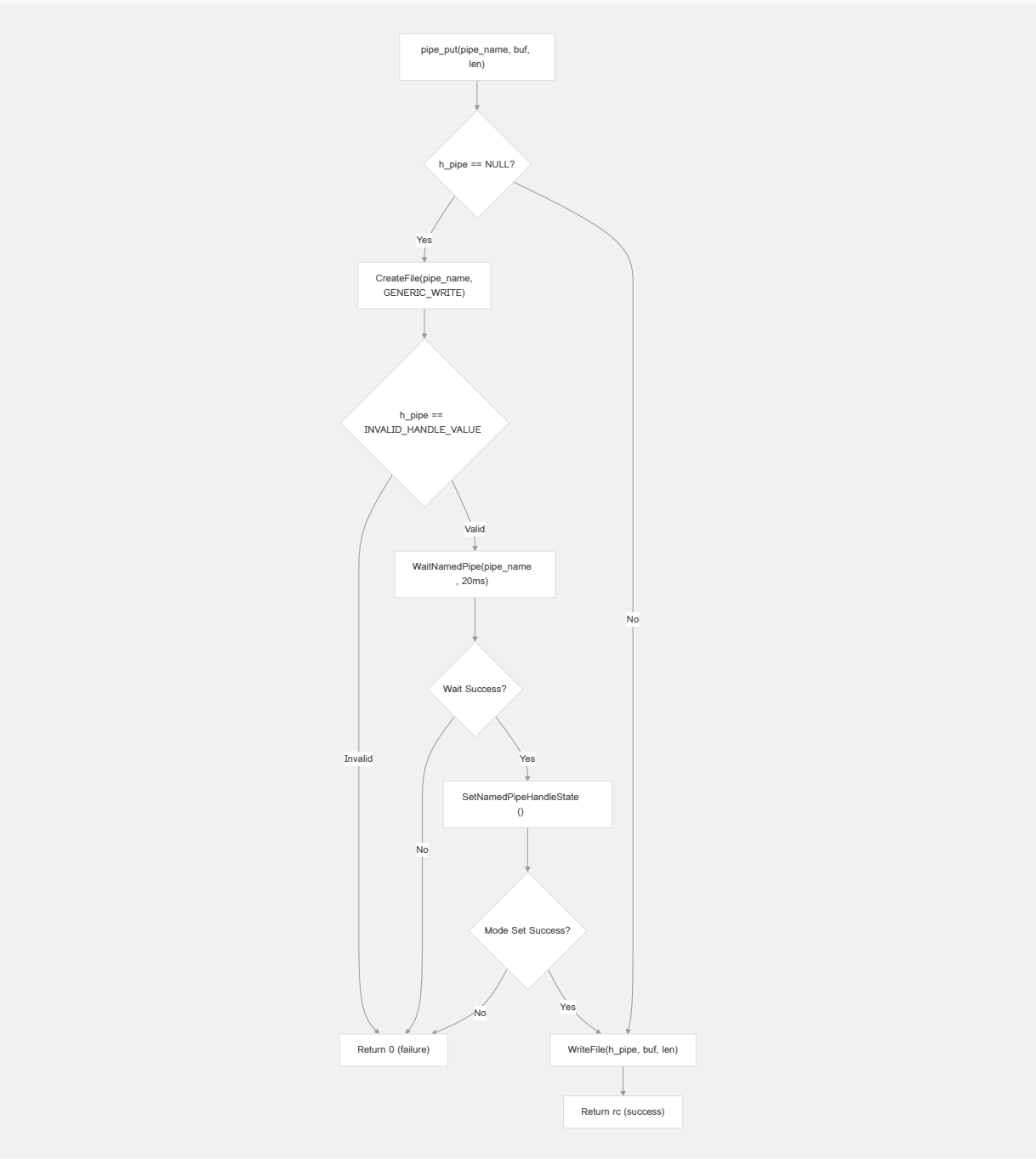
Pipe Naming Convention:

```
\\\\.\\pipe\\opc_client_hda_namedpipe{N}
```

Where `{N}` is the instance ID plus 1 (e.g., `opc_client_hda_namedpipe1` for instance 0).

Pipe Write Operation

The `pipe_put()` method handles lazy connection and message transmission over the named pipe.


pipe_put(pipe_name, buf, len)
h_pipe == NULL?
Yes
CreateFile(pipe_name, GENERIC_WRITE)
h_pipe == INVALID_HANDLE_VALUE
Valid
WaitNamedPipe(pipe_name, 20ms)
No
Wait Success?
Invalid
Yes
SetNamedPipeHandleState()
No
Mode Set Success?
No
Yes
Return 0 (failure)
WriteFile(h_pipe, buf, len)
Return rc (success)

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 234-269

Process Termination

Graceful Shutdown

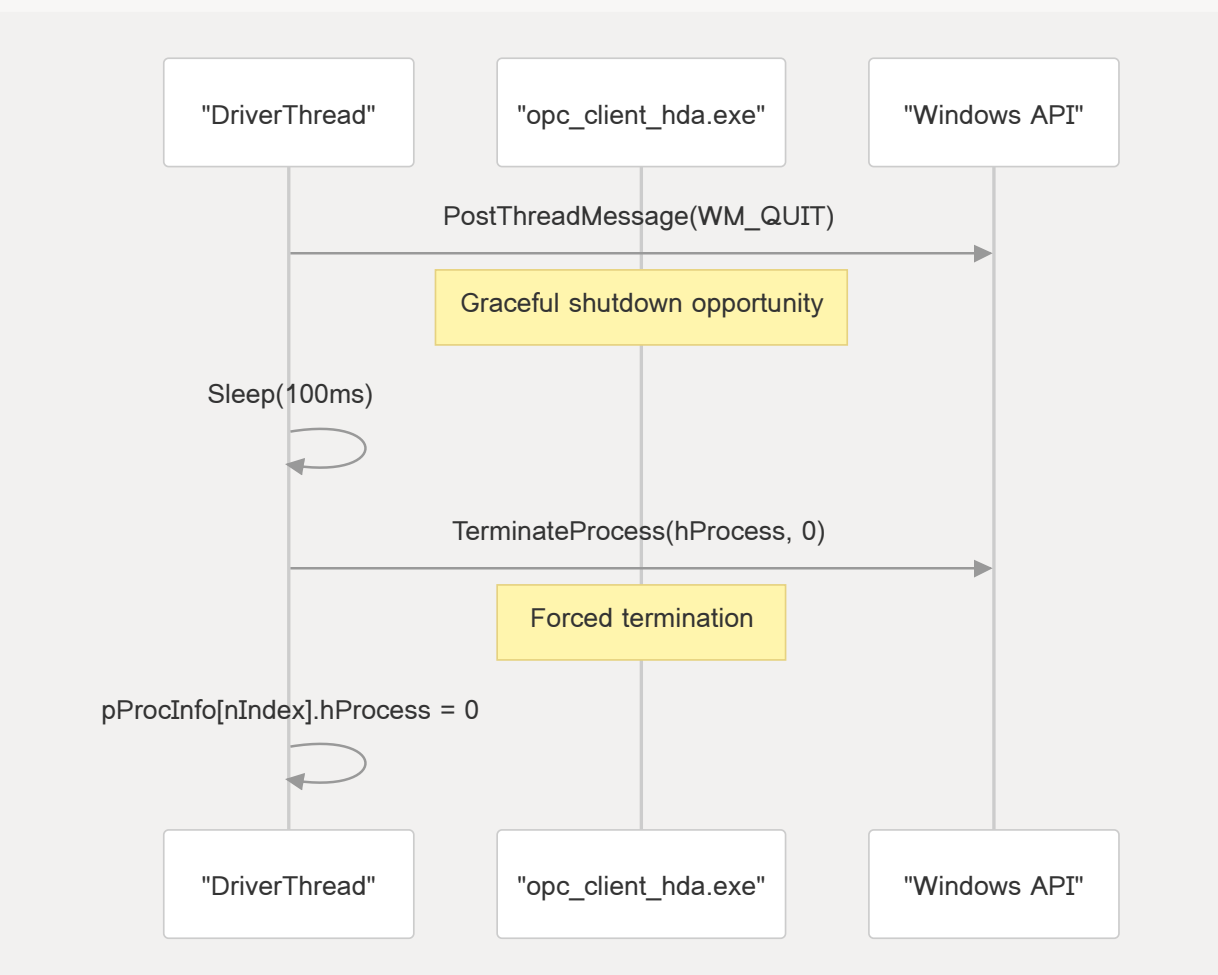
The `TerminateProtocol()` method provides graceful shutdown of the driver thread and child process.

Termination Sequence:

1. Call `EndProcess(nIndex)` to stop child process
2. Set `fRequestHalt = true` to signal thread exit
3. Thread exits monitoring loop and completes

EndProcess Implementation

The `EndProcess()` method attempts graceful termination before forcing process kill.


"DriverThread"
"opc_client_hda.exe"
"Windows API"
PostThreadMessage(WM_QUIT)
Graceful shutdown opportunity
Sleep(100ms)
TerminateProcess(hProcess, 0)
Forced termination
pProcInfo[nIndex].hProcess = 0

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 204-218 `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 271-278

Configuration Parameters

Driver Instance Configuration

The OPC HDA driver requires configuration parameters stored in the `Opc_client_hda_Instance` configuration structure:

Parameter	Type	Purpose	Command Line Flag
`OpcServerIPAddress`	`char[]`	IP address or hostname of OPC HDA server	`-a`
`OpcServerProgID`	`char[]`	OPC server ProgID for server identification	`-p`
`OpcclassId`	`char[]`	OPC server CLSID (class identifier)	`-q`
`SampleTime`	`int`	Polling interval in milliseconds	`-t`

Note: The instance ID (`-l` flag) is automatically derived from the driver instance index and is not stored in the configuration structure.

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 38-50

Comparison with Sibling Drivers

The OPC HDA driver thread follows an identical pattern to OPC DA and OPC AE drivers. The only differences are:

Aspect	OPC DA	OPC AE	OPC HDA
Executable	`opc_client_da.exe`	`opc_client_ae.exe`	`opc_client_hda.exe`
Pipe Name	`opc_client_da_namedpipe{N}`	`opc_client_ae_namedpipe{N}`	`opc_client_hda_namedpipe{N}`
Primary Purpose	Real-time data access	Alarm/event subscription	Historical data retrieval
Class Name	`Opc_client_da_DriverThread`	`Opc_client_ae_DriverThread`	`Opc_client_hda_DriverThread`

All three share the same:

- Child process management logic
- Health monitoring and automatic restart
- Named pipe keepalive mechanism
- Process termination procedures

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 1-296 `src/drivers/opc_client_ae/opc_client_aedriverthread.cpp` 1-280 `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 1-279

Error Handling and Recovery

Automatic Restart Logic

The driver thread automatically detects and recovers from child process crashes:

Crash Detection:

- Every 10 seconds, calls `GetExitCodeProcess()` to check child process status
- If exit code is not `STILL_ACTIVE`, process has terminated

Recovery Procedure:

1. **Terminate Orphan:** Send terminate message (type 37, counter 5) via pipe
2. **Wait Period:** If orphan exists (`rc != 0`), wait 30 seconds for cleanup
3. **Close Pipe:** Close pipe handle to prepare for fresh connection
4. **Restart:** Call `StartProcess()` to launch new child process instance
5. **Resume Monitoring:** Continue health monitoring loop

Error Logging:

All error conditions are logged to `stderr` with descriptive messages:

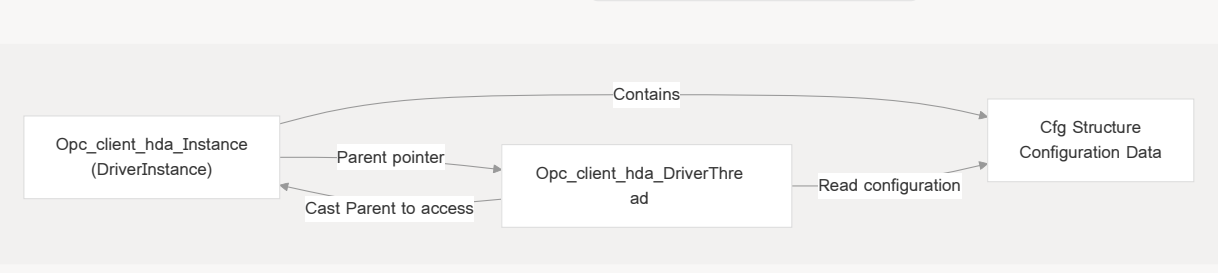
- Process start failures: `"Failed to start program '%s', error code = %d"`
- Process restart: `"Restarted process %s"`
- Restart failures: `"Failed to restart process %s"`
- API failures: `"GetExitCodeProcess failed, error code = %d"`

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 76-121

Integration Points

Parent Instance Relationship

The driver thread is tightly coupled to its parent `Opc_client_hda_Instance`:


Contains
Opc_client_hda_Instance (DriverInstance)
Parent pointer
Cast Parent to access
Opc_client_hda_DriverThread
Read configuration
Cfg Structure Configuration Data

Access Pattern:

```
instanceID = ((Opc_client_hda_Instance*)Parent)->instanceID;
((Opc_client_hda_Instance*)Parent)->Cfg.OpcServerIPAddress
((Opc_client_hda_Instance*)Parent)->Cfg.SampleTime
```

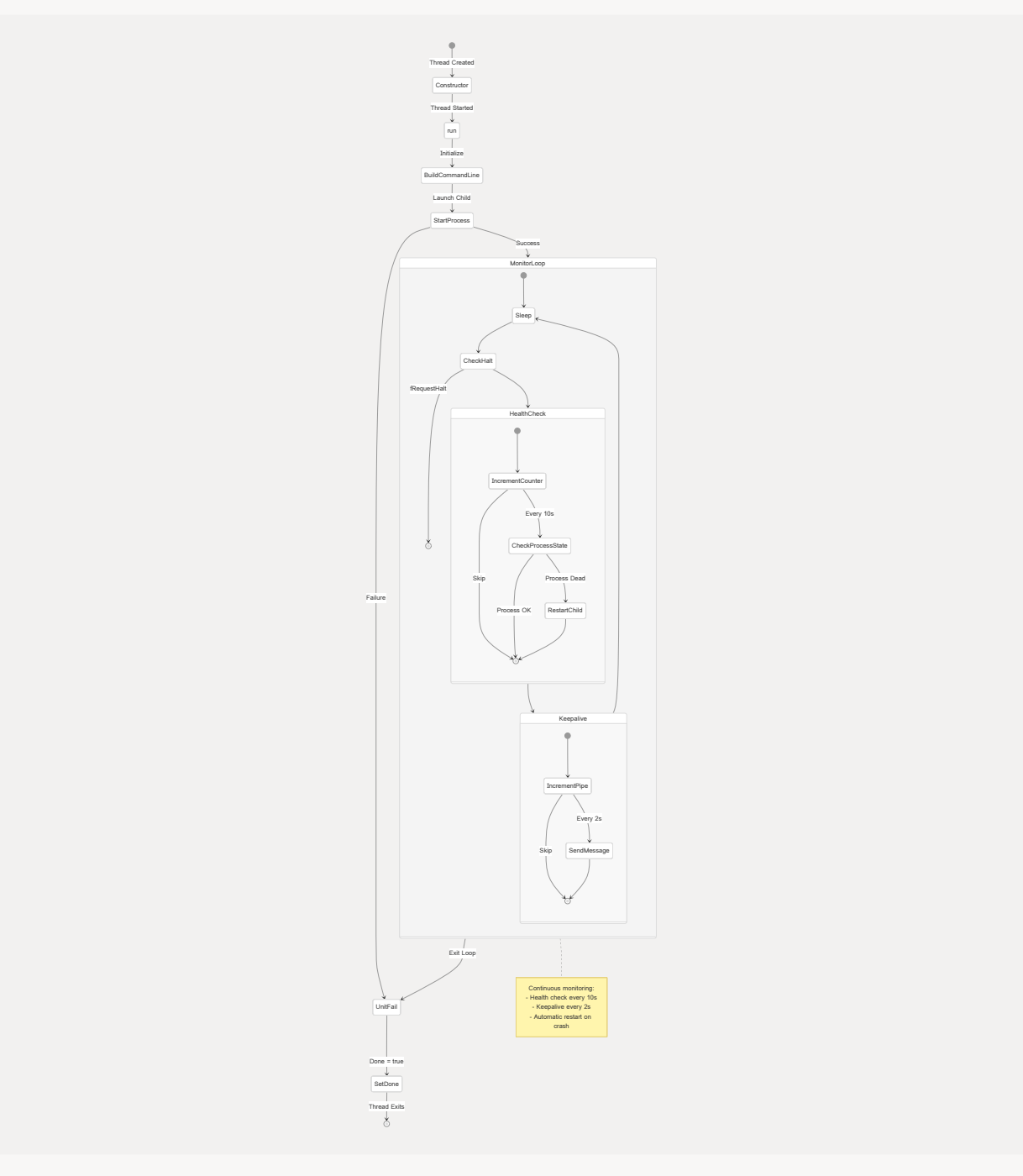
Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.h` 47 `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 38-50

Middleware Integration

While this document covers the driver thread that manages the child process, the actual OPC HDA protocol implementation and middleware integration (ORTE/RTPS publishing) occurs within the `opc_client_hda.exe` child process, which is not included in the provided source files.

Thread Lifecycle

Complete Lifecycle Diagram


Thread Created
Constructor
Thread Started
run
Initialize
BuildCommandLine
Launch Child
StartProcess
Success
MonitorLoop
Sleep
CheckHalt
fRequestHalt
HealthCheck
IncrementCounter
Every 10s
CheckProcessState
Skip
Process Dead
Process OK
RestartChild
Failure
Keepalive
IncrementPipe
Every 2s
Skip
SendMessage
Exit Loop
UntilFail
Continuous monitoring: - Health check every 10s - Keepalive every 2s - Automatic restart on crash
Done = true
SetDone
Thread Exits

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 22-135

Constants and Limits

Constant	Value	Purpose
`nMaxProcCount`	127	Maximum number of child processes (only 1 used)
`nBufferSize`	500	Buffer size for command line and working directory strings
Health Check Period	10 seconds	Interval for process liveness check
Keepalive Period	2 seconds	Interval for pipe keepalive messages
Orphan Wait Time	30 seconds	Wait time before restarting after detecting orphan

Process Start Delay	100 milliseconds	Stabilization time after CreateProcess
Graceful Termination Delay	100 milliseconds	Time allowed for graceful shutdown before forced kill
Pipe Wait Timeout	20 milliseconds	Timeout for WaitNamedPipe operation

Sources: `src/drivers/opc_client_hda/opc_client_hdadriverthread.h` 20-21 `src/drivers/opc_client_hda/opc_client_hdadriverthread.cpp` 76-127

OPC Classic Clients

› Relevant source files

Purpose and Scope

This page provides an overview of the three OPC Classic client driver implementations in IndigoSCADA: OPC Data Access (DA), OPC Alarms & Events (AE), and OPC Historical Data Access (HDA). These drivers enable IndigoSCADA to acquire real-time data, alarms, and historical information from Windows-based OPC servers using the OPC Classic COM/DCOM interfaces.

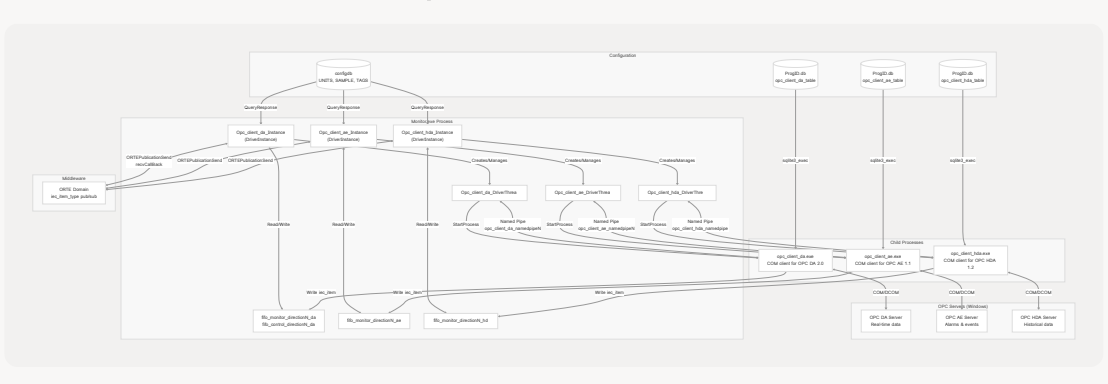
All three OPC client drivers follow the common protocol driver architecture pattern (see Protocol Driver Architecture) with child process execution and middleware integration. For detailed information on individual OPC specifications:

- OPC DA client details: OPC DA Client
- OPC AE client details: OPC AE Client
- OPC HDA client details: OPC HDA Client

Architecture Overview

The OPC Classic client drivers implement a two-process architecture where the parent process (`monitor.exe`) manages a child process that handles the COM/DCOM complexity of communicating with OPC servers.

OPC Classic Client Driver Components



Sources: `src/drivers/opc_client_da/opc_client_da_instance.h` 60-230
`src/drivers/opc_client_da/opc_client_dadriverthread.h` 22-74
`src/drivers/opc_client_ae/opc_client_aedriverthread.h` 24-76
`src/drivers/opc_client_hda/opc_client_hdadriverthread.h` 23-74

Common Components

All three OPC Classic client drivers share a common structural pattern with minor variations.

DriverInstance Classes

Driver Type	Instance Class	Header File	Key Responsibilities
OPC DA	`Opc_client_da_Instance`	`opc_client_da_instance.h` 60	Real-time data acquisition, command execution, IOA-to-ItemID mapping
OPC AE	`Opc_client_ae_Instance`	opc_client_ae_instance.h	Alarm/event subscription, event filtering, acknowledgment handling
OPC HDA	`Opc_client_hda_Instance`	opc_client_hda_instance.h	Historical data queries, time-range selection, aggregate retrieval

All instance classes extend `DriverInstance` and implement:

- Configuration loading from `configdb` via `QueryResponse()` method
- State machine execution in `Tick()` method (100ms intervals)
- FIFO-based communication with child processes via `get_items_from_local_fifo()`
- Middleware integration through ORTE publication/subscription
- IOA-to-name mapping using `IOANameMap ioa_name_map`

Sources: `src/drivers/opc_client_da/opc_client_da_instance.h` 60-230
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 1-11

DriverThread Classes

Driver Type	Thread Class	Child Executable	Named Pipe Pattern
OPC DA	`Opc_client_da_DriverThread`	`opc_client_da.exe`	`\\\\.\\pipe\\opc_client_da_namedpipeN`
OPC AE	`Opc_client_ae_DriverThread`	`opc_client_ae.exe`	`\\\\.\\pipe\\opc_client_ae_namedpipeN`
OPC HDA	`Opc_client_hda_DriverThread`	`opc_client_hda.exe`	`\\\\.\\pipe\\opc_client_hda_namedpipeN`

Common thread responsibilities (defined in header files):

- `PROCESS_INFORMATION pProcInfo[nMaxProcCount]` - Process handle tracking
- `char pCommandLine[nBufferSize+1]` - Command line construction
- `char pipe_name[150]` - Named pipe identifier
- `HANDLE h_pipe` - Pipe handle for IPC
- `int restart_count` - Child process health monitoring counter
- `StartProcess()` - Launch child with COM apartment model
- `EndProcess()` - Graceful/forceful termination
- `send_ack_to_child()` - Keepalive packet transmission
- `pipe_put()` - Named pipe write operations

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.h` 22-74
`src/drivers/opc_client_ae/opc_client_aedriverthread.h` 24-76
`src/drivers/opc_client_hda/opc_client_hdadriverthread.h` 23-74

Configuration Structure

Unit Properties (PROPS Table)

Each OPC client driver instance stores configuration in the `PROPS` table with key format:

```
SKEY = '<unit_name>'
IKEY = '(default)' or '<recipe_name>'
DVAL = '<space-separated configuration values>'
```

Configuration parsing (from `opc_client_da_instance.cpp` 276-312):

```
QString s = UndoEscapeSQLText(GetConfigureDb()->GetString("DVAL"));
QTextIStream is(&s);
is >> OpcItems;              // Number of OPC items
is >> Cfg.SampleTime;        // Polling interval (milliseconds)
is >> Cfg.OpcServerProgID;   // OPC Server ProgID (e.g., "Matrikon.OPC.Simulation")
is >> Cfg.OpcServerIPAddress; // Remote server IP or "localhost"
is >> Cfg.OpcclassId;        // OPC Server CLSID (GUID)
```

SQLite Configuration Database

Each OPC client driver loads protocol-specific configuration from SQLite at startup:

OPC DA Configuration (`opc_client_da_instance.cpp` 661-761):

- Database file: `<OpcServerProgID>.db` in project directory
- Table: `opc_client_da_table`
- Columns:
  - `opc_server_item_id` - OPC item path (e.g., "Device1.Tag1")
  - `ioa_control_center` - IndigoSCADA IOA (Information Object Address)
  - `iec_type` - IEC 60870-5-104 type mapping (M_ME_TF_1, M_SP_TB_1, etc.)
  - `readable` - Read permission flag
  - `writeable` - Write permission flag
  - `HiHiLimit` / `LoLoLimit` - Measurement range
  - `opc_type` - OPC data type string

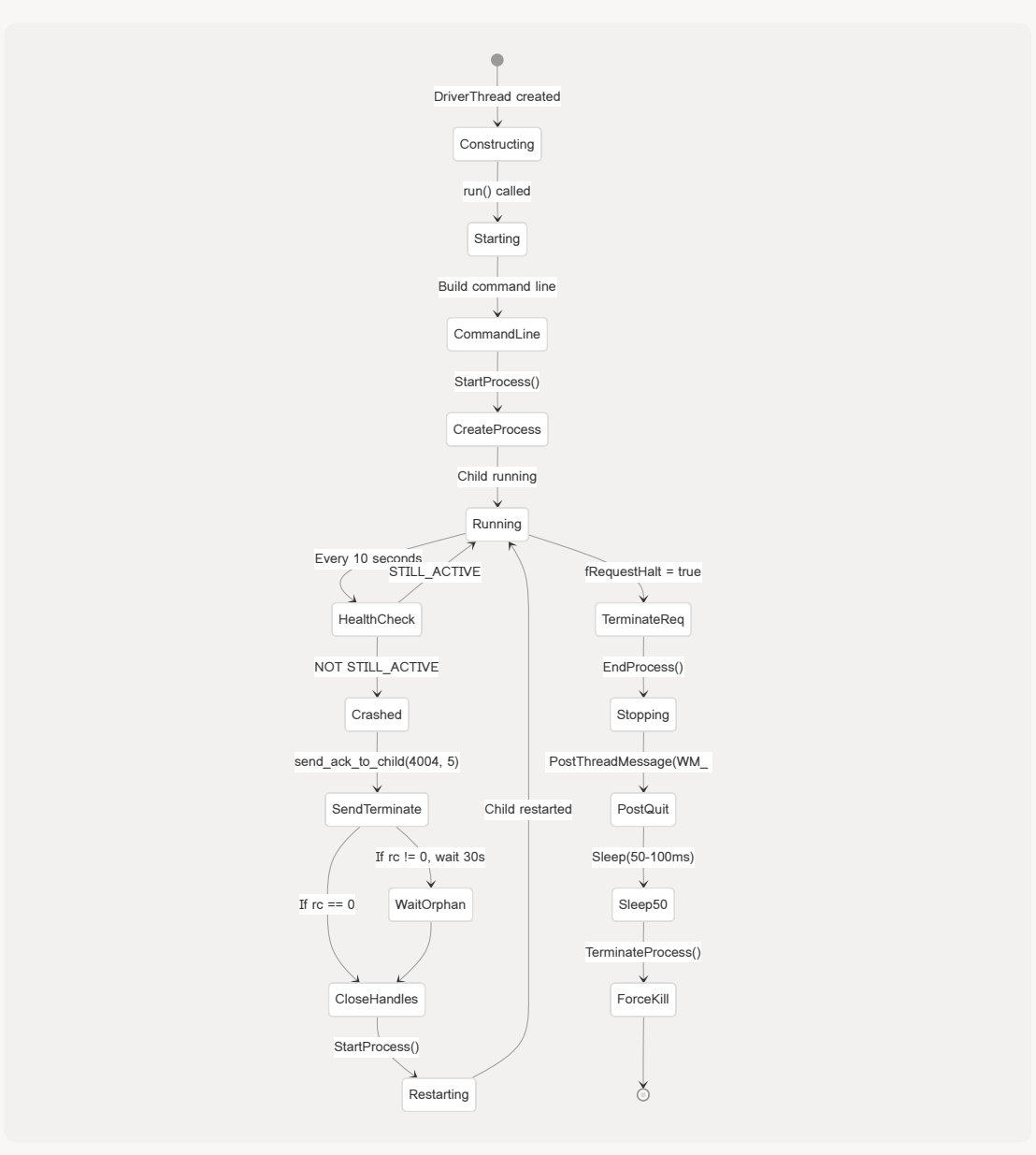
Database callback function (`opc_client_da_instance.cpp` 521-635):

```
static int db_callback(void *NotUsed, int argc, char **argv, char **azColName)
{
    gl_Config_db[gl_row_counter].ioa_control_center = atoi(argv[1]);
    strcpy(gl_Config_db[gl_row_counter].spname, argv[0]);
    // Parse IEC type string to enum
    if(strcmp(argv[2], "M_ME_TF_1") == 0)
        gl_Config_db[gl_row_counter].io_list_iec_type = M_ME_TF_1;
    // ... additional type mappings
    gl_row_counter++;
}
```

Sources: `src/drivers/opc_client_da/opc_client_da_instance.cpp` 276-312
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 661-761
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 521-635

Child Process Management

Child Process Lifecycle



Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 22-135
`src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 204-218

Command Line Construction

From `opc_client_da/opc_client_dadriverthread.cpp` 28-50 :

```
strcpy(pipe_name, "\\\\.\\pipe\\opc_client_da_namedpipe");
itoa(instanceID + 1, line_number, 10);
strcat(pipe_name, line_number);

strcpy(pCommandLine, GetScadaHomeDirectory());
strcat(pCommandLine, "\\bin\\opc_client_da.exe -a ");
strcat(pCommandLine, Cfg.OpcServerIPAddress);
strcat(pCommandLine, " -p ");
strcat(pCommandLine, Cfg.OpcServerProgID);
strcat(pCommandLine, " -q ");
strcat(pCommandLine, Cfg.OpcclassId);
strcat(pCommandLine, " -l ");
strcat(pCommandLine, line_number);
strcat(pCommandLine, " -t ");
strcat(pCommandLine, polling_time);
```

Command line parameters:

- `-a` : OPC server IP address or hostname
- `-p` : OPC server ProgID
- `-q` : OPC server CLSID (Class ID)
- `-l` : Line number (instance ID + 1)
- `-t` : Polling time in milliseconds

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 28-60

Process Creation

From `opc_client_da/opc_client_dadriverthread.cpp` 138-195 :

```
bool StartProcess(char* pCommandLine, char* pWorkingDir)
{
    STARTUPINFO startUpInfo = {
        sizeof(STARTUPINFO),NULL,"",NULL,0,0,0,0,0,0,0,
        STARTF_USESHOWWINDOW,0,0,NULL,0,0,0
    };

    startUpInfo.wShowWindow = SW_SHOWMINIMIZED;  // Minimize child window
    startUpInfo.lpDesktop = NULL;  // Use current desktop

    DWORD dwCreateFlg = CREATE_NEW_CONSOLE;  // Separate console window

    if(CreateProcess(NULL, pCommandLine, NULL, NULL, TRUE,
                     dwCreateFlg, NULL, pWorkingDir,
                     &startUpInfo, pProcInfo + nIndex))
    {
        Sleep(100);
        return TRUE;
    }
    return FALSE;
}
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 138-195

Health Monitoring and Restart Logic

From `opc_client_da/opc_client_dadriverthread.cpp` 74-121 :

```
restart_count++;

if(restart_count % 10 == 0)  // Check every 10 seconds
{
    if(pProcInfo[nIndex].hProcess)
    {
        DWORD dwCode;
        if(::GetExitCodeProcess(pProcInfo[nIndex].hProcess, &dwCode))
        {
            if(dwCode != STILL_ACTIVE)
            {
                pProcInfo[nIndex].hProcess = 0;

                // Attempt graceful shutdown of orphan
                rc = send_ack_to_child(4004, 5, pipe_name);

                if(rc != 0)  // Orphan still exists
                {
                    CloseHandle(h_pipe);
                    h_pipe = NULL;
                    Sleep(30000);  // Wait for orphan to exit
                }

                // Restart child process
                if(StartProcess(pCommandLine, pWorkingDir))
                {
                    fprintf(stderr, "Restarted process %s\n", pCommandLine);
                }
            }
        }
    }
}
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 74-121

Communication Mechanisms

Named Pipe IPC

Named pipes provide bidirectional communication between parent and child processes:

Pipe Naming Convention:

- OPC DA: `\\\\.\\pipe\\opc_client_da_namedpipe<N>` where N = instanceID + 1
- OPC AE: `\\\\.\\pipe\\opc_client_ae_namedpipe<N>`
- OPC HDA: `\\\\.\\pipe\\opc_client_hda_namedpipe<N>`

Pipe Write Implementation (`opc_client_da/opc_client_dadriverthread.cpp` 234-269):

```
int pipe_put(char* pipe_name, char *buf, int len)
{
    if(h_pipe == NULL)
    {
        h_pipe = CreateFile(pipe_name, GENERIC_WRITE, 0, NULL,
                            OPEN_EXISTING, 0, NULL);
        if(h_pipe == INVALID_HANDLE_VALUE)
            return 0;

        if(!WaitNamedPipe(pipe_name, 20))
            return 0;

        mode = PIPE_READMODE_MESSAGE;
        SetNamedPipeHandleState(h_pipe, &mode, NULL, NULL);
    }

    rc = WriteFile(h_pipe, buf, len, &written, NULL);
    return rc;
}
```

Keepalive Packet (`opc_client_da/opc_client_dadriverthread.cpp` 123-128):

```
pipe_sends_cont++;
if(pipe_sends_cont % 2 == 0)  // Every 2 seconds
{
    rc = send_ack_to_child(4004, 33, pipe_name);
}
```

The keepalive packet structure (`opc_client_da/opc_client_dadriverthread.cpp` 220-232):

```
int send_ack_to_child(int address, int data, char* pipeName)
{
    struct iec_item item_to_send;
    memset(&item_to_send, 0x00, sizeof(struct iec_item));
    item_to_send.iec_type = 37;  // M_IT_TB_1
    item_to_send.iec_obj.ioa = address;
    item_to_send.iec_obj.o.type37.counter = data;
    item_to_send.msg_id = msg_id++;
    item_to_send.checksum = clearCrc((unsigned char *)&item_to_send,
                                     sizeof(struct iec_item));
    return pipe_put(pipeName, (char *)&item_to_send, sizeof(struct iec_item));
}
```

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 220-269

FIFO Buffers

Local FIFO buffers provide efficient message passing between middleware and driver instance:

FIFO Naming (`opc_client_da/opc_client_da_instance.h` 154-166):

```
char fifo_monitor_name[150];
itoa(instance_id + 1, str_instance_id, 10);
strcpy(fifo_monitor_name, "fifo_monitor_direction");
strcat(fifo_monitor_name, str_instance_id);
strcat(fifo_monitor_name, "da");  // "ae" for AE, "hda" for HDA

char fifo_control_name[150];
strcpy(fifo_control_name, "fifo_control_direction");
strcat(fifo_control_name, str_instance_id);
strcat(fifo_control_name, "da");
```

FIFO Read Operation (`opc_client_da/opc_client_da_instance.cpp` 834-892):

```
void get_items_from_local_fifo(void)
{
    unsigned char buf[sizeof(struct iec_item)];
    const unsigned wait_limit_ms = 1;
    struct iec_item* p_item;

    for(int i = 0;
        (len = fifo_get(fifo_monitor_direction, (char*)buf,
                        sizeof(struct iec_item), wait_limit_ms)) >= 0;
        i += 1)
    {
        p_item = (struct iec_item*)buf;

        // Checksum validation
        unsigned char rc = clearCrc((unsigned char *)buf,
                                    sizeof(struct iec_item));
        if(rc != 0) ExitProcess(1);

        // Process based on IEC type
        switch(p_item->iec_type) {
            case M_ME_TF_1: // Floating point measurement
            case M_SP_TB_1: // Single point with time
            // ... handle each type
        }
    }
}
```

Sources: `src/drivers/opc_client_da/opc_client_da_instance.h` 154-166
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 834-892

ORTE Middleware Integration

ORTE (Open Real-Time Ethernet) provides publish/subscribe communication with the global monitor direction:

ORTE Callback Registration (`opc_client_da/opc_client_da_instance.cpp` 89-113):

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance,
                  void *recvCallBackParam)
{
    Opc_client_da_Instance * cl = (Opc_client_da_Instance*)recvCallBackParam;
    iec_item_type *item1 = (iec_item_type*)vinstance;

    switch (info->status)
    {
        case NEW_DATA:
        {
            struct iec_item item2;
            rebuild_iec_item_message(&item2, item1);
            fifo_put(cl->fifo_monitor_direction, (char *)&item2,
                    sizeof(struct iec_item));
        }
        break;
    }
}
```

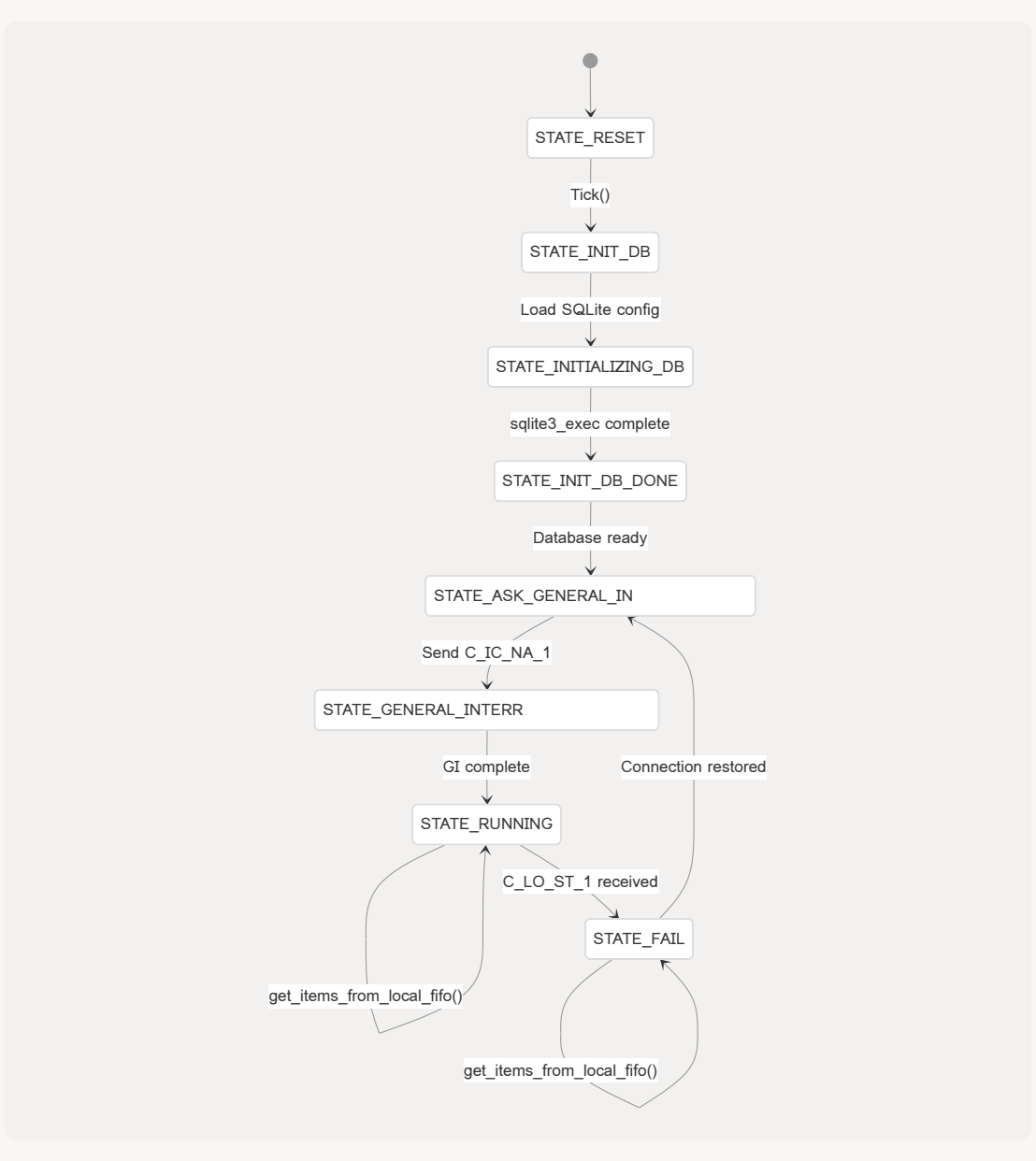
ORTE Publication (`opc_client_da/opc_client_da_instance.cpp` 813-828):

```
if(global_publisher)
{
    // Prepare publication data
    memset(&global_instanceSend, 0x00, sizeof(iec_item_type));
    global_instanceSend.iec_type = p_item->iec_type;
    memcpy(&(global_instanceSend.iec_obj), &(p_item->iec_obj),
          sizeof(struct iec_object));
    global_instanceSend.cause = p_item->cause;
    global_instanceSend.msg_id = p_item->msg_id;
    global_instanceSend.ioa_control_center = p_item->ioa_control_center;
    global_instanceSend.casdu = p_item->casdu;
    global_instanceSend.is_neg = p_item->is_neg;
    global_instanceSend.checksum = p_item->checksum;

    ORTEPublicationSend(global_publisher);
}
```

Sources: `src/drivers/opc_client_da/opc_client_da_instance.cpp` 89-113
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 813-828

State Machine

All OPC client drivers implement a common state machine pattern in their `Tick()` method:



State Definitions (`opc_client_da/opc_client_da_instance.h` 119-129):

```
enum {
    STATE_IDLE = 0,
    STATE_RESET,
    STATE_INIT_DB,
    STATE_INIT_DB_DONE,
    STATE_ASK_GENERAL_INTERROGATION,
    STATE_GENERAL_INTERROGATION_DONE,
    STATE_FAIL,
    STATE_RUNNING
};
```

General Interrogation Command (`opc_client_da/opc_client_da_instance.cpp` 773-809):

```
case STATE_ASK_GENERAL_INTERROGATION:
{
    struct iec_item item_to_send;
    memset(&item_to_send, 0x00, sizeof(struct iec_item));
    item_to_send.iec_type = C_IC_NA_1;  // General interrogation command
    item_to_send.iec_obj.ioa = 0;
    item_to_send.iec_obj.o.type100.qoi = 1;  // Station interrogation
    item_to_send.msg_id = msg_sent_in_control_direction++;
    item_to_send.checksum = clearCrc((unsigned char *)&item_to_send,
                                     sizeof(struct iec_item));

    ORTEPublicationSend(publisher);
    State = STATE_GENERAL_INTERROGATION_DONE;
}
```

Sources: `src/drivers/opc_client_da/opc_client_da_instance.h` 119-129
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 646-831

Data Type Mapping

OPC Classic clients convert OPC data types to IEC 60870-5-104 types for uniform representation:

OPC DA Data Type Mapping

OPC Data Type	IEC 60870-5-104 Type	Description	Code Reference
`VT_R4` (float)	`M_ME_TF_1`	Short floating point measurement with time	`opc_client_da_instance.cpp` 554-556
`VT_BOOL`	`M_SP_TB_1`	Single point with time	`opc_client_da_instance.cpp` 557-559
`VT_I2` (2-bit)	`M_DP_TB_1`	Double point with time	`opc_client_da_instance.cpp` 560-562
`VT_I4` (counter)	`M_IT_TB_1`	Integrated total with time	`opc_client_da_instance.cpp` 573-575
Custom normalized	`M_ME_TN_1`	Normalized measurement	`opc_client_da_instance.cpp` 577-579

Mapping Code (`opc_client_da/opc_client_da_instance.cpp` 532-627):

```
static int db_callback(void *NotUsed, int argc, char **argv, char **azColName)
{
    // Column 3: iec_type
    if(strcmp(argv[2], "M_ME_TF_1") == 0)
        gl_Config_db[gl_row_counter].io_list_iec_type = M_ME_TF_1;
    else if(strcmp(argv[2], "M_SP_TB_1") == 0)
        gl_Config_db[gl_row_counter].io_list_iec_type = M_SP_TB_1;
    else if(strcmp(argv[2], "M_DP_TB_1") == 0)
        gl_Config_db[gl_row_counter].io_list_iec_type = M_DP_TB_1;
    // ... additional mappings
}
```

Processing Received Data

From `opc_client_da/opc_client_da_instance.cpp` 874-1078 :

```
switch(p_item->iec_type)
{
    case M_ME_TF_1:  // Floating point measurement
    {
        if(it != ioa_name_map.end())
        {
            sp_name = it.data();
            iec_type36 var = p_item->iec_obj.o.type36;

            IECValue v(VALUE_TAG, &var, M_ME_TF_1);
            IECValueList l;
            l.insert(l.end(), v);

            PostList(sp_name, l);  // Post to monitor.exe
        }
    }
    break;

    case M_SP_TB_1:  // Single point with time
    {
        sp_name = it.data();
        iec_type30 var = p_item->iec_obj.o.type30;
        IECValue v(VALUE_TAG, &var, M_SP_TB_1);
        // ... post to monitor
    }
    break;
}
```

Sources: `src/drivers/opc_client_da/opc_client_da_instance.cpp` 532-627
`src/drivers/opc_client_da/opc_client_da_instance.cpp` 874-1078

Comparison of OPC Client Variants

Feature	OPC DA	OPC AE	OPC HDA
Purpose	Real-time data acquisition	Alarm and event monitoring	Historical data retrieval
OPC Specification	OPC DA 2.0	OPC AE 1.1	OPC HDA 1.2
Primary Use Case	Current process values	Alarm notifications	Trend analysis, reporting
Data Direction	Bidirectional (read/write)	Unidirectional (subscribe)	Pull (query-based)
Update Mechanism	Polling or subscription	Event-driven callbacks	Time-range queries
Child Executable	`opc_client_da.exe`	`opc_client_ae.exe`	`opc_client_hda.exe`
SQLite Table	`opc_client_da_table`	`opc_client_ae_table`	`opc_client_hda_table`
FIFO Suffix	`_da`	`_ae`	`_hda`
Named Pipe Pattern	`opc_client_da_namedpipeN`	`opc_client_ae_namedpipeN`	`opc_client_hda_namedpipeN`
Typical IEC Types	M_ME_TF_1, M_SP_TB_1, M_IT_TB_1	Alarm-specific types	Time-series types
Command Support	C_SC_NA_1, C_SE_TC_1	Acknowledgment commands	Query parameter commands
Configuration Keys	ItemID, IOA, readable, writeable	EventID, CategoryID, filter	ItemID, start time, end time, aggregate

Sources: `src/drivers/opc_client_da/opc_client_da_instance.h` 1-230
`src/drivers/opc_client_ae/opc_client_ae_instance.cpp` 1-11
`src/drivers/opc_client_hda/opc_client_hda_instance.cpp` 1-11

Windows-Specific Implementation

OPC Classic clients are inherently Windows-only due to their reliance on COM/DCOM technology:

COM Requirements:

- All OPC servers implement COM interfaces (`IOPCServer`, `IOPCEventServer`, `IOPCHDA_Server`)
- DCOM configuration required for remote OPC servers
- Process must initialize COM apartment model (typically STA or MTA)
- Class IDs (CLSIDs) and Program IDs (ProgIDs) stored in Windows Registry

Process Isolation Benefits:

- Child process crashes don't affect `monitor.exe`
- COM/DCOM initialization isolated from main SCADA process
- Easier debugging of OPC-specific issues
- Independent restart capability without stopping entire system

Platform-Specific Code:

- Windows process creation: `CreateProcess()` `opc_client_dadriverthread.cpp` 179
- Named pipes: `CreateFile()`, `WriteFile()` `opc_client_dadriverthread.cpp` 241-266
- Process monitoring: `GetExitCodeProcess()` `opc_client_dadriverthread.cpp` 82
- Thread messaging: `PostThreadMessage(WM_QUIT)` `opc_client_dadriverthread.cpp` 211

Sources: `src/drivers/opc_client_da/opc_client_dadriverthread.cpp` 138-218

Error Handling and Recovery

Connection Loss Detection

From `opc_client_da/opc_client_da_instance.cpp` 847-868 :

```
case C_LO_ST_1:  // Lost station message from child
{
    if(State != STATE_FAIL)
    {
        printf("OPC DA client on line %d has lost connection to OPC DA server.\n",
               instanceID + 1);

        QString msg;
        msg.sprintf("OPC DA client on line %d has lost connection to OPC DA server.",
                    instanceID + 1);
        FailUnit(msg);

        State = STATE_FAIL;
    }
}
break;
```

Connection Restoration

From `opc_client_da/opc_client_da_instance.cpp` 843-864 :

```
if(State == STATE_FAIL)
{
    if(p_item->iec_type != C_LO_ST_1)  // Any message except lost station
    {
        QString msg;
        msg.sprintf("OPC DA client on line %d is now connected to OPC DA server.",
                    instanceID + 1);
        UnFailUnit(msg);
        State = STATE_ASK_GENERAL_INTERROGATION;
    }
}
```

Timeout Monitoring

From `opc_client_da/opc_client_da_instance.cpp` 1155-1177 :

```
if(msg_received_in_monitor_direction && (wait_for_message > 60))  // 60 seconds
{
    wait_for_message = 0;
    msg_received_in_monitor_direction = 0;

    QString msg;
    msg.sprintf("OPC DA client on line %d has been stopped...", instanceID + 1);
    FailUnit(msg);

    State = STATE_FAIL;

    // Terminate child process
    pConnect->TerminateChild();
}
```

Sources: `src/drivers/opc_client_da/opc_client_da_instance.cpp` 843-868 `src/drivers/opc_client_da/opc_client_da_instance.cpp` 1155-1177

OPC UA Client

› Relevant source files

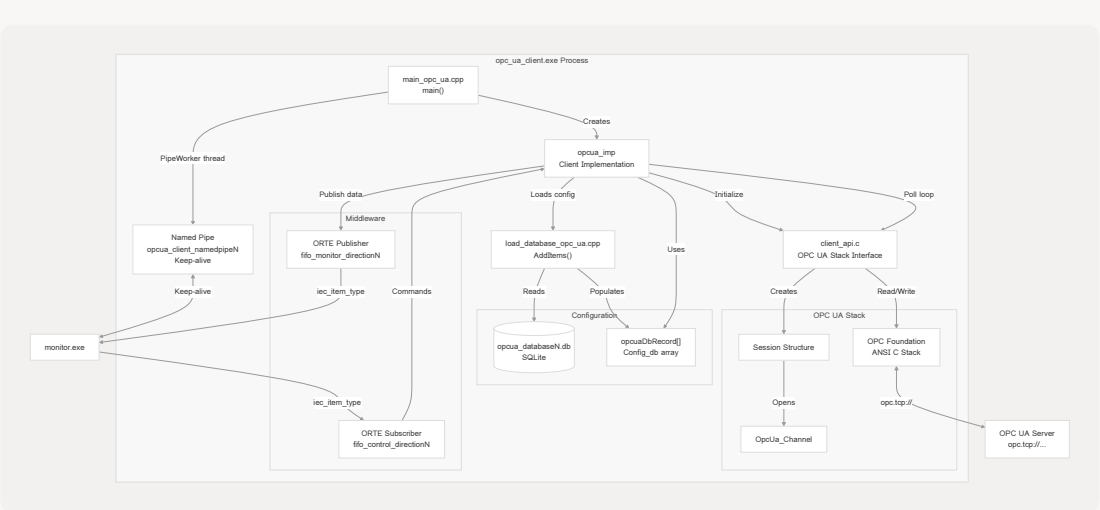
Purpose

This document describes the OPC Unified Architecture (OPC UA) client driver implementation in IndigoSCADA. The OPC UA client connects to OPC UA servers, reads node values periodically, and supports writing commands back to the server. It translates OPC UA data types to IEC 60870-5-104 format for integration with the IndigoSCADA monitoring system.

For information about other OPC implementations, see OPC Classic Clients. For general protocol driver architecture patterns, see Protocol Driver Architecture. For middleware integration details, see Middleware and IPC.

Architecture Overview

The OPC UA client driver follows IndigoSCADA's standard protocol driver architecture but operates as a standalone process without a child process pattern. It uses the OPC Foundation ANSI C Stack for OPC UA protocol implementation.


opc_ua_client.exe Process
main_opc_ua.cpp
main()
Creates
opcua_imp
Client Implementation
PipeWorker thread
Publish data
Loads config
Initialize
Poll loop
Middleware
Named Pipe
opcua_client_namedpipeN
Keep-alive
ORTE Publisher
fifo_monitor_directionN
load_database_opc_ua.cpp
AddItems()
Uses
client_api.c
OPC UA Stack Interface
Keep-alive
iec_item_type
Commands
Reads
Populates
Creates
Read/Write
Configuration
OPC UA Stack
monitor.exe
opcua_databaseN.db
SQLite
opcuaDbRecord[]
Config_db array
Session Structure
OPC Foundation
ANSI C Stack
Opens
opc.tcp://
iec_item_type
ORTE Subscriber
fifo_control_directionN
OpcUa_Channel
OPC UA Server
opc.tcp://...

Sources: `src/drivers/opc_ua/protocol/client/main_opc_ua.cpp` 1-232
`src/drivers/opc_ua/protocol/client/opc_ua_imp.h` 33-85
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 111-186

Main Components

opcua_imp Class

The `opcua_imp` class is the core implementation of the OPC UA client driver.

Member	Type	Purpose
`Config_db`	`opcuaDbRecord*`	Array of node configurations
`db_n_rows`	`int`	Number of configured nodes
`session`	`Session`	OPC UA session structure
`serverURL`	`char[80]`	Server endpoint URL
`lineNumber`	`int`	Driver instance identifier
`pollingTime`	`unsigned long`	Polling interval in milliseconds
`is_connected`	`bool`	Connection status flag
`general_interrogation`	`bool`	Initial data acquisition flag
`publisher`	`ORTEPublication*`	Middleware publisher
`subscriber`	`ORTESubscription*`	Middleware subscriber

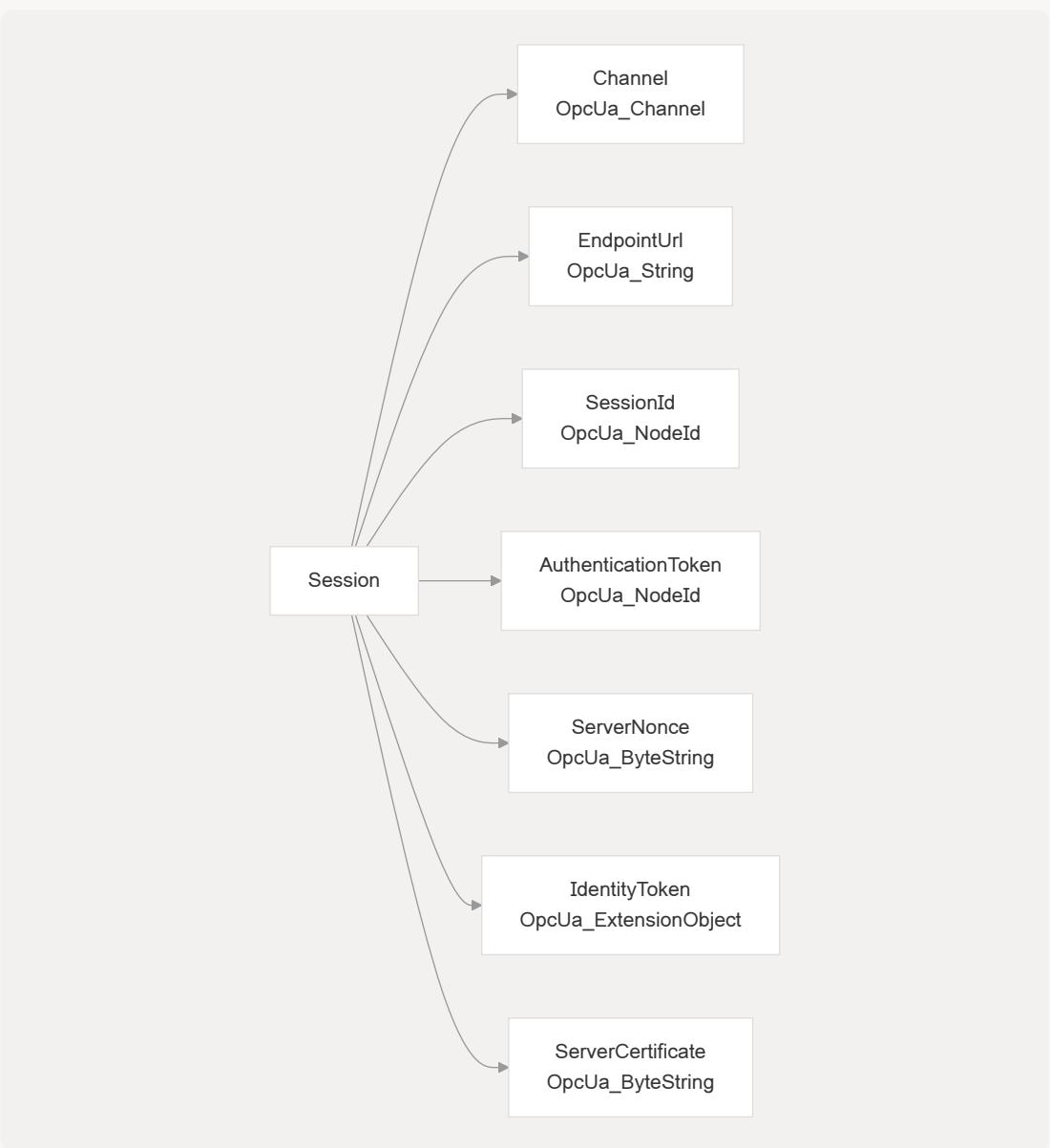
Key Methods:

- `Start()` - Initializes OPC UA stack and creates session
- `PollServer()` - Main polling loop
- `PollItems()` - Reads all configured nodes
- `check_for_commands()` - Processes control commands
- `AddItems()` - Loads node configuration from database

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.h` 33-85
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 111-199

Session Structure

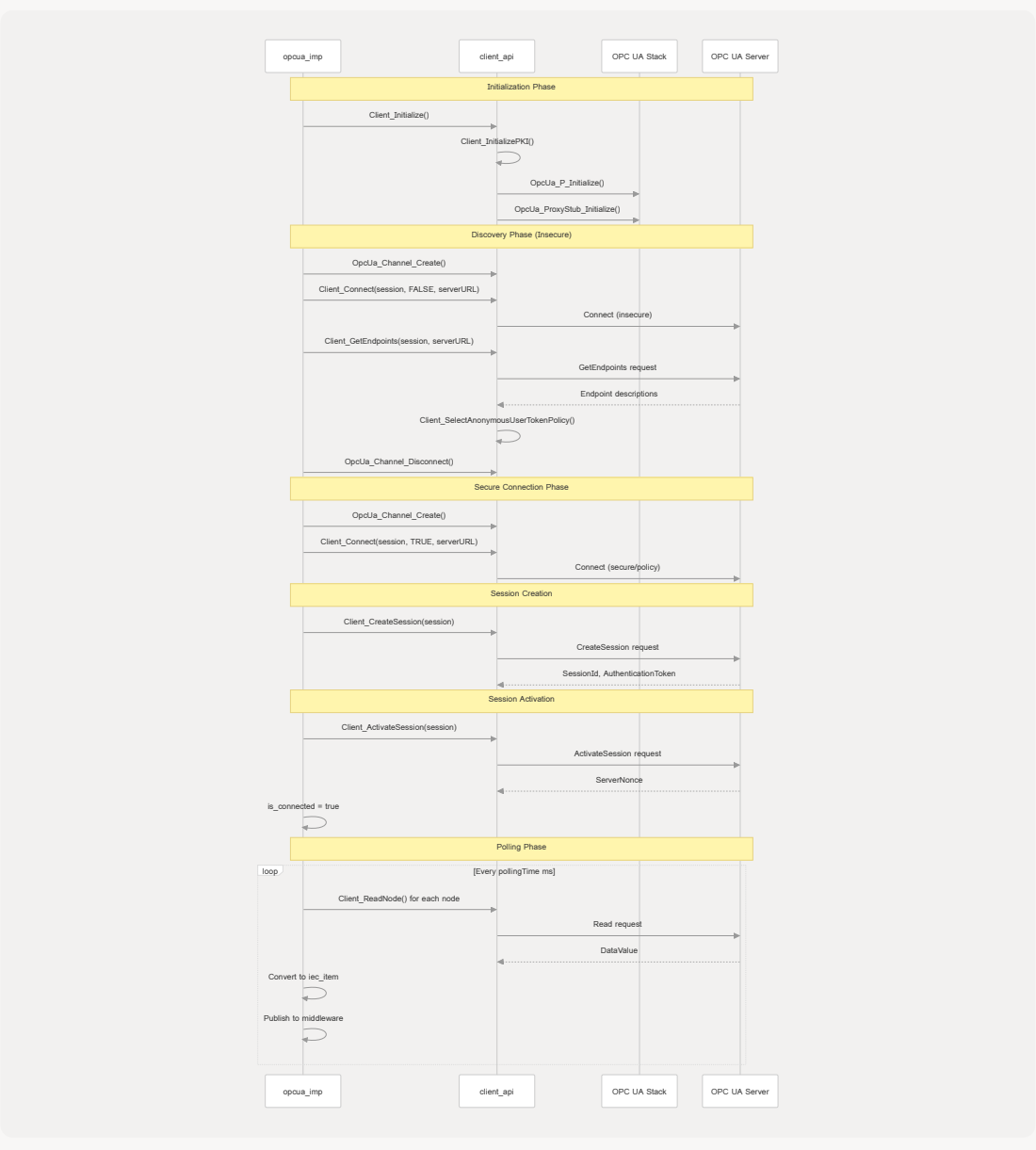
The `Session` structure manages the OPC UA session state:


Session
Channel
OpcUa_Channel
EndpointUrl
OpcUa_String
SessionId
OpcUa_NodeId
AuthenticationToken
OpcUa_NodeId
ServerNonce
OpcUa_ByteString
IdentityToken
OpcUa_ExtensionObject
ServerCertificate
OpcUa_ByteString

Sources: `src/drivers/opc_ua/protocol/client/client_api.h` 129-148
`src/drivers/opc_ua/protocol/client/client_api.c` 318-339

Session Lifecycle

The OPC UA client establishes and maintains a session through multiple phases:


opcua_imp
client_api
OPC UA Stack
OPC UA Server
Initialization Phase
Client_Initialize()
Client_InitializePKI()
OpcUa_P_Initialize()
OpcUa_ProxyStub_Initialize()
Discovery Phase (insecure)
OpcUa_Channel_Create()
Client_Connect(session, FALSE, serverURL)
Connect (insecure)
Client_GetEndpoints(session, serverURL)
GetEndpoints request
Endpoint descriptions
Client_SelectAnonymousUserTokenPolicy()
OpcUa_Channel_Disconnect()
Secure Connection Phase
OpcUa_Channel_Create()
Client_Connect(session, TRUE, serverURL)
Connect (securePolicy)
Session Creation
Client_CreateSession(session)
CreateSession request
SessionId, AuthenticationToken
Session Activation
Client_ActivateSession(session)
ActivateSession request
ServerNonce
is_connected = true
Polling Phase
loop
[Every pollingTime ms]
Client_ReadNode() for each node
Read request
DataValue
Convert to iec_item
Publish to middleware

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 304-396
`src/drivers/opc_ua/protocol/client/client_api.c` 344-406
`src/drivers/opc_ua/protocol/client/client_api.c` 497-566
`src/drivers/opc_ua/protocol/client/client_api.c` 571-669
`src/drivers/opc_ua/protocol/client/client_api.c` 674-744

Security Configuration

The client supports multiple security modes configured at compile time:

Define	Default	Purpose
`UACLIENT_USE_SECURE_COMMUNICATION`	0	Enable/disable security
`UACLIENT_SECURITY_POLICY`	None	Security policy URI
`UACLIENT_SECURITY_MODE`	None	Message security mode
`UACLIENT_CERTIFICATE_LOCATION`	-	Client certificate path
`UACLIENT_PRIVATE_KEY_LOCATION`	-	Client private key path

When security is disabled, the client connects with `OpcUa_SecurityPolicy_None` and `OpcUa_MessageSecurityMode_None`.

Sources: `src/drivers/opc_ua/protocol/client/client_api.h` 40-110
`src/drivers/opc_ua/protocol/client/client_api.c` 80-181

Node Configuration

Database Schema

Node configurations are stored in SQLite database `opcua_databaseN.db` where N is the line number.

Table: opcua_table

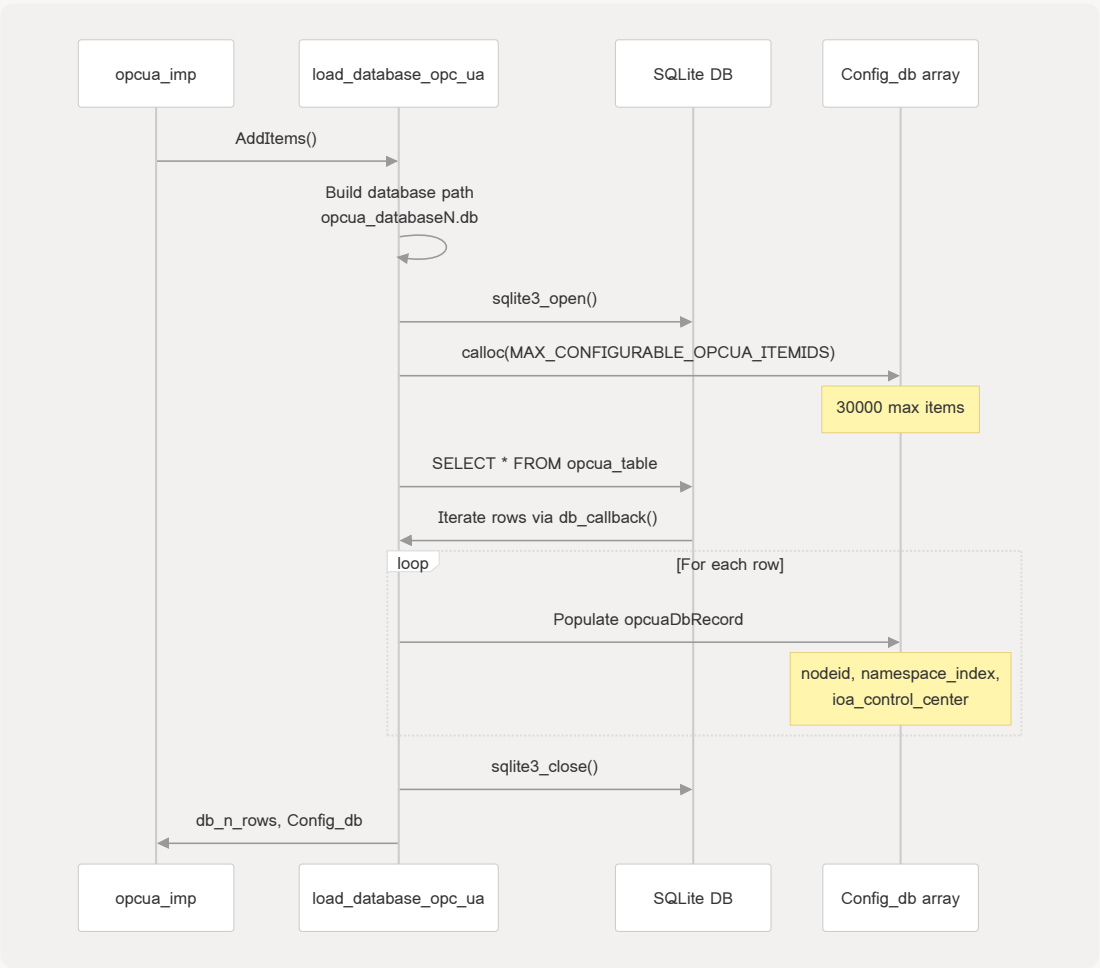
Column	Type	Description
`nodeid`	TEXT	OPC UA node identifier (string format)
`namespace_index`	INTEGER	Node namespace index
`ioa_control_center`	INTEGER	IOA (Information Object Address) for mapping

opcuaDbRecord Structure:

```
struct opcuaDbRecord
{
    char nodeid[50];               // OPC UA node ID string
    int namespace_index;           // namespace index
    int dataType;                  // data type of the node (auto-detected)
    unsigned int ioa_control_center; // unique IOA inside CASDU
};
```

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_db.h` 16-24
`src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 32-80

Configuration Loading

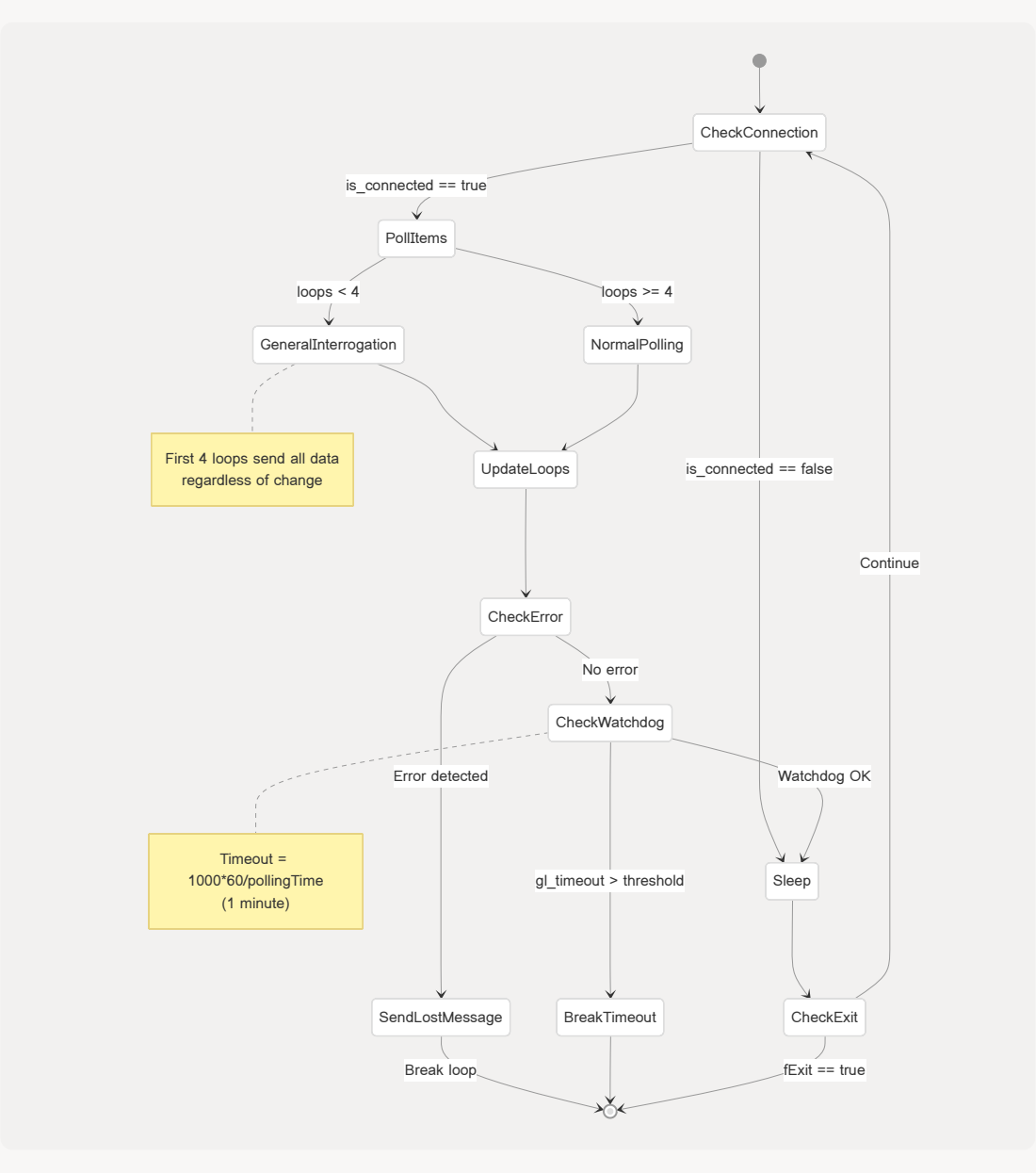

opcua_imp
load_database_opc_ua
SQLite DB
Config_db array
AddItems()
Build database path
opcua_databaseN.db
sqlite3_open()
calloc(MAX_CONFIGURABLE_OPCUA_ITEMIDS)
30000 max items
SELECT * FROM opcua_table
Iterate rows via db_callback()
loop
[For each row]
Populate opcuaDbRecord
nodeid, namespace_index,
ioa_control_center
sqlite3_close()
db_n_rows, Config_db

Sources: `src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 84-165
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 317-320

Data Acquisition (Polling)

Polling Loop

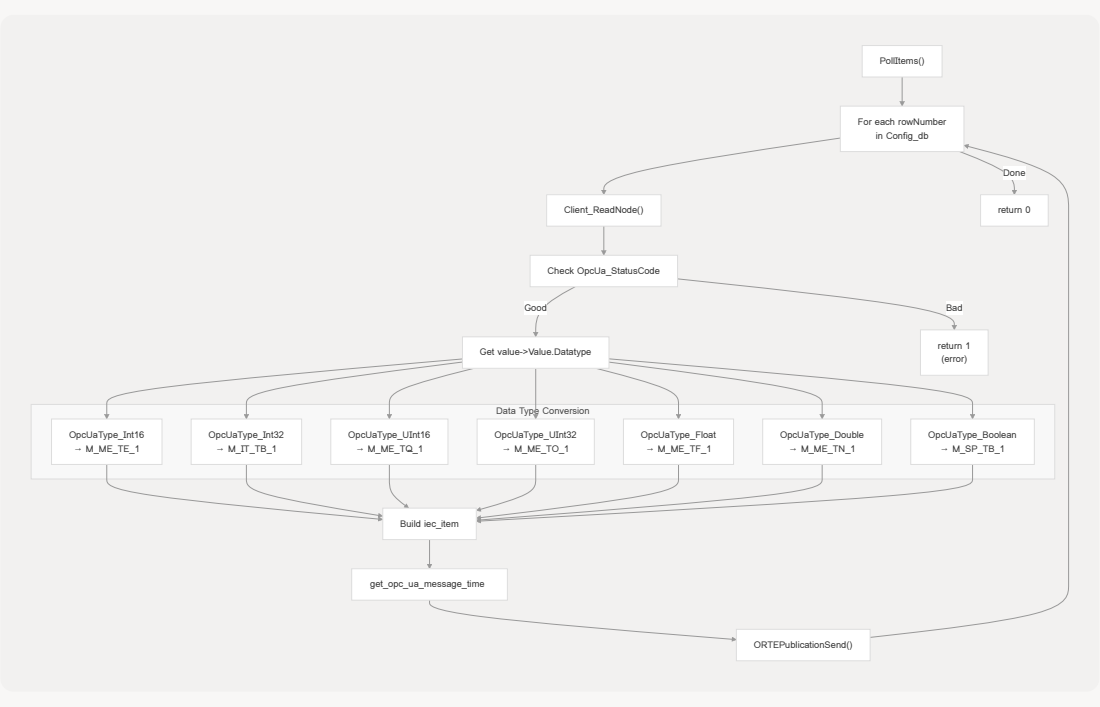
The `PollServer()` method implements the main polling loop with watchdog monitoring:


CheckConnection
is_connected == true
PollItems
loops < 4
loops >= 4
GeneralInterrogation
NormalPolling
First 4 loops send all data regardless of change
UpdateLoops
is_connected == false
Continue
CheckError
No error
CheckWatchdog
Error detected
Watchdog OK
Timeout = 1000*60/pollingTime (1 minute)
gl_timeout > threshold
Sleep
SendLostMessage
BreakTimeout
CheckExit
Break loop
fExit == true

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 203-302

PollItems Implementation

The `PollItems()` method reads all configured nodes and publishes data to the middleware:


PollItems()
For each rowNumber in Config_db
Done
return 0
Client_ReadNode()
Check OpcUa_StatusCode
Good
Bad
return 1 (error)
Get value->Value.Datatype
Data Type Conversion
OpcUaType_Int16 → M_ME_TE_1
OpcUaType_Int32 → M_IT_TB_1
OpcUaType_UInt16 → M_ME_TQ_1
OpcUaType_UInt32 → M_ME_TO_1
OpcUaType_Float → M_ME_TF_1
OpcUaType_Double → M_ME_TN_1
OpcUaType_Boolean → M_SP_TB_1
Build iec_item
get_opc_ua_message_time
ORTEPublicationSend()

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 581-782

Data Type Mapping

The OPC UA client maps OPC UA data types to IEC 60870-5-104 types:

OPC UA Type	IEC Type	IEC Type Code	Field	Description
`OpcUaType_Int16`	`M_ME_TE_1`	35	`type35.mv`	Measured value, scaled
`OpcUaType_Int32`	`M_IT_TB_1`	37	`type37.counter`	Integrated totals
`OpcUaType_UInt16`	`M_ME_TQ_1`	153	`type153.mv`	Measured value, unsigned
`OpcUaType_UInt32`	`M_ME_TO_1`	151	`type151.mv`	Measured value, unsigned long
`OpcUaType_Float`	`M_ME_TF_1`	36	`type36.mv`	Measured value, short float
`OpcUaType_Double`	`M_ME_TN_1`	150	`type150.mv`	Measured value, double
`OpcUaType_Boolean`	`M_SP_TB_1`	30	`type30.sp`	Single point with time

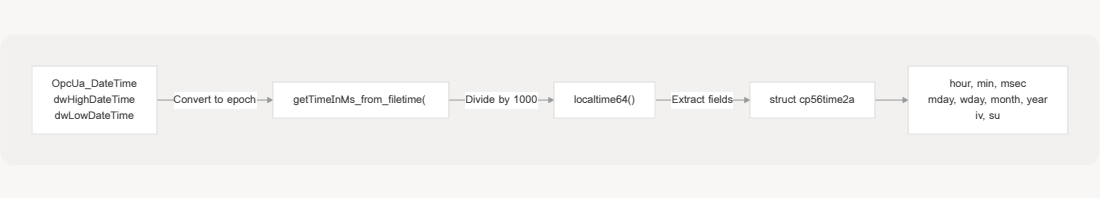
All measurements include:

- **IOA**: Mapped from `Config_db[rowNumber].ioa_control_center`
- **Cause**: 0x03 (spontaneous)
- **Timestamp**: Extracted from `value->SourceTimestamp` using `get_opc_ua_message_time()`
- **Invalid bit**: 0 (valid)

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 618-738

Timestamp Conversion

OPC UA timestamps (Windows FILETIME format) are converted to IEC 60870-5 CP56Time2a:


OpcUa_DateTime
dwHighDateTime
dwLowDateTime
Convert to epoch
getTimeInMs_from_filetime(
Divide by 1000
localtime64()
Extract fields
struct cp56time2a
hour, min, msec
mday, wday, month, year
iv, su

Timestamp fields:

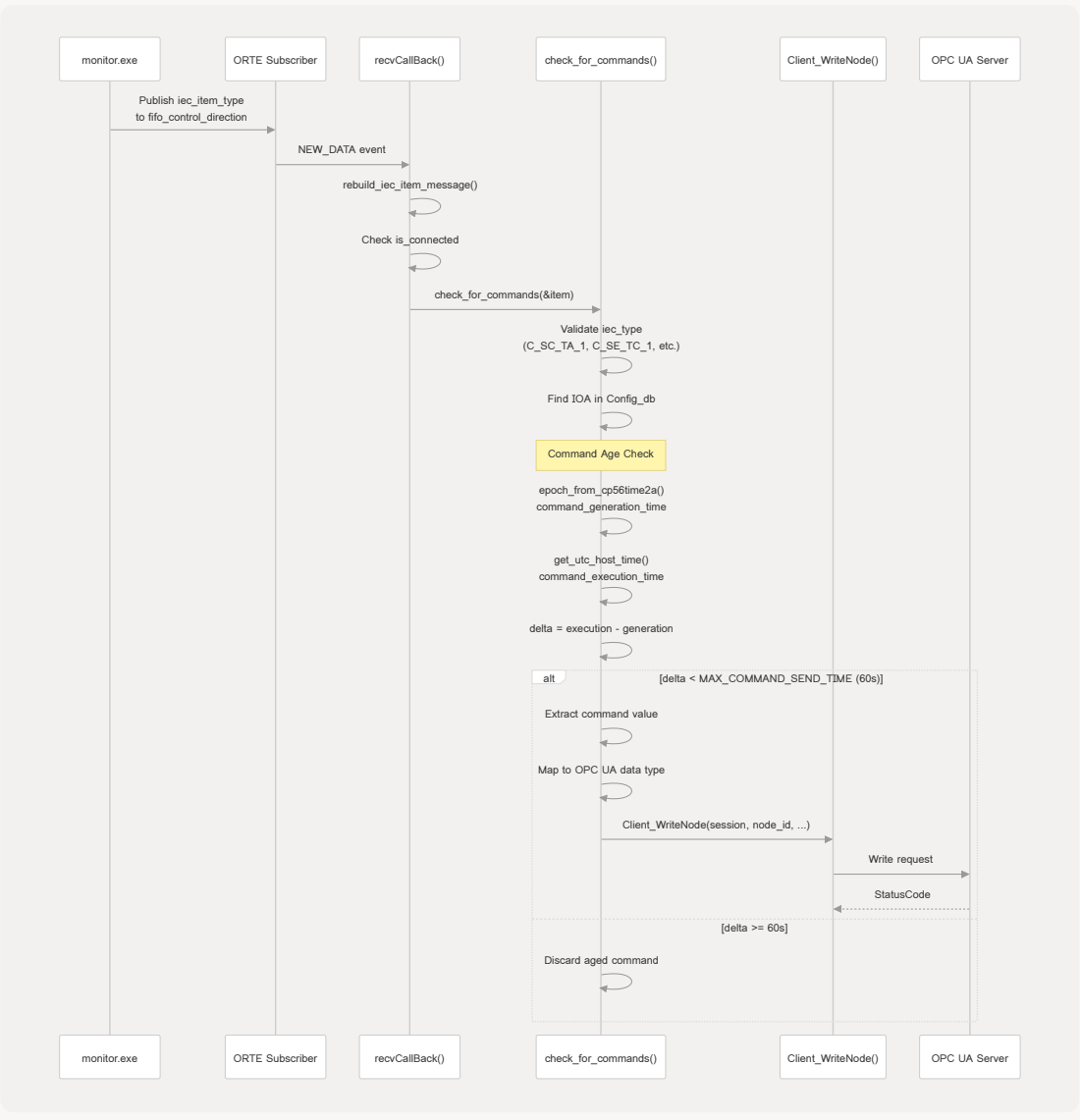
- **hour**: 0-23
- **min**: 0-59
- **msec**: 0-59999 (includes seconds * 1000 + milliseconds)
- **mday**: 1-31 (day of month)
- **wday**: 1-7 (day of week, Sunday=7)
- **month**: 1-12
- **year**: 0-99 (years since 2000)
- **iv**: Invalid flag (0=valid, 1=invalid)
- **su**: Summer time flag

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 442-481
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 457-481

Command Handling

Command Reception

The OPC UA client receives commands from the control direction via ORTE subscription:


monitor.exe
ORTE Subscriber
recvCallBack()
check_for_commands()
Client_WriteNode()
OPC UA Server
Publish iec_item_type
to fifo_control_direction
NEW_DATA event
rebuild_iec_item_message()
Check is_connected
check_for_commands(&item)
Validate iec_type
(C_SC_TA_1, C_SE_TC_1, etc.)
Find IOA in Config_db
Command Age Check
epoch_from_cp56time2a()
command_generation_time
get_utc_host_time()
command_execution_time
delta = execution - generation
alt
[delta < MAX_COMMAND_SEND_TIME (60s)]
Extract command value
Map to OPC UA data type
Client_WriteNode(session, node_id, ...)
Write request
StatusCode
[delta >= 60s]
Discard aged command

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 77-104
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 896-1298

Supported Command Types

IEC Type	Code	Command Type	Timestamp
`C_SC_TA_1`	58	Single command with time	Yes
`C_SC_NA_1`	45	Single command	No
`C_DC_TA_1`	59	Double command with time	Yes
`C_DC_NA_1`	46	Double command	No
`C_SE_TA_1`	61	Set point, normalized with time	Yes
`C_SE_NA_1`	48	Set point, normalized	No
`C_SE_TB_1`	62	Set point, scaled with time	Yes
`C_SE_NB_1`	49	Set point, scaled	No
`C_SE_TC_1`	63	Set point, short float with time	Yes
`C_SE_NC_1`	50	Set point, short float	No
`C_BO_TA_1`	64	Bitstring with time	Yes
`C_BO_NA_1`	51	Bitstring	No
`C_IC_NA_1`	100	General interrogation	-
`C_EX_IT_1`	103	Exit process	-

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 904-916

Command to OPC UA Type Mapping

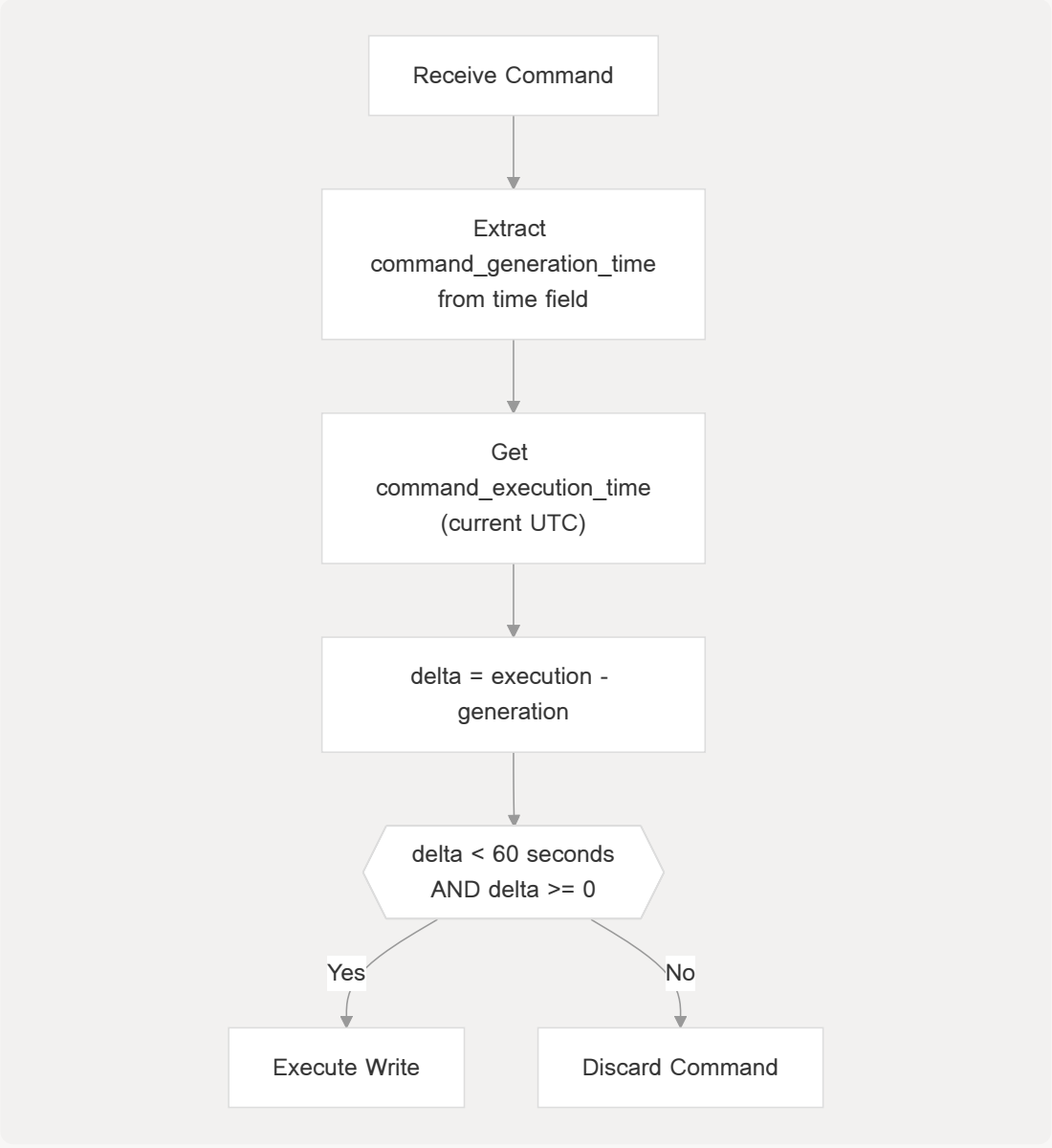
The `C_SE_TC_1` (set point, short float) command is mapped to OPC UA types based on the configured node's data type:

```
// From Config_db[rowNumber].dataType
switch(Config_db[rowNumber].dataType)
{
    case OpcUaType_Int16:
        ValueWrite.Value.Datatype = OpcUaType_Int16;
        ValueWrite.Value.Value.Int16 = (int)cmd_val.f;
        break;
    case OpcUaType_Float:
        ValueWrite.Value.Datatype = OpcUaType_Float;
        ValueWrite.Value.Value.Float = cmd_val.f;
        break;
    // ... other types
}
```

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 1152-1213

Command Age Validation

Commands include a timestamp in the `time` field. The client validates command freshness:


Receive Command
Extract
command_generation_time
from time field
Get
command_execution_time
(current UTC)
delta = execution -
generation
delta < 60 seconds
AND delta >= 0
Yes
No
Execute Write
Discard Command

The threshold `MAX_COMMAND_SEND_TIME` is defined as 60 seconds.

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 23
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 949-1105

Middleware Integration

ORTE Publication and Subscription

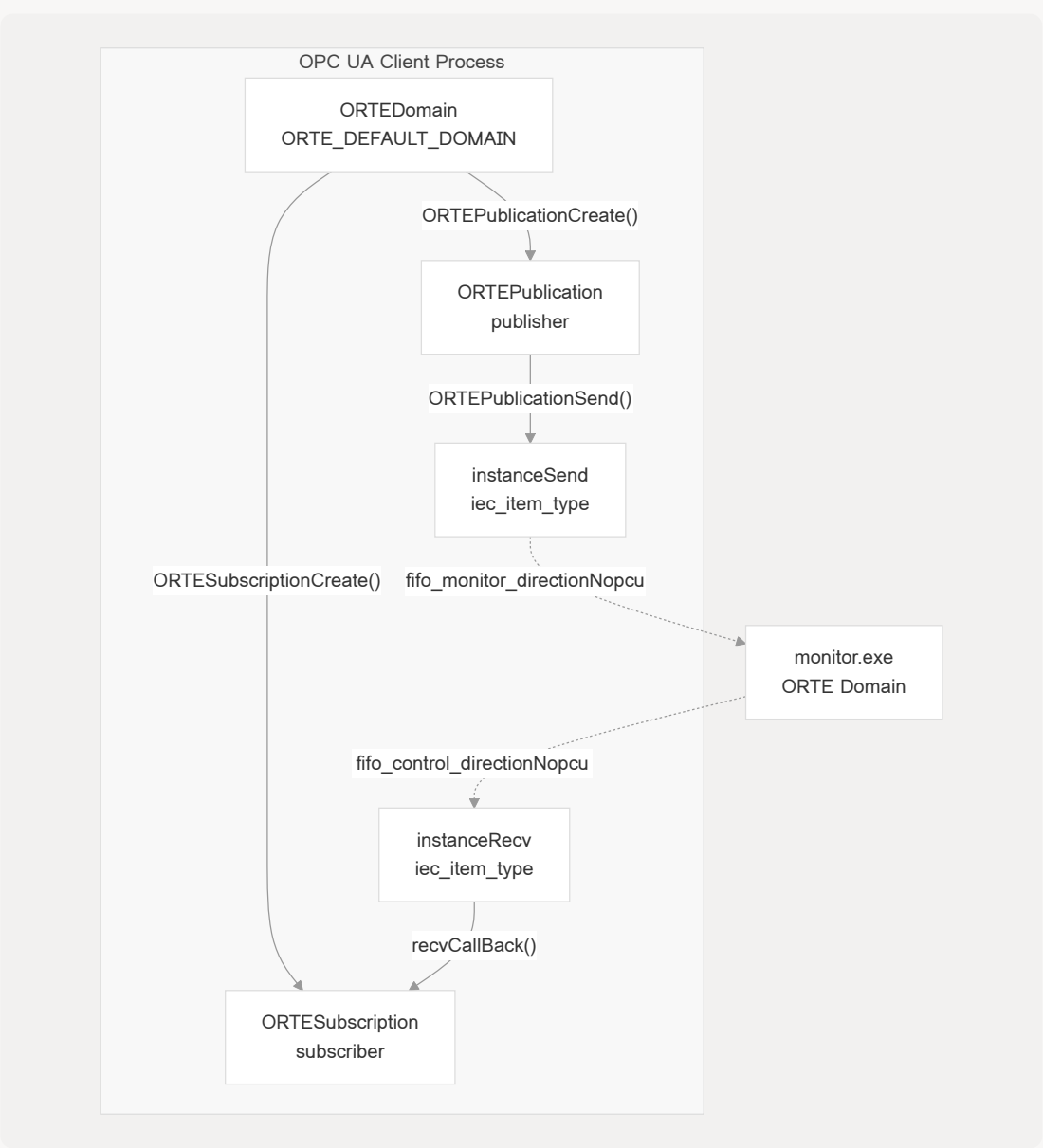
The OPC UA client uses ORTE (Open Real-Time Ethernet) middleware for communication with `monitor.exe`:

Publisher Configuration:

- Topic: `fifo_monitor_direction<lineNumber>opcua`
- Type: `iec_item_type`
- Persistence: 5 seconds
- Direction: From OPC UA client to monitor

Subscriber Configuration:

- Topic: `fifo_control_direction<lineNumber>opcua`
- Type: `iec_item_type`
- Deadline: 3 seconds
- Direction: From monitor to OPC UA client
- Callback: `recvCallBack()`


OPC UA Client Process
ORTEDomain
ORTE_DEFAULT_DOMAIN
ORTEPublicationCreate()
ORTEPublication
publisher
ORTEPublicationSend()
instanceSend
iec_item_type
ORTESubscriptionCreate()
fifo_monitor_directionNopcu
monitor.exe
ORTE Domain
fifo_control_directionNopcu
instanceRecv
iec_item_type
recvCallBack()
ORTESubscription
subscriber

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 117-185

Keep-Alive Mechanism

A separate thread (`PipeWorker`) manages keep-alive communication via named pipe:

Named Pipe: `\\.\pipe\opcua_client_namedpipe<lineNumber>`

The pipe receives periodic messages with IOA 4004 from monitor.exe to reset the watchdog counter:

```
if(p_item->iec_obj.ioa == 4004)
{
    gl_timeout_connection_with_parent = 0;
    fprintf(stdout, "wdg %d\r", p_item->msg_id);
    fflush(stdout);
}
```

If `gl_timeout_connection_with_parent` exceeds the threshold (60 seconds), the polling loop exits.

Sources: `src/drivers/opc_ua/protocol/client/main_opc_ua.cpp` 238-379
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 286-295

Process Startup

Command Line Arguments

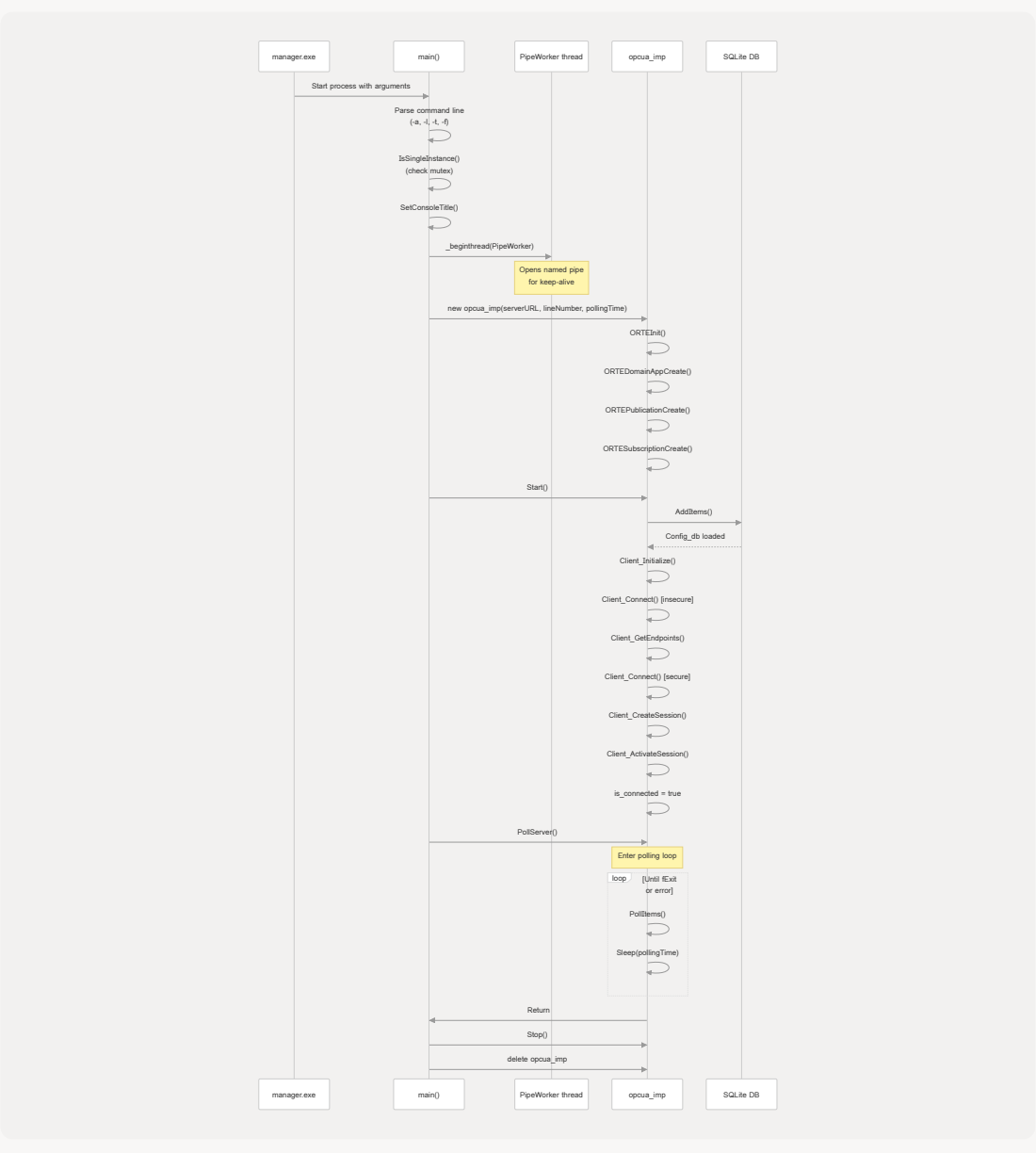
The OPC UA client is started with the following arguments:

```
opc_ua_client.exe -a <server_url> -l <line_number> -t <polling_time> [-f <log_file>]
```

Argument	Description	Example
`-a`	OPC UA server endpoint URL	`opc.tcp://localhost:4840`
`-l`	Line number (driver instance)	`1`
`-t`	Polling time in milliseconds	`1000`
`-f`	Optional log file prefix	`opcua_log`

Sources: `src/drivers/opc_ua/protocol/client/main_opc_ua.cpp` 62
`src/drivers/opc_ua/protocol/client/main_opc_ua.cpp` 100-142

Startup Sequence


manager.exe
main()
PipeWorker thread
opcua_imp
SQLite DB
Start process with arguments
Parse command line
(-a, -l, -t, -f)
IsSingleInstance()
(check mutex)
SetConsoleTitle()
_beginthread(PipeWorker)
Opens named pipe
for keep-alive
new opcua_imp(serverURL, lineNumber, pollingTime)
ORTEInit()
ORTEDomainAppCreate()
ORTEPublicationCreate()
ORTESubscriptionCreate()
Start()
AddItems()
Config_db loaded
Client_Initialize()
Client_Connect() (insecure)
Client_GetEndpoints()
Client_Connect() (secure)
Client_CreateSession()
Client_ActivateSession()
is_connected = true
PollServer()
Enter polling loop
loop
[Until fExit or error]
PollItems()
Sleep(pollingTime)
Return
Stop()
delete opcua_imp

Sources: `src/drivers/opc_ua/protocol/client/main_opc_ua.cpp` 70-232
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 304-396

Error Handling

Connection Loss Detection

The client monitors connection status and publishes a LOST message when disconnected:

```
if(rc) // PollItems() returned error
{
    // Build C_LO_ST_1 message with IOA 0
    struct iec_item item_to_send;
    memset(&item_to_send, 0x00, sizeof(struct iec_item));

    item_to_send.iec_obj.ioa = 0;
    item_to_send.cause = 0x03;
    item_to_send.iec_type = C_LO_ST_1;
    item_to_send.iec_obj.o.type30.sp = 0;
    item_to_send.iec_obj.o.type30.time = actual_time;

    ORTEPublicationSend(publisher);

    break; // Exit polling loop
}
```

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 227-276

General Interrogation

The first 4 polling loops send all data regardless of change detection (if implemented):

```
// In PollServer()
loops = 0;
general_interrogation = true;

// ... polling loop ...

loops++;
if(loops == 4)
{
    general_interrogation = false;
}

// In PollItems()
if(send_item || general_interrogation)
{
    // Send item to middleware
}
```

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 209-225
`src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 741-776

Build Configuration

Project Files

The OPC UA client is built as a console application:

Project: `opc_ua_client.dsp` **Target:** `c:\scada\bin\opc_ua_client.exe`

Key include directories:

- `.\Stack\core` - OPC UA stack core
- `.\Stack\platforms\win32` - Windows platform layer
- `.\Stack\stackcore` - Stack core functionality
- `.\Stack\securechannel` - Security layer
- `.\Stack\transport\tcp` - TCP transport
- `.\Stack\proxystub\clientproxy` - Client proxy stubs
- `..\..\..\common` - IndigoSCADA common headers
- `..\..\..\middleware\rtps\include` - RTPS middleware

Libraries:

- `ws2_32.lib` - Winsock 2
- `rtps.lib` - RTPS middleware
- `utilities.lib` - IndigoSCADA utilities

Sources: `src/drivers/opc_ua/protocol/opc_ua_client.dsp` 1-797

OPC UA Stack Integration

The client integrates the complete OPC Foundation ANSI C Stack, including:

Core Components:

- Binary encoder/decoder
- Built-in types
- String handling
- DateTime handling
- Memory management
- GUID generation

Platform Layer (Win32):

- Socket implementation (`opcua_p_socket.c`)
- Thread management (`opcua_p_thread.c`)
- Mutex/semaphore (`opcua_p_mutex.c`, `opcua_p_semaphore.c`)
- Cryptography (OpenSSL integration)
- PKI provider

Transport:

- TCP connection (`opcua_tcpconnection.c`)

- TCP listener (`opcua_tcplistener.c`)
- Secure channel (`opcua_tcpsecurechannel.c`)

Client Proxy:

- Channel management (`opcua_channel.c`)
- Client API (`opcua_clientapi.c`)
- Async call state (`opcua_asynccallstate.c`)

Sources: `src/drivers/opc_ua/protocol/opc_ua_client.dsp` 91-785

Monitor Process

> Relevant source files

Purpose and Scope

The Monitor Process (`monitor.exe`) serves as the central data acquisition engine in IndigoSCADA. It orchestrates all protocol drivers, manages sample point state, coordinates with the results processor for alarm evaluation, and maintains bidirectional communication with the HMI and field devices. This document covers the monitor's architecture, tick-based state machine, driver coordination, command routing, and database synchronization mechanisms.

For information about individual protocol drivers, see 3. Protocol Drivers. For alarm evaluation and historical data storage, see 4.2 Results Processing and Alarming. For the calculated driver, see 4.3 Calculated Driver.

Overview

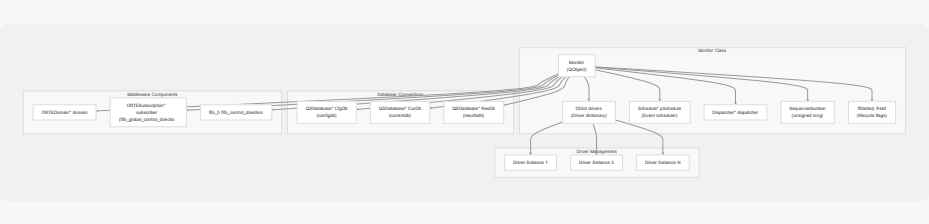
The Monitor process is a long-running Qt application that:

- **Orchestrates Protocol Drivers**: Loads, starts, stops, and monitors health of all configured driver instances
- **Manages Sample Points**: Tracks enabled sample points with their tags, alarm states, and statistics
- **Routes Commands**: Dispatches control commands from HMI to appropriate drivers based on IOA mappings
- **Synchronizes Databases**: Updates `currentdb` (real-time values) and coordinates with `configdb` (configuration) and `resultsdb` (historical data)
- **Publishes Events**: Broadcasts tick notifications, alarm events, and system state changes via the Dispatcher
- **Tick-Based Processing**: Executes a 100ms cycle for state updates, alarm evaluation, and database synchronization

Sources: `src/monitor/monitor.h` 1-145 `src/monitor/monitor.cpp` 1-150

Architecture Components

Core Class: Monitor


Monitor Class
Monitor (QObject)
Middleware Components
Database Connections
Driver Management
Driver Instance 1
Driver Instance 2
Driver Instance N

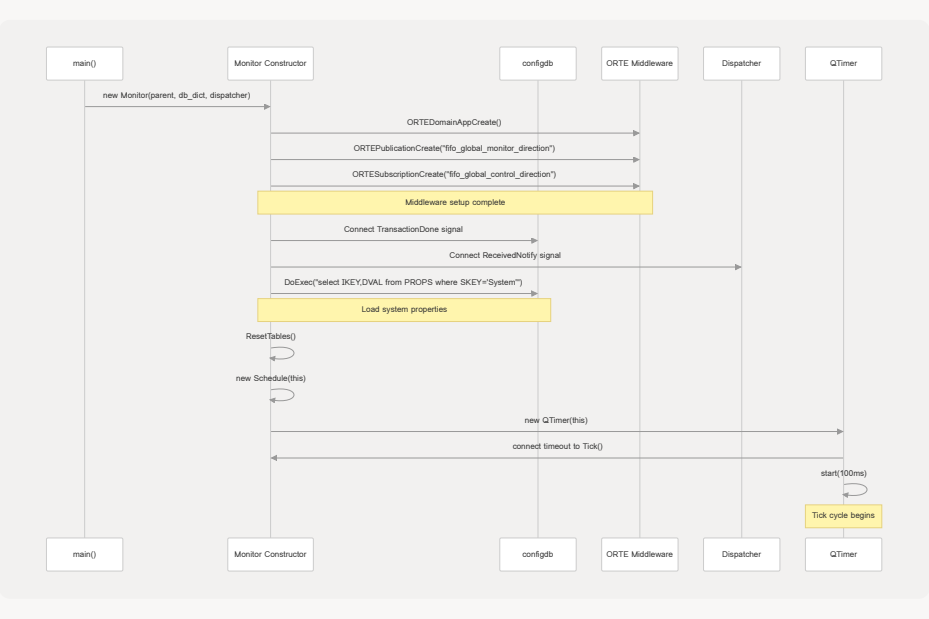
Key Members:

- `drivers` (`DDict`): `std::map<QString, Driver*>` mapping driver names to driver instances
- `pSchedule`: Points to Schedule object managing automated tasks
- `dispatcher`: Connection to notification system for pub/sub events
- `SequenceNumber`: Monotonically increasing counter providing machine-independent time ordering (incremented every tick)
- `CfgDb`, `CurDb`, `ResDb`: Database client connections
- `domain`: ORTE middleware domain for pub/sub
- `fifo_control_direction`: Global FIFO receiving commands from all drivers

Sources: `src/monitor/monitor.h` 55-142 `src/monitor/monitor.cpp` 149-319

Monitor Process Lifecycle

Initialization Sequence


main()
Monitor Constructor
configdb
ORTE Middleware
Dispatcher
QTimer
new Monitor(parent, db_dict, dispatcher)
ORTEDomainAppCreate()
ORTEPublicationCreate("fifo_global_monitor_direction")
ORTESubscriptionCreate("fifo_global_control_direction")
Middleware setup complete
Connect TransactionDone signal
Connect ReceivedNotify signal
DoExec("select IKEY,DVAL from PROPS where SKEY='System'")
Load system properties
ResetTables()
new Schedule(this)
new QTimer(this)
connect timeout to Tick()
start(100ms)
Tick cycle begins

The Monitor constructor performs the following steps:

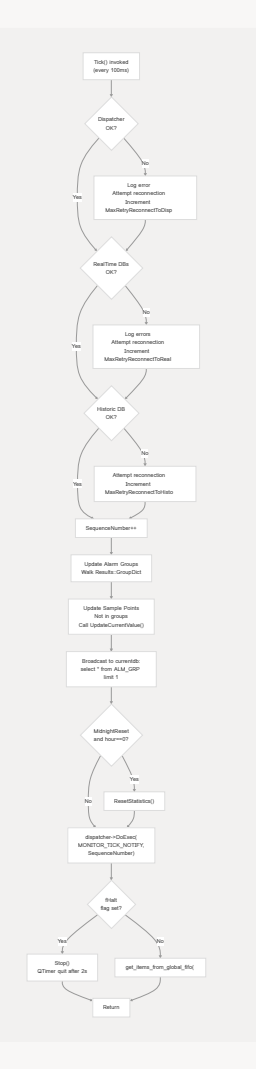
1. **Middleware Initialization** `src/monitor/monitor.cpp` 182-235:
   - Creates ORTE domain with default properties
   - Registers `iec_item_type` data type
   - Creates global publisher (`fifo_global_monitor_direction`)
   - Creates global subscriber (`fifo_global_control_direction`)
   - Opens local FIFO (`fifo_control_dir`) for receiving commands
2. **Database Connections** `src/monitor/monitor.cpp` 243-289:
   - Resolves database pointers from `db_dictionary`
   - Connects to `configdb`, `currentdb`, `resultsdb` signal handlers
   - Optionally connects to historic database if configured
3. **Dispatcher Integration** `src/monitor/monitor.cpp` 292-294:
   - Connects to dispatcher's `ReceivedNotify` signal
   - Enables receiving system-wide notifications
4. **Initialization Queries** `src/monitor/monitor.cpp` 306-310:
   - Fetches system properties (language, timezone, etc.)
   - Calls `ResetTables()` to clear alarm group states
   - Removes system manager lock
5. **Scheduler and Timer** `src/monitor/monitor.cpp` 313-318:
   - Creates `Schedule` object for automated events
   - Starts 100ms tick timer (`TICK_CYCLE`)

Sources: `src/monitor/monitor.cpp` 151-319

Tick-Based State Machine

The Monitor operates on a **100ms tick cycle** (defined by `TICK_CYCLE` constant). Each tick executes a sequence of state updates and health checks.

Tick Processing Flow


Tick() invoked every 100ms
Dispatcher OK?
Log error Attempt reconnection Increment MaxRetryReconnectToDisp
RealTime DBs OK?
Historic DB OK?
SequenceNumber++
Update Alarm Groups Walk Results::GroupDict
Update Sample Points Not in groups Call UpdateCurrentValue()
MidnightReset and hour==0?
ResetStatistics()
fHalt flag set?
get_items_from_global_fifo()
Return
Yes
No

Tick Implementation Details

Connection Health Checks `src/monitor/monitor.cpp` 462-557:

- Tests `GetDispatcher()->Ok()`, reconnects if necessary
- Tests all realtime database connections, logs errors and reconnects
- Tests historic database if configured
- Tracks retry counts in `MaxRetryReconnectTo*` variables

Sequence Number `src/monitor/monitor.cpp` 562:

- Increments on every tick
- Provides machine-independent ordering for database records
- Written to `SEQNO` column in all current value tables

Alarm Group Processing `src/monitor/monitor.cpp` 560-628:

- Iterates `Results::GroupDict`
- For each changed group:
  - Updates member sample points if not already updated
  - Tracks largest alarm state in group
  - Writes to `ALM_GRP` and `ALM_GRP_STATE` tables

Sample Point Updates `src/monitor/monitor.cpp` 633-645:

- Processes sample points not in alarm groups
- Calls `UpdateCurrentValue()` for changed points
- Marks points as not changed (`fChanged = 0`)

Midnight Reset `src/monitor/monitor.cpp` 654-664:

- Detects hour transition from 23 to 0
- Calls `ResetStatistics()` to clear daily stats
- Controlled by `MidnightReset` flag

Broadcast Notification `src/monitor/monitor.cpp` 668:

- Sends `MONITOR_TICK_NOTIFY` with current `SequenceNumber`
- Informs all subscribers that monitor is alive and processing

Global FIFO Processing `src/monitor/monitor.cpp` 679:

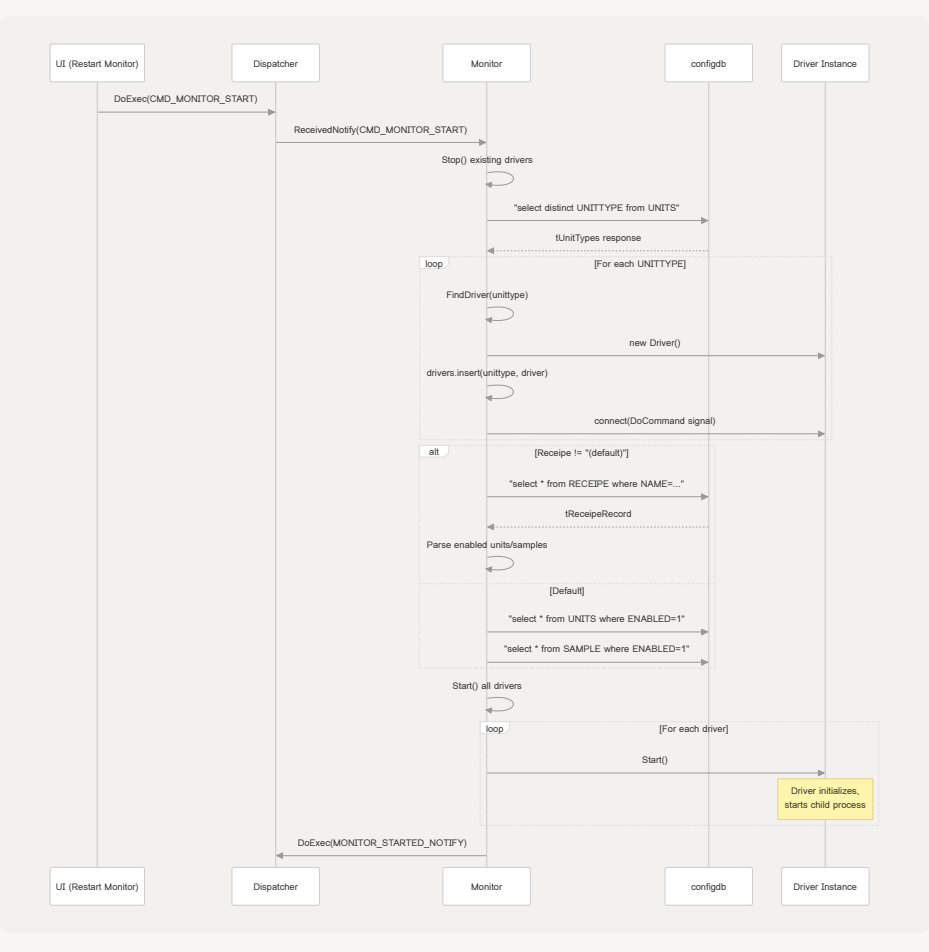
- Calls `get_items_from_global_fifo()` to receive commands from drivers

Sources: `src/monitor/monitor.cpp` 458-680

Driver Coordination

The Monitor acts as the supervisor for all protocol driver instances. It loads drivers dynamically based on configuration, manages their lifecycle, and routes commands to them.

Driver Loading and Management


UI (Restart Monitor)
Dispatcher
Monitor
configdb
Driver Instance
DoExec(CMD_MONITOR_START)
ReceivedNotify(CMD_MONITOR_START)
Stop() existing drivers
"select distinct UNITTYPE from UNITS"
tUnitTypes response
loop
[For each UNITTYPE]
FindDriver(unittype)
new Driver()
drivers.insert(unittype, driver)
connect(DoCommand signal)
alt
[Receipe != "(default)"]
"select * from RECEIPE where NAME=..."
tReceipeRecord
Parse enabled units/samples
[Default]
"select * from UNITS where ENABLED=1"
"select * from SAMPLE where ENABLED=1"
Start() all drivers
[For each driver]
Start()
Driver initializes, starts child process
DoExec(MONITOR_STARTED_NOTIFY)

Driver Dictionary (`DDict drivers`):

- Type: `std::map<QString, Driver*, std::less<QString>>`
- Maps unit type names (e.g., "iec104driver", "modbus_driver") to driver instances
- Populated from `UNITS` table query results `src/monitor/monitor.cpp` 934-972

Loading Process:

1. **Stop Existing Drivers** `src/monitor/monitor.cpp` 936:
   - Clears any previously loaded drivers
   - Ensures clean state before reconfiguration
2. **Query Unit Types** `src/monitor/monitor.cpp` 941-992:
   - Executes `select distinct UNITTYPE from UNITS`
   - Response handled in `tUnitTypes` case
3. **Dynamic Driver Loading** `src/monitor/monitor.cpp` 945-972:
   - For each unique unit type:
     - Calls `FindDriver(unittype)` to load DLL
     - Creates driver instance
     - Inserts into `drivers` map
     - Connects `DoCommand` signal for command routing
4. **Receipe Handling** `src/monitor/monitor.cpp` 979-991:
   - If receipe is "(default)": loads all enabled units/samples
   - Otherwise: loads specific units/samples from `RECEIPE` table
   - Populates `DriverInstance::EnabledUnits` and `Results::GetEnabledPoints()`
5. **Driver Startup** `src/monitor/monitor.cpp` 810-823:
   - Iterates through `drivers` dictionary
   - Calls `Start()` on each driver instance
   - Logs startup events

Signal Connections:

The Monitor uses Qt signals to route commands to drivers:

```
QObject::connect(this, SIGNAL(DoCommand(const QString &, BYTE, LPVOID, DWORD, DWORD)),
                 p, SLOT(Command(const QString &, BYTE, LPVOID, DWORD, DWORD)));
```

`src/monitor/monitor.cpp` 963-964

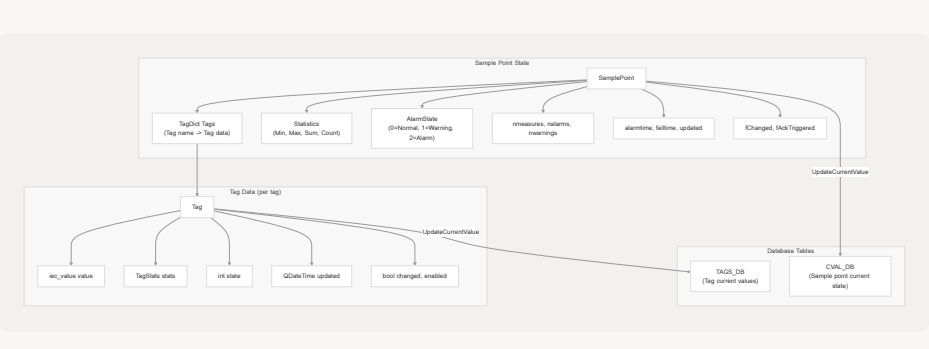
This enables `Monitor::Command()` to broadcast commands to all drivers, which filter based on unit name.

Sources: `src/monitor/monitor.cpp` 803-992 `src/monitor/monitor.h` 81-94

Sample Point Management

Sample Point Structure and Updates

Sample points are the fundamental data entities in IndigoSCADA, representing measurable or controllable values from field devices. The Monitor maintains the current state of all enabled sample points.


Sample Point State
SamplePoint
TagDict Tags (Tag name -> Tag data)
Statistics (Min, Max, Sum, Count)
AlarmState (0=Normal, 1=Warning, 2=Alarm)
nmeasures, nalarms, nwarnings
alarmtime, failtime, updated
fChanged, fAckTriggered
UpdateCurrentValue
Tag Data (per tag)
Tag
iec_value value
TagStats stats
int state
QDateTime updated
bool changed, enabled
UpdateCurrentValue
Database Tables
TAGS_DB (Tag current values)
CVAL_DB (Sample point current state)

UpdateCurrentValue() Method

The `UpdateCurrentValue()` method writes sample point and tag data to `currentdb`:

Tag Updates `src/monitor/monitor.cpp` 361-425:

```
update TAGS_DB set
  UPDTIME={timestamp},
  VAL={value},
  MINVAL={min},
  MAXVAL={max},
  SUMVAL={sum},
  SUM2VAL={sum2},
  NVAL={count},
  STATE={state},
  SEQNO={SequenceNumber}
where NAME='{spname}' and TAGNAME='{tagname}';
```

- Iterates through `sp.Tags` dictionary
- Only updates tags marked as `changed` and `enabled`
- Handles NaN values by substituting 0
- Writes statistics (min, max, sum, sum2, count)
- Includes alarm state and sequence number

Sample Point Updates `src/monitor/monitor.cpp` 432-446:

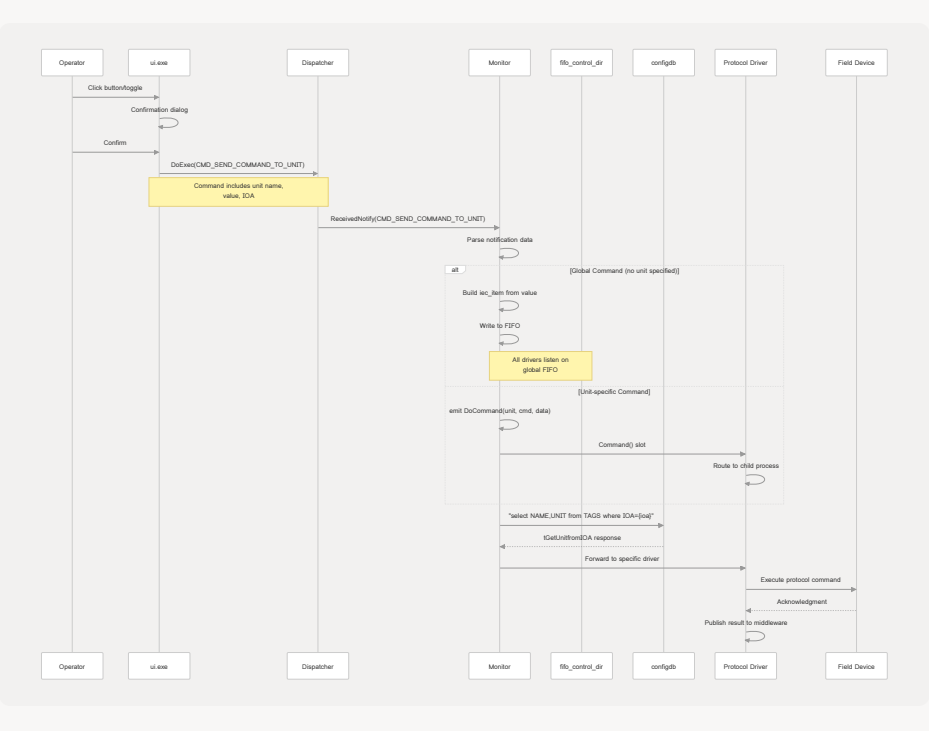
```
update CVAL_DB set
  UPDTIME={timestamp},
  ALMTIME={alarmtime},
  NMEASURE={nmeasures},
  NALARM={nalarms},
  NWARNING={nwarnings},
  STATE={AlarmState},
  COMMENT='{Comment}',
  FAILTIME={failtime},
  ACKFLAG={fAckTriggered},
  SEQNO={SequenceNumber}
where NAME='{name}';
```

- Updates overall sample point state
- Includes measurement and alarm counts
- Clears `fAckTriggered` flag after writing
- Marks sample point as not changed (`fChanged = 0`)

Sources: `src/monitor/monitor.cpp` 349-451

Command Routing and Dispatcher Integration

Command Flow from HMI to Field Device


Operator
Dispatcher
Monitor
fifo_control_dir
configdb
Protocol Driver
Field Device
Confirmation dialog
Confirm
DoExec(CMD_SEND_COMMAND_TO_UNIT)
ReceivedNotify(CMD_SEND_COMMAND_TO_UNIT)
Parse notification data
Write to FIFO
Forward to specific driver
Execute protocol command
Acknowledgment
Publish result to middleware

ReceivedNotify Handler

The `ReceivedNotify()` method processes dispatcher notifications `src/monitor/monitor.cpp` 1313-1481:

Key Notification Types:

- `CMD_MONITOR_STOP` [line 1321]: Calls `Stop()`, broadcasts `MONITOR_STOPPED_NOTIFY`
- `CMD_MONITOR_START` [line 1328]: Reloads configuration, calls `Start()`
- `CMD_SEND_COMMAND_TO_UNIT` [line 1354]: Routes command to driver via `DoCommand` signal
- `CMD_ACK_ALARM` [line 1395]: Acknowledges specific sample point alarm
- `CMD_ACK_ALL` [line 1405]: Acknowledges all active alarms

Command Structure:

Commands carry additional data via `dispatcher_extra_params` structure:

```
typedef struct _dispatcher_extra_params {
    char string1[30];      // Unit name or sample point name
    char string2[30];      // Additional parameter
    char string3[20];      // Additional parameter
    cp56time2a time_stamp; // Command timestamp
    double value;          // Command value
    unsigned char iec_command_type; // IEC 60870-5 type
} dispatcher_extra_params;
```

`src/common/dispatch.h` 89-96

Global FIFO Command Processing

The `get_items_from_global_fifo()` method retrieves commands from drivers `src/monitor/monitor.cpp` 684-794:

Processing Loop:

1. Calls `fifo_get()` with 1ms timeout
2. Validates checksum with `clearCrc()`
3. Extracts value based on `iec_type`:
   - `C_SC_TA_1` / `C_SC_NA_1`: Single command
   - `C_DC_TA_1` / `C_DC_NA_1`: Double command
   - `C_SE_TA_1` / `C_SE_NA_1`: Setpoint command (short)
   - `C_SE_TB_1` / `C_SE_NB_1`: Setpoint command (scaled)
   - `C_SE_TC_1` / `C_SE_NC_1`: Setpoint command (float)
4. Queries `configdb` for unit mapping via IOA:

   ```
   select NAME,UNIT from TAGS where IOA={ioa};
   ```

5. Forwards command to appropriate driver

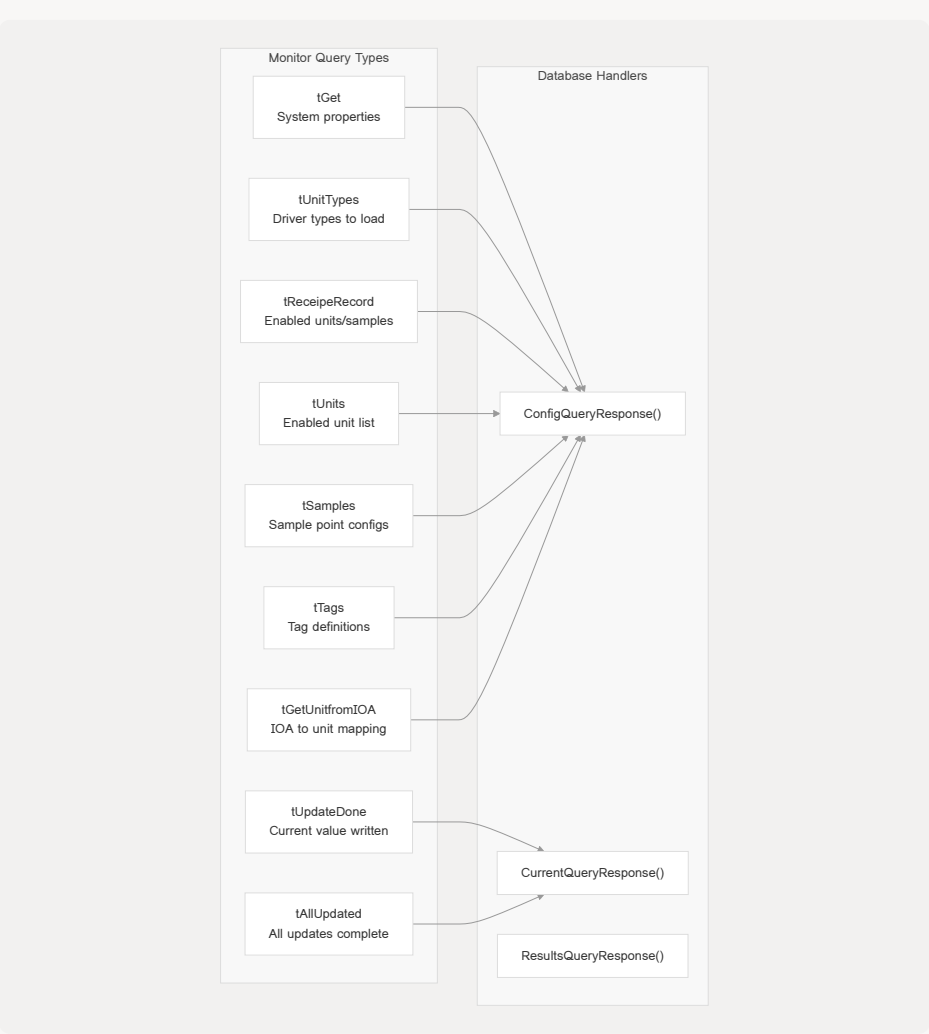
Rate Limiting: Processes maximum 50 items per tick to prevent blocking [line 789-792]

Sources: `src/monitor/monitor.cpp` 684-794 `src/monitor/monitor.cpp` 1313-1481

Database Interactions

The Monitor maintains connections to three primary databases and interacts with them through asynchronous query/response patterns.

Database Query Response Handlers


Monitor Query Types
tGet
System properties
tUnitTypes
Driver types to load
tReceipeRecord
Enabled units/samples
tUnits
Enabled unit list
tSamples
Sample point configs
tTags
Tag definitions
tGetUnitfromIOA
IOA to unit mapping
tUpdateDone
Current value written
tAllUpdated
All updates complete
Database Handlers
ConfigQueryResponse()
CurrentQueryResponse()
ResultsQueryResponse()

ConfigQueryResponse Handler

Processes configuration database responses `src/monitor/monitor.cpp` 906-1304:

tGet [line 915-932]:

- Loads system properties (language, timezone)
- Installs translation if not English

tUnitTypes [line 934-993]:

- Loads distinct driver types
- Creates driver instances via `FindDriver()`
- Populates `drivers` dictionary
- Handles receipe selection

tSamples [line 1051-1202]:

- Loads enabled sample point configurations
- Populates `Results::GetEnabledPoints()` dictionary
- Parses alarm thresholds and configuration
- Creates tag dictionaries within sample points

tTags [line 1204-1276]:

- Loads tag definitions and IOA mappings
- Associates tags with sample points
- Configures tag-specific alarm settings

tGetUnitfromIOA [line 1278-1304]:

- Maps Information Object Address to unit/sample point
- Used for routing commands from FIFO
- Forwards command to appropriate driver

CurrentQueryResponse Handler

Handles real-time database updates `src/monitor/monitor.cpp` 2077-2126:

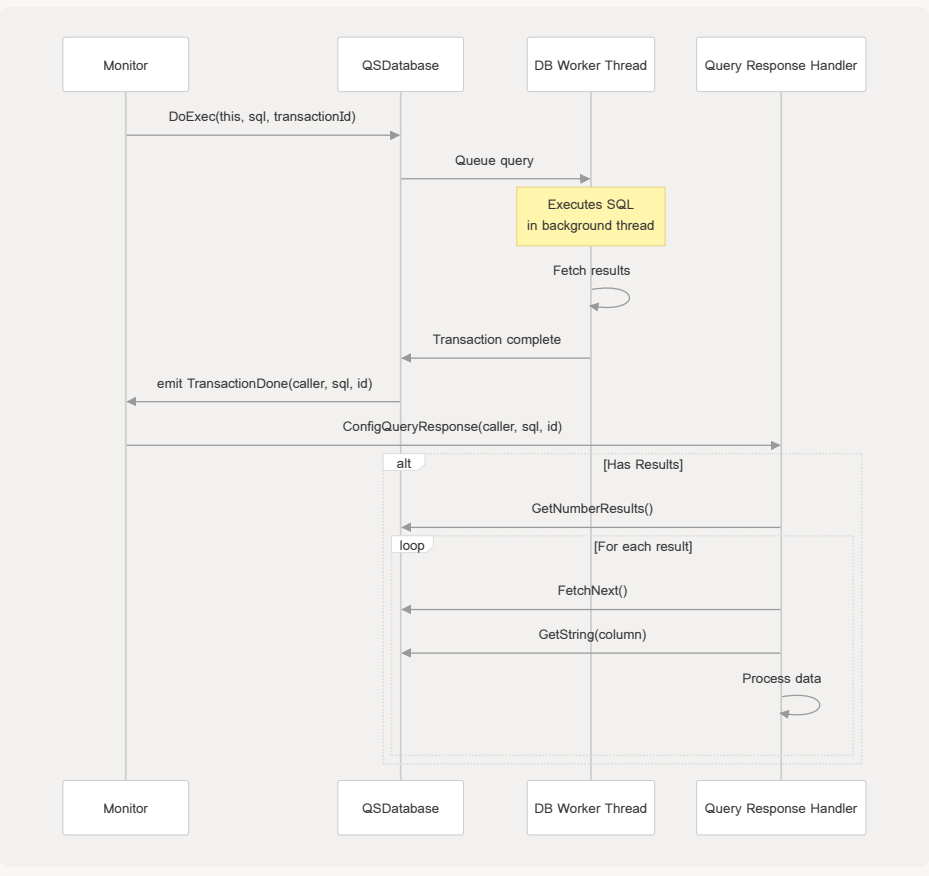
tUpdateDone [line 2081-2094]:

- Confirms single sample point update written
- Logs sequence number written
- Used for debugging/tracing updates

tAllUpdated [line 2096-2125]:

- Confirms all alarm group updates complete
- Triggers dispatcher notification if enabled
- Broadcasts `UPDATE_NOTIFY` to subscribers

Database Transaction Pattern


Monitor
QSDatabase
DB Worker Thread
Query Response Handler
DoExec(this, sql, transactionId)
Queue query
Executes SQL in background thread
Fetch results
Transaction complete
emit TransactionDone(caller, sql, id)
ConfigQueryResponse(caller, sql, id)
alt
[Has Results]
GetNumberResults()
loop
[For each result]
FetchNext()
GetString(column)
Process data

All database operations are **asynchronous**:

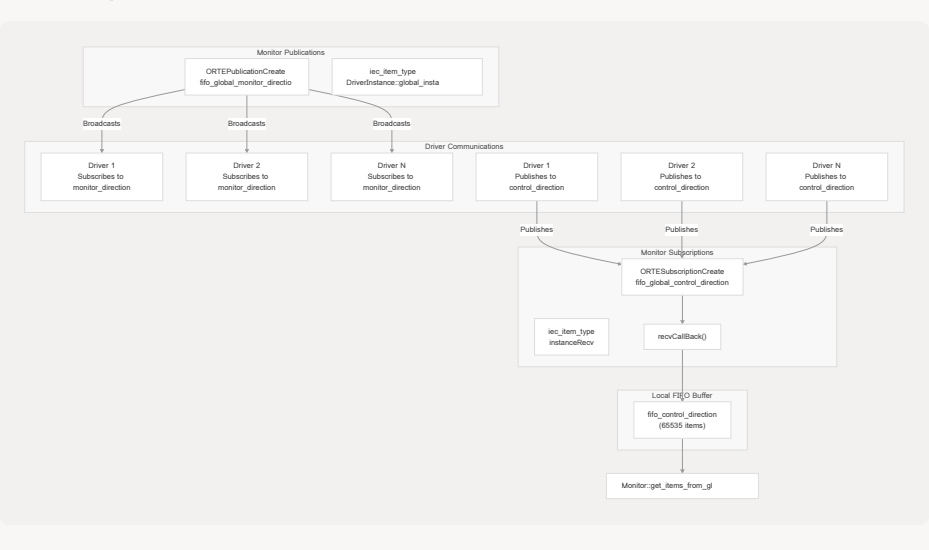
- `DoExec()` queues query and returns immediately
- Worker thread executes query
- `TransactionDone` signal delivers results
- Response handler processes data by transaction ID

Sources: `src/monitor/monitor.cpp` 906-1304 `src/monitor/monitor.cpp` 2077-2126

Middleware Communication

The Monitor uses ORTE (Open Real-Time Ethernet) middleware for publish-subscribe communication with drivers and other processes.

Publisher/Subscriber Architecture


Monitor Publications
Broadcasts
Driver Communications
Driver 1
Driver 2
Driver N
Publishes
Monitor Subscriptions
recvCallBack()
Local FIFO Buffer

ORTE Domain Setup

Domain Creation `src/monitor/monitor.cpp` 182-202:

```
ORTEDomainAppCreate(ORTE_DEFAULT_DOMAIN, &dp, &events, ORTE_FALSE);
```

- Creates application domain with default properties
- Registers `iec_item_type` data type for communication
- Sets up failure callbacks via `events.onRegFail`

Global Publisher `src/monitor/monitor.cpp` 207-218:

```
DriverInstance::global_publisher = ORTEPublicationCreate(
    domain,
    "fifo_global_monitor_direction",  // Topic name
    "iec_item_type",                  // Data type
    &(DriverInstance::global_instanceSend),  // Data instance
    &persistence,                     // 5 second persistence
    strength,
    NULL, NULL, NULL
);
```

- Published by Monitor, subscribed by all drivers
- Used for monitoring direction (device → HMI)
- Carries `iec_item` messages with process data

Global Subscriber `src/monitor/monitor.cpp` 220-234:

```
subscriber = ORTESubscriptionCreate(
    domain,
    IMMEDIATE,
    BEST_EFFORTS,
    "fifo_global_control_direction",  // Topic name
    "iec_item_type",                  // Data type
    &instanceRecv,                    // Receive buffer
    &deadline,                        // 3 second deadline
    &minimumSeparation,
    recvCallBack,                     // Callback function
    this,                             // Callback parameter
    smIPAddress
);
```

- Subscribed by Monitor, published by all drivers
- Used for control direction (HMI → device)
- Triggers `recvCallBack()` on new data

Receive Callback

The `recvCallBack()` function is invoked by ORTE when data arrives `src/monitor/monitor.cpp` 116-139:

Callback Flow:

1. Receives `ORTERecvInfo` and `iec_item_type` instance
2. On `NEW_DATA` event:
   - Calls `rebuild_iec_item_message()` to reconstruct `iec_item`
   - Validates checksum
   - Writes to `fifo_control_direction` FIFO
3. Monitor's `get_items_from_global_fifo()` retrieves items from FIFO during tick

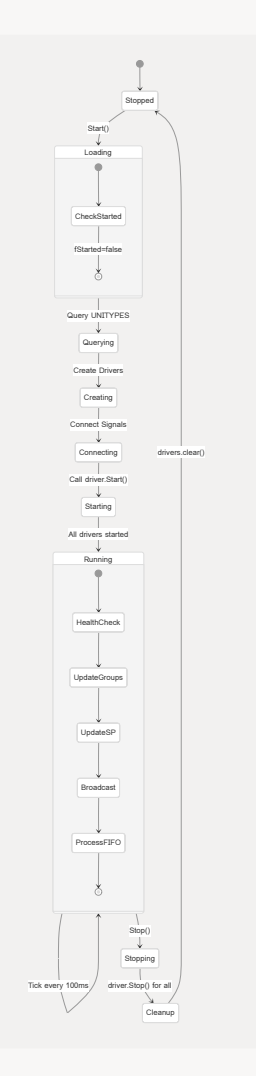
Buffering Strategy:

- ORTE callback writes to FIFO immediately (interrupt context)
- Monitor reads from FIFO during tick (application context)
- Decouples real-time middleware from application processing
- FIFO size: 65535 items (`MAX_FIFO_SIZE`)

Sources: `src/monitor/monitor.cpp` 116-139 `src/monitor/monitor.cpp` 173-235

Lifecycle Management

Start Sequence


Stopped
Start()
Loading
CheckStarted
fStarted=false
Query UNITTYPES
Querying
Create Drivers
Creating
Connect Signals
Connecting
drivers.clear()
Call driver Start()
Starting
All drivers started
Running
HealthCheck
UpdateGroups
UpdateSP
Broadcast
ProcessFIFO
Stop()
Stopping
Tick every 100ms
driver.Stop() for all
Cleanup

Start() Method `src/monitor/monitor.cpp` 803-865:

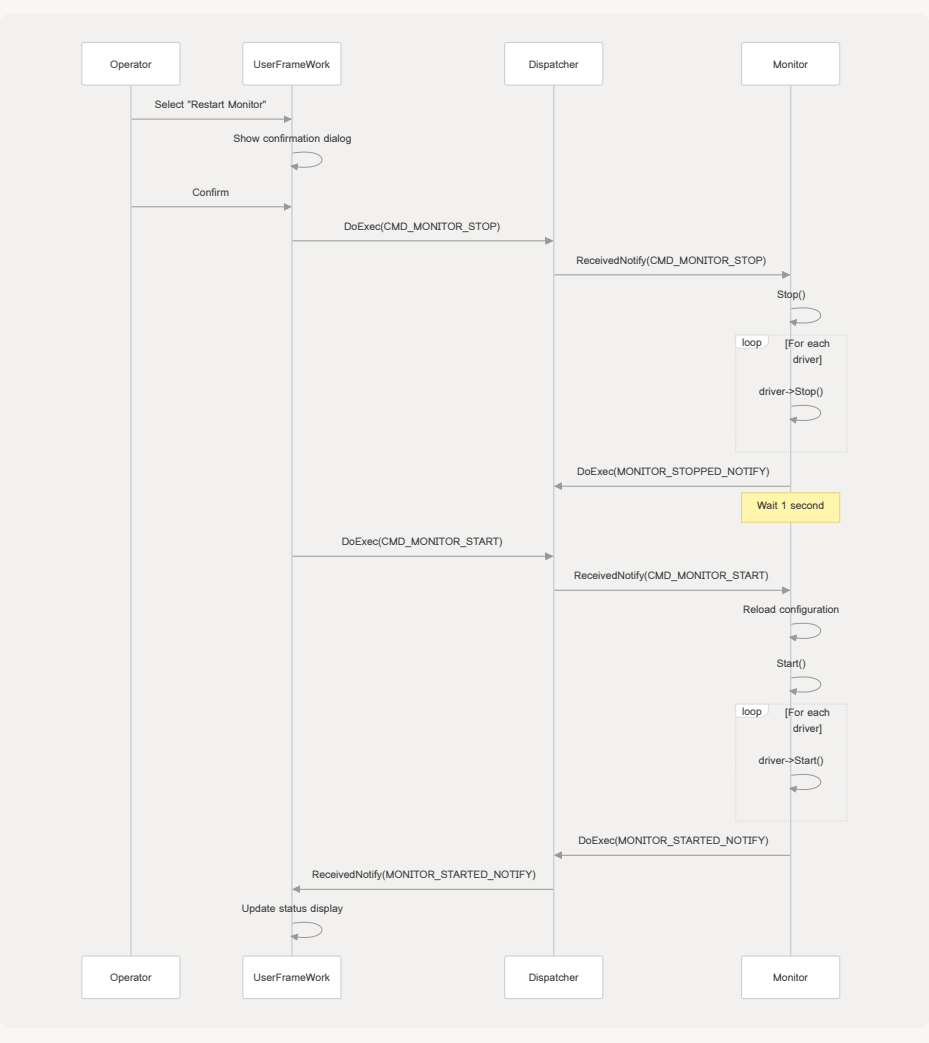
1. **Precondition Check**: Only starts if `!fStarted`
2. **Driver Iteration**: Walks `drivers` dictionary
3. **Driver Start**: Calls `driver->Start()` for each
4. **Logging**: Logs "Starting {unitname}" events
5. **Flag Set**: Sets `fStarted = true`
6. **Notification**: Broadcasts `MONITOR_STARTED_NOTIFY`

Stop() Method `src/monitor/monitor.cpp` 873-898:

1. **Precondition Check**: Only stops if `fStarted`
2. **Driver Iteration**: Walks `drivers` dictionary
3. **Driver Stop**: Calls `driver->Stop()` for each
4. **Dictionary Clear**: Calls `drivers.clear()`
5. **Flag Reset**: Sets `fStarted = false`
6. **Properties Clear**: Clears `DriverInstance::Props`
7. **Notification**: Broadcasts `MONITOR_STOPPED_NOTIFY`

Restart Flow

Restart is initiated from the UI via dispatcher:


Operator
UserFrameWork
Dispatcher
Monitor
Select "Restart Monitor"
Show confirmation dialog
Confirm
DoExec(CMD_MONITOR_STOP)
ReceivedNotify(CMD_MONITOR_STOP)
Stop()
loop
[For each driver]
driver->Stop()
DoExec(MONITOR_STOPPED_NOTIFY)
Wait 1 second
DoExec(CMD_MONITOR_START)
ReceivedNotify(CMD_MONITOR_START)
Reload configuration
Start()
driver->Start()
DoExec(MONITOR_STARTED_NOTIFY)
ReceivedNotify(MONITOR_STARTED_NOTIFY)
Update status display

The restart process in UI code `src/ui/user.cpp` 1197-1213:

```
void UserFrameWork::restart() {
    if(!YESNO(tr("Restart Monitoring"), tr("Restart Monitoring Are You Sure?"))) {
        GetDispatcher()->DoExec(NotificationEvent::CMD_MONITOR_STOP);
        QTimer::singleShot(1000, this, SLOT(StartMonitor()));
    }
}

void UserFrameWork::StartMonitor() {
    GetDispatcher()->DoExec(NotificationEvent::CMD_MONITOR_START);
}
```

Sources: `src/monitor/monitor.cpp` 803-898 `src/ui/user.cpp` 1197-1213

Key Data Structures

iec_item Message Format

The universal data format for all communications:

```
struct iec_item {
    unsigned int iec_type;        // IEC 60870-5-104 type identifier
    struct iec_object iec_obj;    // Type-specific data payload
    unsigned char cause;          // Cause of transmission
    unsigned int msg_id;          // Message sequence number
    unsigned int ioa_control_center; // IOA at control center
    unsigned int casdu;           // Common ASDU address
    unsigned char is_neg;         // Negative confirmation flag
    unsigned char checksum;       // 1-byte integrity check
};
```

Checksum Validation: Uses `clearCrc()` function to verify message integrity `src/monitor/monitor.cpp` 95-104

Sample Point Dictionary

Sample points are stored in `Results::GetEnabledPoints()`:

```
typedef std::map<QString, SamplePoint, std::less<QString>> SamplePointDictWrap;
```

- Key: Sample point name (from `SAMPLE` table)
- Value: `SamplePoint` object with tags, alarms, statistics
- Populated during configuration loading `src/monitor/monitor.cpp` 1051-1202
- Updated during tick processing `src/monitor/monitor.cpp` 633-645

Sources: `src/monitor/monitor.cpp` 80-114 `src/monitor/monitor.cpp` 1051-1202

Integration with Other Components

Results Processor

Monitor triggers results processing via shared data structures:

- **Shared Dictionary**: `Results::GetEnabledPoints()` accessed by both Monitor and Results processor
- **Alarm Evaluation**: Results processor reads sample point states, evaluates thresholds
- **State Updates**: Results processor sets `AlarmState`, `fAckTriggered` flags
- **Notification**: Results processor publishes alarm events via middleware

For details, see 4.2 Results Processing and Alarming.

Calculated Driver

Monitor coordinates with calculated driver:

- **Trigger Mechanism**: Tick broadcasts trigger calculated script execution
- **Value Posting**: Calculated driver posts computed values back to Monitor via middleware
- **FIFO Integration**: Uses same global FIFO for command routing

For details, see 4.3 Calculated Driver.

User Interface

Monitor provides tick notifications to HMI:

- **MONITOR_TICK_NOTIFY**: Sent every tick with `SequenceNumber` `src/monitor/monitor.cpp` 668
- **UPDATE_NOTIFY**: Sent when current values updated `src/monitor/monitor.cpp` 2120
- **HMI Response**: UI queries `currentdb` and refreshes widgets

For HMI details, see 5.1 User Interface Framework and 5.2 HMI Widgets and Displays.

Sources: `src/monitor/monitor.cpp` 668 `src/monitor/monitor.cpp` 2077-2126

Summary

The Monitor Process is the **central orchestrator** of IndigoSCADA's data acquisition infrastructure. Its key responsibilities are:

Responsibility	Implementation	Files
Driver Management	Loads drivers dynamically, manages lifecycle	`monitor.cpp` 803-992
Tick Processing	100ms cycle updates state, checks health	`monitor.cpp` 458-680
Sample Points	Maintains current values, statistics, alarms	`monitor.cpp` 349-451
Command Routing	Routes HMI commands to drivers via IOA	`monitor.cpp` 684-794
Database Sync	Writes to currentdb, reads from configdb	`monitor.cpp` 906-2126
Middleware	Publishes/subscribes via ORTE, FIFOs	`monitor.cpp` 116-235
Notifications	Broadcasts system events via Dispatcher	`monitor.cpp` 1313-1481

Ask Devin about jonathanxavier/IndigoSCADA

Fast

The Monitor's **tick-based architecture** ensures predictable, periodic processing suitable for real-time SCADA applications. Its use of **asynchronous database transactions** and **middleware**

buffering prevents blocking during I/O operations. The **global FIFO and ORTE domain** enable scalable communication with an arbitrary number of protocol drivers.

Sources: `src/monitor/monitor.h` 1-145 `src/monitor/monitor.cpp` 1-2126

Results Processing and Alarming

Relevant source files

Purpose and Scope

This document describes the Results Processing and Alarming subsystem within IndigoSCADA's data acquisition pipeline. The Results Processor is responsible for:

- Receiving measurement data from the Monitor process
- Evaluating alarm conditions based on configured thresholds
- Managing alarm state transitions and alarm groups
- Filing historical data to the results database
- Executing configurable alarm actions (SQL commands or shell scripts)
- Updating current value statistics and quality flags

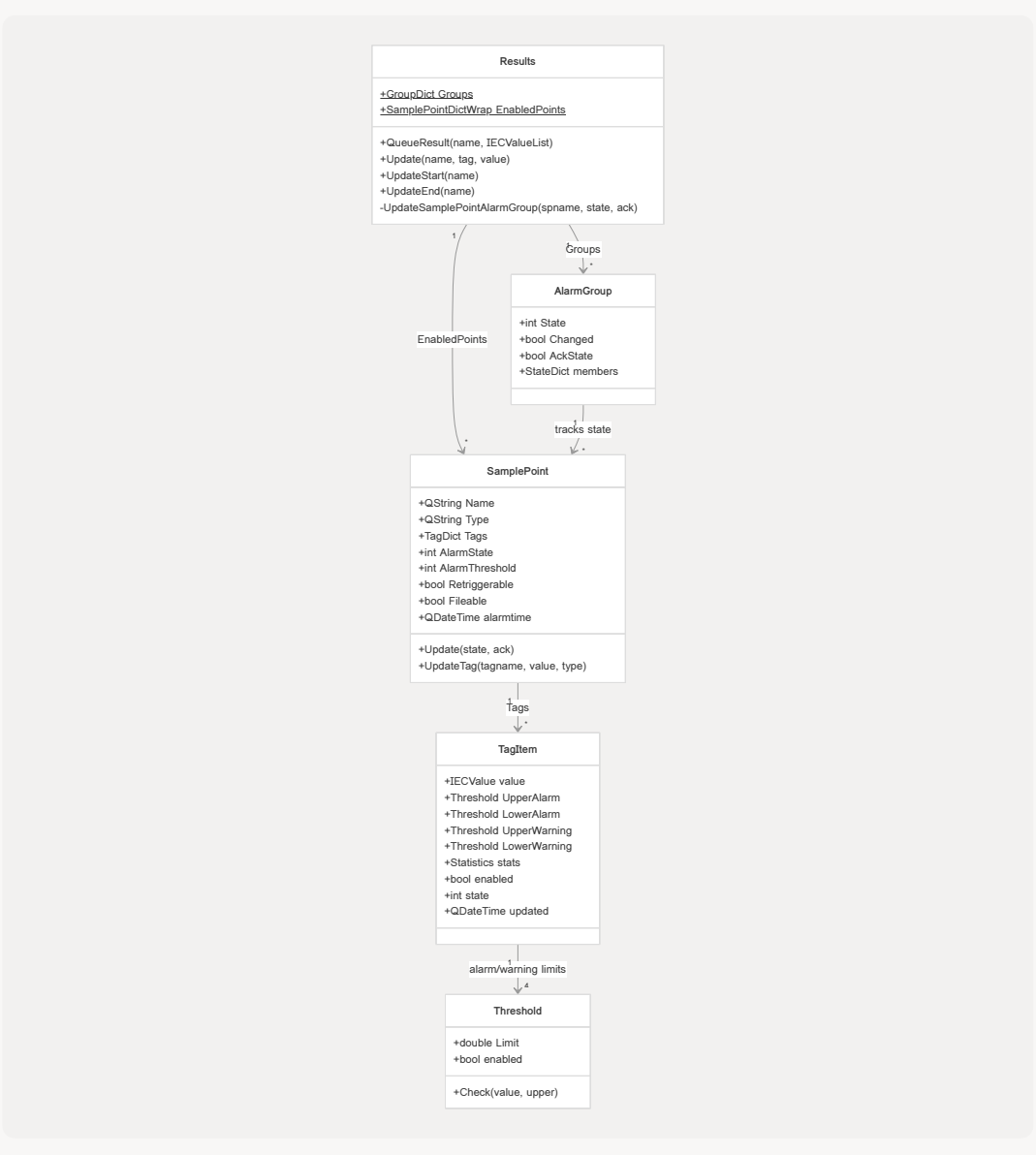
For information about how data reaches the Results Processor, see Monitor Process. For details on how the calculated driver integrates with results processing, see Calculated Driver. For alarm display in the HMI, see Alarm and Event Displays.

Sources: `src/database/results.cpp` 1-895 `src/monitor/monitor.cpp` 1-2529

System Architecture

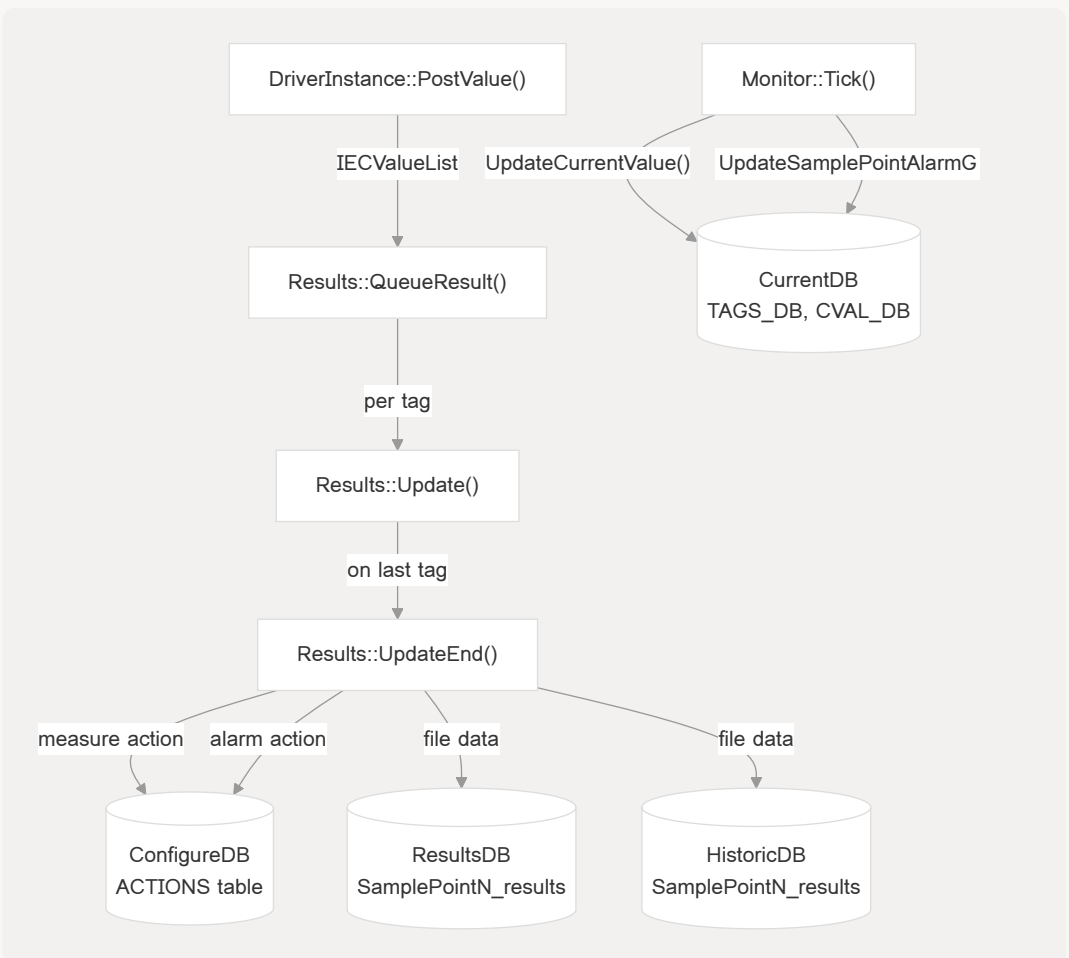
The Results Processing subsystem is centered around the `Results` class, which manages a collection of enabled sample points and evaluates their alarm states. Each sample point contains multiple tags, and each tag has configured threshold limits.

Core Classes


Results
+GroupDict Groups
+SamplePointDictWrap EnabledPoints
+QueueResult(name, IECValueList)
+Update(name, tag, value)
+UpdateStart(name)
+UpdateEnd(name)
-UpdateSamplePointAlarmGroup(spname, state, ack)
Groups
EnabledPoints
AlarmGroup
+int State
+bool Changed
+bool AckState
+StateDict members
tracks state
SamplePoint
+QString Name
+QString Type
+TagDict Tags
+int AlarmState
+int AlarmThreshold
+bool Retriggerable
+bool Fileable
+QDateTime alarmtime
+Update(state, ack)
+UpdateTag(tagname, value, type)
Tags
TagItem
+IECValue value
+Threshold UpperAlarm
+Threshold LowerAlarm
+Threshold UpperWarning
+Threshold LowerWarning
+Statistics stats
+bool enabled
+int state
+QDateTime updated
alarm/warning limits
Threshold
+double Limit
+bool enabled
+Check(value, upper)

Sources: `src/database/results.cpp` 25-305 `src/database/results.h` 1-100

Integration Points


DriverInstance::PostValue()
Monitor::Tick()
IECValueList
UpdateCurrentValue()
UpdateSamplePointAlarmG
Results::QueueResult()
CurrentDB
TAGS_DB, CVAL_DB
per tag
Results::Update()
on last tag
Results::UpdateEnd()
measure action
alarm action
file data
file data
ConfigureDB
ACTIONS table
ResultsDB
SamplePointN_results
HistoricDB
SamplePointN_results

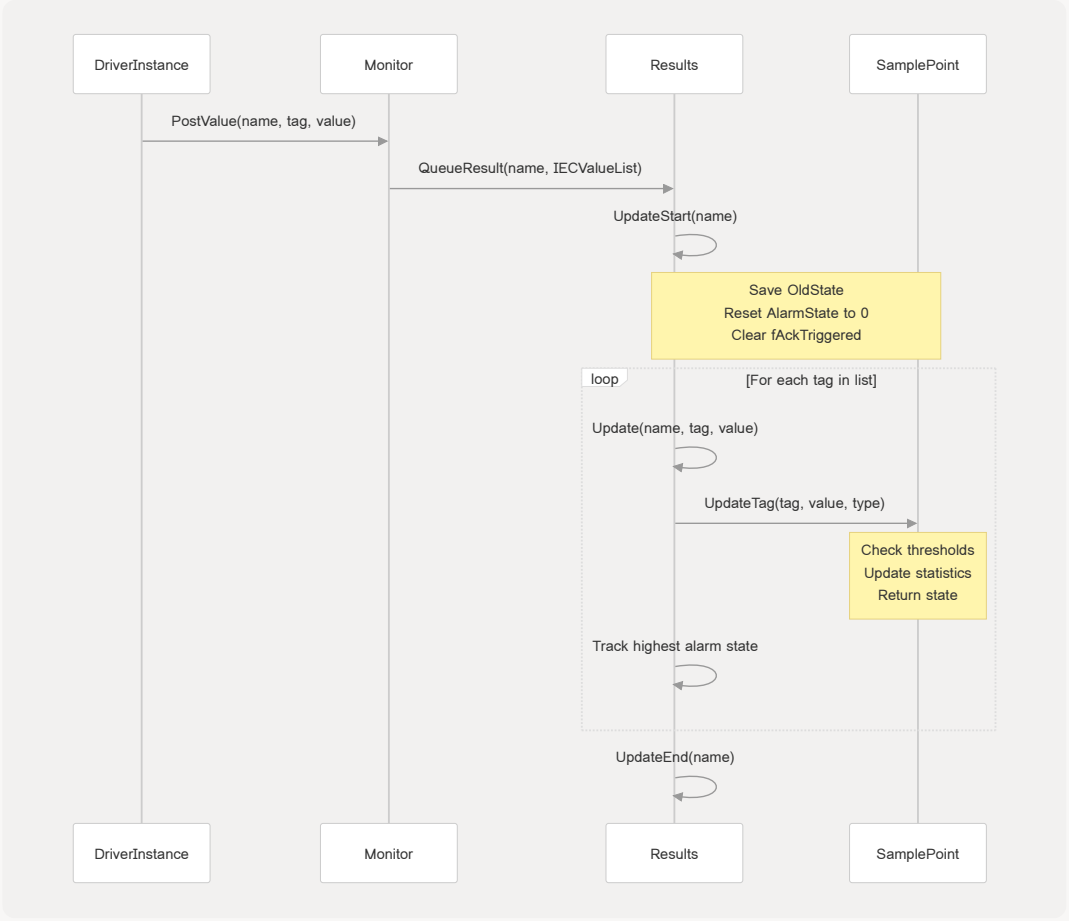
Sources: `src/database/results.cpp` 330-666 `src/monitor/monitor.cpp` 348-450

Data Processing Pipeline

The results processor operates in three phases for each sample point update:

Phase 1: Update Start

When new data arrives from a driver, the Monitor calls `Results::QueueResult()` which initiates processing:

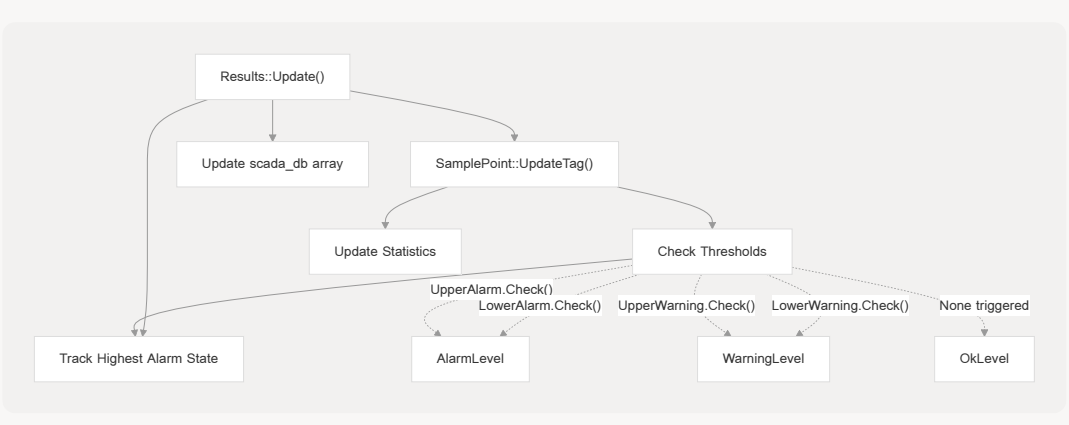

DriverInstance
Monitor
Results
SamplePoint
PostValue(name, tag, value)
QueueResult(name, IECValueList)
UpdateStart(name)
Save OldState
Reset AlarmState to 0
Clear fAckTriggered
loop
[For each tag in list]
Update(name, tag, value)
UpdateTag(tag, value, type)
Check thresholds
Update statistics
Return state
Track highest alarm state
UpdateEnd(name)

The `UpdateStart()` method prepares the sample point by saving the previous alarm state and resetting the current state to the lowest level. This allows the subsequent tag updates to determine the new overall alarm state.

Sources: `src/database/results.cpp` 330-347 `src/database/results.cpp` 482-495

Phase 2: Tag Value Updates

For each tag in the incoming value list, `Results::Update()` processes the value:

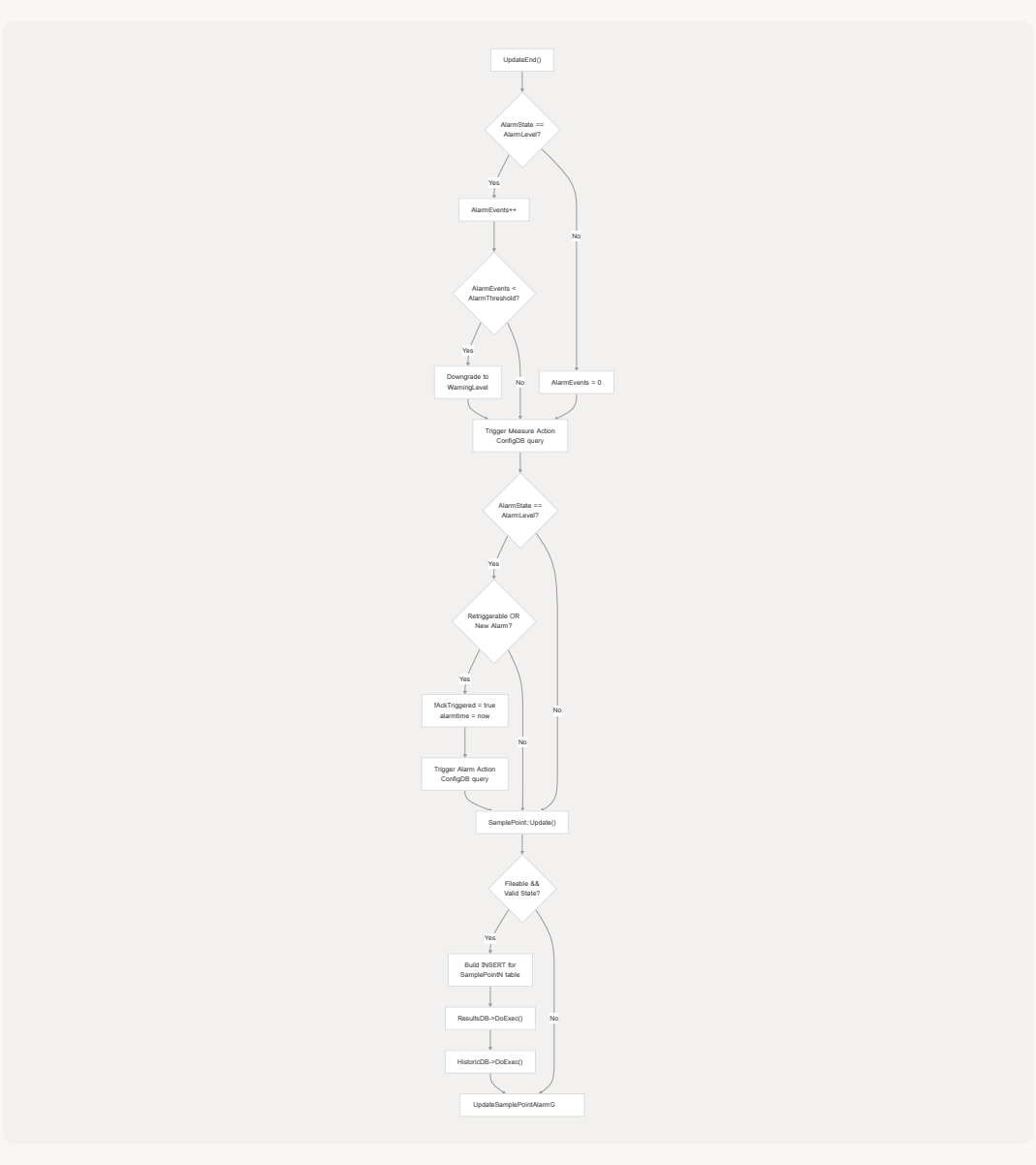

Results::Update()
Update scada_db array
SamplePoint::UpdateTag()
Update Statistics
Check Thresholds
UpperAlarm.Check()
LowerAlarm.Check()
UpperWarning.Check()
LowerWarning.Check()
None triggered
Track Highest Alarm State
AlarmLevel
WarningLevel
OkLevel

The `UpdateTag()` method checks the value against configured thresholds and returns the alarm state for that tag. The `Update()` method tracks the highest alarm state across all tags.

Sources: `src/database/results.cpp` 504-536 `src/database/results.cpp` 771-853

Phase 3: Update Finalization

The `UpdateEnd()` method completes processing by evaluating alarm thresholds, triggering actions, and filing data:


UpdateEnd()
AlarmState == AlarmLevel?
Yes
No
AlarmEvents++
AlarmEvents < AlarmThreshold?
Downgrade to WarningLevel
AlarmEvents = 0
Trigger Measure Action ConfigDB query
AlarmState == AlarmLevel?
Retriggerable OR New Alarm?
fAckTriggered = true alarmtime = now
Trigger Alarm Action ConfigDB query
SamplePoint::Update()
Fileable && Valid State?
Build INSERT for SamplePointN table
ResultsDB->DoExec()
HistoricDB->DoExec()
UpdateSamplePointAlarmG

Sources: `src/database/results.cpp` 546-666

Alarm State Evaluation

Alarm Levels

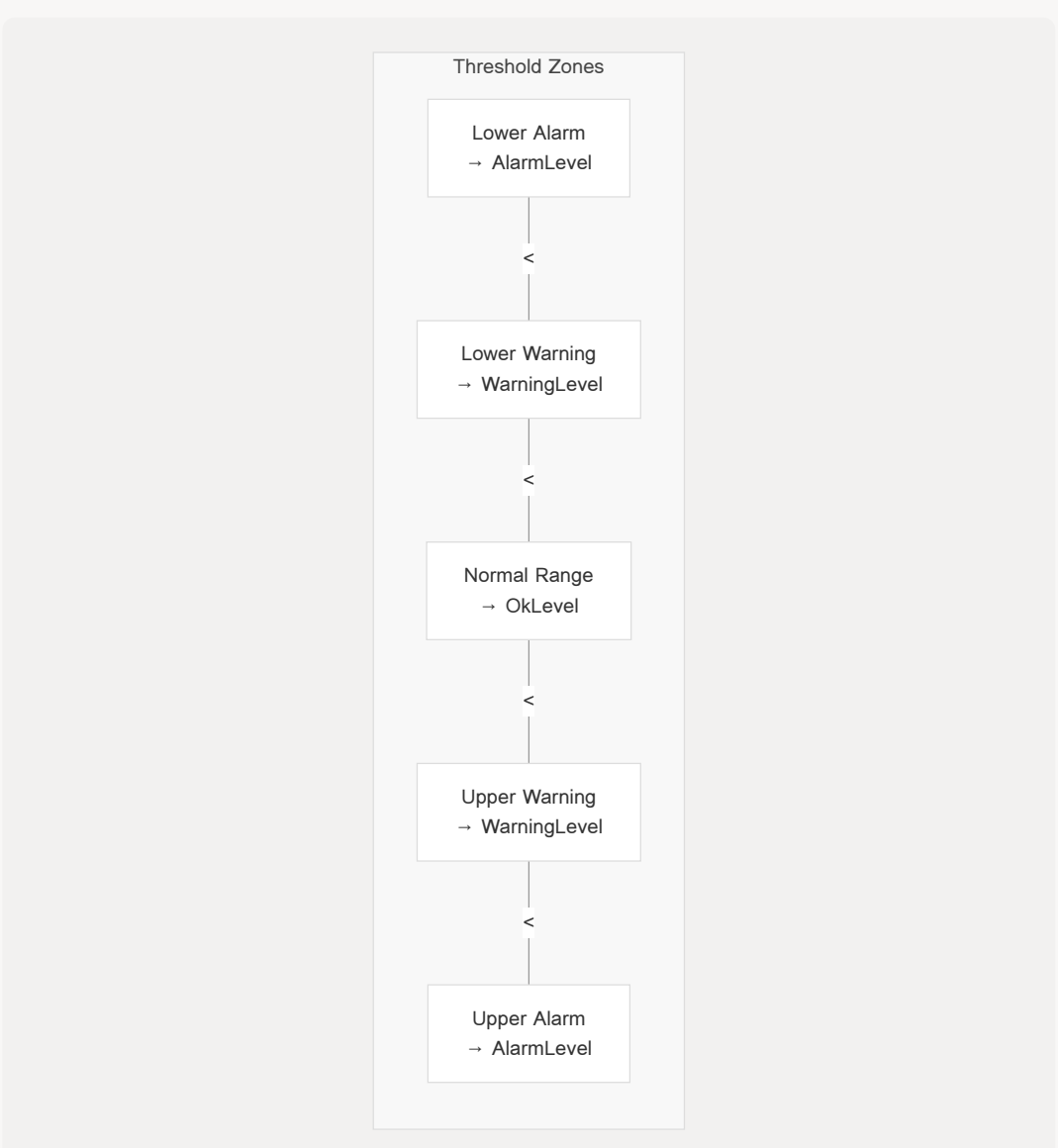
IndigoSCADA defines five alarm states for sample points, evaluated during `UpdateTag()`:

State	Value	Description
NoLevel	0	No data received
OkLevel	1	Value within normal range
WarningLevel	2	Value exceeded warning threshold
AlarmLevel	3	Value exceeded alarm threshold
FailureLevel	4	Communication failure or bad quality

Sources: `src/database/sptypes.h` 1-100 `src/database/results.cpp` 771-853

Threshold Checking Logic

For analog measurement types (`M_ME_NA_1`, `M_ME_TD_1`, `M_ME_TF_1`, etc.), each tag can have four configured thresholds:


Threshold Zones
Lower Alarm
→ AlarmLevel
<
Lower Warning
→ WarningLevel
<
Normal Range
→ OkLevel
<
Upper Warning
→ WarningLevel
<
Upper Alarm
→ AlarmLevel

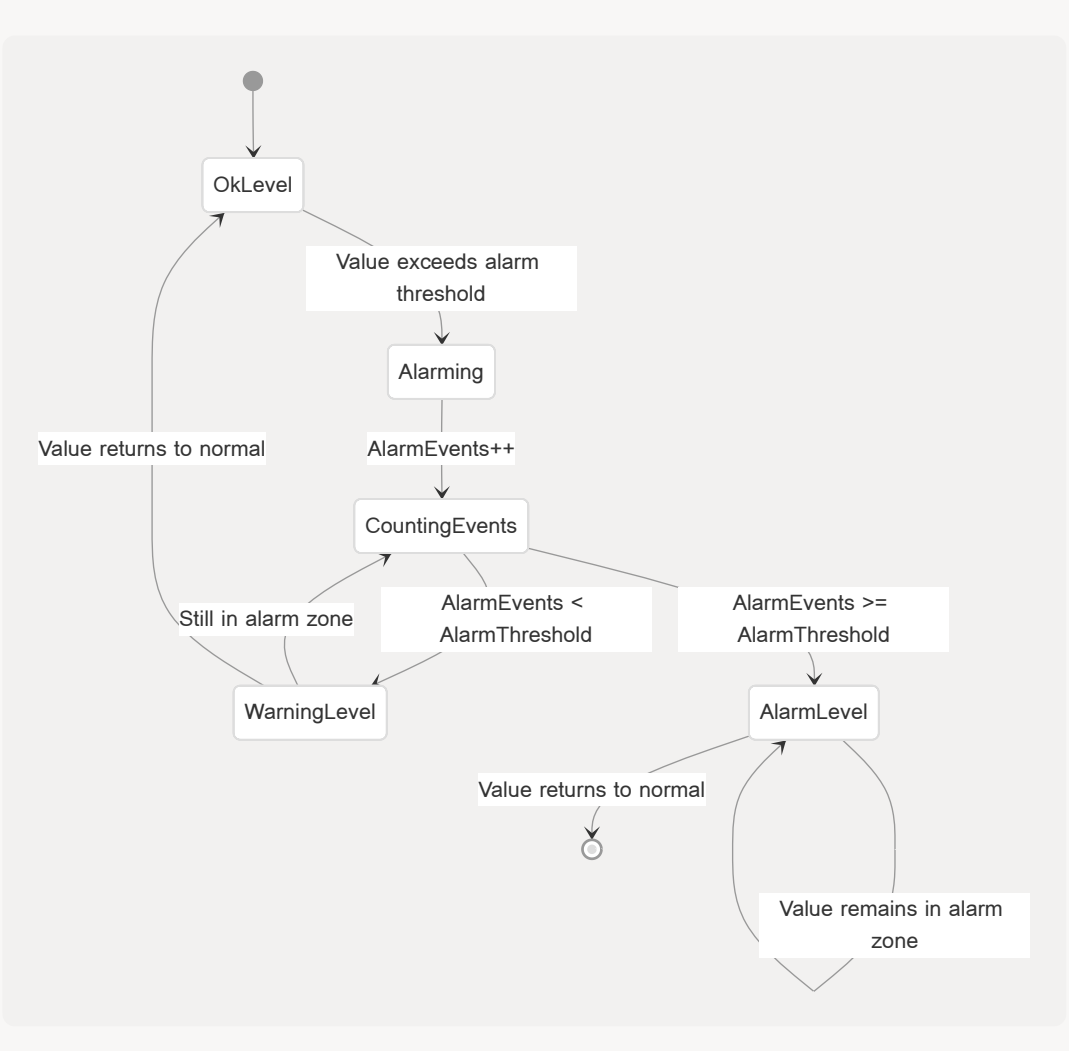
The threshold check order in code (from `src/database/results.cpp` 802-834):

1. **Upper Alarm**: If `UpperAlarm.Check(value, true)` returns true → `AlarmLevel`
2. **Upper Warning**: Else if `UpperWarning.Check(value, true)` returns true → `WarningLevel`
3. **Lower Alarm**: If `LowerAlarm.Check(value, false)` returns true → `AlarmLevel`
4. **Lower Warning**: Else if `LowerWarning.Check(value, false)` returns true → `WarningLevel`
5. **Default**: If no thresholds triggered → `OkLevel`

Sources: `src/database/results.cpp` 792-853

Alarm Threshold Debouncing

To prevent alarm flapping, IndigoSCADA implements alarm threshold counting. When a value enters the alarm zone, it must remain there for multiple consecutive samples before the alarm is confirmed:


OkLevel
Value exceeds alarm threshold
Alarming
Value returns to normal
AlarmEvents++
CountingEvents
Still in alarm zone
AlarmEvents <
AlarmThreshold
AlarmEvents >=
AlarmThreshold
WarningLevel
AlarmLevel
Value returns to normal
Value remains in alarm zone

From `src/database/results.cpp` 555-566:

```
if((*i).second.AlarmState == AlarmLevel)
{
    (*i).second.AlarmEvents++;
    if((*i).second.AlarmEvents < (*i).second.AlarmThreshold)
    {
        (*i).second.AlarmState = WarningLevel;
    };
}
else
{
    (*i).second.AlarmEvents = 0; // reset
};
```

Sources: `src/database/results.cpp` 555-566

Retriggerable Alarms

Sample points can be configured as `Retriggerable`, which controls acknowledgment behavior:

Retriggerable	Behavior
`true`	Every time the value enters the alarm zone, `fAckTriggered` is set, requiring re-acknowledgment
`false`	Only the first transition to alarm state triggers `fAckTriggered`

From `src/database/results.cpp` 581-590:

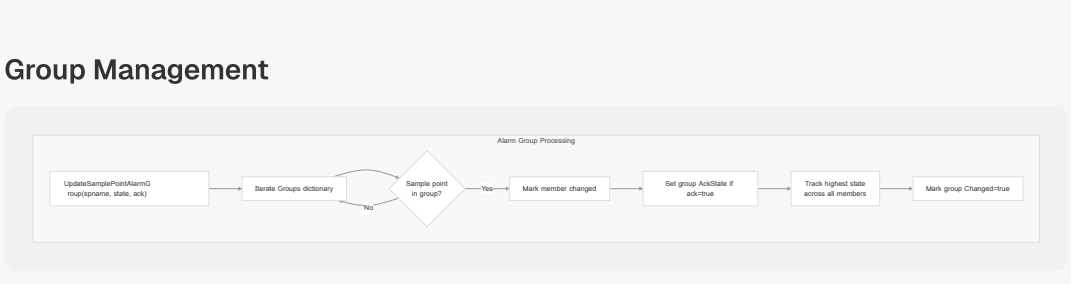
```
if((*i).second.Retriggerable)
{
    (*i).second.fAckTriggered = true;
};

if((*i).second.AlarmState > (*i).second.OldState)
{
    (*i).second.fAckTriggered = true; // always trigger on state increase
};
```

Sources: `src/database/results.cpp` 575-601

Alarm Groups

Alarm groups allow multiple sample points to be managed as a logical unit, with the group state reflecting the highest alarm level of any member.

Group Management


Alarm Group Processing
UpdateSamplePointAlarmG (spname, state, ack)
Iterate Groups dictionary
Sample point in group?
Yes
No
Mark member changed
Set group AckState if ack=true
Track highest state across all members
Mark group Changed=true

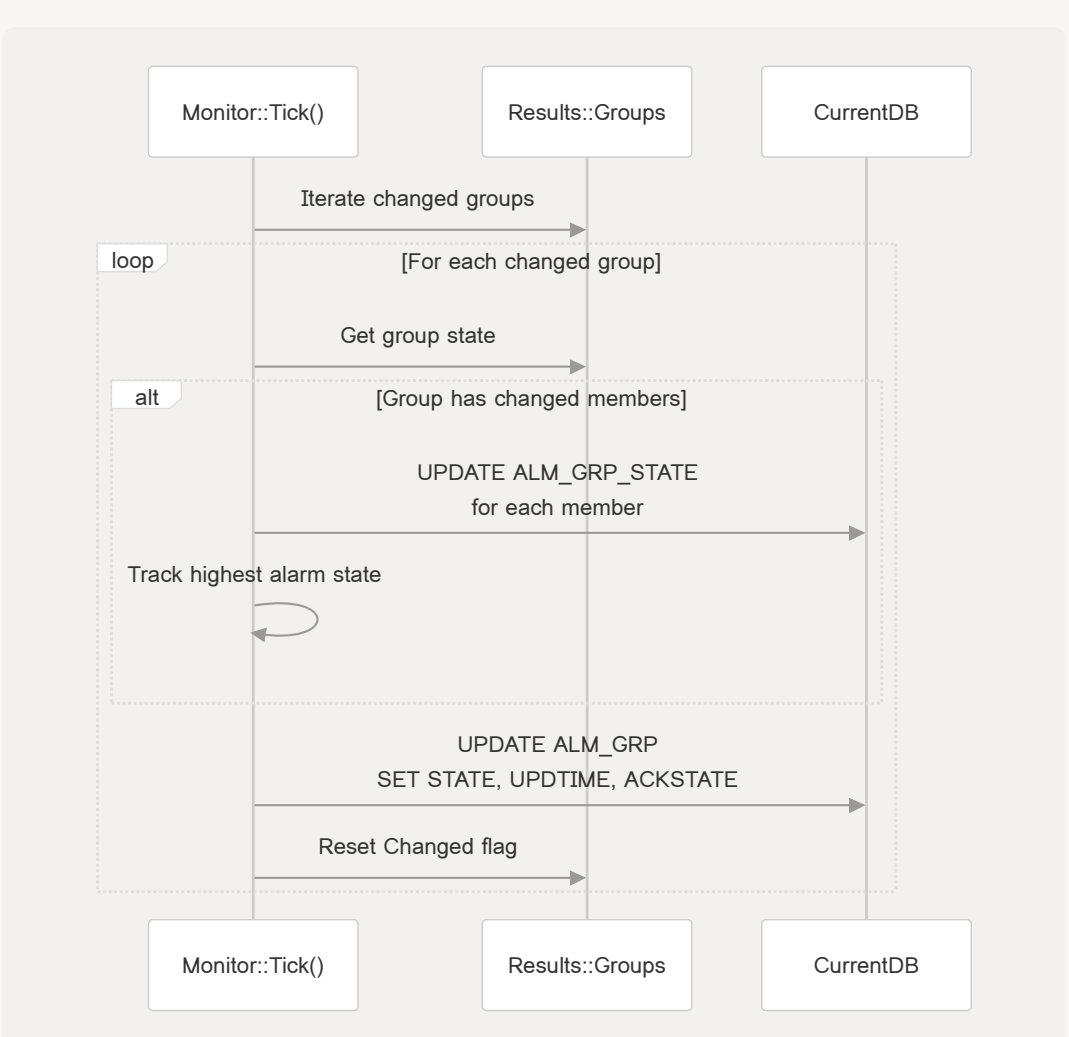
From `src/database/results.cpp` 445-473:

```
GroupDict::iterator i = Groups.begin();
for(; !(i == Groups.end()); i++)
{
    StateDict::iterator j = (*i).second.find(spname);
    if(!(j == (*i).second.end()))
    {
        (*j).second = true; // mark member changed
        if(ack) (*i).second.AckState = true;
        if((*i).second.State < state)
        {
            (*i).second.State = state; // track highest
        };
        (*i).second.Changed = true;
    };
};
```

Sources: `src/database/results.cpp` 445-473

Monitor Integration

The Monitor's `Tick()` method processes updated alarm groups and writes their state to the `ALM_GRP` and `ALM_GRP_STATE` tables:


Monitor::Tick()
Results::Groups
CurrentDB
Iterate changed groups
loop
[For each changed group]
Get group state
alt
[Group has changed members]
UPDATE ALM_GRP_STATE
for each member
Track highest alarm state
UPDATE ALM_GRP
SET STATE, UPDTIME, ACKSTATE
Reset Changed flag

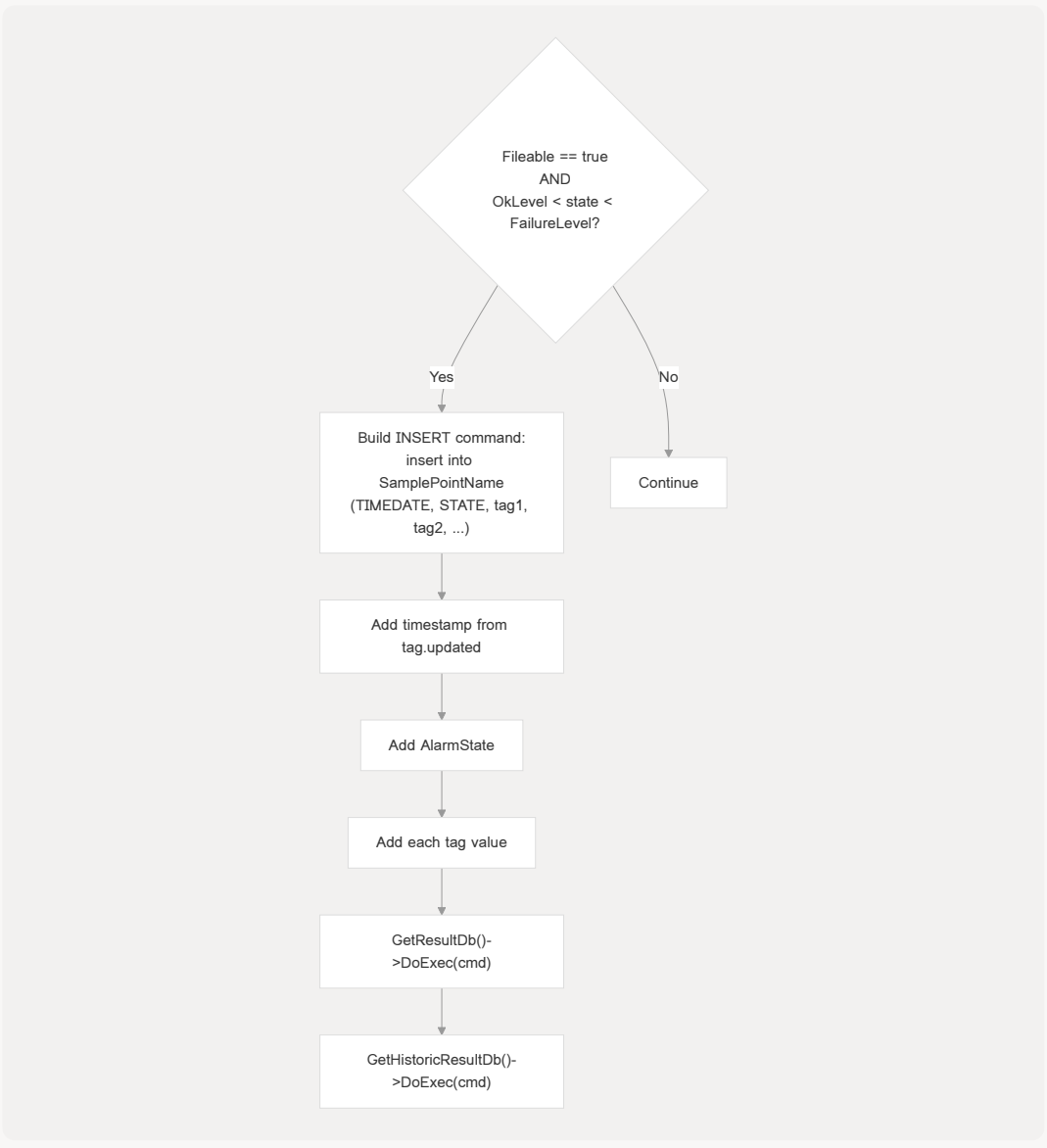
From `src/monitor/monitor.cpp` 560-628 the group processing logic updates the `ALM_GRP_STATE` table for each member sample point and the `ALM_GRP` table for the group itself.

Sources: `src/monitor/monitor.cpp` 560-628

Data Filing

Historical Data Storage

When the `Fileable` flag is set for a sample point, and the alarm state is valid (between `OkLevel` and `AlarmLevel`), the results processor files the data to historical storage:


Fileable == true
AND
OkLevel < state <
FailureLevel?
Yes
No
Build INSERT command:
insert into
SamplePointName
(TIMEDATE, STATE, tag1,
tag2, ...)
Continue
Add timestamp from
tag.updated
Add AlarmState
Add each tag value
GetResultDb()-
>DoExec(cmd)
GetHistoricResultDb()-
>DoExec(cmd)

From `src/database/results.cpp` 608-661:

```
if((*i).second.Fileable)
{
    if(((*i).second.AlarmState > NoLevel) &&
       ((*i).second.AlarmState < FailureLevel))
    {
        QString cmd = "insert into " + name + " (TIMEDATE,STATE";
        // Add tag column names
        for tag in Tags:
            cmd += "," + tagname;

        cmd += ") values( " + timestamp;
        cmd += "," + AlarmState;

        // Add tag values
        for tag in Tags:
            cmd += "," + value;

        cmd += ");";

        GetResultDb()->DoExec(0,cmd,0);      // FastDB
        GetHistoricResultDb()->DoExec(0,cmd,0); // GigaBASE
    };
};
```

Each sample point gets its own table named after the sample point (e.g., `SamplePoint1_results`), with columns for `TIMEDATE`, `STATE`, and each configured tag.

Sources: `src/database/results.cpp` 608-661

Database Targets

Database	Type	Purpose	Filed From
`resultsdb`	FastDB	Short-term historical storage	`GetResultDb()->DoExec()`
`historicdb`	GigaBASE	Long-term historical archive	`GetHistoricResultDb()->DoExec()`
`currentdb`	FastDB	Current values and statistics	`Monitor::UpdateCurrentValue()`

The dual-database strategy allows fast recent data access (FastDB) while maintaining long-term archives (GigaBASE with better compression).

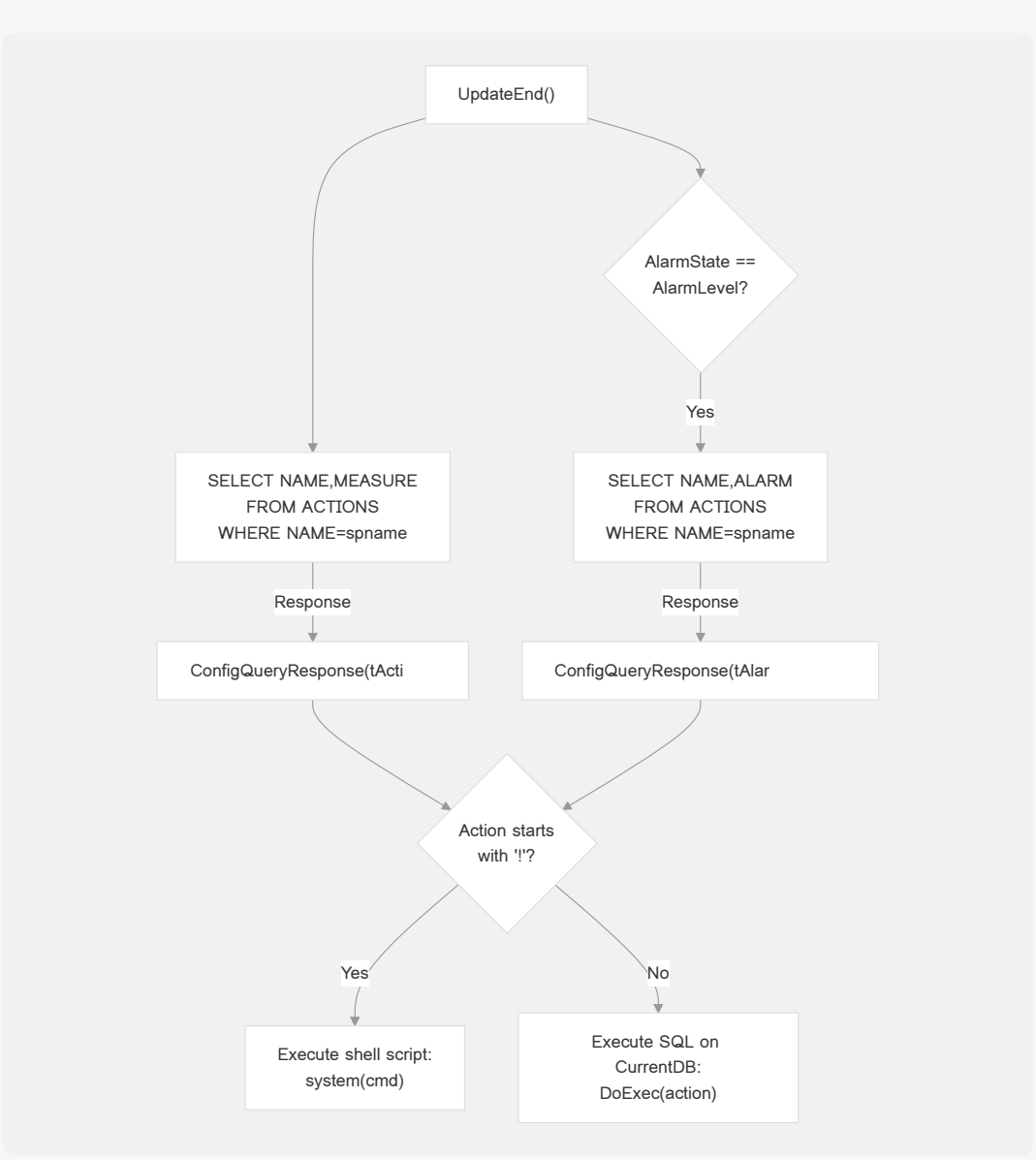
Sources: `src/database/results.cpp` 651-659 `src/monitor/monitor.cpp` 348-450

Alarm Actions

Action Configuration

Alarm actions are configured in the `ACTIONS` table with two types:

1. **MEASURE** actions - Executed on every value update
2. **ALARM** actions - Executed when alarm condition is triggered


UpdateEnd()
AlarmState ==
AlarmLevel?
Yes
SELECT NAME,MEASURE
FROM ACTIONS
WHERE NAME=spname
SELECT NAME,ALARM
FROM ACTIONS
WHERE NAME=spname
Response
Response
ConfigQueryResponse(tActi
ConfigQueryResponse(tAlar
Action starts
with '!'?
Yes
No
Execute shell script:
system(cmd)
Execute SQL on
CurrentDB:
DoExec(action)

From `src/database/results.cpp` 571-601:

```
// Trigger measure action for every update
GetConfigureDb()->DoExec(this,
    "select NAME,MEASURE from ACTIONS where NAME='" + name + "';",
    tActions);

// Trigger alarm action when entering alarm state
if((*i).second.AlarmState == AlarmLevel)
{
    if((*i).second.fAckTriggered)
    {
        GetConfigureDb()->DoExec(this,
            "select NAME,ALARM from ACTIONS where NAME='" + name + "';",
            tAlarmActions);
    };
};
```

Sources: `src/database/results.cpp` 571-601 `src/database/results.cpp` 363-435

Action Execution

From `src/database/results.cpp` 363-435 action execution supports two modes:

1. **Shell Commands** (action starts with `!`):

   ```
   if(s[0] == '!')
   {
       QString cmd = s.mid(1) + "&";
       system((const char *)cmd); // Background process
   }
   ```

2. **SQL Commands** (no `!` prefix):

   ```
   GetCurrentDb()->DoExec(0, action_sql, 0);
   ```

This allows flexible responses such as:

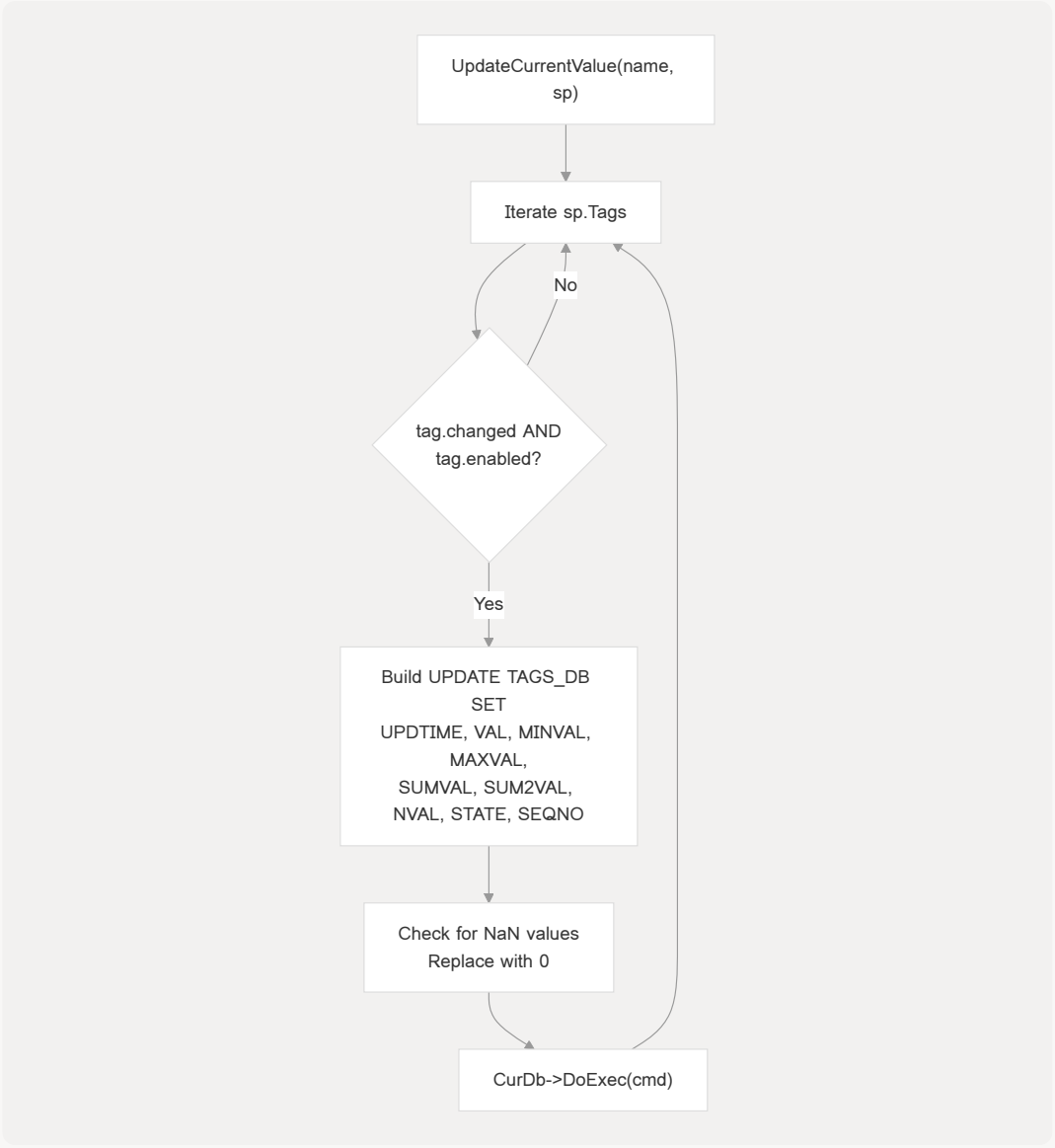
- Setting digital outputs via SQL updates to control relays
- Executing custom scripts for notifications (email, SMS, etc.)
- Updating calculated values or derived states

Sources: `src/database/results.cpp` 375-428

Current Value Updates

The Monitor process updates the current value database after results processing completes:

TAGS_DB Table Updates


UpdateCurrentValue(name,
sp)
Iterate sp.Tags
No
tag.changed AND
tag.enabled?
Yes
Build UPDATE TAGS_DB
SET
UPDTIME, VAL, MINVAL,
MAXVAL,
SUMVAL, SUM2VAL,
NVAL, STATE, SEQNO
Check for NaN values
Replace with 0
CurDb->DoExec(cmd)

From `src/monitor/monitor.cpp` 349-425:

```
void Monitor::UpdateCurrentValue(const QString &name, SamplePoint &sp)
{
    SamplePoint::TagDict::iterator i = sp.Tags.begin();
    for(; !(i == sp.Tags.end()); i++)
    {
        if((*i).second.changed && (*i).second.enabled)
        {
            (*i).second.changed = 0;
            cmd = "update TAGS_DB set ";
            cmd += "UPDTIME=" + ISO_TIMESTAMP;
            cmd += ",VAL=" + value;
            cmd += ",MINVAL=" + stats.Min();
            cmd += ",MAXVAL=" + stats.Max();
            cmd += ",SUMVAL=" + stats.sum();
            cmd += ",SUM2VAL=" + stats.sum2();
            cmd += ",NVAL=" + stats.samples();
            cmd += ",STATE=" + state;
            cmd += ",SEQNO=" + SequenceNumber;
            cmd += " where NAME='" + name + "' and TAGNAME='" + tagname + "';";
            CurDb->DoExec(0, cmd, 0);
        };
    };
}
```

Sources: `src/monitor/monitor.cpp` 349-425

CVAL_DB Table Updates

After updating individual tags, the Monitor updates the sample point's overall state in `CVAL_DB`:

From `src/monitor/monitor.cpp` 432-446:

```
cmd = "update CVAL_DB set UPDTIME=" + updated_time
    + ",ALMTIME=" + alarm_time
    + ",NMEASURE=" + nmeasures
    + ",NALARM=" + nalarms
    + ",NWARNING=" + nwarnings
    + ",STATE=" + AlarmState
    + ",COMMENT='" + EscapeSQLText(Comment) + "'"
    + ",FAILTIME=" + failtime;

if(sp.fAckTriggered)
{
    cmd += ",ACKFLAG=" + fAckTriggered;
    sp.fAckTriggered = 0;
};

cmd += ",SEQNO=" + SequenceNumber;
cmd += " where NAME='" + name + "';";
CurDb->DoExec(this, cmd, tUpdateDone, SequenceNumber);
```

The `SEQNO` field provides a machine-independent ordering of updates across the system.

Sources: `src/monitor/monitor.cpp` 432-446

Quality Flags and IEC Values

IEC Value Types

The results processor handles multiple IEC 60870-5 data types through the `IECValue` union structure:

Type	Description	Extracted By
`M_SP_NA_1`	Single-point without time	`v1_q.sp`
`M_DP_NA_1`	Double-point without time	`v3_q.dp`
`M_ME_NA_1`	Normalized measured value	`v9_q.mv`
`M_ME_NB_1`	Scaled measured value	`v11_q.mv`
`M_ME_NC_1`	Short floating point	`v13_q.mv`
`M_SP_TB_1`	Single-point with time	`v30_q.sp`
`M_ME_TD_1`	Normalized measured value with time	`v34_q.mv`
`M_ME_TF_1`	Short floating point with time	`v36_q.mv`

Sources: `src/database/results.cpp` 30-77

Quality Flag Processing

The `get_quality_of_iec_value()` function extracts quality information from IEC values:

From `src/database/results.cpp` 134-211:

```
int get_quality_of_iec_value(IECValue &v)
{
    switch(v.type)
    {
        case M_SP_NA_1: return v.v1_q.iv;
        case M_ME_NA_1: return v.v9_q.iv;
        case M_ME_TD_1: return v.v34_q.iv;
        // ... etc for all types
        default: return 0; // good quality
    }
}
```

The `iv` (invalid) flag indicates whether the value is valid:

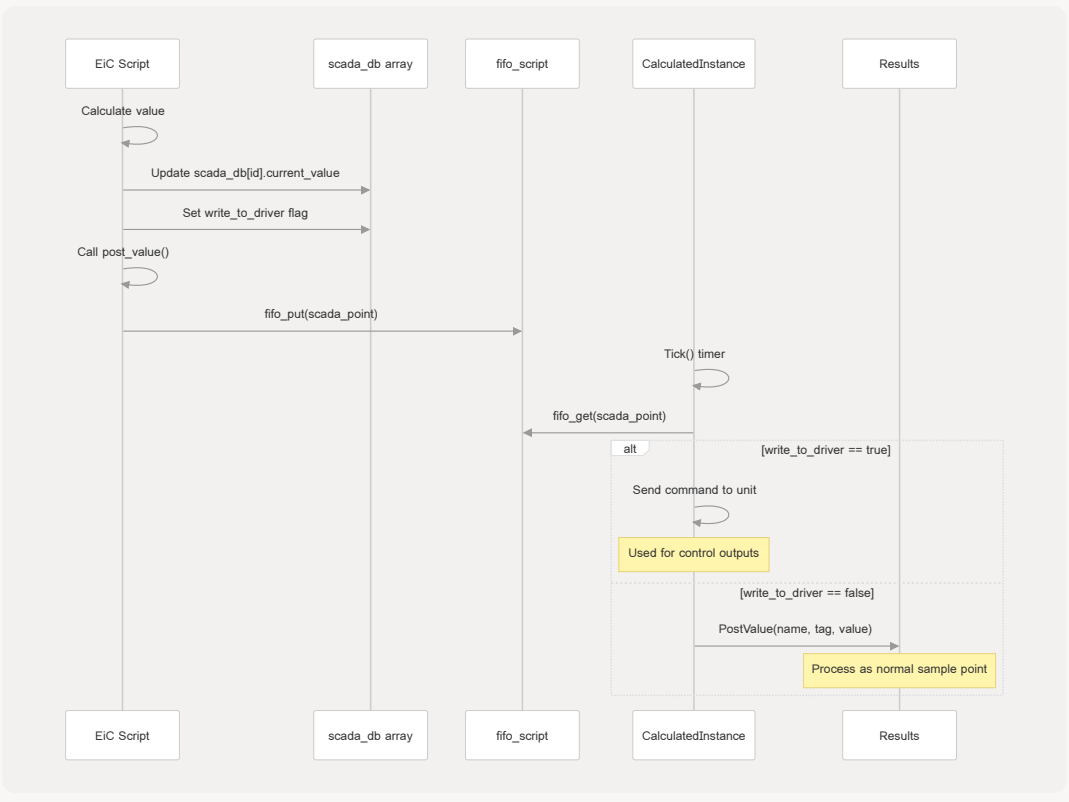
- `0` = Valid data
- `1` = Invalid data (triggers `FailureLevel` state)

Quality flags propagate through the system and affect alarm evaluation. Invalid quality automatically sets the sample point to `FailureLevel`, bypassing threshold checks.

Sources: `src/database/results.cpp` 134-211

Integration with Calculated Driver

The Calculated driver posts computed values back to the results processor through a shared memory mechanism:


EiC Script
scada_db array
fifo_script
CalculatedInstance
Results
Calculate value
Update scada_db[id].current_value
Set write_to_driver flag
Call post_value()
fifo_put(scada_point)
Tick() timer
fifo_get(scada_point)
alt
[write_to_driver == true]
Send command to unit
Used for control outputs
[write_to_driver == false]
PostValue(name, tag, value)
Process as normal sample point

From `src/drivers/Calculated/Calculated.cpp` 595-715:

```
void CalculatedInstance::Tick()
{
    for(fifo_get(p_fifo_script, buf, ...) >= 0; )
    {
        p_item = (struct scada_point*)buf;

        if(p_item->write_to_driver)
        {
            // Send command to unit
            GetConfigureDb()->DoExec(this, cmd, tUnit, value, name);
        }
        else
        {
            // Post value directly to Results
            PostValue(p_item->name, p_item->tag, p_item->current_value);
        }
    }
}
```

The `scada_db` array provides a shared memory interface between the EiC interpreter (running calculated scripts) and the main monitor process.

Sources: `src/drivers/Calculated/Calculated.cpp` 595-715 `src/drivers/Calculated/embedEiC.c` 78-121

Summary

The Results Processing and Alarming subsystem provides:

1. **Structured Data Flow**: Three-phase processing (UpdateStart, Update, UpdateEnd) ensures consistent state management
2. **Threshold-Based Alarming**: Four configurable thresholds per tag with debouncing via alarm event counting
3. **Alarm Groups**: Logical grouping of sample points with state aggregation
4. **Historical Archiving**: Dual-database strategy for short-term (FastDB) and long-term (GigaBASE) storage

Ask Devin about jonathanxavier/IndigoSCADA
Fast

5. **Configurable Actions**: Shell scripts and SQL commands triggered on measurements and alarms
6. **Quality Management**: IEC 60870-5 quality flags propagate through the system
7. **Current Value Tracking**: Real-time statistics (min, max, sum, samples) updated in CurrentDB

The system integrates tightly with the Monitor process for database updates and the Calculated driver for computed value processing.

Sources: `src/database/results.cpp` 1-895 `src/monitor/monitor.cpp` 348-680

HMI Widgets and Displays

Relevant source files

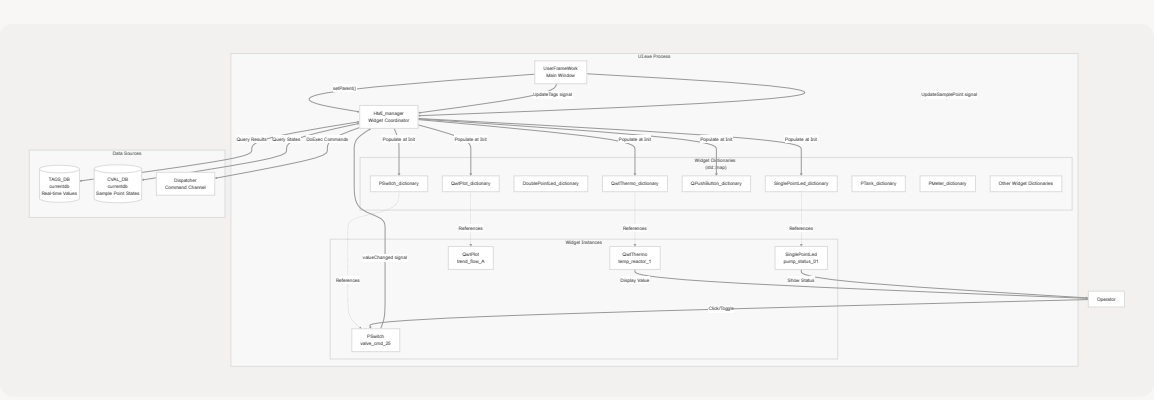
Purpose and Scope

This document describes the HMI widget system in IndigoSCADA, which provides graphical visualization and control of field data. The system uses the Qwt 4.2.0 library extended with custom SCADA widgets to display process values, status indicators, and operator controls. This page covers widget types, the HMI_manager coordination mechanism, update flows, and widget naming conventions.

For information about the overall UI framework and authentication, see User Interface Framework. For alarm display components, see Alarm and Event Displays. For creating custom screens, see HMI Designer and Custom Screens.

Architecture Overview

The HMI widget system consists of a central coordinator (`HMI_manager`) that manages multiple widget dictionaries and mediates between the real-time database and on-screen widgets.



Sources: `src/ui/hmi_mng.h` 21-67 `src/ui/hmi_mng.cpp` 33-51

Widget Type Categories

IndigoSCADA provides three functional categories of widgets, each serving distinct roles in the HMI.

Display Widgets (Read-Only)

Widget Class	Purpose	Update Method	Data Type
`QwtThermo`	Thermometer visualization	`setValue(double)`	Analog
`QLCDNumber`	Basic numeric display	`display(double)`	Analog
`PLCDNumber`	Enhanced LCD number	`display(double)`	Analog
`PTank`	Tank level indicator	`setValue(double)`	Analog
`PThermometer`	Graphical thermometer	`setValue(double)`	Analog
`PMeter`	Meter/gauge display	`setValue(double)`	Analog
`QwtPlot`	Trend chart (1000 points)	`setCurveData()`	Time series

Status Indicators (LEDs)

Widget Class	Purpose	State Representation
`SinglePointLed`	Single-point status	Green (0), Red (1), White (invalid), Blue (failure)
`DoublePointLed`	Double-point status	Yellow (0-indeterminate), Green (1-off), Red (2-on), Yellow (3-indeterminate), White (invalid), Blue (failure)
`PSinglePointLed`	Custom single LED	Same as SinglePointLed with custom rendering
`PDoublePointLed`	Custom double LED	Same as DoublePointLed with custom rendering

Control Widgets (Interactive)

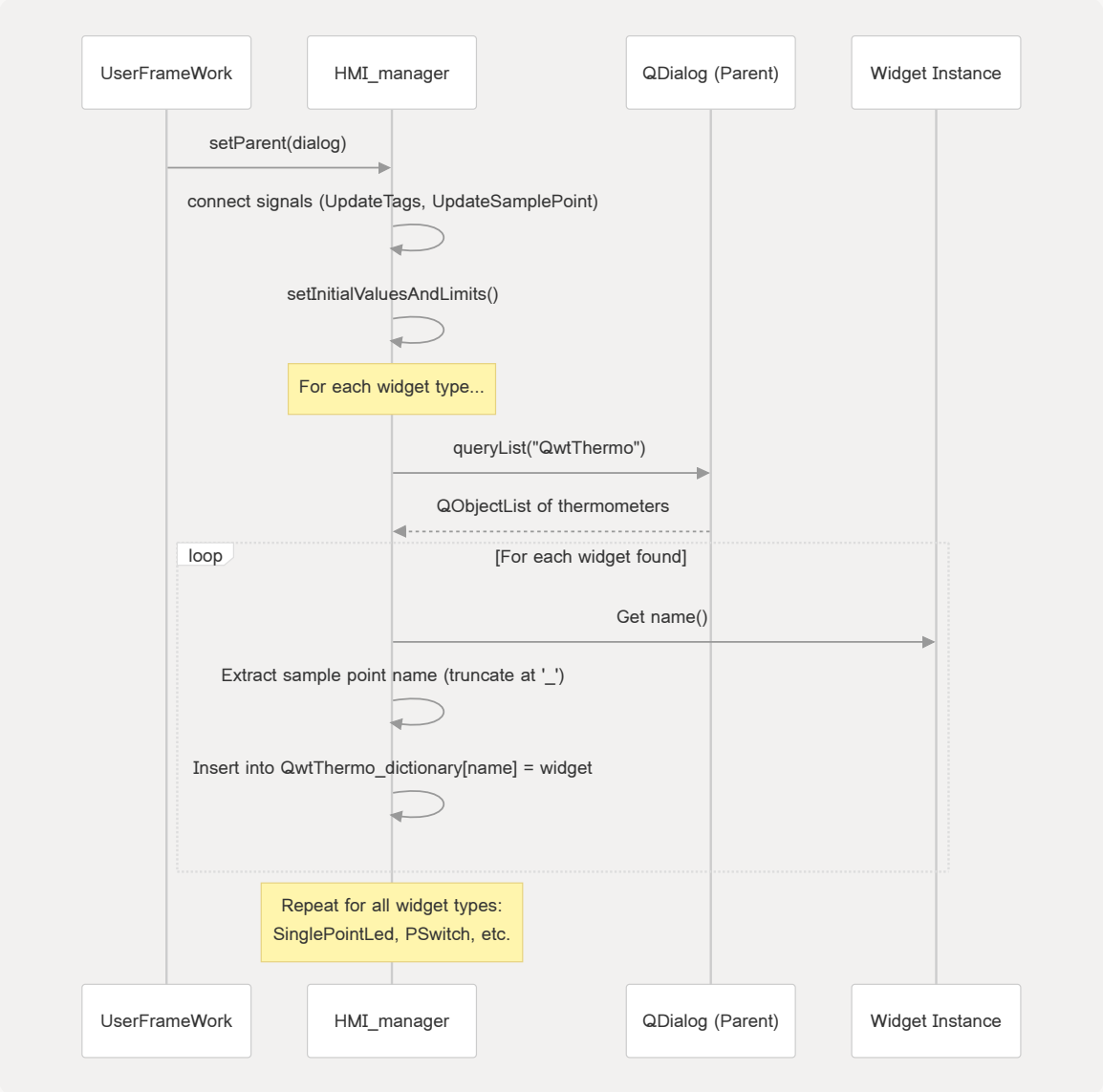
Widget Class	Purpose	Command Generation
`PSwitch`	Toggle switch	Emits `valueChanged(bool)`
`Breaker`	Circuit breaker	Emits `valueChanged(bool)`
`DoubleBreaker`	Double breaker	Emits `valueChanged(int)` with 0-3 states
`QPushButton`	Command button	Detected via `isDown()` polling

Sources: `src/ui/hmi_mng.h` 31-44 `src/ui/hmi_mng.cpp` 59-514 `src/ui/qwt-4.2.0/designer/qwtplugin.cpp` 70-118

Widget Management System

Initialization and Dictionary Population

The `HMI_manager` discovers and registers all widgets at initialization using Qt's introspection API:


UserFrameWork
HMI_manager
QDialog (Parent)
Widget Instance
setParent(dialog)
connect signals (UpdateTags, UpdateSamplePoint)
setInitialValuesAndLimits()
For each widget type...
queryList("QwtThermo")
QObjectList of thermometers
loop
[For each widget found]
Get name()
Extract sample point name (truncate at '_')
Insert into QwtThermo_dictionary[name] = widget
Repeat for all widget types:
SinglePointLed, PSwitch, etc.

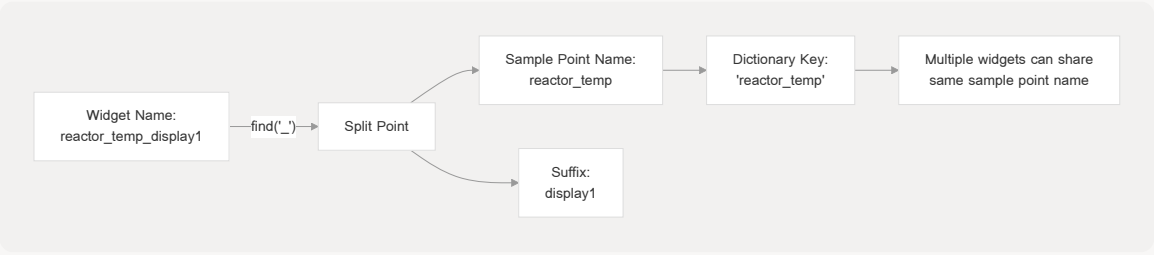
The initialization process follows this pattern for each widget type:

1. Query parent dialog for all widgets of type `T`
2. Iterate through returned list
3. Extract widget name and truncate at underscore
4. Insert `(sample_point_name, widget_pointer)` into appropriate dictionary
5. Set initial widget state

Sources: `src/ui/hmi_mng.cpp` 59-514

Widget Naming Convention

Widget instances must follow the naming pattern: `<sample_point_name>_<suffix>`


Widget Name:
reactor_temp_display1
find('_')
Split Point
Sample Point Name:
reactor_temp
Suffix:
display1
Dictionary Key:
'reactor_temp'
Multiple widgets can share
same sample point name

Examples:

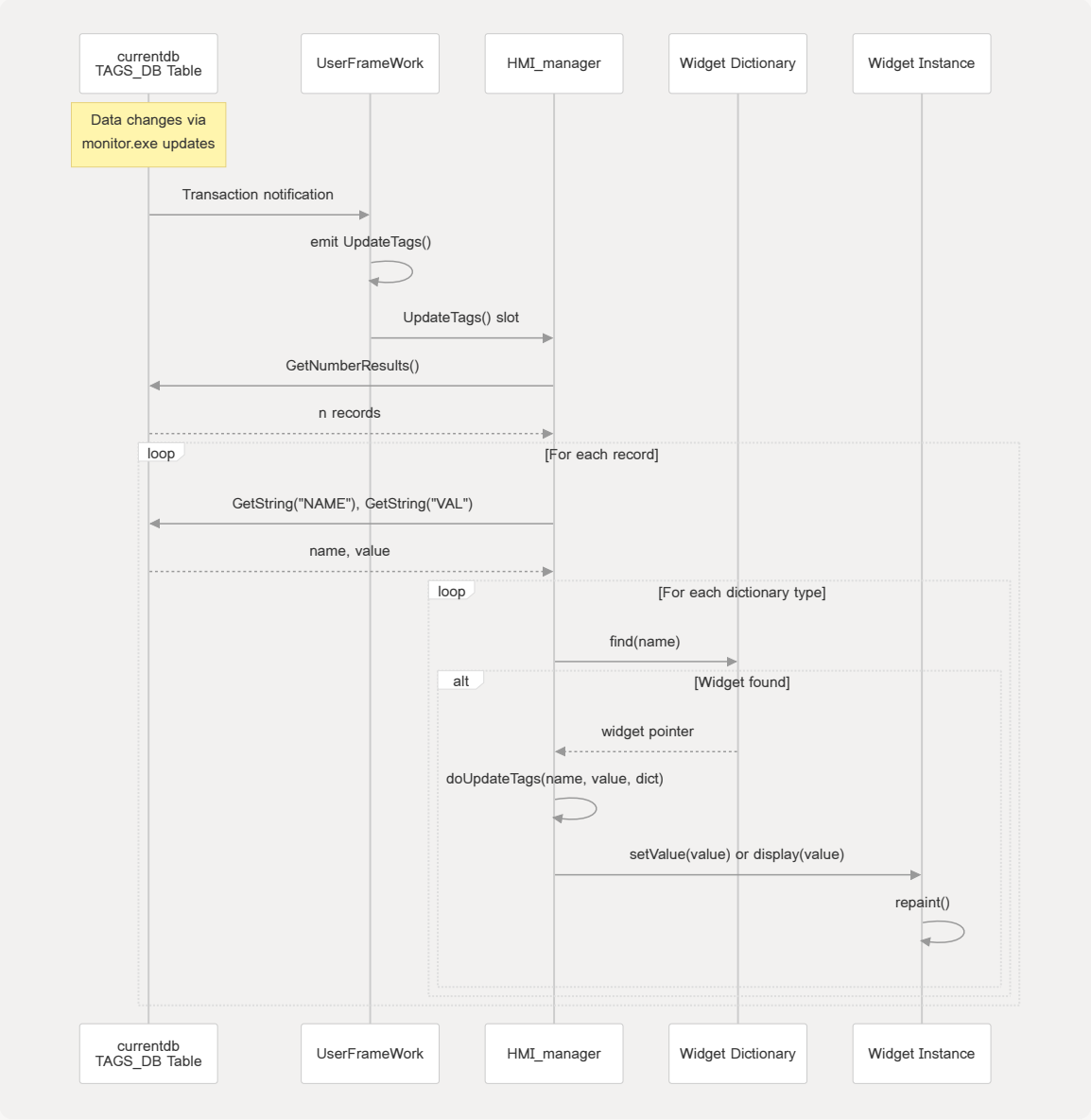
- Widget named `flow_line_A_gauge1` → dictionary key `flow_line_A`
- Widget named `flow_line_A_gauge2` → dictionary key `flow_line_A`
- Both widgets update when sample point `flow_line_A` changes

Sources: `src/ui/hmi_mng.cpp` 73-78 `src/ui/hmi_mng.cpp` 103-104

Update Mechanisms

The HMI_manager provides two distinct update paths: value updates and alarm state updates.

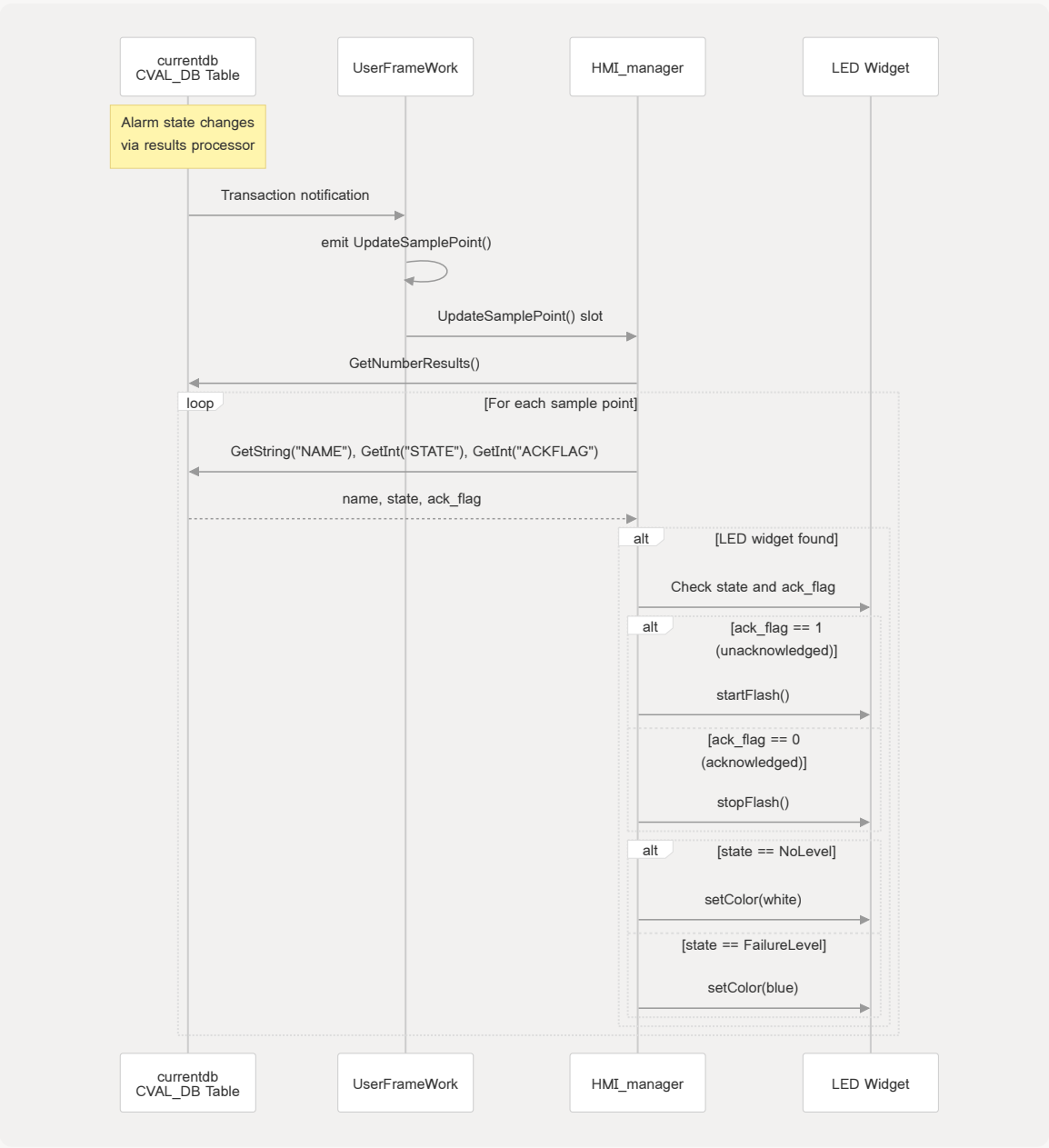
Value Update Flow (UpdateTags)


currentdb
TAGS_DB Table
UserFrameWork
HMI_manager
Widget Dictionary
Widget Instance
Data changes via
monitor.exe updates
Transaction notification
emit UpdateTags()
UpdateTags() slot
GetNumberResults()
n records
loop
[For each record]
GetString("NAME"), GetString("VAL")
name, value
loop
[For each dictionary type]
find(name)
alt
[Widget found]
widget pointer
doUpdateTags(name, value, dict)
setValue(value) or display(value)
repaint()

The `UpdateTags()` method `src/ui/hmi_mng.cpp` 812-841:

1. Retrieves all records from `TAGS_DB` query result
2. Extracts `NAME` and `VAL` for each record
3. Calls `doUpdateTags()` for each widget dictionary
4. Updates widget display if found in dictionary

Alarm State Update Flow (UpdateSamplePoint)


currentdb
CVAL_DB Table
UserFrameWork
HMI_manager
LED Widget
Alarm state changes
via results processor
Transaction notification
emit UpdateSamplePoint()
UpdateSamplePoint() slot
GetNumberResults()
loop
[For each sample point]
GetString("NAME"), GetInt("STATE"), GetInt("ACKFLAG")
name, state, ack_flag
alt
[LED widget found]
Check state and ack_flag
alt
[ack_flag == 1
(unacknowledged)]
startFlash()
[ack_flag == 0
(acknowledged)]
stopFlash()
alt
[state == NoLevel]
setColor(white)
[state == FailureLevel]
setColor(blue)

Sources: `src/ui/hmi_mng.cpp` 812-841 `src/ui/hmi_mng.cpp` 1012-1042 `src/ui/hmi_mng.cpp` 843-1002

Widget Value Mapping and State Representation

LED State Mapping

The `doUpdateTags()` method implements state-to-color mapping:

SinglePointLed `src/ui/hmi_mng.cpp` 536-576:

- Value 0 → Green (state off)
- Value 1 → Red (state on)
- Value 2-3 → Yellow (unused but supported)
- Default → White (invalid/none)

DoublePointLed `src/ui/hmi_mng.cpp` 577-617:

- Value 0 → Yellow (indeterminate/intermediate)
- Value 1 → Green (determinate state off)
- Value 2 → Red (determinate state on)
- Value 3 → Yellow (indeterminate)
- Default → White (invalid/none)

Alarm State Overlay

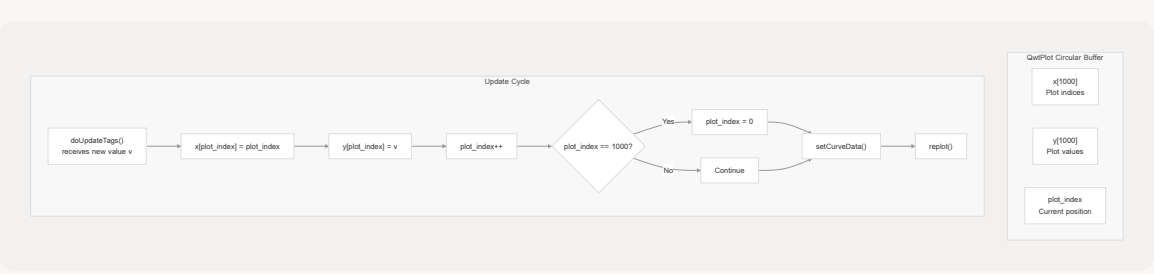
The `doUpdateSamplePoint()` method overlays alarm states onto LED widgets:

- `NoLevel` alarm state → Force color to white
- `FailureLevel` alarm state → Force color to blue (communication failure)
- `ack_flag == 1` → Start flashing via `timer` at 1-second interval
- `ack_flag == 0` → Stop flashing

Sources: `src/ui/hmi_mng.cpp` 843-1002 `src/ui/qwt-4.2.0/src/single_point_led.cpp` 89-110

Plot Widget Time Series Display

The `QwtPlot` widget provides real-time trending with a circular buffer:


Update Cycle
doUpdateTags()
receives new value v
x[plot_index] = plot_index
y[plot_index] = v
plot_index++
plot_index >= 1000?
Yes
plot_index = 0
No
Continue
setCurveData()
replot()
QwtPlot Circular Buffer
x[1000]
Plot indices
y[1000]
Plot values
plot_index
Current position

Plot initialization `src/ui/hmi_mng.cpp` 508-510:

```
((QwtPlot*)obj)->curve1 = ((QwtPlot*)obj)->insertCurve("Graph 1");
((QwtPlot*)obj)->plot_index = 0;
```

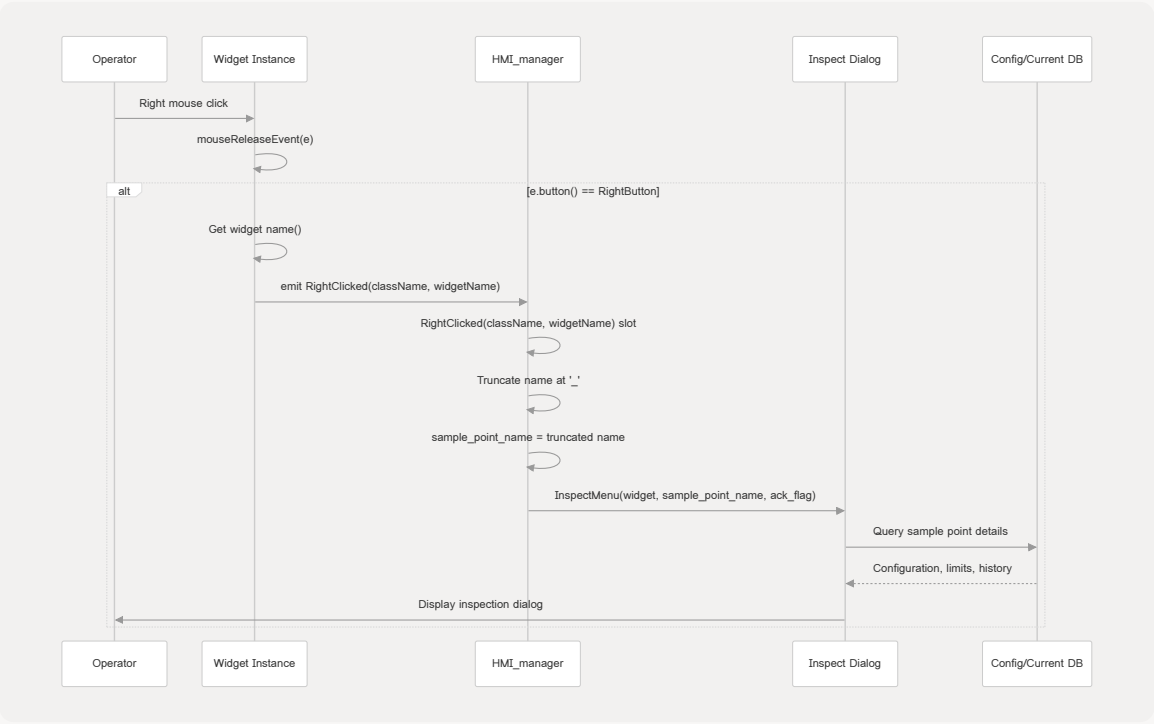
Plot update `src/ui/hmi_mng.cpp` 794-803:

```
((QwtPlot*)obj)->x[((QwtPlot*)obj)->plot_index] = ((QwtPlot*)obj)->plot_index;
((QwtPlot*)obj)->y[((QwtPlot*)obj)->plot_index] = v;
((QwtPlot*)obj)->plot_index++;
if(((QwtPlot*)obj)->plot_index == 1000) ((QwtPlot*)obj)->plot_index = 0;
((QwtPlot*)obj)->setCurveData(((QwtPlot*)obj)->curve1, ((QwtPlot*)obj)->x,
                              ((QwtPlot*)obj)->y, ((QwtPlot*)obj)->plot_index);
((QwtPlot*)obj)->replot();
```

Sources: `src/ui/hmi_mng.cpp` 483-513 `src/ui/hmi_mng.cpp` 793-804 `src/ui/qwt-4.2.0/include/qwt_plot.h` 429-432

Right-Click Inspection

Most widgets support right-click context menus for detailed inspection. The mechanism uses Qt's signal/slot system:


Operator
Widget Instance
HMI_manager
Inspect Dialog
Config/Current DB
Right mouse click
mouseReleaseEvent(e)
alt
[e.button() == RightButton]
Get widget name()
emit RightClicked(className, widgetName)
RightClicked(className, widgetName) slot
Truncate name at '_'
sample_point_name = truncated name
InspectMenu(widget, sample_point_name, ack_flag)
Query sample point details
Configuration, limits, history
Display inspection dialog

Widget-side implementation example `src/ui/qwt-4.2.0/src/single_point_led.cpp` 115-122:

```
void SinglePointLed::mouseReleaseEvent(QMouseEvent * e)
{
    if(e->button() == Qt::RightButton)
    {
        QString name = this->name();
        emit RightClicked(QString("SinglePointLed"), name);
    }
}
```

HMI_manager-side handling `src/ui/hmi_mng.cpp` 1126-1553:

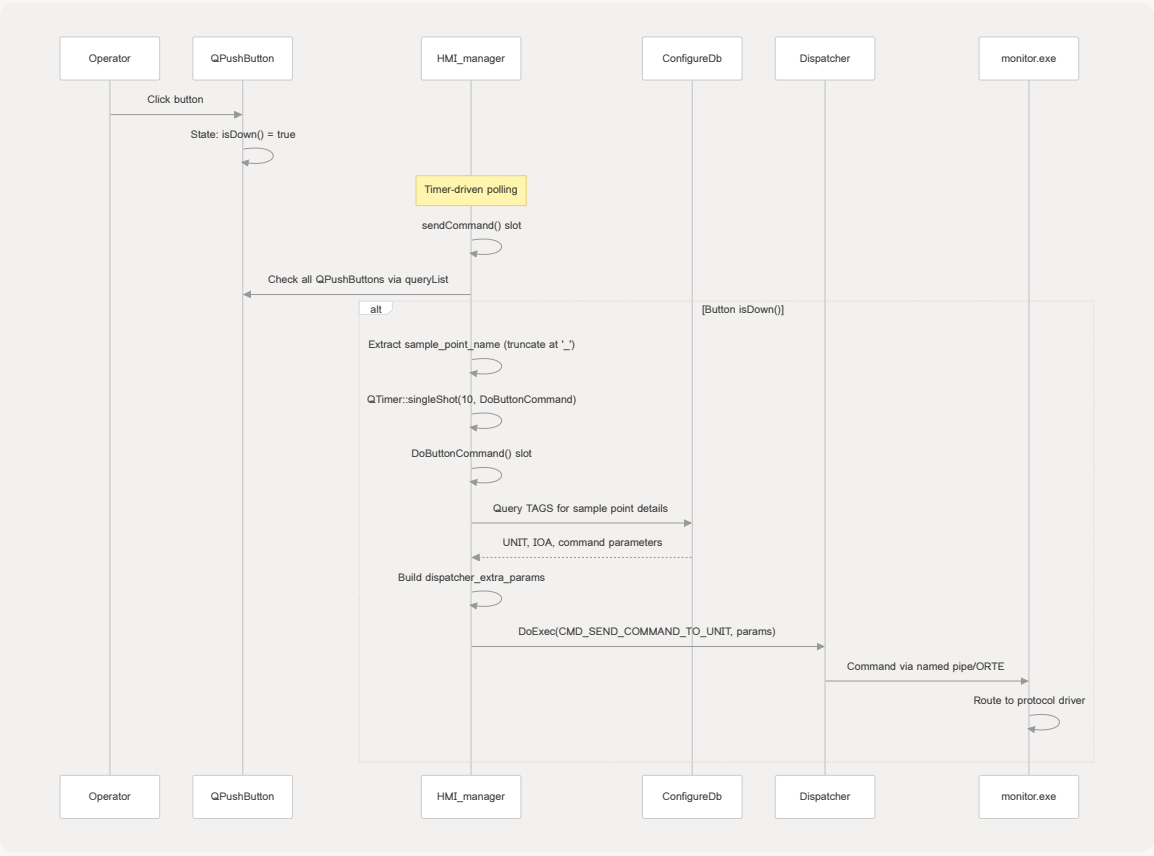
- Receives `RightClicked(class_name, widget_name)` signal
- Queries parent dialog for specific widget by name
- Extracts sample point name (truncate at underscore)
- Retrieves flash state if LED widget
- Calls `InspectMenu()` with widget pointer and sample point name

Sources: `src/ui/hmi_mng.cpp` 1126-1553 `src/ui/qwt-4.2.0/src/single_point_led.cpp` 115-122 `src/ui/qwt-4.2.0/src/elswitch.cpp` 51-65

Command Widget Interaction

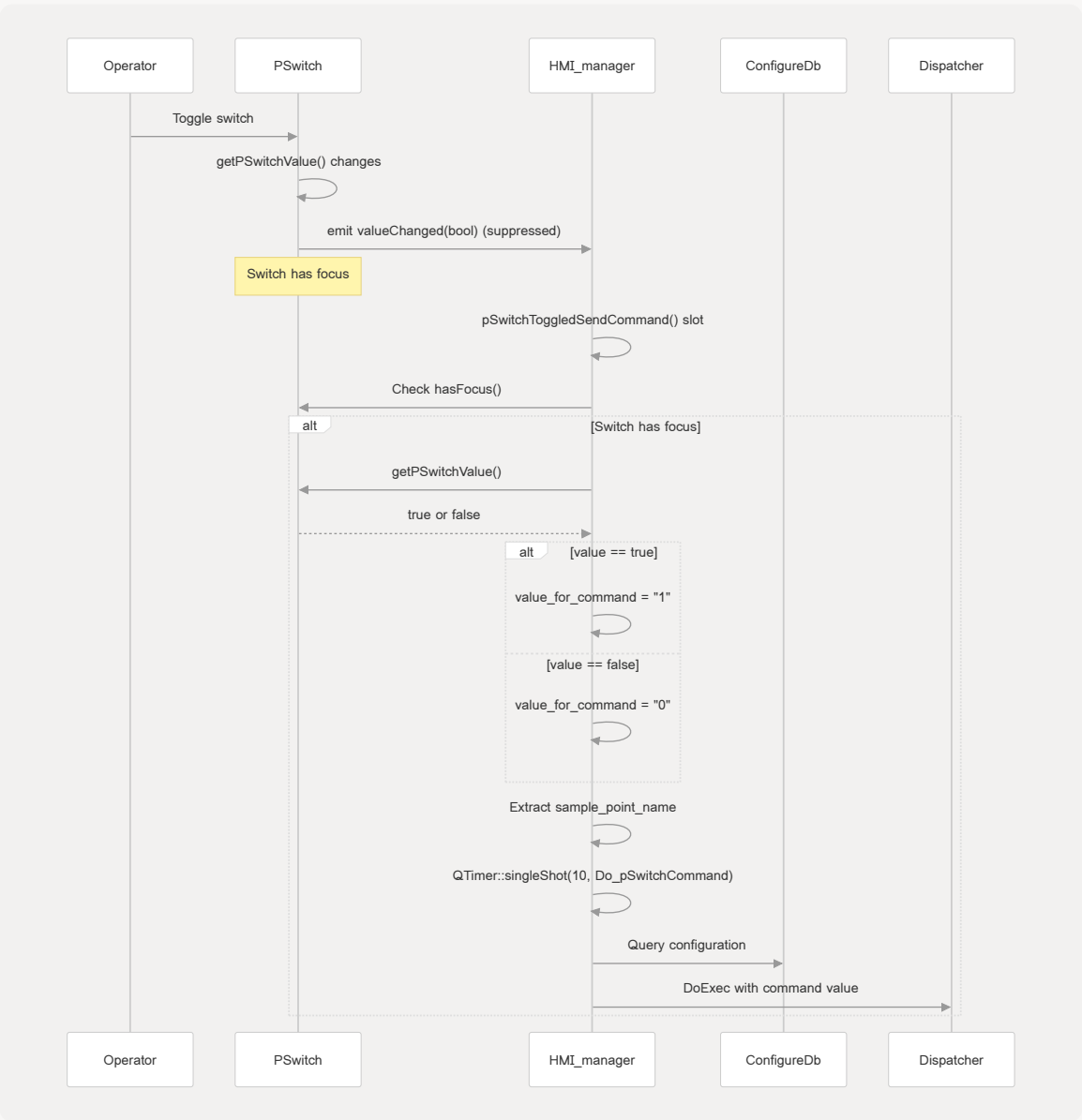
Control widgets enable operators to send commands to field devices through the HMI.

Push Button Command Flow


Operator
QPushButton
HMI_manager
ConfigureDb
Dispatcher
monitor.exe
Click button
State: isDown() = true
Timer-driven polling
sendCommand() slot
Check all QPushButtons via queryList
alt
[Button isDown()]
Extract sample_point_name (truncate at '_')
QTimer::singleShot(10, DoButtonCommand)
DoButtonCommand() slot
Query TAGS for sample point details
UNIT, IOA, command parameters
Build dispatcher_extra_params
DoExec(CMD_SEND_COMMAND_TO_UNIT, params)
Command via named pipe/ORTE
Route to protocol driver

Sources: `src/ui/hmi_mng.cpp` 1045-1074

PSwitch Toggle Flow


Operator
PSwitch
HMI_manager
ConfigureDb
Dispatcher
Toggle switch
getPSwitchValue() changes
emit valueChanged(bool) (suppressed)
Switch has focus
pSwitchToggledSendCommand() slot
Check hasFocus()
alt
[Switch has focus]
getPSwitchValue()
true or false
alt
[value == true]
value_for_command = "1"
[value == false]
value_for_command = "0"
Extract sample_point_name
QTimer::singleShot(10, Do_pSwitchCommand)
Query configuration
DoExec with command value

Key difference: PSwitch uses focus detection to determine which switch was toggled, while QPushButton uses `isDown()` state polling.

Sources: `src/ui/hmi_mng.cpp` 1077-1116

Command Button Privilege Control

By default, all command buttons are disabled for security `src/ui/hmi_mng.cpp` 290-314:

```
QObjectList *l = dialog_parent->queryList( "QPushButton" );
// ...
while((obj = it.current()) != 0)
{
    QString name = obj->name();

    //By default commands are not enabled
    ((QPushButton*)obj)->setEnabled( FALSE );

    int idx = name.find('_');
    name.truncate(idx);

    WidgetDict::value_type pr(name, obj);
    QPushButton_dictionary.insert(pr);
}
```

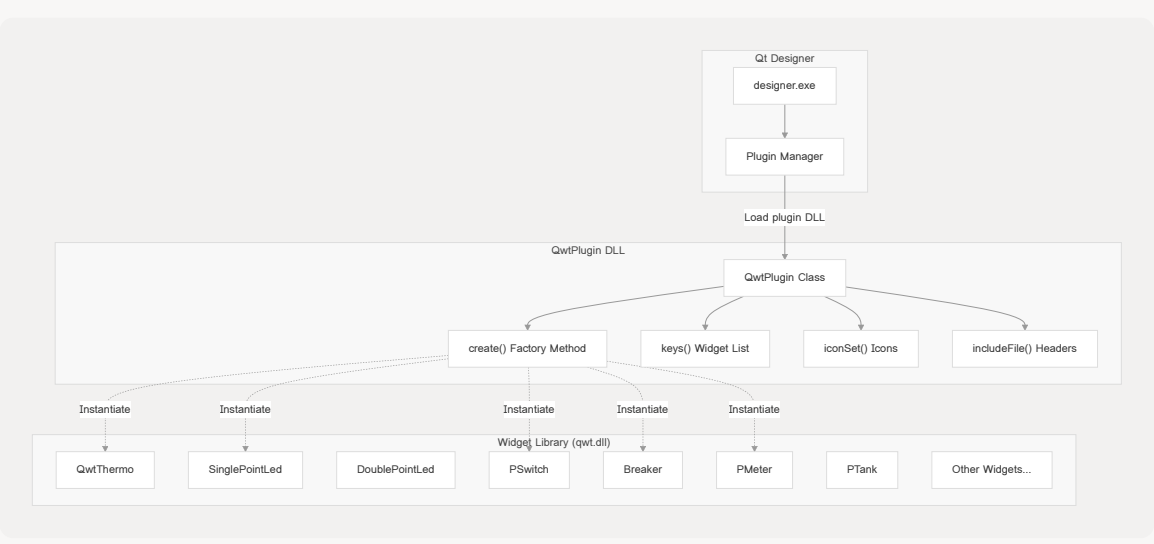
Buttons are enabled based on user privileges set in the `UserFrameWork` (see User Interface Framework for details on privilege system).

Sources: `src/ui/hmi_mng.cpp` 290-314

Qt Designer Integration

IndigoSCADA extends Qt Designer with custom SCADA widgets through a plugin system.

Widget Plugin Architecture


Qt Designer
designer.exe
Plugin Manager
Load plugin DLL
QwtPlugin DLL
QwtPlugin Class
create() Factory Method
keys() Widget List
iconSet() Icons
includeFile() Headers
Instantiate
Instantiate
Instantiate
Instantiate
Instantiate
Widget Library (qwt.dll)
QwtThermo
SinglePointLed
DoublePointLed
PSwitch
Breaker
PMeter
PTank
Other Widgets...

Supported Designer Widgets

The plugin registers all custom widgets in its constructor `src/ui/qwt-4.2.0/designer/qwtplugin.cpp` 70-118:

```
vec.append(Entry("QwtPlot", "qwt_plot.h", "qwtplot.png", "QwtPlot", "whatsthis"));
vec.append(Entry("SinglePointLed", "single_point_led.h", "led_rg0000.png", "SinglePointLed",
vec.append(Entry("DoublePointLed", "double_point_led.h", "led_rg0001.png", "DoublePointLed",
vec.append(Entry("PSwitch", "pswitch.h", "pswitchicon.png", "PSwitch", "whatsthis"));
vec.append(Entry("Breaker", "elswitch.h", "pswitchicon.png", "Breaker", "whatsthis"));
// ... etc
```

Factory method creates widget instances `src/ui/qwt-4.2.0/designer/qwtplugin.cpp` 121-173:

```
QWidget* QwtPlugin::create(const QString &key, QWidget* parent, const char* name)
{
    if ( key == "QwtPlot" )
        return new QwtPlot( parent, name );
    else if ( key == "SinglePointLed" )
        return new SinglePointLed(parent, name );
    else if ( key == "PSwitch" )
        return new PSwitch(parent, name );
    // ... etc
}
```

Designer Widget Group

All SCADA widgets are grouped under "SCADA Widgets" in the designer palette `src/ui/qwt-4.2.0/designer/qwtplugin.cpp` 186-191:

```
QString QwtPlugin::group( const QString& feature ) const
{
    if (entry(feature) != NULL )
        return QString("SCADA Widgets");
    return QString::null;
}
```

Sources: `src/ui/qwt-4.2.0/designer/qwtplugin.cpp` 70-235 `src/ui/qwt-4.2.0/designer/qwtplugin.dsp` 1-195

Widget Implementation Details

LED Flash Timer Mechanism

LED widgets implement flashing for unacknowledged alarms using QTimer `src/ui/qwt-4.2.0/src/single_point_led.cpp` 89-110:

```
void SinglePointLed::startFlash()
{
   if(timer == NULL)
   {
       isFlashing = true;
       timer = new QTimer(this);
       timer->start(1000);  // 1-second interval
       connect(timer,SIGNAL(timeout()), this, SLOT(timerSlot()));
   }
}

void SinglePointLed::stopFlash()
{
   if(timer)
   {
      isFlashing = false;
      timer->stop();
      delete timer;
      timer = NULL;
      on();  // Ensure LED is visible after stopping flash
   }
}

void SinglePointLed::timerSlot()
{
    toggleState();  // Alternate between On/Off
}
```

This creates a visual 1Hz blink to attract operator attention to unacknowledged alarms.

Sources: `src/ui/qwt-4.2.0/src/single_point_led.cpp` 74-110 `src/ui/qwt-4.2.0/include/single_point_led.h` 55-61

Breaker Widget State Visualization

The Breaker widget uses pixmaps for visual representation `src/ui/qwt-4.2.0/src/elswitch.cpp` 25-38:

```
Breaker::Breaker(QWidget *parent, const char *name)
        : QWidget(parent,name)
{
    OnString="on";
    OffString="off";
    // ... size calculation ...
    value=false;
    setFocusPolicy(QWidget::StrongFocus);

    lOnpixmap = new QPixmap((const char **)breaker_closed_xpm);
    lOffpixmap = new QPixmap((const char **)breaker_opened_xpm);
}
```

Drawing logic `src/ui/qwt-4.2.0/src/elswitch.cpp` 85-121:

```
void Breaker::drawSwitch()
{
    QPainter painter(this);

    if(value)
    {
        painter.begin(this);
        painter.drawPixmap(0, 0, *lOnpixmap);  // Closed breaker
        painter.end();
    }
    else
    {
        painter.begin(this);
        painter.drawPixmap(0, 0, *lOffpixmap);  // Open breaker
        painter.end();
    }
}
```

The pixmap resources are embedded XPM files:

- `src/ui/qwt-4.2.0/src/breaker_closed.xpm` 1-69 - Visual representation of closed state
- `src/ui/qwt-4.2.0/src/breaker_opened.xpm` 1-69 - Visual representation of open state

Sources: `src/ui/qwt-4.2.0/src/elswitch.cpp` 25-121 `src/ui/qwt-4.2.0/include/elswitch.h` 28-113

Widget Dictionary Type Definition

The widget dictionary uses STL map for efficient lookup `src/ui/hmi_mng.h` 17-19:

```
#ifdef USE_STD_MAP
typedef std::map<QString, QObject*, std::less<QString> > WidgetDict;
#endif
```

Dictionary members in HMI_manager `src/ui/hmi_mng.h` 31-44:

```
WidgetDict QwtThermo_dictionary;
WidgetDict QLCDNumber_dictionary;
WidgetDict PLCDNumber_dictionary;
WidgetDict SinglePointLed_dictionary;
WidgetDict DoublePointLed_dictionary;
WidgetDict PSinglePointLed_dictionary;
WidgetDict PDoublePointLed_dictionary;
WidgetDict PSwitch_dictionary;
WidgetDict Breaker_dictionary;
WidgetDict PTank_dictionary;
WidgetDict PThermometer_dictionary;
WidgetDict QPushButton_dictionary;
WidgetDict PMeter_dictionary;
WidgetDict QwtPlot_dictionary;
```

Each dictionary maps sample point names (QString) to widget pointers (QObject*), allowing multiple widget instances to display the same process value.

Sources: `src/ui/hmi_mng.h` 17-44

Summary

The HMI widget system provides a flexible, extensible framework for visualizing and controlling industrial processes. Key design principles include:

- **Separation of concerns**: HMI_manager coordinates between database and widgets
- **Dictionary-based lookup**: Efficient O(log n) widget access by sample point name
- **Dual update paths**: Value updates separate from alarm state updates
- **Multiple widgets per point**: Many widgets can display the same process variable
- **Qt Designer integration**: Custom widgets available in graphical designer
- **Right-click inspection**: Contextual access to detailed information
- **Privilege-based control**: Command widgets respect user access levels

The system leverages Qt 3.3.8 and Qwt 4.2.0 as foundational libraries, extended with custom SCADA-specific widgets optimized for industrial monitoring and control applications.

HMI Designer and Custom Screens

› Relevant source files

Purpose and Scope

This document describes the HMI screen designer tool (`hmi_designer.exe`) and the mechanism for creating and loading custom graphical user interfaces in IndigoSCADA. The designer provides a visual environment for creating HMI screens using SCADA-specific widgets, while the runtime system (`ui.exe`) dynamically loads and animates these screens with real-time data.

For information about runtime widget management, updates, and interaction, see HMI Widgets and Displays. For details on the main UI framework and user authentication, see User Interface Framework.

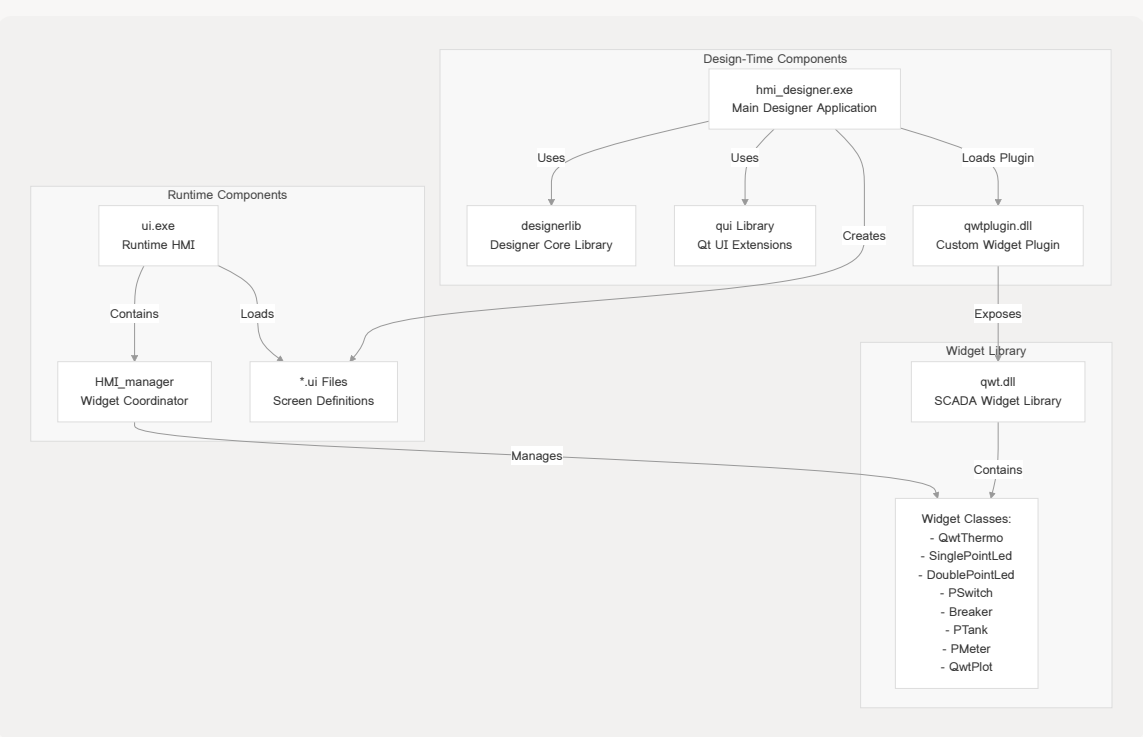
Architecture Overview

The HMI designer system consists of two distinct phases:

Design-Time Phase: The `hmi_designer.exe` application provides a graphical environment for creating HMI screens. It is built on Qt Designer 3.3.8 and extended with custom SCADA widgets through a plugin mechanism.

Runtime Phase: The `ui.exe` application dynamically loads the `.ui` files created by the designer and populates them with live data from the SCADA system.

Component Architecture


Design-Time Components
hmi_designer.exe
Main Designer Application
Uses
Uses
Loads Plugin
designerlib
Designer Core Library
qui Library
Qt UI Extensions
Creates
qwtplugin.dll
Custom Widget Plugin
Runtime Components
ui.exe
Runtime HMI
Contains
Loads
HMI_manager
Widget Coordinator
*.ui Files
Screen Definitions
Exposes
Widget Library
qwt.dll
SCADA Widget Library
Contains
Manages
Widget Classes:
- QwtThermo
- SinglePointLed
- DoublePointLed
- PSwitch
- Breaker
- PTank
- PMeter
- QwtPlot

Design-Time Flow:

1. `hmi_designer.exe` starts and loads Qt Designer framework
2. Designer loads `qwtplugin.dll` plugin from `C:\scada\plugins\designer\`
3. Plugin exposes custom SCADA widgets from `qwt.dll` to the designer palette
4. User drags widgets onto forms and configures properties
5. Designer saves screens as `.ui` files (XML format)

Runtime Flow:

1. `ui.exe` starts and initializes `HMI_manager`
2. User framework loads `.ui` files dynamically using Qt's UI loader
3. `HMI_manager` scans loaded widgets and populates dictionaries
4. Widgets are bound to sample points based on naming convention
5. Real-time updates flow through widget dictionaries

Sources:

- `src/daq/daq.dsw` 231-245
- `src/daq/daq.dsw` 249-258
- `src/daq/daq.dsw` 870-879
- `src/daq/daq.dsw` 882-891
- `src/ui/hmi_mng.cpp` 33-51

Custom SCADA Widget Library

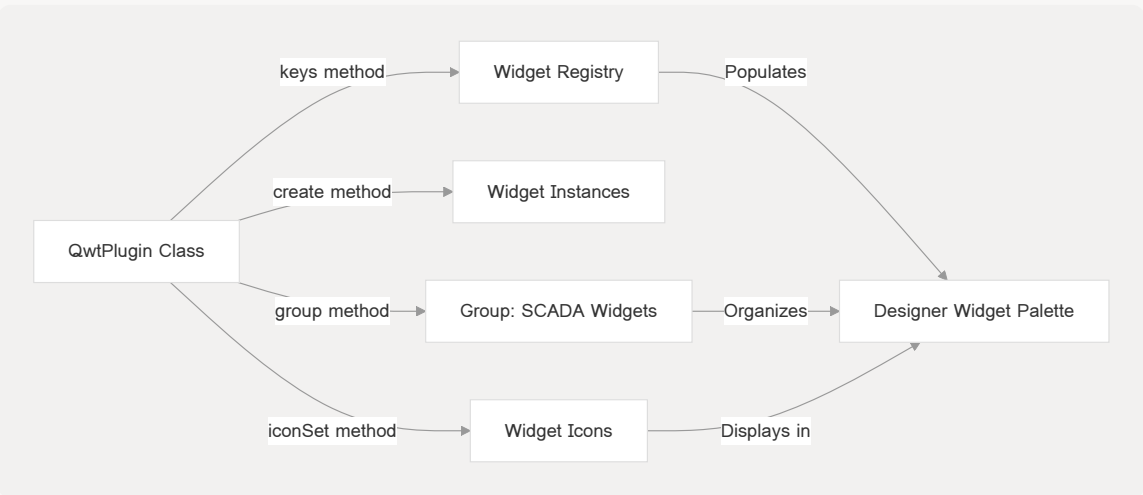
IndigoSCADA extends Qt Designer with a comprehensive set of SCADA-specific widgets built on the Qwt 4.2.0 library. These widgets are registered through the `qwtplugin` plugin and appear in the designer's widget palette under the "SCADA Widgets" group.

Available Widgets

Widget Class	Header File	Purpose	Visual Type
`QwtThermo`	qwt_thermo.h	Temperature/level display	Analog indicator
`SinglePointLed`	single_point_led.h	Single-state indicator	LED (Green/Red/White)
`DoublePointLed`	double_point_led.h	Double-point state indicator	LED (Yellow/Green/Red)
`PSinglePointLed`	psinglepointled.h	Styled single-point LED	Graphical LED
`PDoublePointLed`	pdoublepointled.h	Styled double-point LED	Graphical LED
`PSwitch`	pswitch.h	Controllable switch	Interactive switch
`Breaker`	elswitch.h	Circuit breaker	Electrical symbol
`DoubleBreaker`	double_elswitch.h	Double-position breaker	Electrical symbol
`PTank`	ptank.h	Tank level visualization	Tank graphic
`PThermometer`	pthermometer.h	Thermometer visualization	Thermometer graphic
`PMeter`	pmeter.h	Analog meter	Dial gauge
`PLCDNumber`	plcdnumber.h	Numeric display	LCD digits
`QwtPlot`	qwt_plot.h	Trend chart	Time-series plot

Widget Plugin Registration

The `qwtplugin` registers widgets with Qt Designer through a standard Qt Designer plugin interface:


keys method
Widget Registry
Populates
create method
Widget Instances
QwtPlugin Class
group method
Group: SCADA Widgets
Organizes
Designer Widget Palette
iconSet method
Widget Icons
Displays in

The plugin implementation defines how each widget appears in the designer:

Key Methods:

- `keys()` - Returns list of widget class names
- `create()` - Instantiates widget objects in designer
- `group()` - Returns "SCADA Widgets" group name
- `iconSet()` - Provides widget icons (PNG/XPM files)
- `includeFile()` - Specifies header file for code generation
- `toolTip()` - Provides hover text

Sources:

- `src/ui/qwt-4.2.0/designer/qwtplugin.cpp` 70-118
- `src/ui/qwt-4.2.0/designer/qwtplugin.cpp` 176-191

Widget Properties and Capabilities

Each widget class exposes Qt properties that can be configured in the designer:

Example: `Breaker` Widget Properties

- `switchValue` (bool) - Current breaker state (open/closed)
- `OnLabel` (QString) - Text for closed state
- `OffLabel` (QString) - Text for open state
- `Onpixmap` (QPixmap) - Image for closed state
- `Offpixmap` (QPixmap) - Image for open state

Example: `SinglePointLed` Widget

- `state` (enum) - On/Off state
- `color` (QColor) - LED color
- Supports flashing for unacknowledged alarms
- Emits `RightClicked` signal for context menus

Sources:

- `src/ui/qwt-4.2.0/include/elswitch.h` 28-113
- `src/ui/qwt-4.2.0/include/single_point_led.h` 39-96
- `src/ui/qwt-4.2.0/src/elswitch.cpp` 25-38

Screen Design Workflow

Creating a New HMI Screen

Step 1: Launch Designer The designer is launched from `C:\scada\bin\hmi_designer.exe`. On startup, it loads the custom widget plugin from `C:\scada\plugins\designer\qwtplugin.dll`.

Step 2: Create Form

- Create a new dialog or main window form
- The form becomes the container for SCADA widgets
- Set form properties (size, title, etc.)

Step 3: Add Widgets Widgets are dragged from the "SCADA Widgets" palette group onto the form. The designer allows:

- Visual positioning and sizing
- Property configuration via property editor
- Layout management using Qt layouts

Step 4: Widget Naming Convention

Widget names must follow a specific convention for runtime binding:

```
<SamplePointName>_<UniqueIdentifier>
```

- **SamplePointName**: Matches a sample point name in the `TAGS` database table
- **Underscore**: Required separator
- **UniqueIdentifier**: Arbitrary suffix to allow multiple widgets for same point

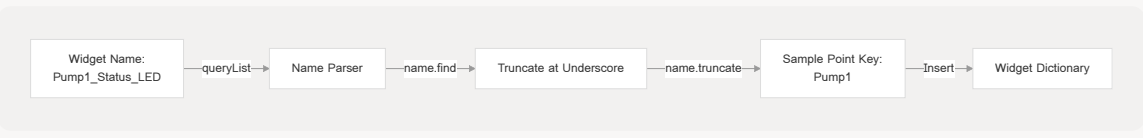
Example Names:

- `Pump1_Status` - LED showing Pump1 status
- `Pump1_Control` - Switch to control Pump1
- `TankLevel_Meter` - Meter showing TankLevel
- `TankLevel_Plot` - Plot showing TankLevel trend

Step 5: Save Screen Screens are saved as `.ui` files (Qt UI XML format) to the project directory.

Widget Name Processing

At runtime, the `HMI_manager` processes widget names:


Widget Name:
Pump1_Status_LED
queryList
Name Parser
name.find
Truncate at Underscore
name.truncate
Sample Point Key:
Pump1
Insert
Widget Dictionary

The truncation logic:

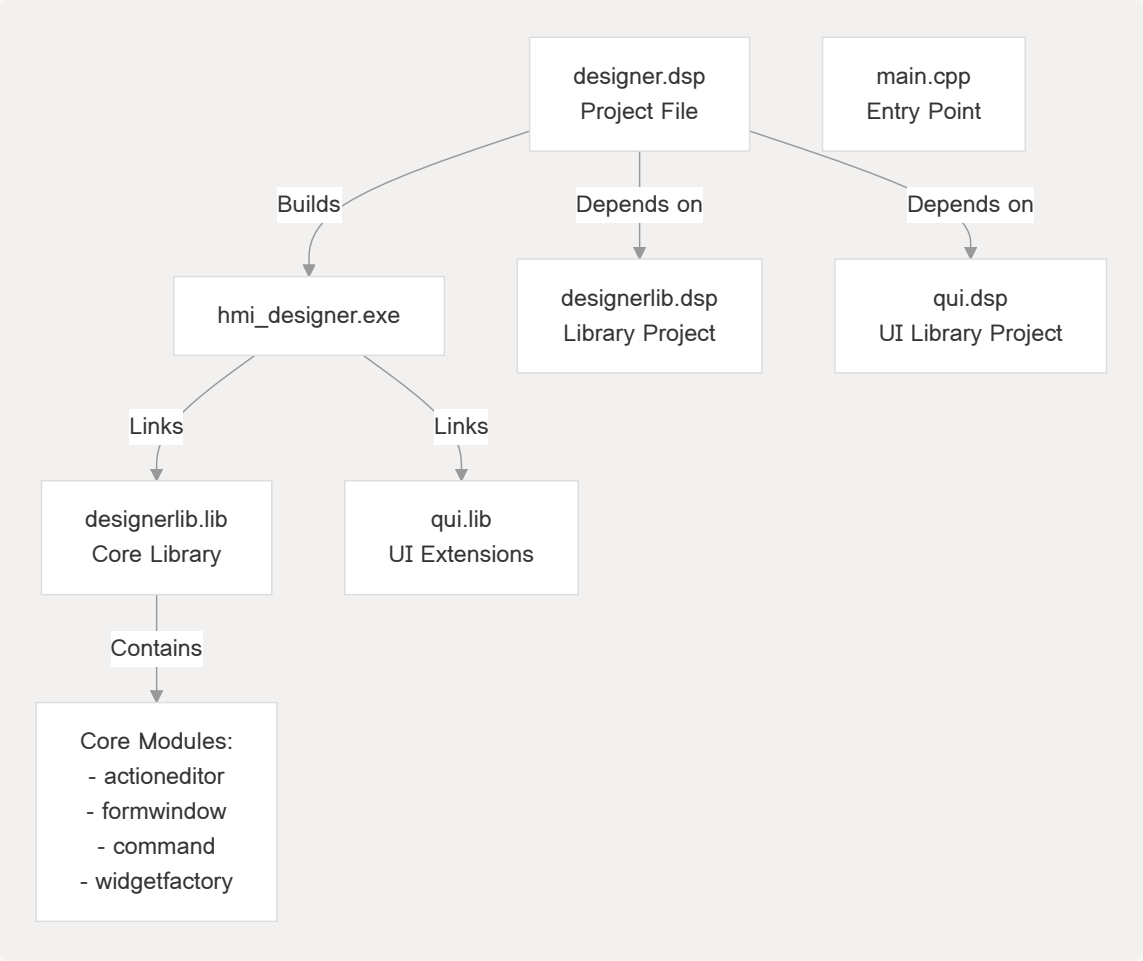
```
QString name = obj->name();          // "Pump1_Status_LED"
int idx = name.find('_');            // Find first underscore
name.truncate(idx);                  // Truncate: "Pump1"
WidgetDict::value_type pr(name, obj); // Create dictionary entry
dictionary.insert(pr);               // Insert into dictionary
```

Sources:

- `src/ui/hmi_mng.cpp` 62-84
- `src/ui/hmi_mng.cpp` 139-177

Designer Project Structure

The designer application is built from multiple components:


designer.dsp
Project File
main.cpp
Entry Point
Builds
Depends on
Depends on
hmi_designer.exe
designerlib.dsp
Library Project
qui.dsp
UI Library Project
Links
Links
designerlib.lib
Core Library
qui.lib
UI Extensions
Contains
Core Modules:
- actioneditor
- formwindow
- command
- widgetfactory

Sources:

- `src/hmi_designer/designer/app/designer.dsp` 1-118
- `src/hmi_designer/designer/designer/designerlib.dsp` 1-80
- `src/hmi_designer/designer/uilib/qui.dsp` 1-87

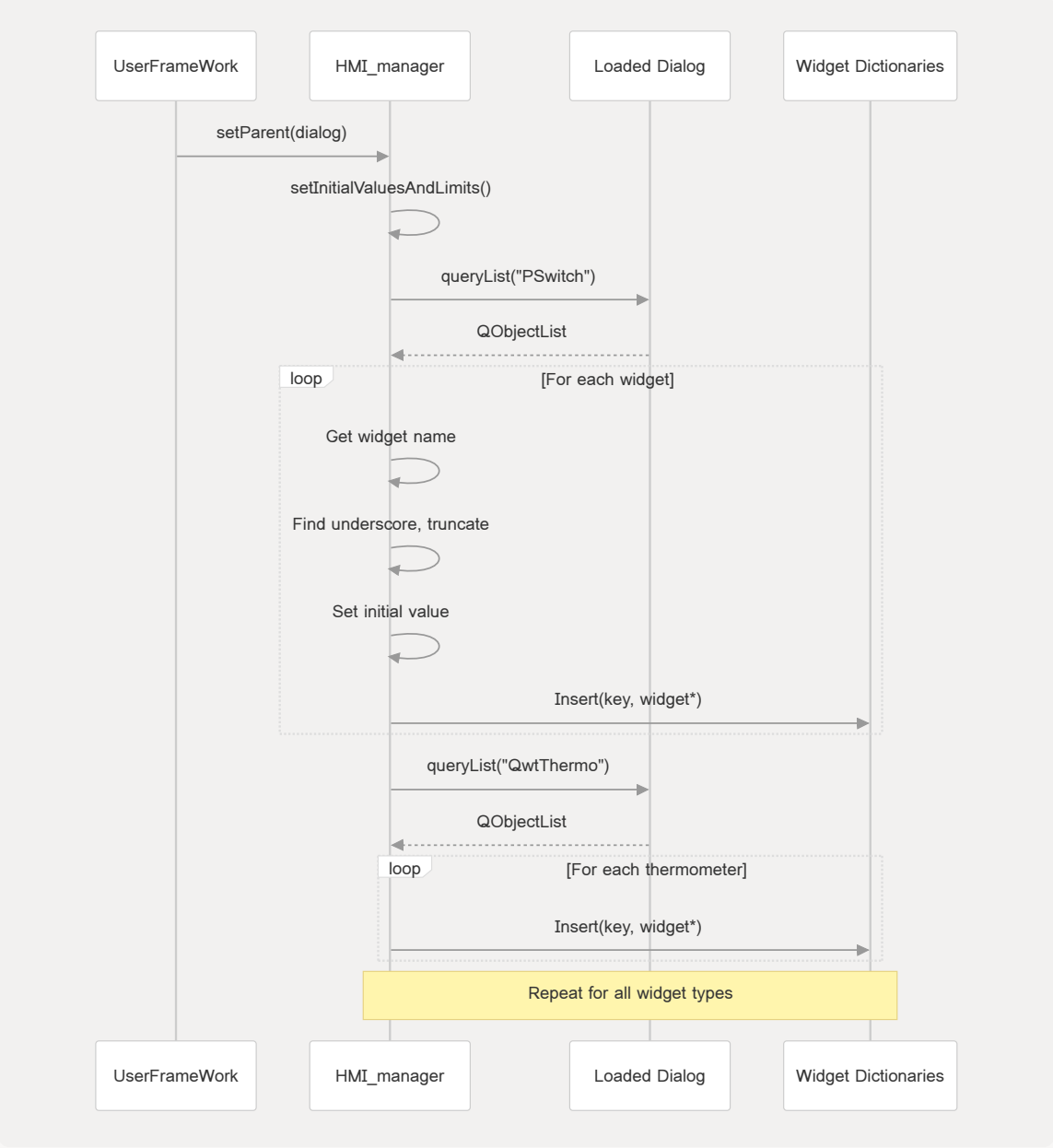
Runtime Loading Mechanism

Dynamic UI File Loading

The runtime HMI (`ui.exe`) loads `.ui` files dynamically using Qt's UI loader mechanism. This happens in the `UserFrameWork` class (see User Interface Framework).

HMI_manager Initialization

When a `.ui` file is loaded, `HMI_manager` scans the widget tree and builds widget dictionaries:


UserFrameWork
HMI_manager
Loaded Dialog
Widget Dictionaries
setParent(dialog)
setInitialValuesAndLimits()
queryList("PSwitch")
QObjectList
loop
[For each widget]
Get widget name
Find underscore, truncate
Set initial value
Insert(key, widget*)
queryList("QwtThermo")
QObjectList
loop
[For each thermometer]
Insert(key, widget*)
Repeat for all widget types
UserFrameWork
HMI_manager
Loaded Dialog
Widget Dictionaries

Dictionary Initialization Process:

The `setInitialValuesAndLimits()` method populates dictionaries for each widget type:

1. **Query Widget Tree**: Use `queryList()` to find all widgets of a specific type
2. **Extract Name**: Get widget object name
3. **Truncate at Underscore**: Extract sample point name
4. **Set Initial State**: Configure widget to initial/invalid state
5. **Insert to Dictionary**: Add (sample_point_name → widget*) mapping

Supported Widget Type Dictionaries:

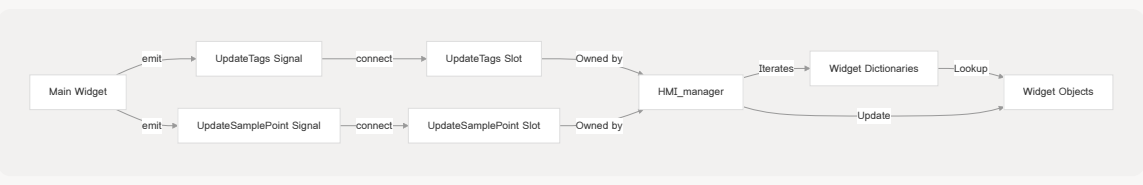
- `QwtThermo_dictionary`
- `QLCDNumber_dictionary`
- `PLCDNumber_dictionary`
- `SinglePointLed_dictionary`
- `DoublePointLed_dictionary`
- `PSinglePointLed_dictionary`
- `PDoublePointLed_dictionary`
- `PSwitch_dictionary`
- `Breaker_dictionary`
- `PTank_dictionary`
- `PThermometer_dictionary`
- `QPushButton_dictionary`
- `PMeter_dictionary`
- `QwtPlot_dictionary`

Sources:

- `src/ui/hmi_mng.cpp` 59-514
- `src/ui/hmi_mng.h` 17-67

Widget Update Mechanism

Once dictionaries are populated, widgets are updated through signal-slot connections:


Main Widget
emit
UpdateTags Signal
connect
UpdateTags Slot
Owned by
emit
UpdateSamplePoint Signal
connect
UpdateSamplePoint Slot
Owned by
HMI_manager
Iterates
Widget Dictionaries
Lookup
Update
Widget Objects

Update Signals:

- `UpdateTags()` - Updates widget values from database
- `UpdateSamplePoint()` - Updates widget alarm states

These signals are emitted by the main application widget and connected during `HMI_manager` initialization:

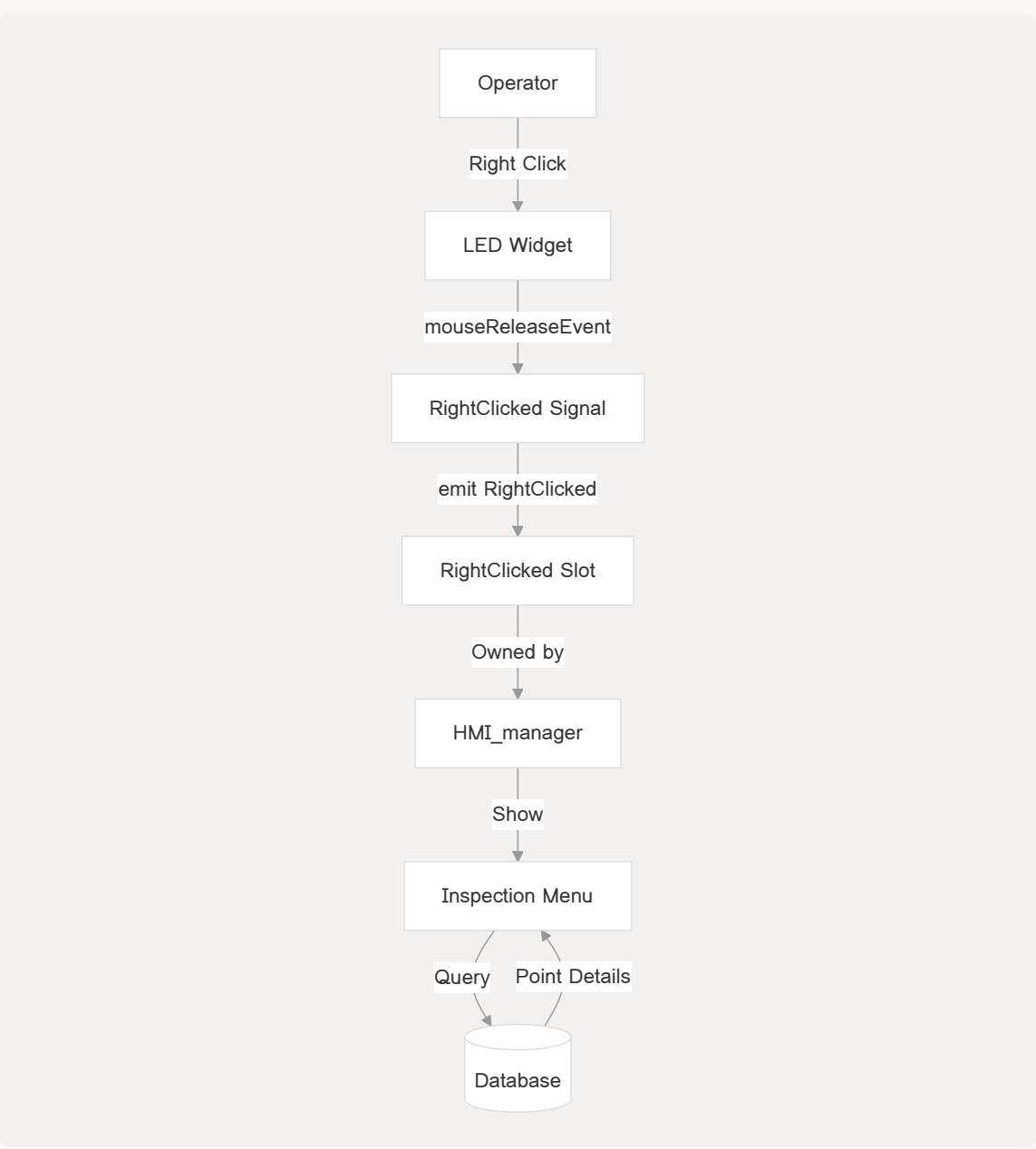
Sources:

- `src/ui/hmi_mng.cpp` 37-38
- `src/ui/hmi_mng.cpp` 812-841
- `src/ui/hmi_mng.cpp` 1012-1042

Widget Interaction and Commands

Right-Click Context Menus

Widgets support right-click inspection to view detailed data point information:


Operator
Right Click
LED Widget
mouseReleaseEvent
RightClicked Signal
emit RightClicked
RightClicked Slot
Owned by
HMI_manager
Show
Inspection Menu
Query
Point Details
Database

The right-click handler receives class name and widget name:

Implementation Pattern:

```
void Widget::mouseReleaseEvent(QMouseEvent *e)
{
    if(e->button() == Qt::RightButton)
    {
        QString name = this->name();
        emit RightClicked(QString("SinglePointLed"), name);
    }
}
```

Sources:

- `src/ui/qwt-4.2.0/src/single_point_led.cpp` 113-122
- `src/ui/qwt-4.2.0/src/elswitch.cpp` 51-65
- `src/ui/hmi_mng.cpp` 1126-1383

Control Commands

Interactive widgets like `PSwitch` and `Breaker` allow operators to send commands:

Command Flow:

1. User clicks/toggles widget
2. Widget changes visual state
3. Widget emits `valueChanged` signal
4. `HMI_manager` receives signal
5. Manager queries widget for new value
6. Manager sends command through dispatcher to `monitor.exe`
7. Monitor routes command to protocol driver
8. Driver executes command on field device

Sources:

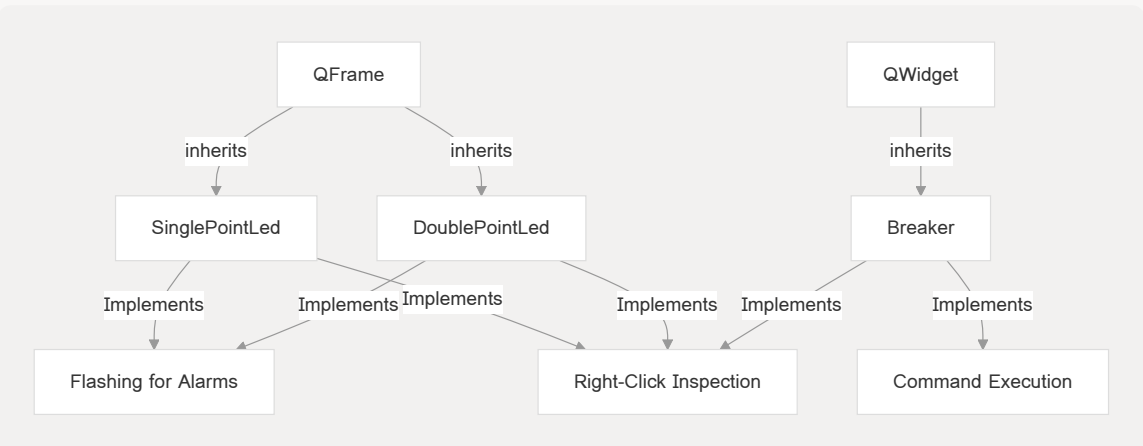
- `src/ui/hmi_mng.cpp` 1045-1074
- `src/ui/hmi_mng.cpp` 1077-1116

Custom Widget Implementation

Widget Base Classes

SCADA widgets inherit from standard Qt widget classes and implement SCADA-specific behaviors:

Inheritance Hierarchy:


QFrame
QWidget
inherits
inherits
inherits
SinglePointLed
DoublePointLed
Breaker
Implements
Implements
Implements
Implements
Implements
Implements
Flashing for Alarms
Right-Click Inspection
Command Execution

Example: Breaker Widget

The `Breaker` widget demonstrates the electrical switch pattern:

Key Features:

- Two pixmaps for open/closed states
- Read-only value property (controlled by data, not user clicks)
- Right-click inspection
- Invalid state visualization (disabled appearance)

Visual States:

- **Closed**: Displays `breaker_closed.xpm` pixmap
- **Opened**: Displays `breaker_opened.xpm` pixmap
- **Invalid**: Displayed as disabled (grayed out)

Property Management: The widget exposes Qt properties for designer configuration:

Sources:

- `src/ui/qwt-4.2.0/include/elswitch.h` 28-113
- `src/ui/qwt-4.2.0/src/elswitch.cpp` 17-39
- `src/ui/qwt-4.2.0/src/elswitch.cpp` 85-121
- `src/ui/qwt-4.2.0/src/breaker_opened.xpm` 1-65
- `src/ui/qwt-4.2.0/src/breaker_closed.xpm` 1-65

Example: SinglePointLed Widget

The `SinglePointLed` demonstrates indicator pattern with alarm flashing:

States:

- Green: State OFF (value = 0)
- Red: State ON (value = 1)
- Yellow: Intermediate (values 2-3)
- White: Invalid/None
- Blue: Communication failure

Alarm Flashing: When an unacknowledged alarm occurs, the LED flashes using a QTimer:

```
void SinglePointLed::startFlash()
{
    if(timer == NULL)
    {
        isFlashing = true;
        timer = new QTimer(this);
        timer->start(1000);  // 1 second interval
        connect(timer, SIGNAL(timeout()), this, SLOT(timerSlot()));
    }
}

void SinglePointLed::timerSlot()
{
    toggleState();  // Toggle between On and Off
}
```

Sources:

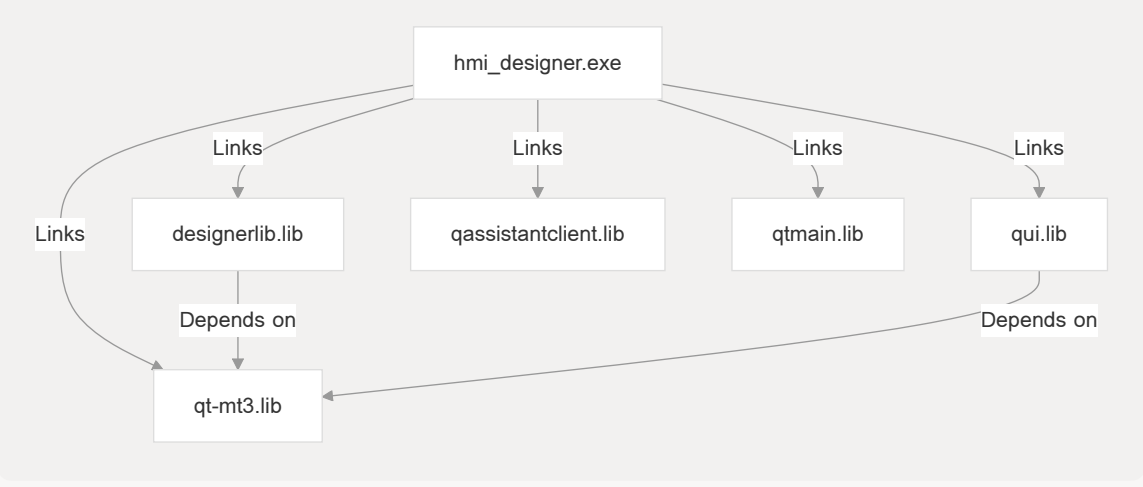
- `src/ui/qwt-4.2.0/include/single_point_led.h` 39-96
- `src/ui/qwt-4.2.0/src/single_point_led.cpp` 15-25
- `src/ui/qwt-4.2.0/src/single_point_led.cpp` 89-110

Build Configuration

Designer Build Dependencies

The designer executable links against multiple libraries:

Build Dependency Graph:


hmi_designer.exe
Links
Links
Links
Links
Links
designerlib.lib
qassistantclient.lib
qtmain.lib
qui.lib
Depends on
Depends on
qt-mt3.lib

Library Paths:

- Output: `c:\scada\bin\hmi_designer.exe`
- Library path: `$(QTDIR)\lib`
- Library path: `c:\scada\lib`

Sources:

- `src/hmi_designer/designer/app/designer.dsp` 32-77

Widget Library Build

The `qwt.dll` library is built as a shared library (DLL):

Compilation Flags:

- `/D "QWT_DLL"` - Export symbols for DLL
- `/D "QWT_MAKEDLL"` - Building DLL (not using)
- `/I "include"` - Widget headers
- `/I "$(QTDIR)\include"` - Qt headers

Output:

- DLL: `c:\scada\bin\qwt.dll`
- Import library: `c:\scada\lib\qwt.lib`

Sources:

- `src/ui/qwt-4.2.0/qwt.dsp` 32-82

Plugin Build

The `qwtplugin.dll` is built as a Qt Designer plugin:

Plugin Requirements:

- Must be placed in `C:\scada\plugins\designer\`
- Links against `qwt.lib`
- Defines `/D "QT_PLUGIN"` to indicate plugin build
- Must export `Q_EXPORT_PLUGIN` macro

Image Resources: Widget icons are embedded using Qt's image collection feature:

Sources:

- `src/ui/qwt-4.2.0/designer/qwtplugin.dsp` 32-82
- `src/ui/qwt-4.2.0/designer/qwtplugin.dsp` 135-158

Summary

The HMI Designer system provides a complete design-time and runtime solution for creating SCADA graphical interfaces:

Design-Time:

- Extended Qt Designer 3.3.8 with SCADA widgets
- Plugin architecture (`qwtplugin.dll`) exposes custom widgets
- Visual design environment with property editor
- Saves screens as standard Qt `.ui` XML files

Runtime:

- `ui.exe` dynamically loads `.ui` files
- `HMI_manager` builds widget dictionaries based on names
- Widget names truncated at underscore to extract sample point key
- Widgets update via signals connected to database queries
- Right-click inspection and command execution supported

Widget Library:

- 14+ SCADA-specific widget types
- Built on Qwt 4.2.0 foundation
- Indicators (LEDs), controls (switches), displays (meters, tanks)
- Trend plotting with `QwtPlot`

Key Convention: Widget naming format: `<SamplePointName>_<Suffix>` where sample point name must match database `TAGS` table entries.

Sources:

- `src/daq/daq.dsw` 231-258
- `src/daq/daq.dsw` 870-891

- src/ui/qwt-4.2.0/designer/qwtplugin.cpp 1-242

Reports and Historical Data Queries

> Relevant source files

This page describes the reporting and historical data query capabilities of IndigoSCADA's HMI system. It covers the generation of system reports (alarms, events, audit logs), custom configurable reports, historical data browsing interfaces, and time-based data retrieval mechanisms.

For information about the configuration of report definitions, see 6.1 Sample Point Configuration. For information about real-time data display widgets, see 5.2 HMI Widgets and Displays.

Report Types

IndigoSCADA provides several categories of reports accessible through the HMI:

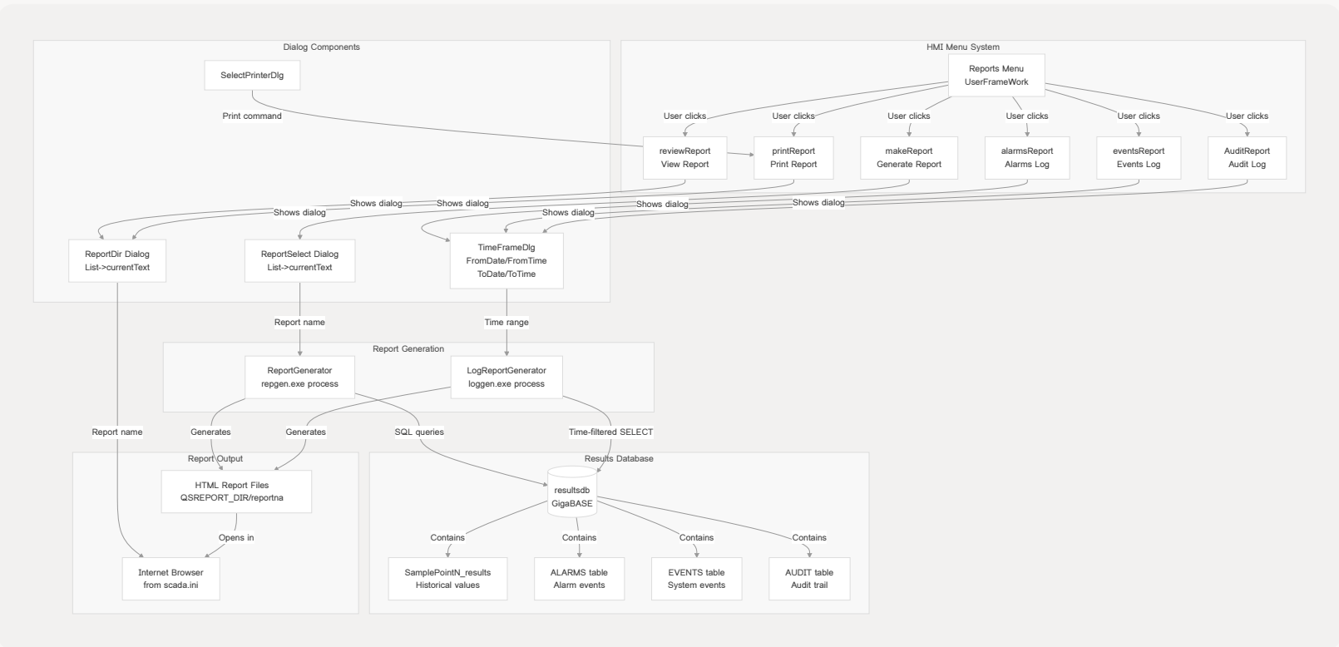
Report Type	Purpose	Generator	Source Database
Configured Reports	Custom reports based on sample point data	`ReportGenerator`	`resultsdb`
Alarms Report	Time-filtered alarm event log	`LogReportGenerator`	`resultsdb.ALARMS`
Events Report	Time-filtered system event log	`LogReportGenerator`	`resultsdb.EVENTS`
Audit Report	Time-filtered audit trail (admin only)	`LogReportGenerator`	`resultsdb.AUDIT`

Privilege Requirements: Report generation requires `PRIVS_REPORTS` privilege. Audit reports additionally require `PRIVS_ALL` privilege.

Sources: `src/ui/user.cpp 560-582` `src/ui/user.h 109-138`

Report Generation Workflow

Architecture Diagram

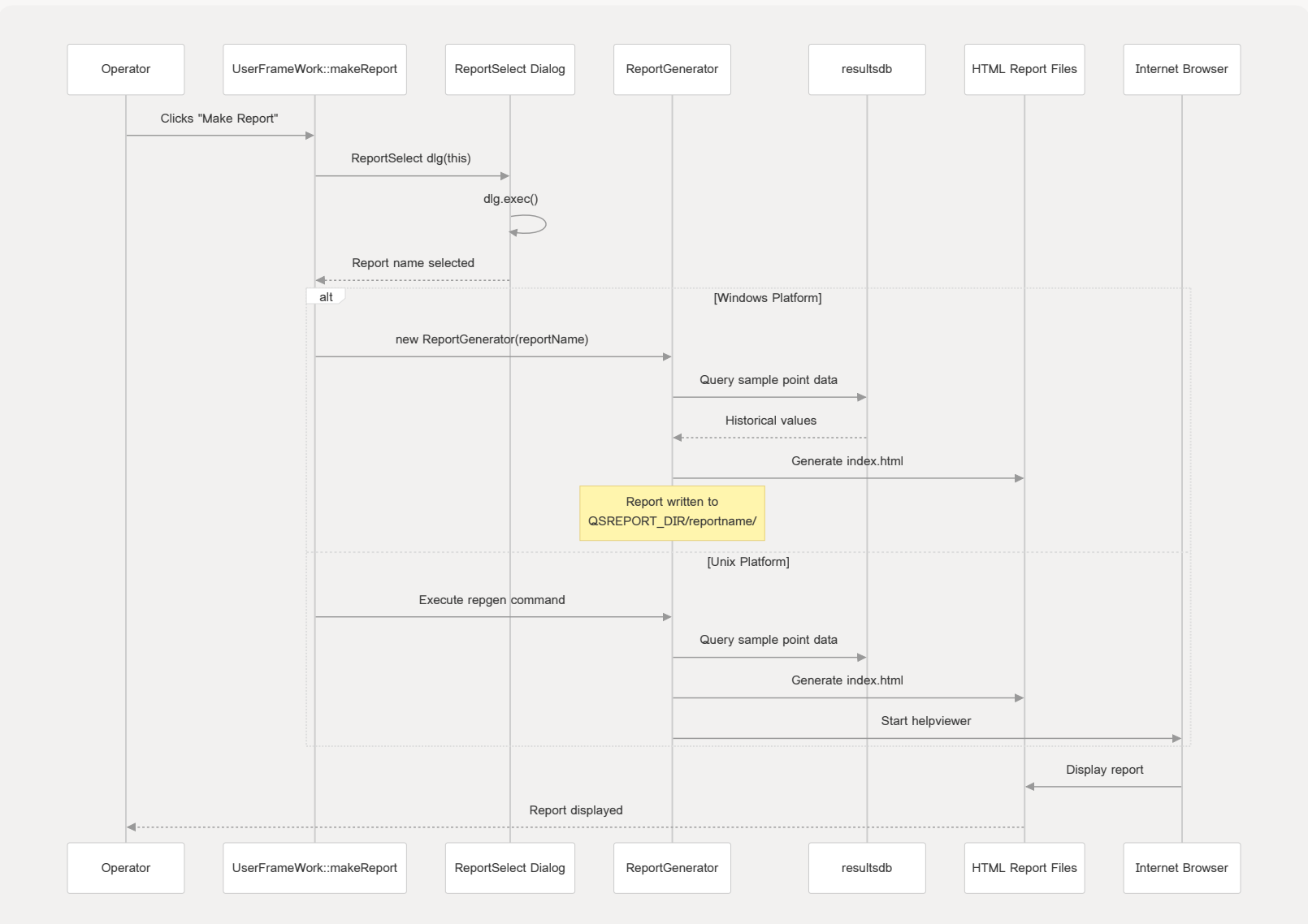

Dialog Components
SelectPrinterDlg
Print command
HMI Menu System
Reports Menu
UserFrameWork
User clicks
User clicks
User clicks
User clicks
User clicks
User clicks
reviewReport
View Report
printReport
Print Report
makeReport
Generate Report
alarmsReport
Alarms Log
eventsReport
Events Log
AuditReport
Audit Log
Shows dialog
Shows dialog
Shows dialog
Shows dialog
Shows dialog
Shows dialog
ReportDir Dialog
List->currentText
ReportSelect Dialog
List->currentText
TimeFrameDlg
FromDate/FromTime
ToDate/ToTime
Report name
Time range
Report Generation
ReportGenerator
repgen.exe process
LogReportGenerator
loggen.exe process
Report name
Generates
Generates
SQL queries
Time-filtered SELECT
Report Output
HTML Report Files
QSREPORT_DIR/reportna
Opens in
Internet Browser
from scada.ini
Results Database
resultsdb
GigaBASE
Contains
Contains
Contains
Contains
SamplePointN_results
Historical values
ALARMS table
Alarm events
EVENTS table
System events
AUDIT table
Audit trail

Report Generation Sequence

1. **User initiates report**: Operator selects report type from Reports menu
2. **Dialog prompts**: System displays selection or time frame dialog
3. **Generator spawned**: Windows process created for report generation executable
4. **Database queries**: Generator executes SQL queries against `resultsdb`
5. **HTML output**: Report written to `QSREPORT_DIR/reportname/index.html`
6. **Browser launch**: Configured browser opens report for viewing

Sources: `src/ui/user.cpp 933-975` `src/ui/user.cpp 982-1040` `src/ui/user.cpp 1046-1078` `src/ui/user.cpp 1084-1119` `src/ui/user.cpp 1125-1162`

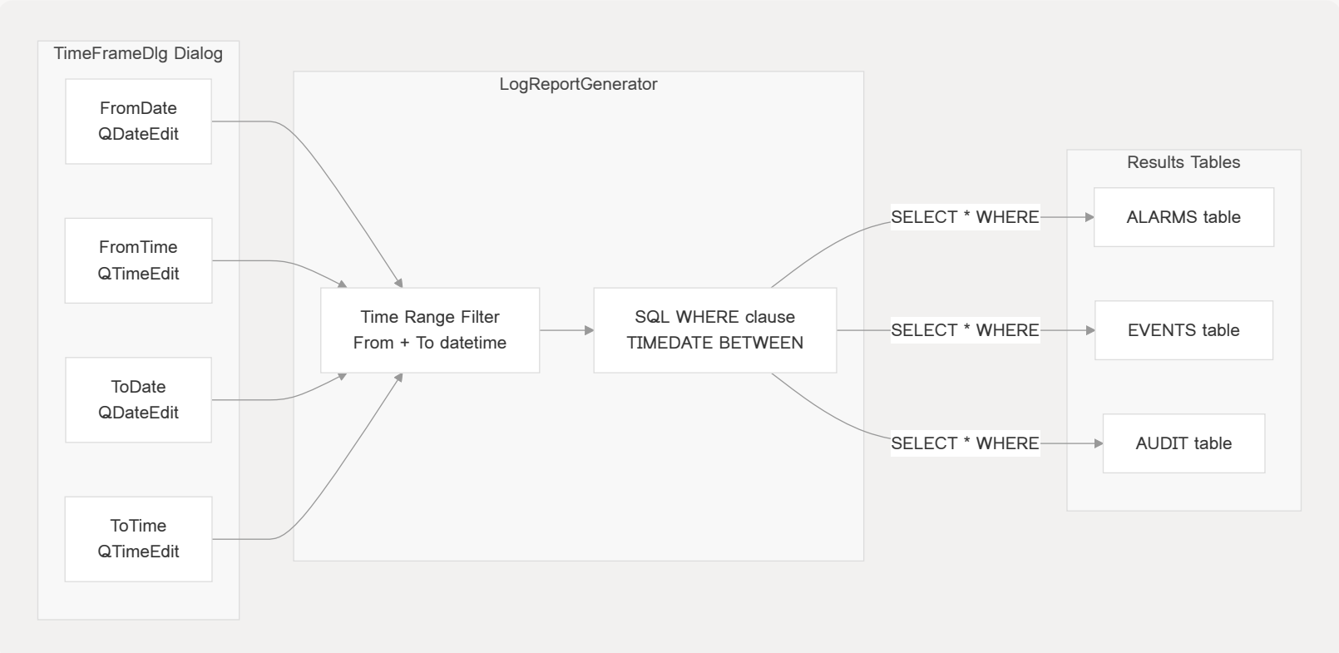
Make Report Implementation


Operator
UserFrameWork::makeReport
ReportSelect Dialog
ReportGenerator
resultsdb
HTML Report Files
Internet Browser
Clicks "Make Report"
ReportSelect dlg(this)
dlg.exec()
Report name selected
alt
[Windows Platform]
new ReportGenerator(reportName)
Query sample point data
Historical values
Generate index.html
Report written to
QSREPORT_DIR/reportname/
[Unix Platform]
Execute repgen command
Query sample point data
Generate index.html
Start helpviewer
Display report
Report displayed

The `makeReport()` function creates a new `ReportGenerator` instance on Windows `src/ui/user.cpp 970-972` which runs report generation in the background. On Unix systems, it spawns the `repgen` executable as a detached process `src/ui/user.cpp 950-965`

Sources: `src/ui/user.cpp 933-975`

Time Frame Selection for Log Reports


TimeFrameDlg Dialog
FromDate
QDateEdit
FromTime
QTimeEdit
ToDate
QDateEdit
ToTime
QTimeEdit
LogReportGenerator
Time Range Filter
From + To datetime
SQL WHERE clause
TIMEDATE BETWEEN
SELECT * WHERE
SELECT * WHERE
SELECT * WHERE
Results Tables
ALARMS table
EVENTS table
AUDIT table

The `TimeFrameDlg` dialog provides date/time pickers for filtering historical data. The selected range is formatted and passed to `LogReportGenerator` `src/ui/user.cpp 1072-1076`:

```
QString From_cmd = dlg.FromDate->text() + " " + dlg.FromTime->text();
QString To_cmd = dlg.ToDate->text() + " " + dlg.ToTime->text();
logrepgen = new LogReportGenerator("ALARMS", From_cmd, To_cmd);
```

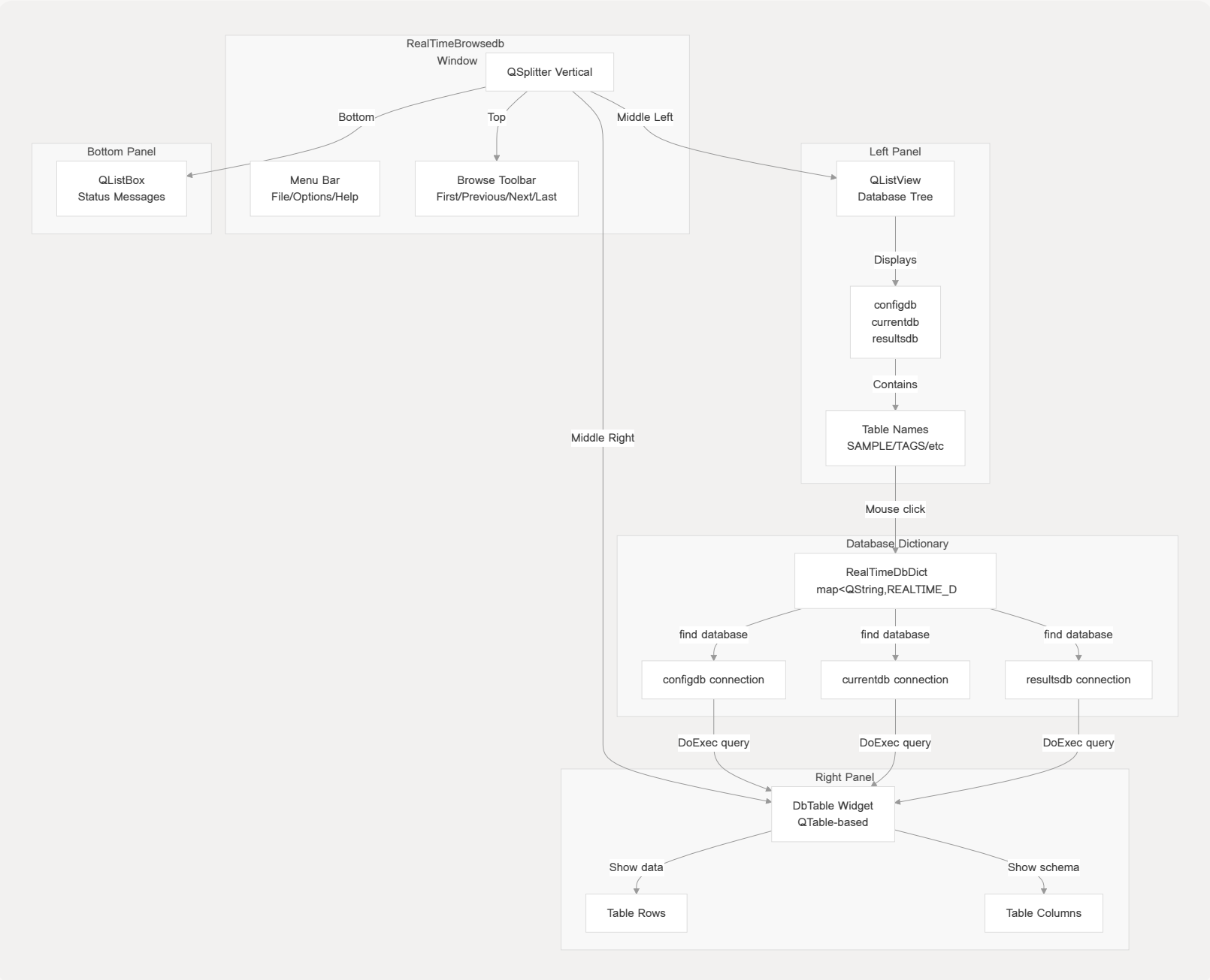
Sources: `src/ui/user.cpp 1050-1078` `src/ui/user.cpp 1088-1119` `src/ui/user.cpp 1129-1162`

Historical Data Browsing

IndigoSCADA provides two database browser interfaces for interactive historical data queries:

- **RealTimeBrowsedb**: Queries `configdb`, `currentdb`, and `resultsdb` (real-time databases)
- **HistoricBrowsedb**: Queries long-term historical archive databases

Real-Time Database Browser Architecture


RealTimeBrowsedb
Window
QSplitter Vertical
Bottom
Top
Middle Left
Bottom Panel
QListBox
Status Messages
Menu Bar
File/Options/Help
Browse Toolbar
First/Previous/Next/Last
Left Panel
QListView
Database Tree
Displays
configdb
currentdb
resultsdb
Contains
Table Names
SAMPLE/TAGS/etc
Middle Right
Mouse click
Database Dictionary
RealTimeDbDict
map<QString,REALTIME_D
find database
find database
find database
configdb connection
currentdb connection
resultsdb connection
DoExec query
DoExec query
DoExec query
Right Panel
DbTable Widget
QTable-based
Show data
Show schema
Table Rows
Table Columns

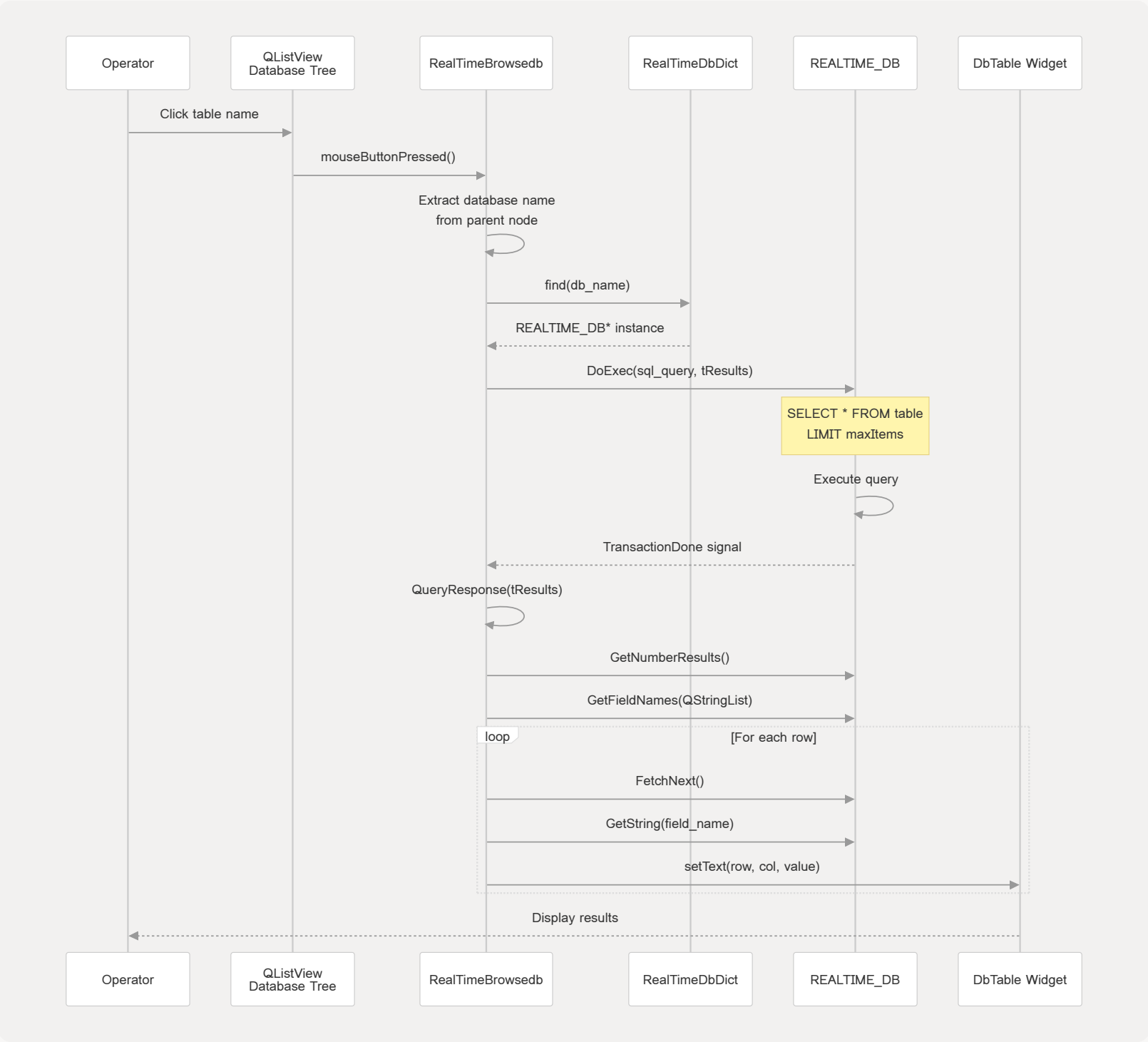
The browser maintains a dictionary of database connections `src/ui/realtime_browsedb.cpp 61-62`:

```
RealTimeDbDict db_dictionary;
```

When initialized, it connects to all online databases and displays them in a tree view `src/ui/realtime_browsedb.cpp 169-176`

Sources: `src/ui/realtime_browsedb.cpp 42-181`

Query Execution Flow


Operator
QListView
Database Tree
RealTimeBrowsedb
RealTimeDbDict
REALTIME_DB
DbTable Widget
Click table name
mouseButtonPressed()
Extract database name
from parent node
find(db_name)
REALTIME_DB* instance
DoExec(sql_query, tResults)
SELECT * FROM table
LIMIT maxItems
Execute query
TransactionDone signal
QueryResponse(tResults)
GetNumberResults()
GetFieldNames(QStringList)
loop
[For each row]
FetchNext()
GetString(field_name)
setText(row, col, value)
Display results

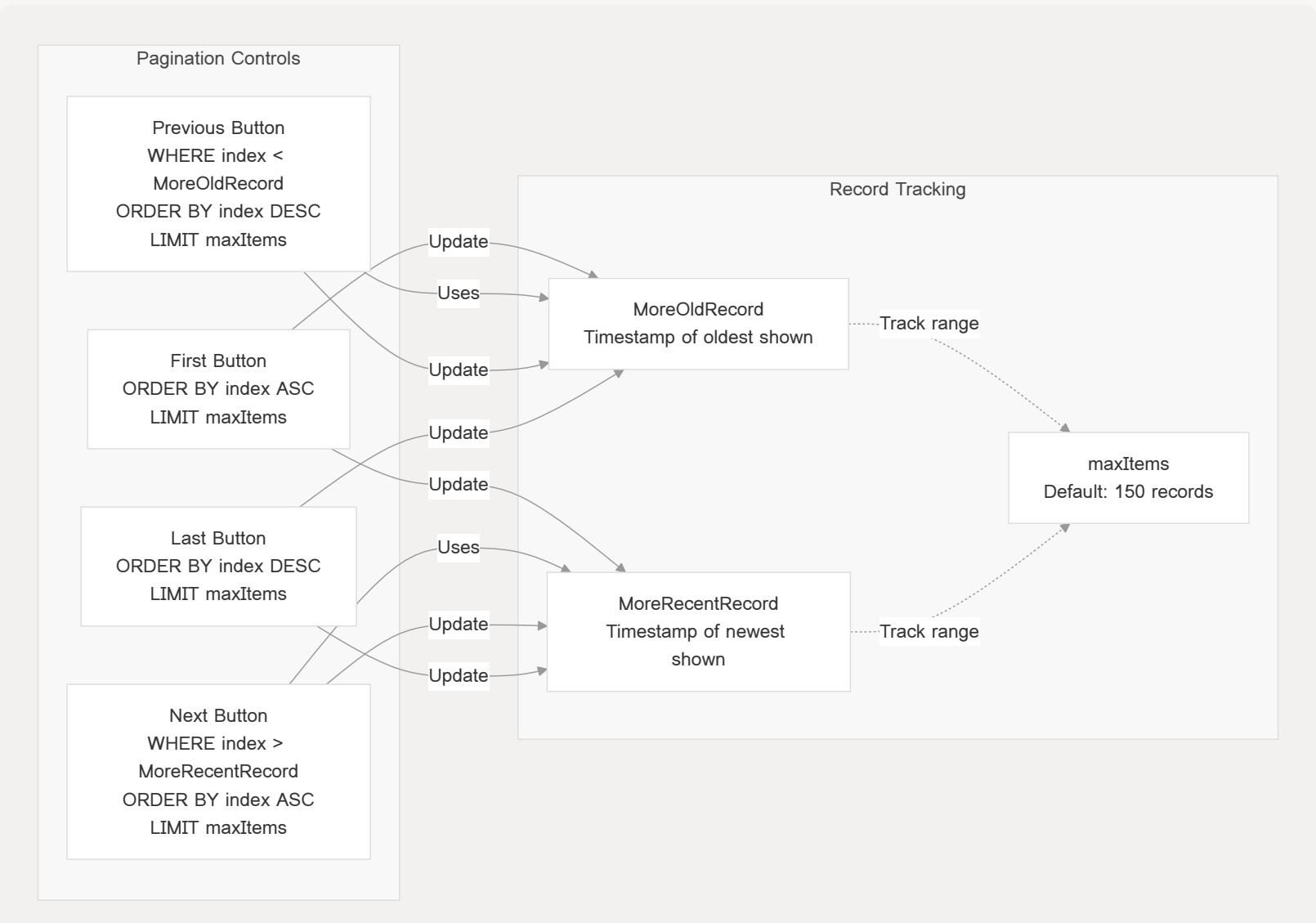
Query responses are handled in `RealTimeBrowsedb::QueryResponse()` `src/ui/realtime_browsedb.cpp 189-590` The method switches on transaction ID to handle different query types:

- `tList`: Build database/table tree `src/ui/realtime_browsedb.cpp 376-396`
- `tResults`: Display query results in table `src/ui/realtime_browsedb.cpp 199-301`
- `tBrowse`: Handle navigation (first/prev/next/last) `src/ui/realtime_browsedb.cpp 468-586`

Sources: `src/ui/realtime_browsedb.cpp 189-590` `src/ui/realtime_browsedb.cpp 636-664`

Result Set Pagination

The browser limits query results to prevent memory overflow and provides navigation buttons for paging through data:


Pagination Controls
Previous Button
WHERE index <
MoreOldRecord
ORDER BY index DESC
LIMIT maxItems
First Button
ORDER BY index ASC
LIMIT maxItems
Last Button
ORDER BY index DESC
LIMIT maxItems
Next Button
WHERE index >
MoreRecentRecord
ORDER BY index ASC
LIMIT maxItems
Update
Uses
Update
Update
Update
Uses
Update
Update
Record Tracking
MoreOldRecord
Timestamp of oldest shown
MoreRecentRecord
Timestamp of newest
shown
Track range
Track range
maxItems
Default: 150 records

The system tracks the oldest and newest record timestamps in the current view `src/ui/realtime_browsedb.cpp 256-277`:

```
if(k == 0)
{
    MoreOldRecord = timestamp;
    MoreRecentRecord = timestamp;
}
else
{
    if(timestamp > MoreRecentRecord)
        MoreRecentRecord = timestamp;
    if(timestamp < MoreOldRecord)
        MoreOldRecord = timestamp;
}
```

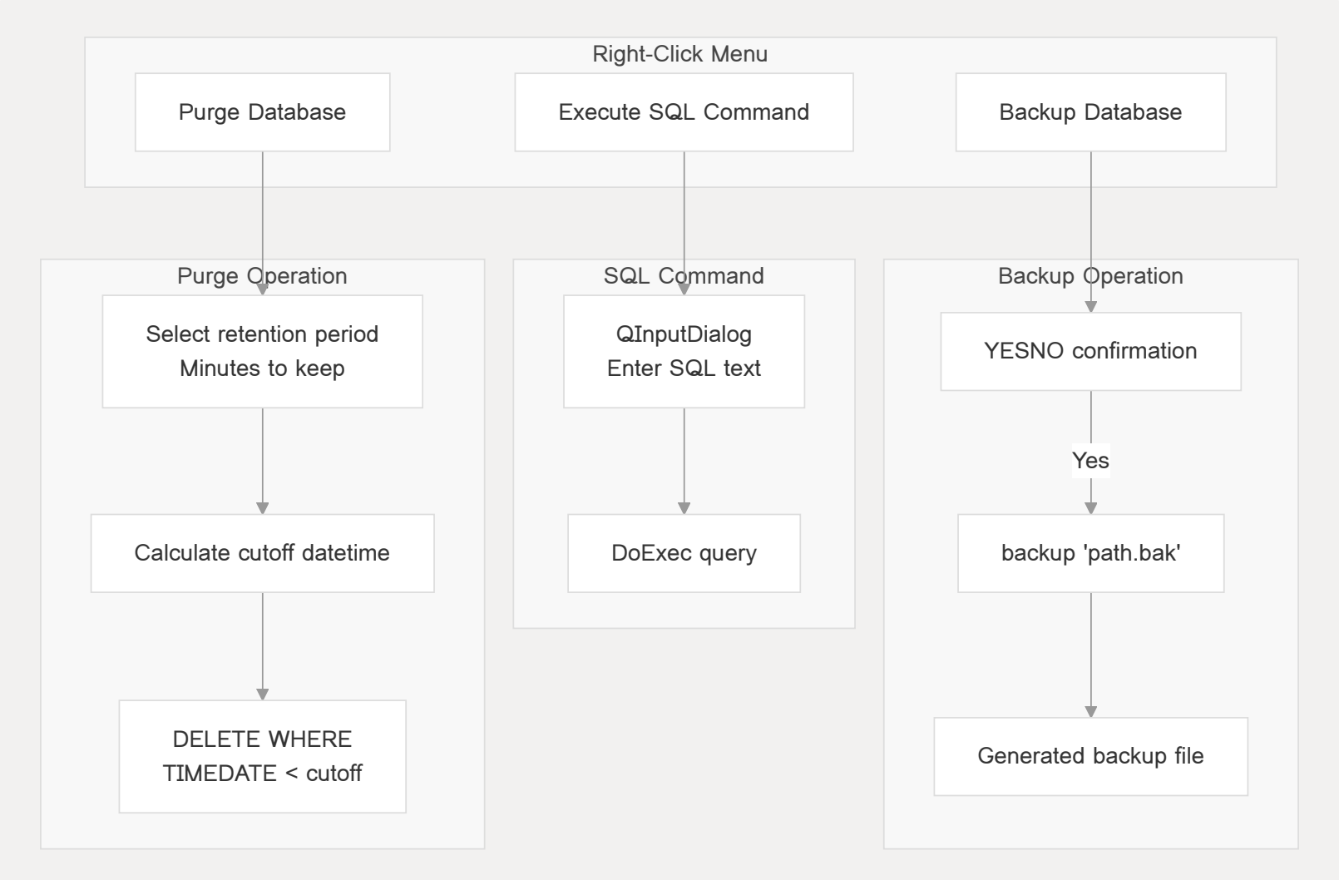
Navigation queries use these boundaries to fetch adjacent result sets `src/ui/realtime_browsedb.cpp 525-533`

Sources: `src/ui/realtime_browsedb.cpp 49-56` `src/ui/realtime_browsedb.cpp 199-301` `src/ui/realtime_browsedb.cpp 468-586`

Database Management Functions

The browser interfaces provide administrative functions accessible via right-click context menus:

Database Operations


Right-Click Menu
Purge Database
Execute SQL Command
Backup Database
Purge Operation
Select retention period
Minutes to keep
Calculate cutoff datetime
DELETE WHERE
TIMEDATE < cutoff
SQL Command
QInputDialog
Enter SQL text
DoExec query
Backup Operation
YESNO confirmation
Yes
backup 'path.bak'
Generated backup file

Purge Database: Removes historical records older than specified retention period `src/ui/realtime_browsedb.cpp 303-354`:

```
int back = QInputDialog::getInteger(tr("Purge Results Database"),
                                    tr("Last number of minutes to keep"),
                                    60, 1, 600, 1, &ok, this);
QDateTime dt = QDateTime::currentDateTime().addSecs(-back*60);
QString cmd = "delete from " + tableName + " where TIMEDATE < " + dt + ";";
```

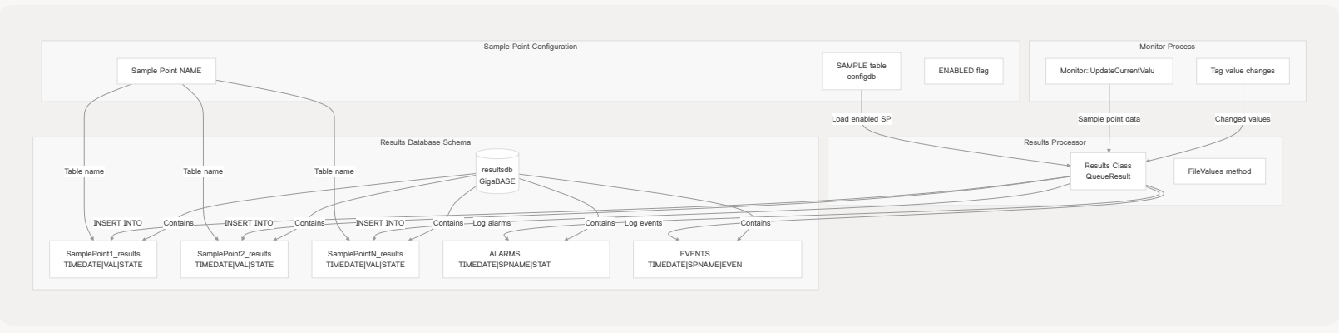
Backup Database: Creates a timestamped backup file `src/ui/realtime_browsedb.cpp 702-721`:

```
QString cmd = QString("backup ") + "'" + QSBIN_DIR + "\\" + DATETIME_NOW + ".bak';";
```

Sources: `src/ui/realtime_browsedb.cpp 668-732` `src/ui/realtime_browsedb.cpp 303-354`

Results Database Architecture

Historical data is stored in the `resultsdb` database (GigaBASE) with dynamic table creation for each sample point:

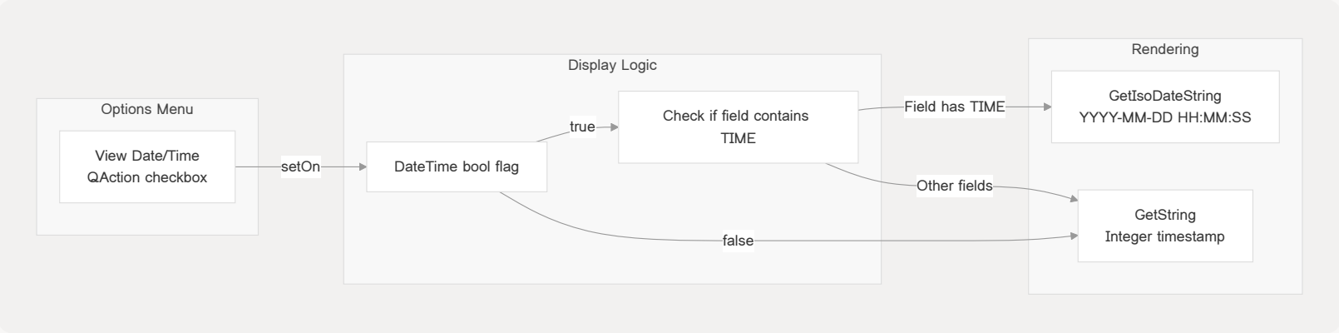

Sample Point Configuration
Sample Point NAME
SAMPLE table
configdb
ENABLED flag
Monitor Process
Monitor::UpdateCurrentValu
Tag value changes
Load enabled SP
Sample point data
Changed values
Results Database Schema
Results Processor
Table name
Table name
Table name
resultsdb
GigaBASE
Results Class
QueueResult
FileValues method
INSERT INTO
Contains
INSERT INTO
Contains
INSERT INTO
Contains
Log alarms
Contains
Log events
Contains
SamplePoint1_results
TIMEDATE|VAL|STATE
SamplePoint2_results
TIMEDATE|VAL|STATE
SamplePointN_results
TIMEDATE|VAL|STATE
ALARMS
TIMEDATE|SPNAME|STAT
EVENTS
TIMEDATE|SPNAME|EVEN

Each sample point gets its own results table created dynamically when the sample point is enabled. The table name is constructed from the sample point name with "_results" suffix.

Sources: From high-level architecture context, see Diagram 5 in the provided system overview

Date/Time Display Options

The browser provides a toggle for displaying timestamps in human-readable format:


Options Menu
View Date/Time
QAction checkbox
setOn
Display Logic
DateTime bool flag
true
Check if field contains
TIME
false
Field has TIME
Other fields
Rendering
GetIsoDateString
YYYY-MM-DD HH:MM:SS
GetString
Integer timestamp

When the Date/Time view option is enabled `src/ui/realtime_browsedb.cpp 120-148` timestamp fields are converted to ISO date format `src/ui/realtime_browsedb.cpp 237-247`:

```
if(DateTime)
{
    if((l[i]).contains("TIME"))
    {
        db_table->setText(k, i, Caller->GetIsoDateString(l[i]));
    }
    else
    {
        db_table->setText(k, i, Caller->GetString(l[i]));
    }
}
```

Sources: `src/ui/realtime_browsedb.cpp 120-148` `src/ui/realtime_browsedb.cpp 237-247`

Report Configuration

Custom reports are configured through the `ReportCfg` dialog (see 6.1 for configuration details). The configuration defines:

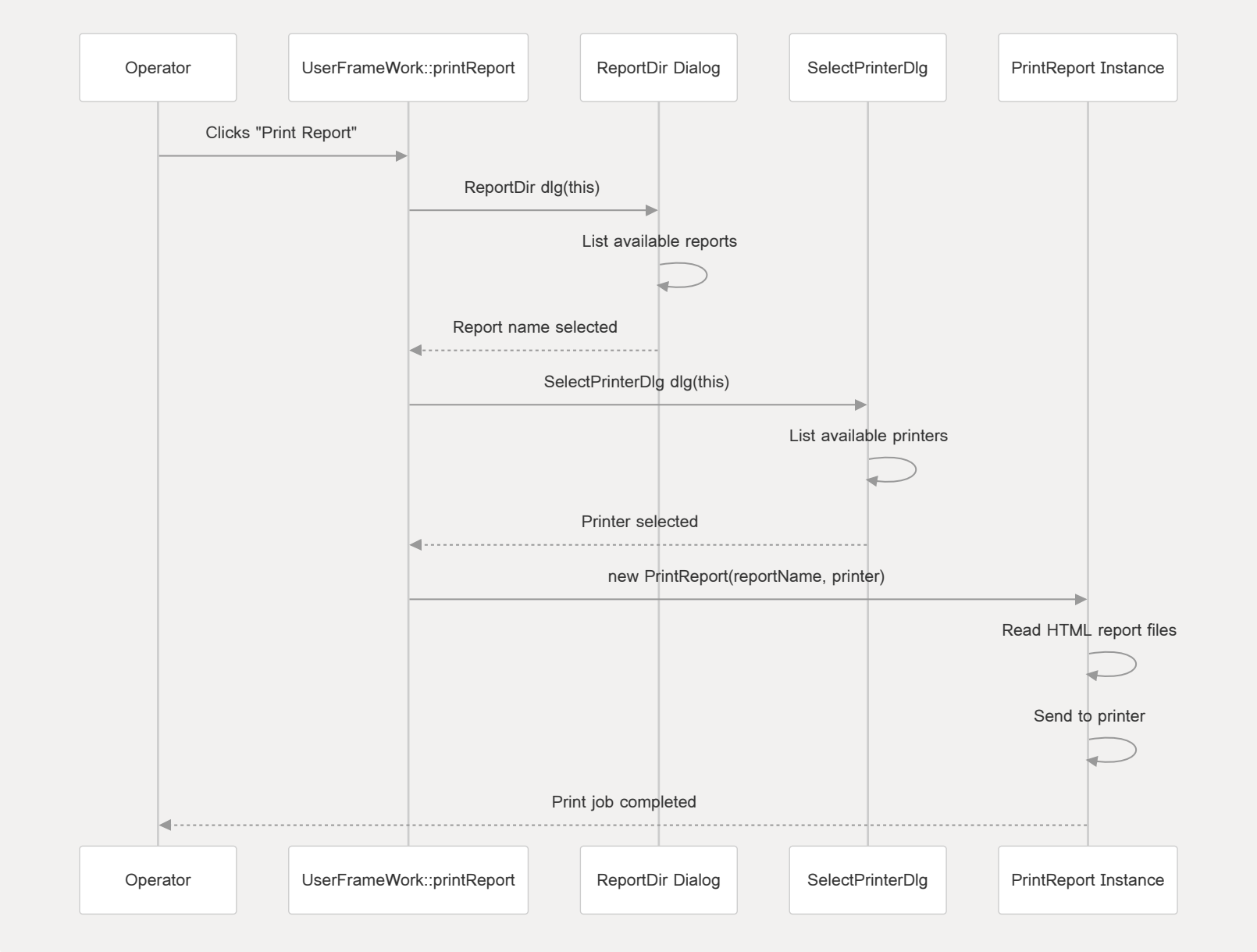
- Sample points to include in the report
- Data aggregation methods (min, max, average)
- Report layout and formatting
- Export format (HTML, PDF, etc.)

Report configurations are stored in the `REPORTS` table in `configdb`. The `makeReport()` function reads this configuration and generates the output files in `QSREPORT_DIR`.

Sources: `src/ui/user.cpp 1168-1174`

Print Report Functionality

The print report feature allows operators to send generated reports to a configured printer:

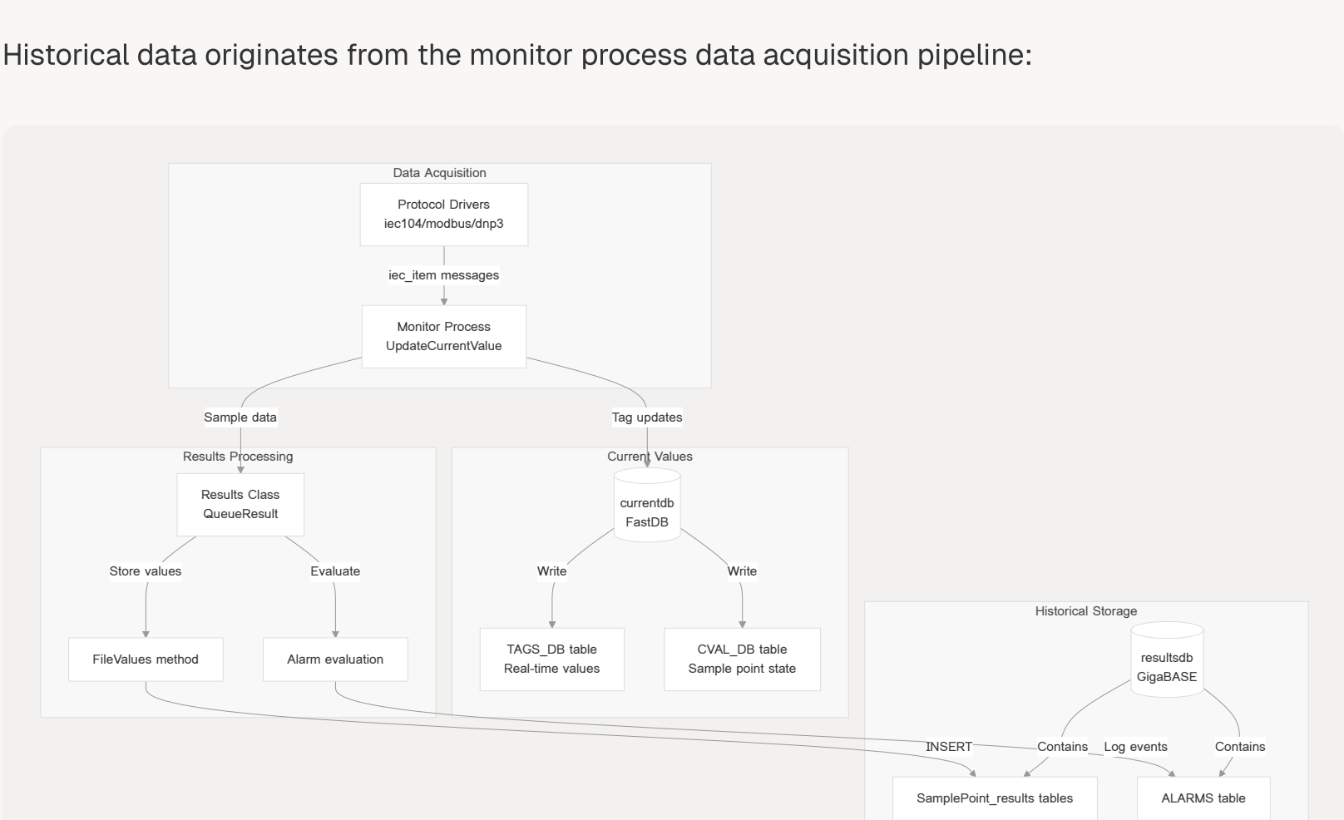

Operator
UserFrameWork::printReport
ReportDir Dialog
SelectPrinterDlg
PrintReport Instance
Clicks "Print Report"
ReportDir dlg(this)
List available reports
Report name selected
SelectPrinterDlg dlg(this)
List available printers
Printer selected
new PrintReport(reportName, printer)
Read HTML report files
Send to printer
Print job completed

The print functionality creates a `PrintReport` instance that handles the printing operation `src/ui/user.h 70`

Sources: `src/ui/user.h 26` `src/ui/user.h 70`

Connection to Real-Time Data Pipeline

Historical data originates from the monitor process data acquisition pipeline:


Data Acquisition
Protocol Drivers
iec104/modbus/dnp3
iec_item messages
Monitor Process
UpdateCurrentValue
Sample data
Tag updates
Results Processing
Results Class
QueueResult
Store values
Evaluate
FileValues method
Alarm evaluation
Current Values
currentdb
FastDB
Write
Write
TAGS_DB table
Real-time values
CVAL_DB table
Sample point state
Historical Storage
resultsdb
GigaBASE
INSERT
Contains
Log events
Contains
SamplePoint_results tables
ALARMS table

The monitor process `src/monitor/monitor.cpp 349-451` updates both the current database (for real-time display) and queues data for the results processor (for historical storage). The `UpdateCurrentValue` method writes to `TAGS_DB` and `CVAL_DB` tables `src/monitor/monitor.cpp 362-449` while the results processor handles long-term archival.

Sources: `src/monitor/monitor.cpp 349-451`

Report Viewer Integration

Reports are viewed using a configured web browser specified in the project's `scada.ini` file:

```
QString ini_file = GetScadaProjectDirectory() + "\\scada.ini";
Inifile iniFile((const char*)ini_file);
QString path = iniFile.find("path","internet_browser");
QString cmd = path + " " + QSREPORT_DIR + "\\" + reportName + "\\index.html";
```

The system creates a new Windows process to launch the browser `src/ui/user.cpp 1011-1028` separating the viewer from the HMI application.

Sources: `src/ui/user.cpp 994-1039`

Unit Configuration

> Relevant source files

Purpose and Scope

Unit Configuration defines and manages **units** in IndigoSCADA. A unit represents a driver instance that communicates with field devices or data sources. Each unit has a unique name, a driver type (e.g., IEC 104, Modbus, OPC), configuration properties, and can have multiple sample points associated with it. This page covers the unit configuration dialog, database schema, driver selection, and unit-specific configuration mechanisms.

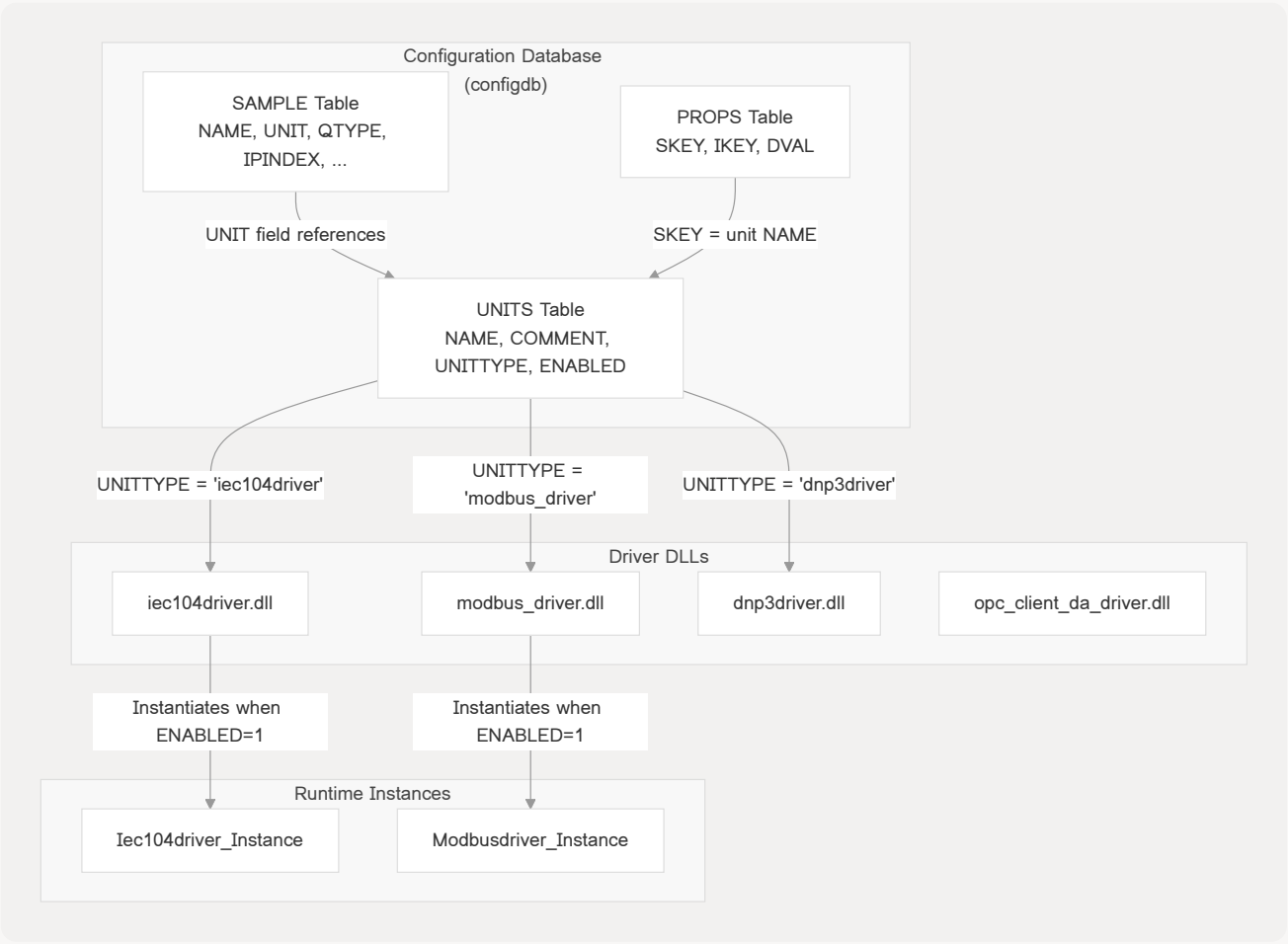
For information about configuring sample points within units, see Sample Point Configuration. For protocol-specific configuration tools, see Protocol Configuration Tools.

Unit Data Model

Units are stored in the `UNITS` table within the `configdb` (FastDB) database. Each unit record defines a driver instance that will be started by the monitor process when enabled.

UNITS Table Schema

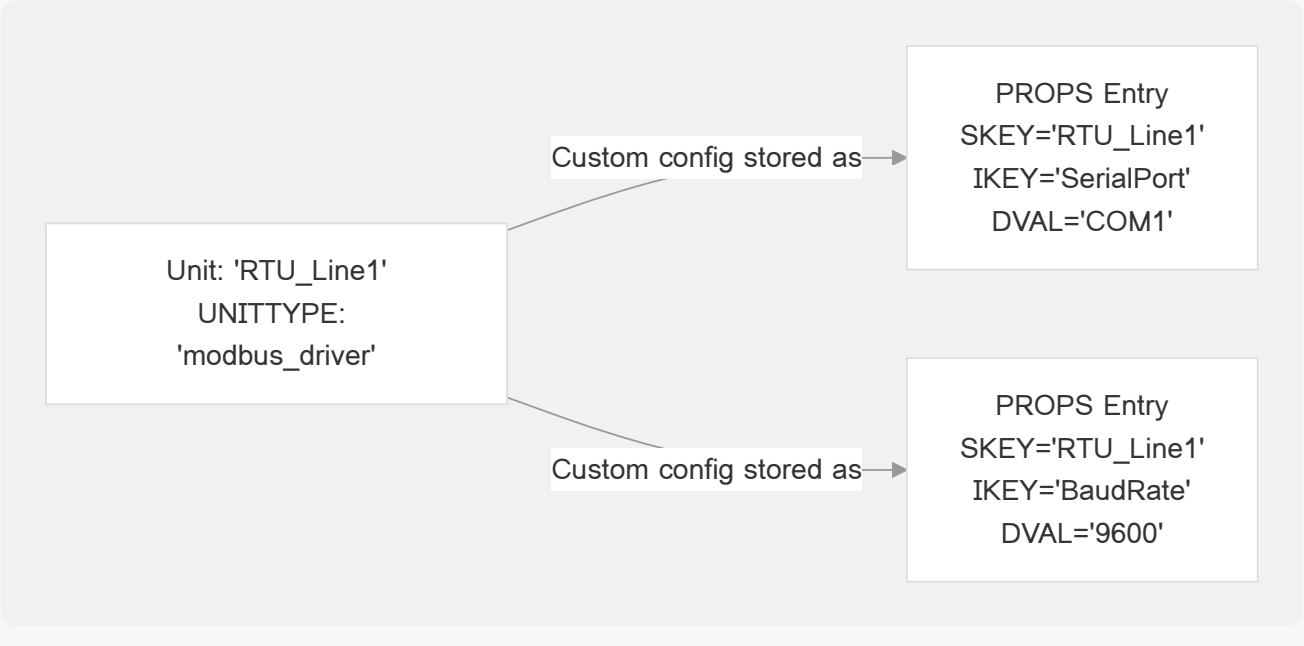
Field	Type	Description
NAME	string	Unique unit identifier (primary key)
COMMENT	string	Human-readable description
UNITTYPE	string	Driver type name (matches driver DLL basename)
ENABLED	boolean	Whether unit should be started on system startup


Configuration Database (configdb)
SAMPLE Table NAME, UNIT, QTYPE, IPINDEX, ...
PROPS Table SKEY, IKEY, DVAL
UNIT field references
SKEY = unit NAME
UNITS Table NAME, COMMENT, UNITTYPE, ENABLED
UNITTYPE = 'iec104driver'
UNITTYPE = 'modbus_driver'
UNITTYPE = 'dnp3driver'
Driver DLLs
iec104driver.dll
modbus_driver.dll
dnp3driver.dll
opc_client_da_driver.dll
Instantiates when ENABLED=1
Instantiates when ENABLED=1
Runtime Instances
Iec104driver_Instance
Modbusdriver_Instance

Sources: `src/ui/sysdialogs/UnitCfg.cpp 1-268` `src/ui/sysdialogs/SampleCfg.cpp 298-308`

Unit Properties Storage

Unit-specific configuration is stored in the `PROPS` table with `SKEY` equal to the unit name. This allows each driver type to store custom configuration data.

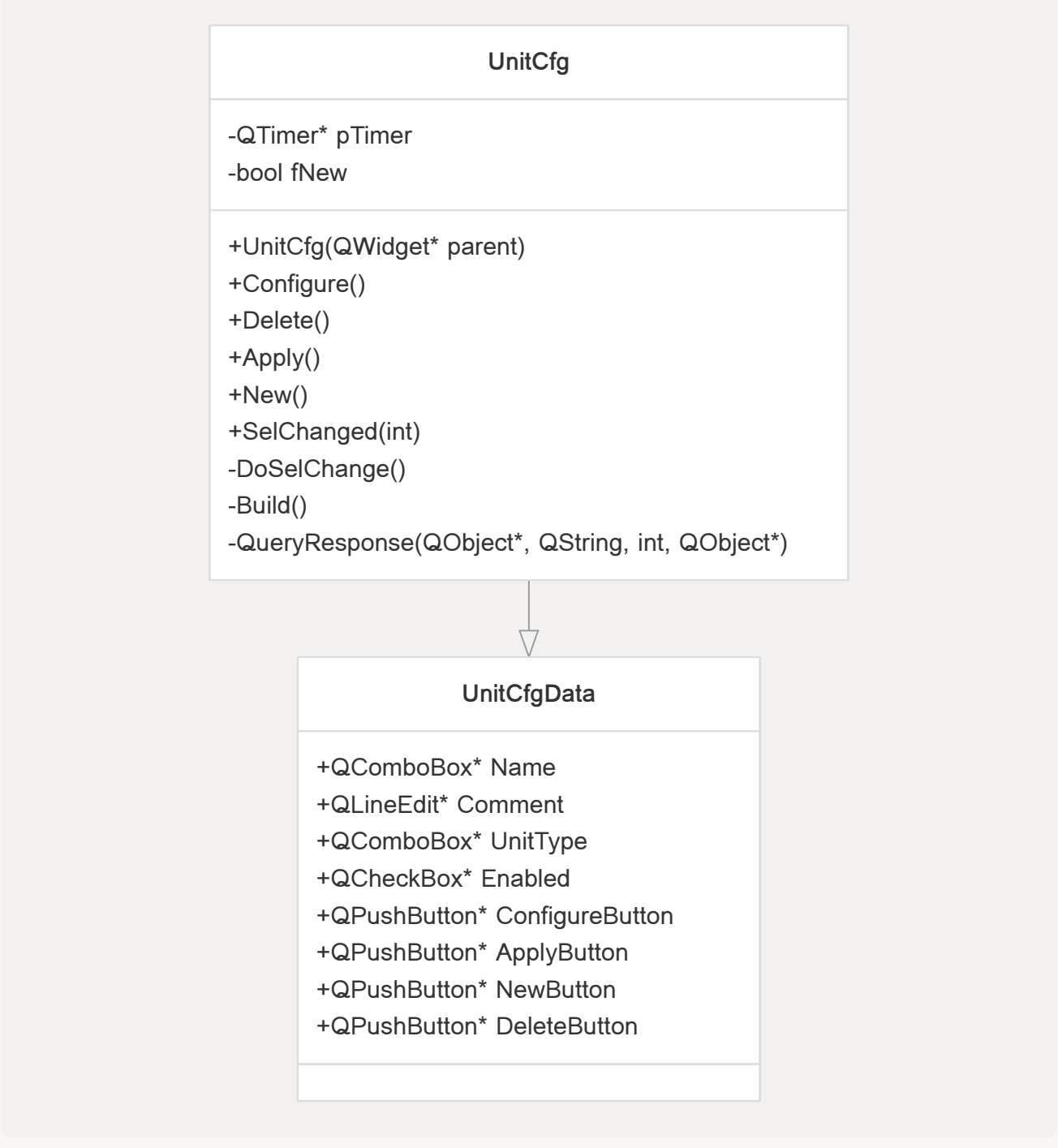

Unit: 'RTU_Line1' UNITTYPE: 'modbus_driver'
Custom config stored as
PROPS Entry SKEY='RTU_Line1' IKEY='SerialPort' DVAL='COM1'
Custom config stored as
PROPS Entry SKEY='RTU_Line1' IKEY='BaudRate' DVAL='9600'

Sources: `src/ui/sysdialogs/UnitCfg.cpp 105` `src/ui/sysdialogs/SampleCfg.cpp 96-98`

Unit Configuration Dialog (UnitCfg)

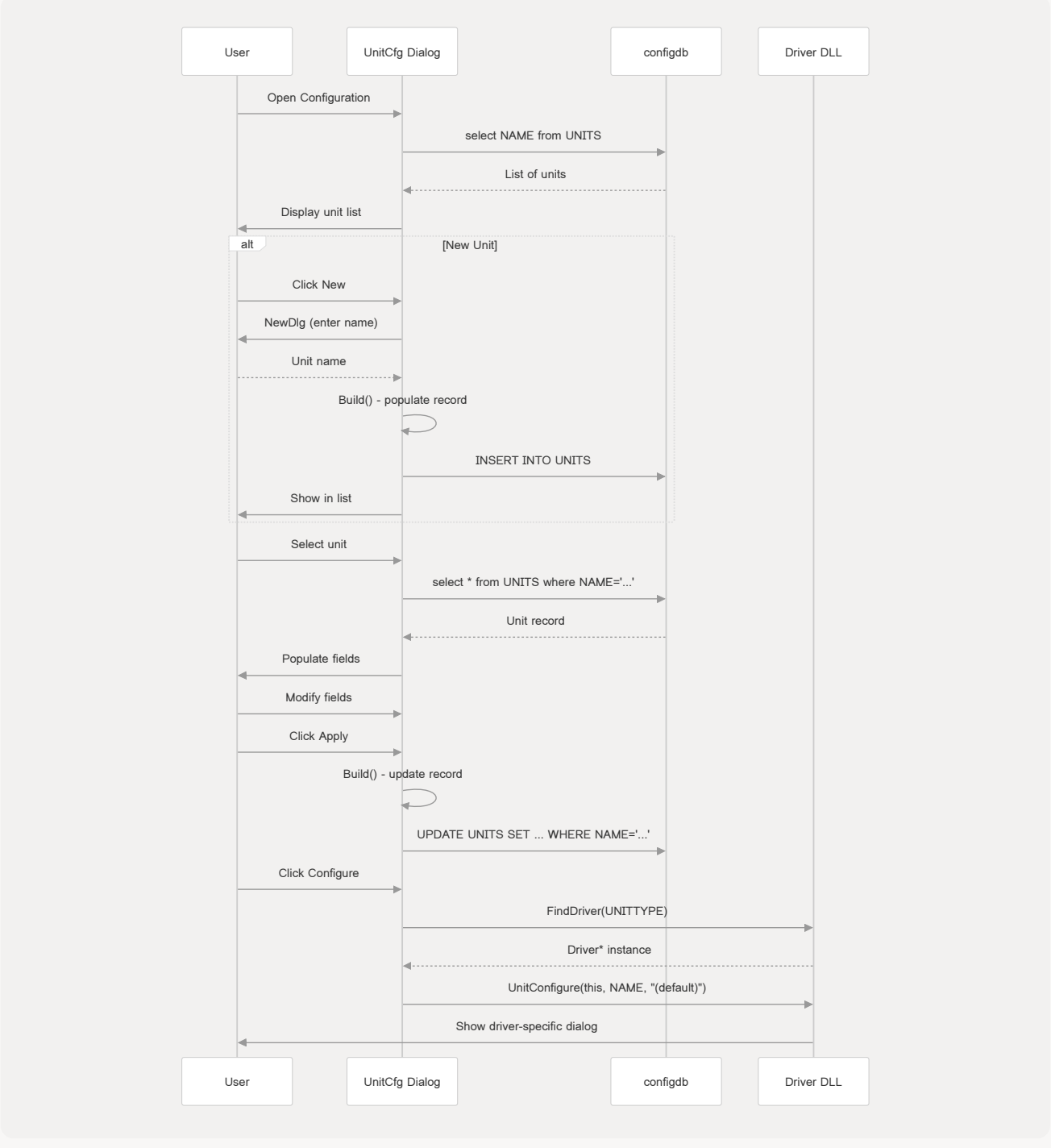
The `UnitCfg` class (`src/ui/sysdialogs/UnitCfg.cpp`) provides the user interface for managing units. It is invoked from the main system configuration menu.

UnitCfg Class Structure


UnitCfg
-QTimer* pTimer
-bool fNew
+UnitCfg(QWidget* parent)
+Configure()
+Delete()
+Apply()
+New()
+SelChanged(int)
-DoSelChange()
-Build()
-QueryResponse(QObject*, QString, int, QObject*)
UnitCfgData
+QComboBox* Name
+QLineEdit* Comment
+QComboBox* UnitType
+QCheckBox* Enabled
+QPushButton* ConfigureButton
+QPushButton* ApplyButton
+QPushButton* NewButton
+QPushButton* DeleteButton

Sources: `src/ui/sysdialogs/UnitCfg.cpp 18-57`

Configuration Workflow


User
UnitCfg Dialog
configdb
Driver DLL
Open Configuration
select NAME from UNITS
List of units
Display unit list
alt
[New Unit]
Click New
NewDlg (enter name)
Unit name
Build() - populate record
INSERT INTO UNITS
Show in list
Select unit
select * from UNITS where NAME='...'
Unit record
Populate fields
Modify fields
Click Apply
Build() - update record
UPDATE UNITS SET ... WHERE NAME='...'
Click Configure
FindDriver(UNITTYPE)
Driver* instance
UnitConfigure(this, NAME, "(default)")
Show driver-specific dialog

Sources: `src/ui/sysdialogs/UnitCfg.cpp 114-157` `src/ui/sysdialogs/UnitCfg.cpp 62-72`

Key Methods

New() - Creating a Unit

`src/ui/sysdialogs/UnitCfg.cpp 137-157` handles unit creation:

1. Displays `NewDlg` for name entry
2. Inserts new record with default values into `UNITS` table
3. Sets `fNew` flag to trigger automatic configuration dialog
4. Generates audit trail entry

Apply() - Saving Changes

`src/ui/sysdialogs/UnitCfg.cpp 122-136` saves modified unit data:

1. Calls `Build()` to populate database record from UI fields
2. Executes `UPDATE UNITS SET ... WHERE NAME='...'`
3. If `fNew` is true, automatically invokes `Configure()` for first-time setup

Delete() - Removing a Unit

`src/ui/sysdialogs/UnitCfg.cpp 73-113` performs cascading deletion:

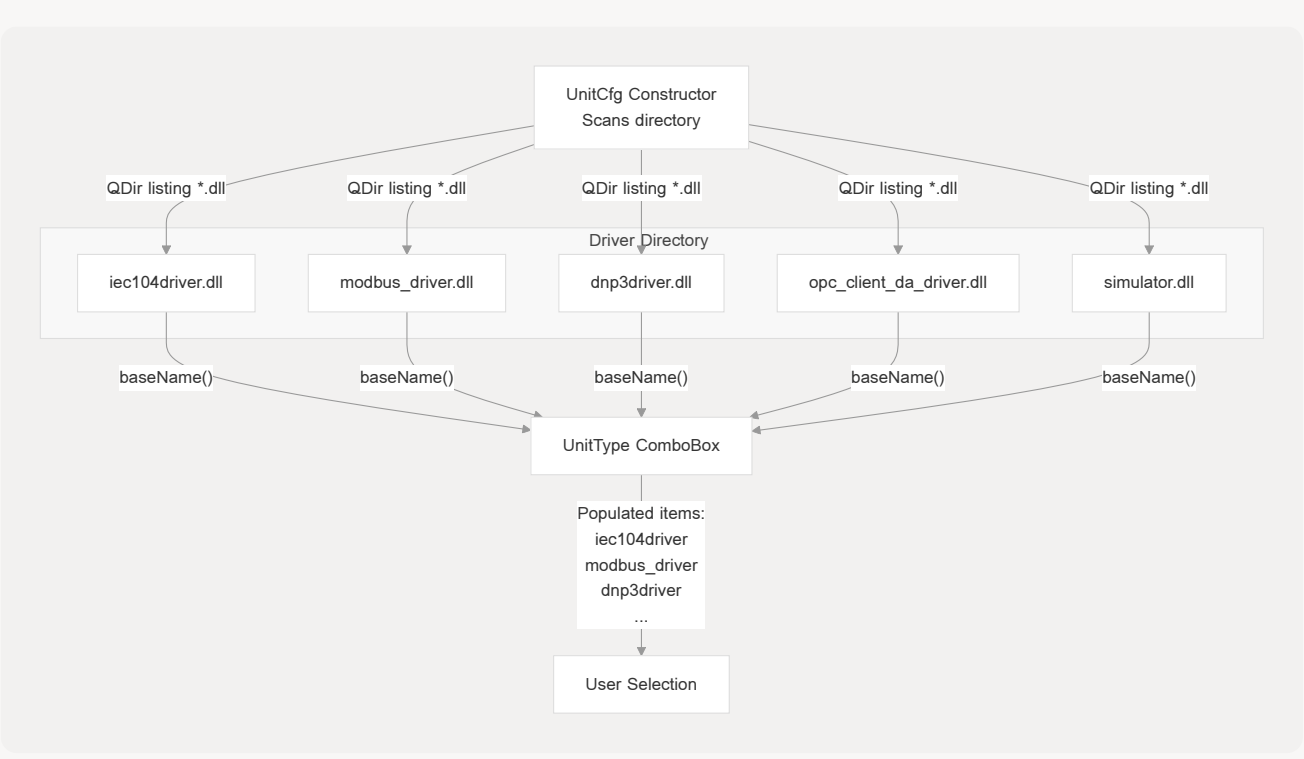
1. Deletes unit record from `UNITS` table
2. Optionally deletes associated sample points (with user confirmation)
3. Removes unit properties from `PROPS` table
4. Drops sample point results tables from `resultsdb`

Sources: `src/ui/sysdialogs/UnitCfg.cpp 73-157`

Unit Types and Driver Selection

Driver Discovery

Unit types are discovered by scanning the driver directory for DLL files. The `UNITTYPE` field must match the basename of a driver DLL.


UnitCfg Constructor Scans directory
QDir listing *.dll
QDir listing *.dll
QDir listing *.dll
QDir listing *.dll
QDir listing *.dll
Driver Directory
iec104driver.dll
modbus_driver.dll
dnp3driver.dll
opc_client_da_driver.dll
simulator.dll
baseName()
baseName()
baseName()
baseName()
baseName()
UnitType ComboBox
Populated items: iec104driver modbus_driver dnp3driver ...
User Selection

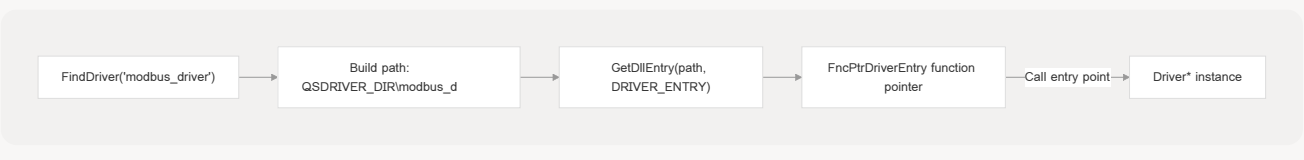
The driver directory scanning code `src/ui/sysdialogs/UnitCfg.cpp 30-43` :

```
QDir d (QSDRIVER_DIR, "*" DLL_EXT, QDir::Name, QDir::Files);
if (d.count ())
{
    const QFileInfoList *pD = d.entryInfoList ();
    QFileInfoListIterator it (*pD);
    QFileInfo *fi;
    while ((fi = it.current ()))
    {
        UnitType->insertItem (fi->baseName ());
        ++it;
    };
};
```

Sources: `src/ui/sysdialogs/UnitCfg.cpp 30-43`

Driver Loading and Interface

The `FindDriver()` helper function loads a driver DLL and returns its interface:

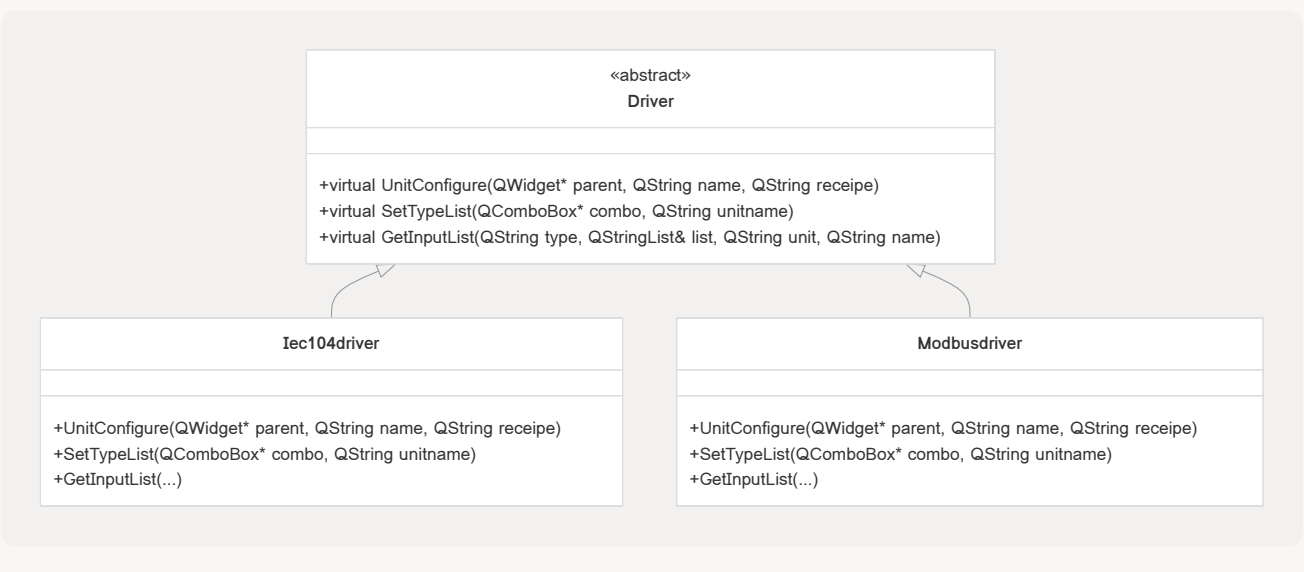

FindDriver('modbus_driver')
Build path: QSDRIVER_DIR\modbus_d
GetDllEntry(path, DRIVER_ENTRY)
FncPtrDriverEntry function pointer
Call entry point
Driver* instance

Sources: `src/drivers/driver.h 527-550`

Unit-Specific Configuration

Driver Configuration Interface

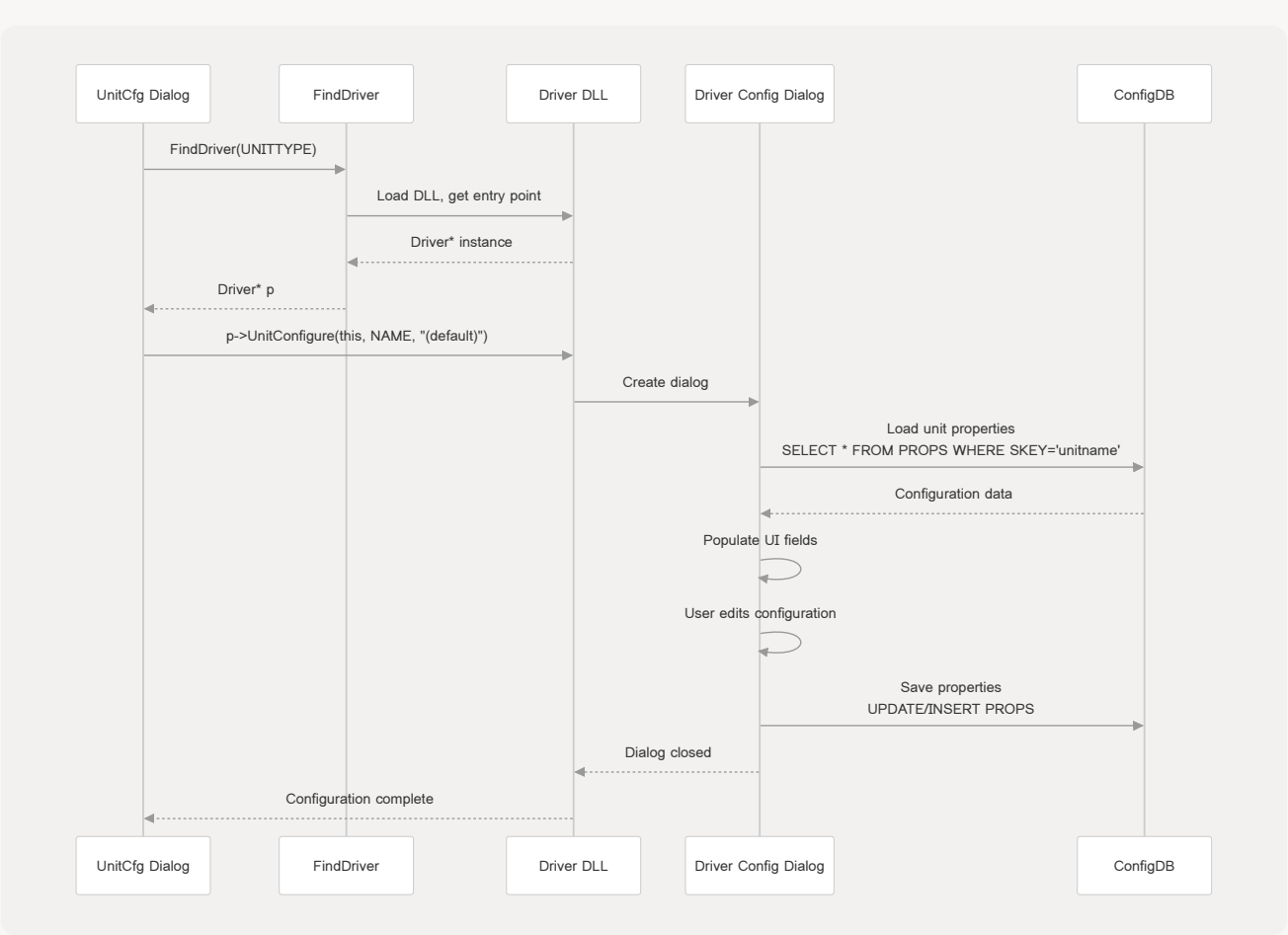
Each driver DLL implements the `Driver::UnitConfigure()` virtual method to provide custom configuration:


«abstract»
Driver
+virtual UnitConfigure(QWidget* parent, QString name, QString receipe)
+virtual SetTypeList(QComboBox* combo, QString unitname)
+virtual GetInputList(QString type, QStringList& list, QString unit, QString name)
Iec104driver
+UnitConfigure(QWidget* parent, QString name, QString receipe)
+SetTypeList(QComboBox* combo, QString unitname)
+GetInputList(...)
Modbusdriver
+UnitConfigure(QWidget* parent, QString name, QString receipe)
+SetTypeList(QComboBox* combo, QString unitname)
+GetInputList(...)

Sources: `src/drivers/driver.h 96-134` `src/drivers/driver.cpp 57`

Configuration Flow

When the user clicks "Configure" in the UnitCfg dialog:


UnitCfg Dialog
FindDriver
Driver DLL
Driver Config Dialog
ConfigDB
FindDriver(UNITTYPE)
Load DLL, get entry point
Driver* instance
Driver* p
p->UnitConfigure(this, NAME, "(default)")
Create dialog
Load unit properties SELECT * FROM PROPS WHERE SKEY='unitname'
Configuration data
Populate UI fields
User edits configuration
Save properties UPDATE/INSERT PROPS
Dialog closed
Configuration complete

Sources: `src/ui/sysdialogs/UnitCfg.cpp 62-72` `src/drivers/driver.h 107`

Example: IEC 104 Unit Configuration

For IEC 104 driver units, configuration includes:

- Server IP address
- Server port
- CASDU (Common ASDU address)
- Protocol logging settings

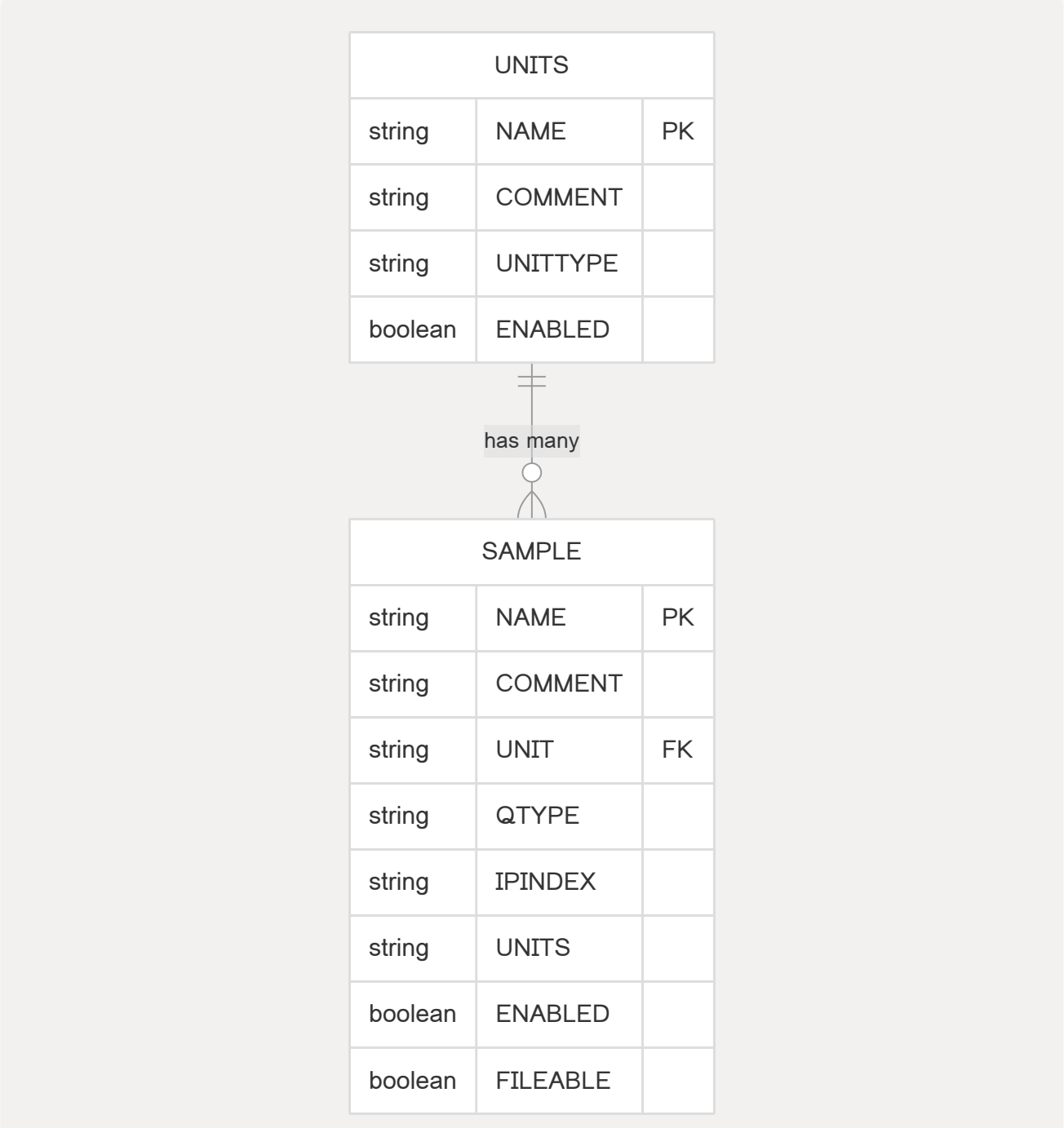
These are stored in `PROPS` table entries and loaded by `Iec104driver_Instance` at startup.

Sources: `src/drivers/iec104/iec104driverthread.cpp 40-58`

Relationship to Sample Points

Unit-Sample Point Association

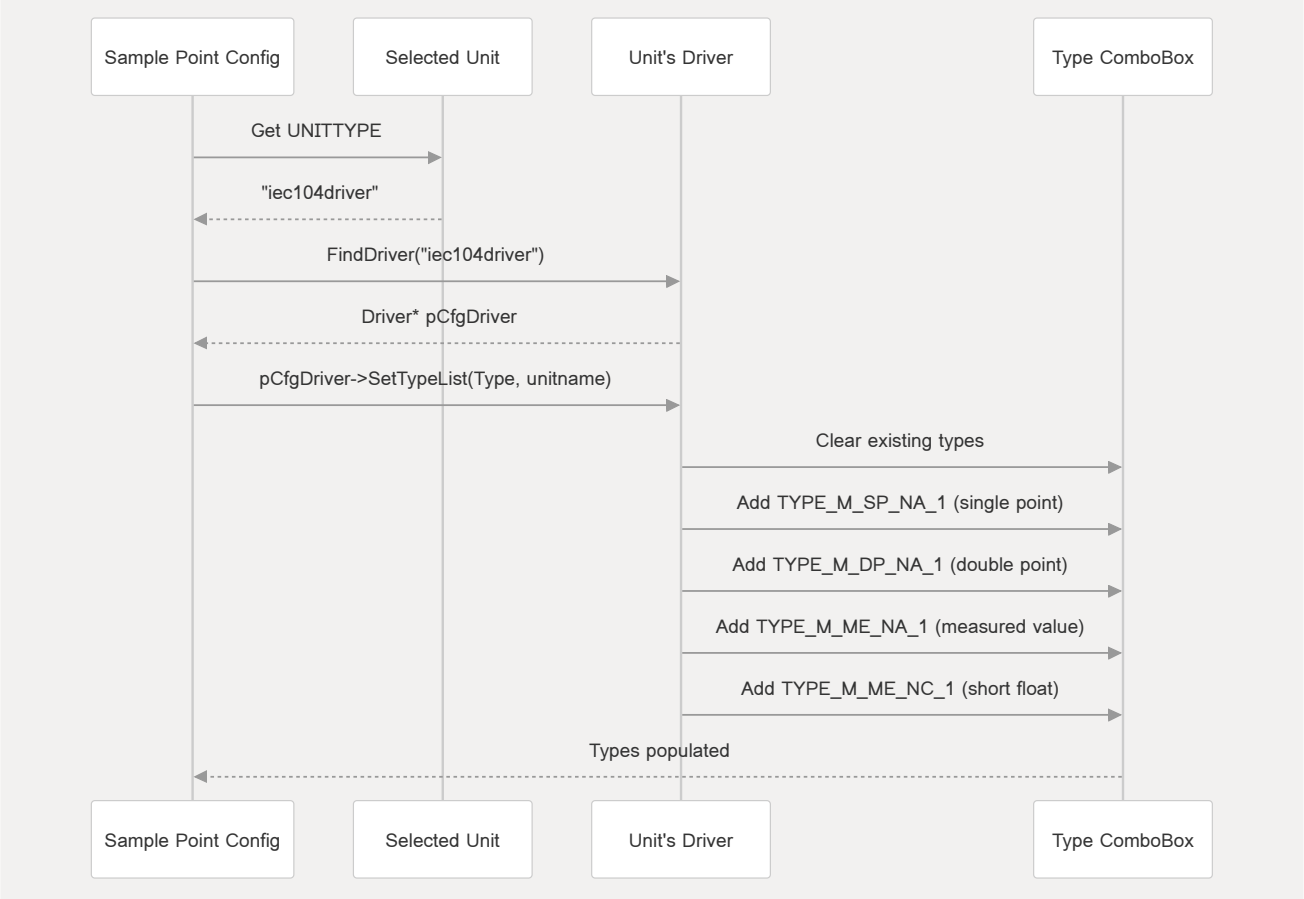
Sample points reference their parent unit through the `UNIT` field in the `SAMPLE` table:


UNITS
string NAME PK
string COMMENT
string UNITTYPE
boolean ENABLED
has many
SAMPLE
string NAME PK
string COMMENT
string UNIT FK
string QTYPE
string IPINDEX
string UNITS
boolean ENABLED
boolean FILEABLE

Sources: `src/ui/sysdialogs/SampleCfg.cpp 39-40` `src/ui/sysdialogs/SampleCfg.cpp 481`

Unit-Driven Sample Point Types

The unit's driver type determines which sample point types are permitted. The `Driver::SetTypeList()` method populates the type combo box in the sample point configuration dialog:


Sample Point Config
Selected Unit
Unit's Driver
Type ComboBox
Get UNITTYPE
"iec104driver"
FindDriver("iec104driver")
Driver* pCfgDriver
pCfgDriver->SetTypeList(Type, unitname)
Clear existing types
Add TYPE_M_SP_NA_1 (single point)
Add TYPE_M_DP_NA_1 (double point)
Add TYPE_M_ME_NA_1 (measured value)
Add TYPE_M_ME_NC_1 (short float)
Types populated

Sources: `src/ui/sysdialogs/SampleCfg.cpp 567-596` `src/drivers/driver.h 109`

Cascading Deletion

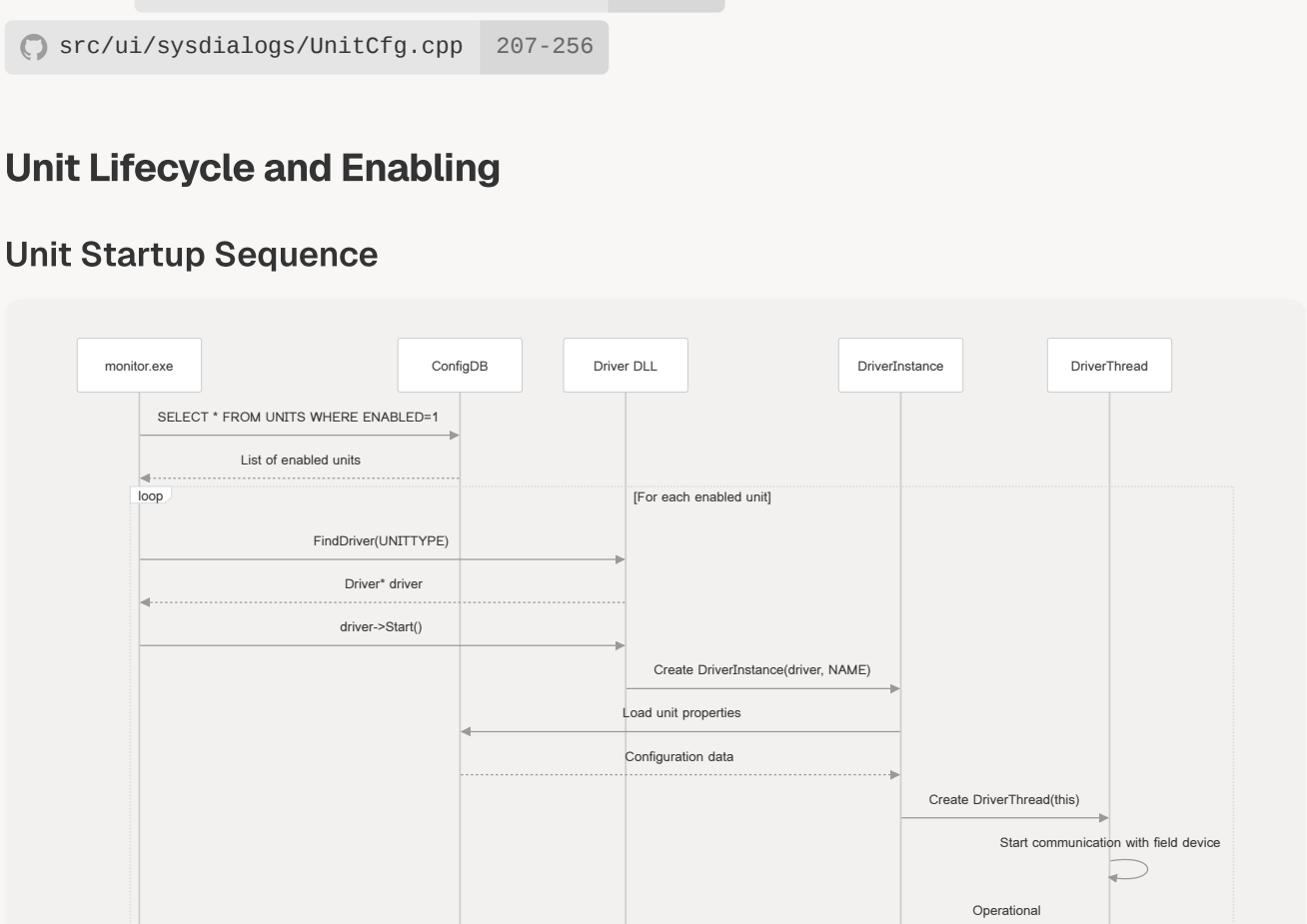
When a unit is deleted, the user is prompted to delete associated sample points. If confirmed, the system:

1. Queries all sample points with matching `UNIT` field
2. Deletes entries from `TAGS`, `TAGS_DB`, `CVAL_DB` tables
3. Drops sample point results tables from `resultsdb` and `historicdb`
4. Calls driver's `DropAllSpecTables()` for protocol-specific cleanup

Sources: `src/ui/sysdialogs/UnitCfg.cpp 85-102` `src/ui/sysdialogs/UnitCfg.cpp 207-256`

Unit Lifecycle and Enabling

Unit Startup Sequence


monitor.exe
ConfigDB
Driver DLL
DriverInstance
DriverThread
SELECT * FROM UNITS WHERE ENABLED=1
List of enabled units
loop
[For each enabled unit]
FindDriver(UNITTYPE)
Driver* driver
driver->Start()
Create DriverInstance(driver, NAME)
Load unit properties
Configuration data
Create DriverThread(this)
Start communication with field device
Operational
Unit started

Sources: `src/drivers/driver.cpp 208-227` `src/drivers/driver.h 161`

Enabled Flag Behavior

The `ENABLED` field in the `UNITS` table controls whether a unit is started automatically:

- **ENABLED = 1**: Unit is started when monitor process initializes. Driver creates instance and begins communication.
- **ENABLED = 0**: Unit is defined but not started. Configuration is preserved but unit remains inactive.

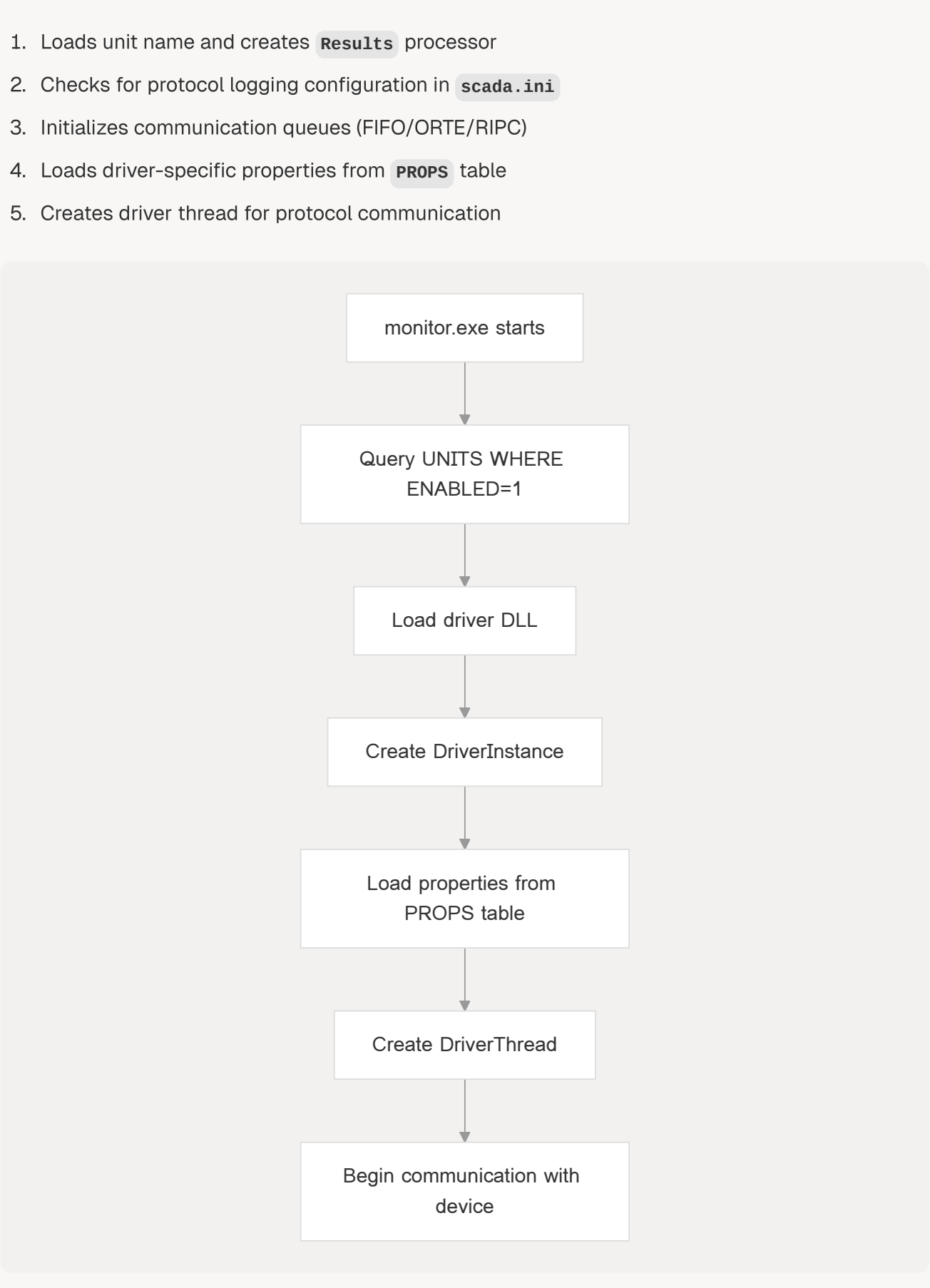
Units can be enabled/disabled through the UnitCfg dialog without deleting the unit configuration.

Sources: `src/ui/sysdialogs/UnitCfg.cpp 173-175` `src/ui/sysdialogs/UnitCfg.cpp 201`

DriverInstance Initialization

When a unit is started, the `DriverInstance` class (`src/drivers/driver.cpp 208-227`) performs initialization:

1. Loads unit name and creates `Results` processor
2. Checks for protocol logging configuration in `scada.ini`
3. Initializes communication queues (FIFO/ORTE/RIPC)
4. Loads driver-specific properties from `PROPS` table
5. Creates driver thread for protocol communication


monitor.exe starts
Query UNITS WHERE ENABLED=1
Load driver DLL
Create DriverInstance
Load properties from PROPS table
Create DriverThread
Begin communication with device

Sources: `src/drivers/driver.cpp 208-227` `src/drivers/driver.h 161-232`

Database Transaction Codes

The UnitCfg dialog uses transaction codes for asynchronous database operations:

Transaction Code	Purpose	SQL Operation
`tList`	Load unit list	`SELECT NAME FROM UNITS ORDER BY NAME`
`tItem`	Load unit details	`SELECT * FROM UNITS WHERE NAME='...'`
`tNew`	Insert new unit	`INSERT INTO UNITS ...`
`tApply`	Update unit	`UPDATE UNITS SET ... WHERE NAME='...'`
`tDelete`	Delete unit	`DELETE FROM UNITS WHERE NAME='...'`
`tDrop`	Drop sample point tables	`DROP TABLE ...`

Transaction responses are handled in the `QueryResponse()` callback method `src/ui/sysdialogs/UnitCfg.cpp 177-266`

Sources: `src/ui/sysdialogs/UnitCfg.cpp 177-266`

Code Entity Reference

Key Classes

- `UnitCfg` - Unit configuration dialog `src/ui/sysdialogs/UnitCfg.cpp`
- `Driver` - Base class for all driver types `src/drivers/driver.h 96-134`
- `DriverInstance` - Runtime instance of a unit `src/drivers/driver.h 140-302`
- `DriverThread` - Protocol communication thread `src/drivers/driver.h 441-510`

Key Functions

- `FindDriver(QString name)` - Loads driver DLL by name `src/drivers/driver.h 527-550`
- `Driver::UnitConfigure()` - Driver-specific configuration `src/drivers/driver.h 107`
- `UnitCfg::Build()` - Builds database record from UI `src/ui/sysdialogs/UnitCfg.cpp 168-176`
- `UnitCfg::Configure()` - Invokes driver config dialog `src/ui/sysdialogs/UnitCfg.cpp 62-72`

Database Tables

- `UNITS` - Unit definitions `src/ui/sysdialogs/UnitCfg.cpp 78-79`
- `PROPS` - Unit-specific properties `src/ui/sysdialogs/UnitCfg.cpp 105`

- `SAMPLE` - Sample points (reference units) `src/ui/sysdialogs/SampleCfg.cpp` 39-40

Sources: `src/ui/sysdialogs/UnitCfg.cpp` 1-268 `src/drivers/driver.h` 1-554 `src/drivers/driver.cpp` 1-771

Protocol Configuration Tools

› Relevant source files

Purpose and Scope

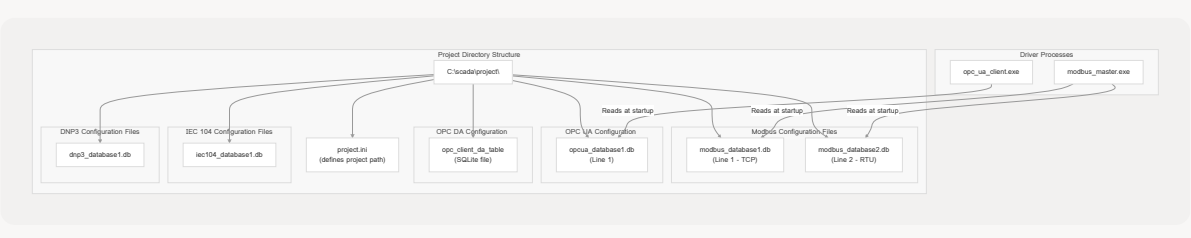
Protocol Configuration Tools manage the protocol-specific configuration for IndigoSCADA driver instances. Each protocol driver (Modbus, OPC UA, DNP3, IEC 104, etc.) requires detailed mapping information between field device data points and the SCADA system's internal addressing scheme. These tools create and manage SQLite database files that store protocol parameters, device addresses, data type mappings, and IOA (Information Object Address) assignments for each communication line.

For general unit and driver instance setup, see Unit Configuration. For sample point definitions and alarm configurations, see Sample Point Configuration.

Configuration Storage Architecture

IndigoSCADA uses SQLite 3 databases to store protocol-specific configuration separately from the main FastDB configuration database. Each protocol driver instance (identified by a "line number") has its own dedicated SQLite database file.

Configuration File Organization


Project Directory Structure
C:\scada\project\
DNP3 Configuration Files
dnp3_database1.db
IEC 104 Configuration Files
iec104_database1.db
project.ini
(defines project path)
OPC DA Configuration
opc_client_da_table
(SQLite file)
OPC UA Configuration
opcua_database1.db
(Line 1)
Modbus Configuration Files
modbus_database1.db
(Line 1 - TCP)
modbus_database2.db
(Line 2 - RTU)
Driver Processes
opc_ua_client.exe
modbus_master.exe
Reads at startup
Reads at startup
Reads at startup

Sources: `src/drivers/modbus/protocol/load_database_modbus.cpp` 184-207 `src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 95-118 `src/CHANGES.txt` 99

Database Naming Convention

Each protocol driver's configuration database follows the pattern:

Protocol	Database File Pattern	Example
Modbus	`modbus_database{N}.db`	`modbus_database1.db`
OPC UA	`opcua_database{N}.db`	`opcua_database1.db`
IEC 104	`iec104_database{N}.db`	`iec104_database1.db`
DNP3	`dnp3_database{N}.db`	`dnp3_database1.db`

Where `{N}` is the line number (instance ID) starting from 1.

Sources: `src/drivers/modbus/protocol/load_database_modbus.cpp` 203-207 `src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 114-118

Modbus Configuration Database

Database Schema

The Modbus configuration database contains a single table `modbus_table` with the following structure:

modbus_table		
int	slave_id	Modbus slave/unit ID
int	modbus_function_read	Function code for reading
int	modbus_function_write	Function code for writing
int	modbus_address	Register/coil address
string	modbus_type	Data type (VT_BOOL, VT_I2, VT_I4, VT_R4, etc)
int	ioa_control_center	IOA for control center mapping
float	deadband	Deadband for analog values

Modbus Data Type Mapping

The `modbus_type` field uses OPC-style data type enumerations defined in `src/drivers/modbus/protocol/modbus_db.h` 18-32 :

Type Code	Description	Size
`VT_BOOL` (11)	Boolean	1 bit
`VT_I1` (16)	Signed 8-bit	1 byte
`VT_UI1` (17)	Unsigned 8-bit	1 byte
`VT_I2` (2)	Signed 16-bit	1 register
`VT_UI2` (18)	Unsigned 16-bit	1 register
`VT_I4` (3)	Signed 32-bit	2 registers
`VT_UI4` (19)	Unsigned 32-bit	2 registers
`VT_R4` (4)	32-bit float	2 registers
`VT_R4SWAP` (22)	32-bit float (LSB/MSB swapped)	2 registers
`VT_R8` (5)	64-bit double	4 registers

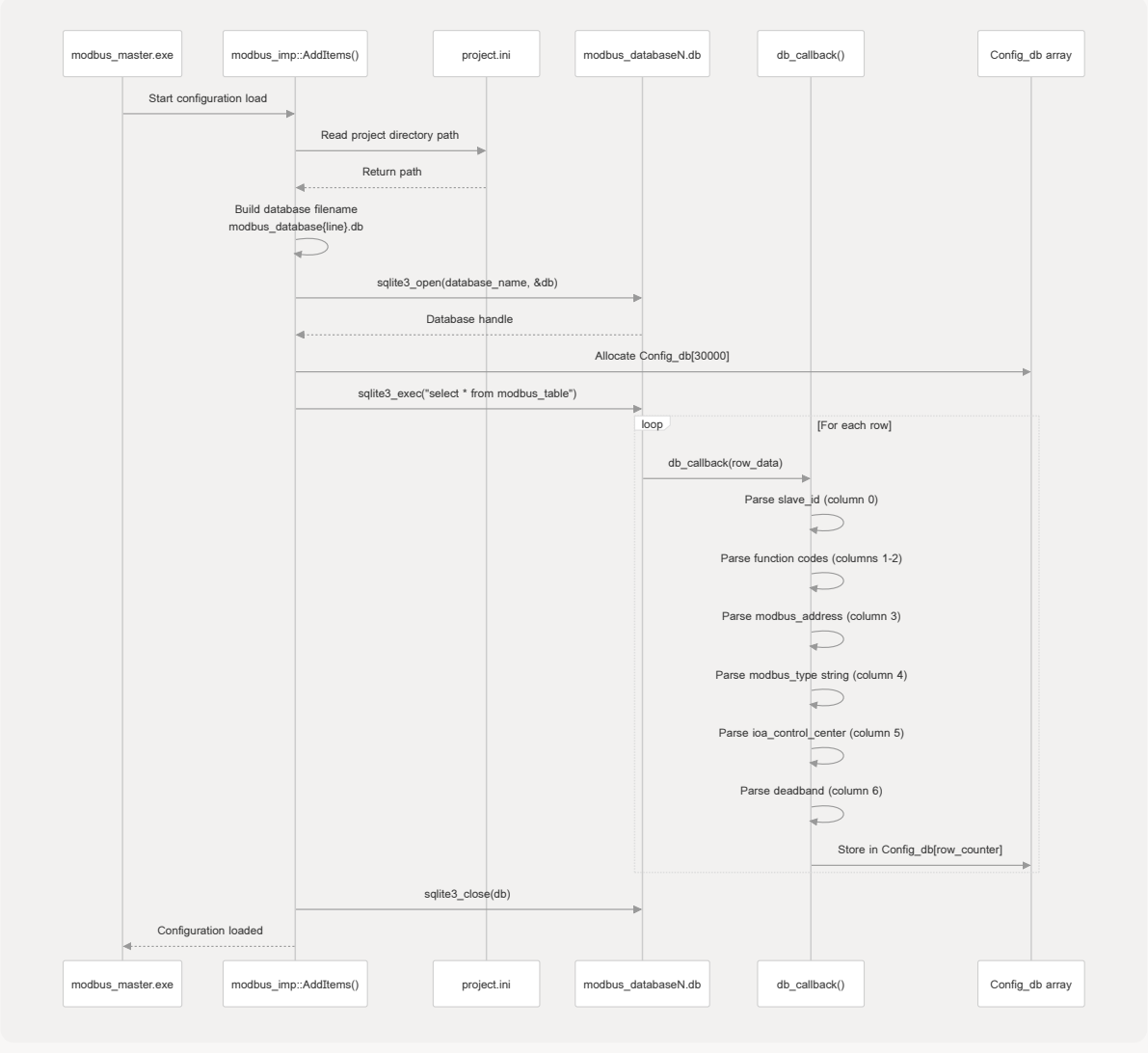
Modbus Function Codes

Function codes supported for read/write operations:

Code	Function	Usage
`0x01`	`FC_READ_COILS`	Read coil status (discrete outputs)
`0x02`	`FC_READ_DISCRETE_INPUTS`	Read input status (discrete inputs)
`0x03`	`FC_READ_HOLDING_REGISTERS`	Read holding registers (analog outputs)
`0x04`	`FC_READ_INPUT_REGISTERS`	Read input registers (analog inputs)
`0x05`	`FC_WRITE_SINGLE_COIL`	Write single coil
`0x06`	`FC_WRITE_SINGLE_REGISTER`	Write single register
`0x0F`	`FC_WRITE_MULTIPLE_COILS`	Write multiple coils
`0x10`	`FC_WRITE_MULTIPLE_REGISTERS`	Write multiple registers

Sources: `src/drivers/modbus/protocol/modbus_imp.h` 63-74 `src/drivers/modbus/protocol/modbus_db.h` 18-50

Configuration Loading Process


modbus_master.exe
modbus_imp::AddItems()
project.ini
modbus_databaseN.db
db_callback()
Config_db array
Start configuration load
Read project directory path
Return path
Build database filename
modbus_database{line}.db
sqlite3_open(database_name, &db)
Database handle
Allocate Config_db[30000]
sqlite3_exec("select * from modbus_table")
loop
[For each row]
db_callback(row_data)
Parse slave_id (column 0)
Parse function codes (columns 1-2)
Parse modbus_address (column 3)
Parse modbus_type string (column 4)
Parse ioa_control_center (column 5)
Parse deadband (column 6)
Store in Config_db[row_counter]
sqlite3_close(db)
Configuration loaded

Sources: `src/drivers/modbus/protocol/load_database_modbus.cpp` 32-169 `src/drivers/modbus/protocol/load_database_modbus.cpp` 173-254

Modbus Configuration Record Structure

The runtime representation in memory is defined by the `modbusDbRecord` structure:

```
struct modbusDbRecord {
    int slave_id;                      // Slave ID of RTU
    int modbus_function_read;          // Modbus function to read
    int modbus_function_write;         // Modbus function to write
    int modbus_address;                // Address of memory to fetch
    int modbus_type;                   // Modbus type (VT_*)
    unsigned int ioa_control_center;   // IOA unique inside CASDU
    float deadband;                    // Deadband for analog values
    union {
        unsigned int ua;
        int a;
        float f;
    } last_value;                      // Cached last value for change detection
};
```

Sources: `src/drivers/modbus/protocol/modbus_db.h` 34-50

OPC UA Configuration Database

Database Schema

The OPC UA configuration database contains a single table `opcua_table`:

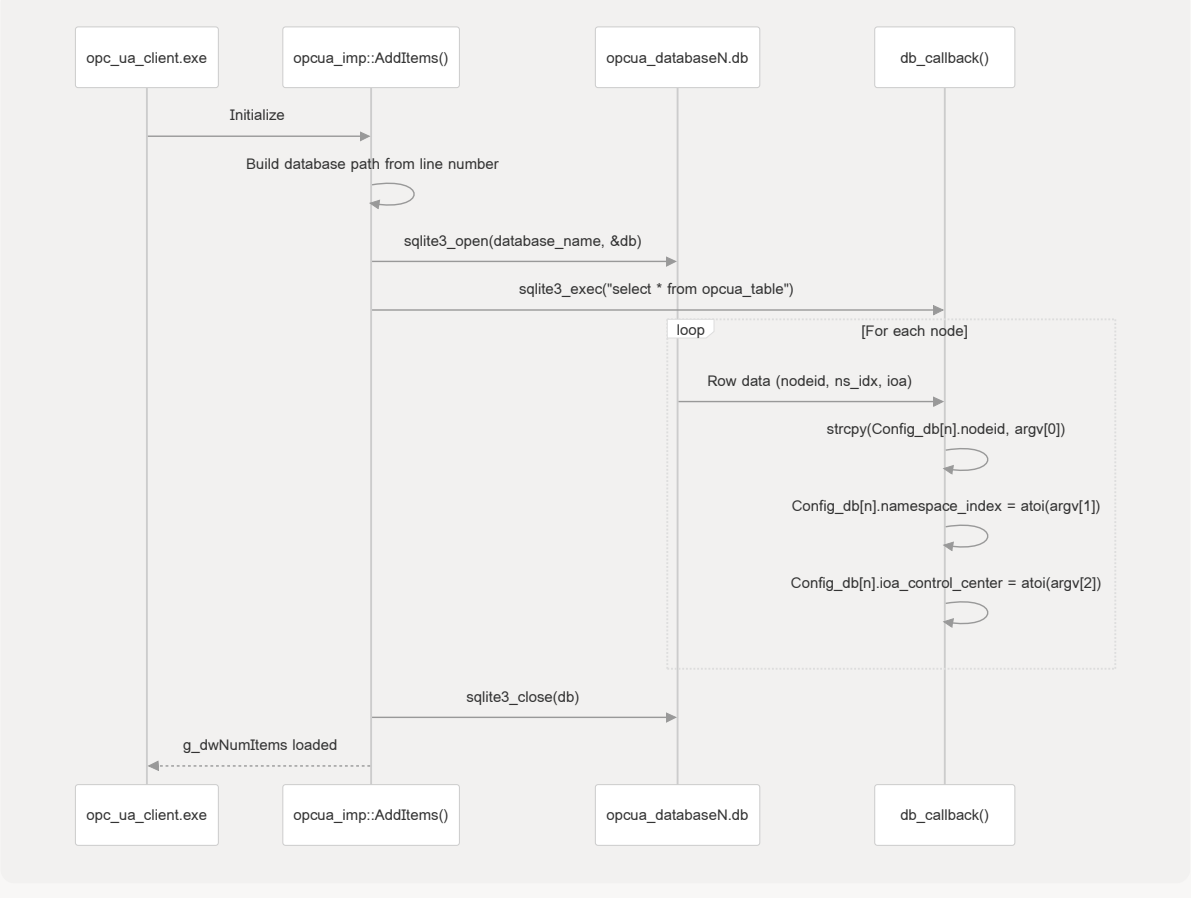
opcua_table		
string	nodeid	OPC UA node ID string
int	namespace_index	Namespace index
int	ioa_control_center	IOA for control center mapping

OPC UA Node Configuration

OPC UA nodes are identified by:

- **NodeId string**: String representation like "2258" (numeric) or "Device1.Temperature" (string)
- **Namespace index**: Integer identifying the namespace (typically 0-N)
- **IOA mapping**: Maps to IndigoSCADA's Information Object Address

Configuration Loading


opc_ua_client.exe
opcua_imp::AddItems()
opcua_databaseN.db
db_callback()
Initialize
Build database path from line number
sqlite3_open(database_name, &db)
sqlite3_exec("select * from opcua_table")
loop
[For each node]
Row data (nodeid, ns_idx, ioa)
strcpy(Config_db[n].nodeid, argv[0])
Config_db[n].namespace_index = atoi(argv[1])
Config_db[n].ioa_control_center = atoi(argv[2])
sqlite3_close(db)
g_dwNumItems loaded

Sources: `src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 32-80 `src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 84-165

OPC UA Configuration Record

Runtime structure defined in `src/drivers/opc_ua/protocol/client/opc_ua_db.h` 16-23 :

```
struct opcuaDbRecord {
    char nodeid[50];                   // OPC UA node ID string
    int namespace_index;               // Namespace index
    int dataType;                      // Data type of the node (auto-detected)
    unsigned int ioa_control_center;   // IOA unique inside CASDU
};
```

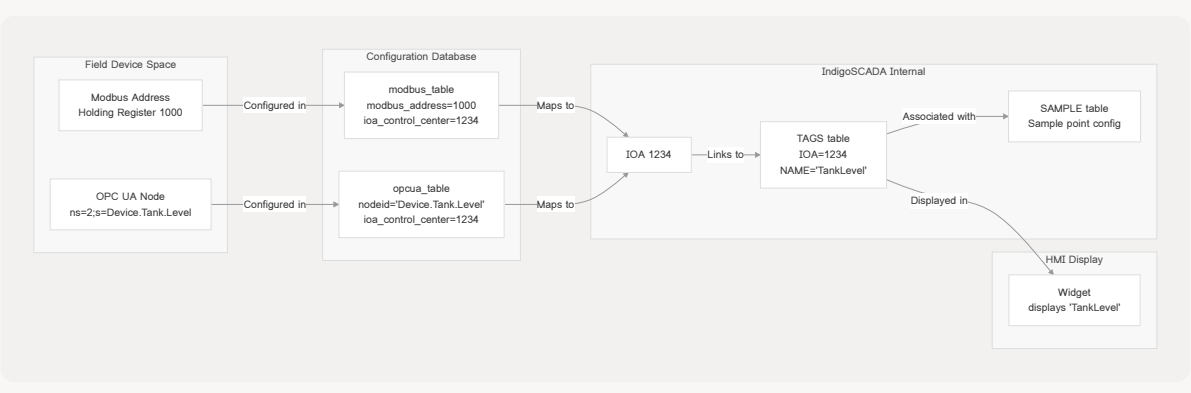
The `dataType` field is auto-detected at runtime from the first read operation (see `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 613-616).

Sources: `src/drivers/opc_ua/protocol/client/opc_ua_db.h` 16-23

IOA (Information Object Address) Mapping

Purpose of IOA Mapping

The IOA field in protocol configuration databases serves as the critical link between field device data points and IndigoSCADA's internal addressing:


Field Device Space
Modbus Address
Holding Register 1000
OPC UA Node
ns=2;s=Device.Tank.Level
Configured in
Configured in
Configuration Database
modbus_table
modbus_address=1000
ioa_control_center=1234
opcua_table
nodeid='Device.Tank.Level'
ioa_control_center=1234
Maps to
Maps to
IndigoSCADA Internal
IOA 1234
Links to
TAGS table
IOA=1234
NAME='TankLevel'
Associated with
SAMPLE table
Sample point config
Displayed in
HMI Display
Widget
displays 'TankLevel'

IOA Assignment Rules

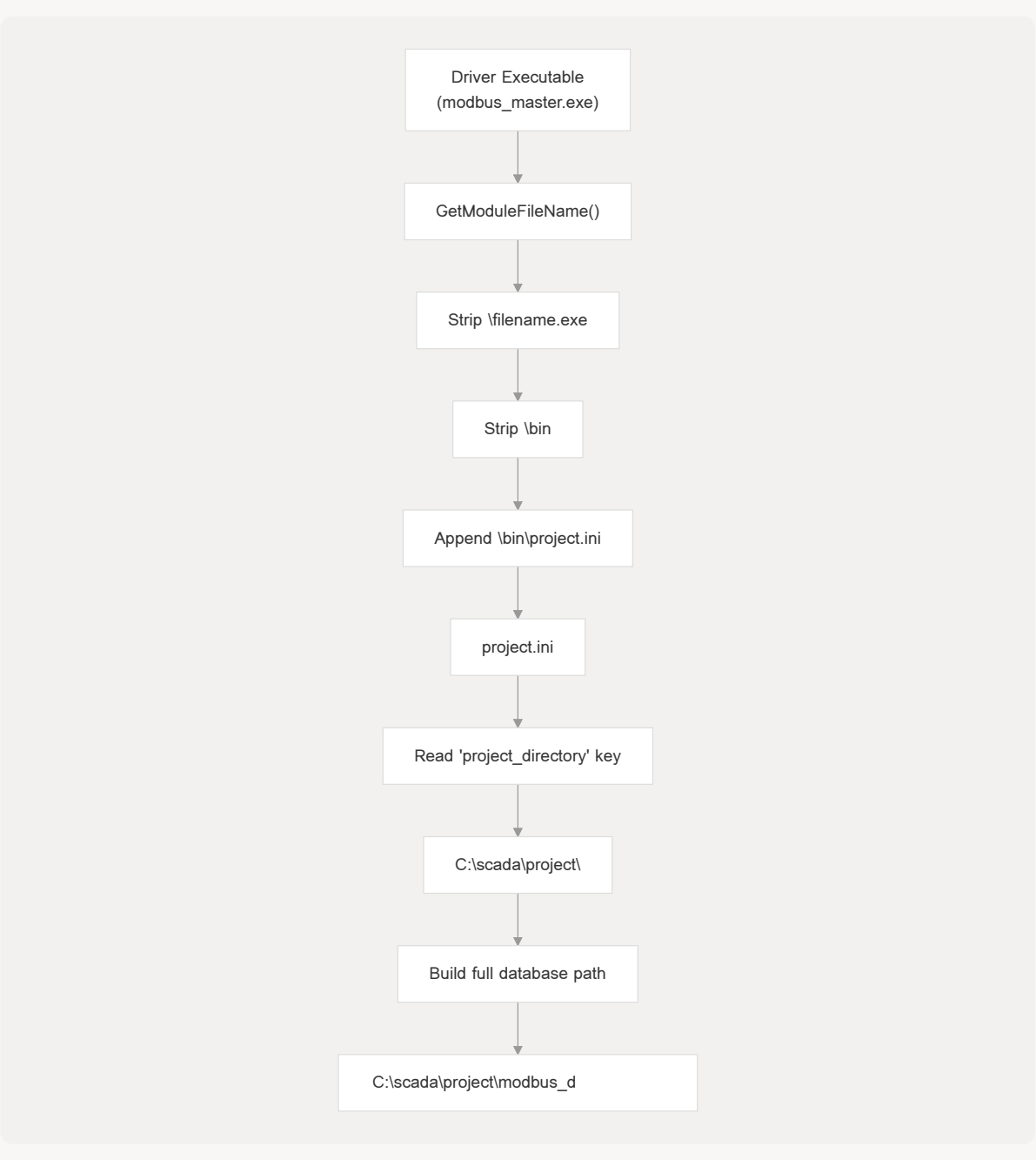
1. **Uniqueness**: Each IOA must be unique within a CASDU (Common Address of ASDU)
2. **Range**: IOA values are unsigned 32-bit integers (0 to 4,294,967,295)
3. **Consistency**: The same physical data point accessed by multiple protocols must use the same IOA
4. **Reserved Values**: IOA 4004 is reserved for keep-alive messages (see `src/drivers/modbus/protocol/main_master.cpp` 492-498)

Sources: `src/drivers/modbus/protocol/modbus_db.h` 43 `src/drivers/opc_ua/protocol/client/opc_ua_db.h` 21 `src/drivers/modbus/protocol/main_master.cpp` 492-498

Configuration File Location

Project Directory Resolution

All protocol configuration databases are stored in the project directory, which is resolved at runtime:


Driver Executable
(modbus_master.exe)
GetModuleFileName()
Strip \filename.exe
Strip \bin
Append \bin\project.ini
project.ini
Read 'project_directory' key
C:\scada\project\
Build full database path
C:\scada\project\modbus_d

Implementation:

```
// From load_database_modbus.cpp:189-201
char project_dir[_MAX_PATH];
char ini_file[_MAX_PATH];

GetModuleFileName(NULL, ini_file, _MAX_PATH);
*(strrchr(ini_file, '\\')) = '\0';  // Strip \filename.exe
*(strrchr(ini_file, '\\')) = '\0';  // Strip \bin
strcat(ini_file, "\\bin\\project.ini");

Inifile iniFile(ini_file);
if(iniFile.find("path","project_directory")) {
    strcpy(project_dir, iniFile.find("path","project_directory"));
}

strcpy(database_name, project_dir);
strcat(database_name, "\\modbus_database");
strcat(database_name, line_number);
strcat(database_name, ".db");
```

Sources: `src/drivers/modbus/protocol/load_database_modbus.cpp` 184-207 `src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 95-118

Configuration Capacity Limits

Each protocol driver pre-allocates memory for configuration records:

Protocol	Maximum Items	Definition Location
Modbus	30,000	`MAX_CONFIGURABLE_MODBUS_ITEMIDS`
OPC UA	30,000	`MAX_CONFIGURABLE_OPCUA_ITEMIDS`

```
// From load_database_modbus.cpp:220-222
#define MAX_CONFIGURABLE_MODBUS_ITEMIDS 30000

g_dwNumItems = MAX_CONFIGURABLE_MODBUS_ITEMIDS;
Config_db = (struct modbusDbRecord*)calloc(1,
    g_dwNumItems * sizeof(struct modbusDbRecord));
```

After loading from the database, `g_dwNumItems` is set to the actual row count:

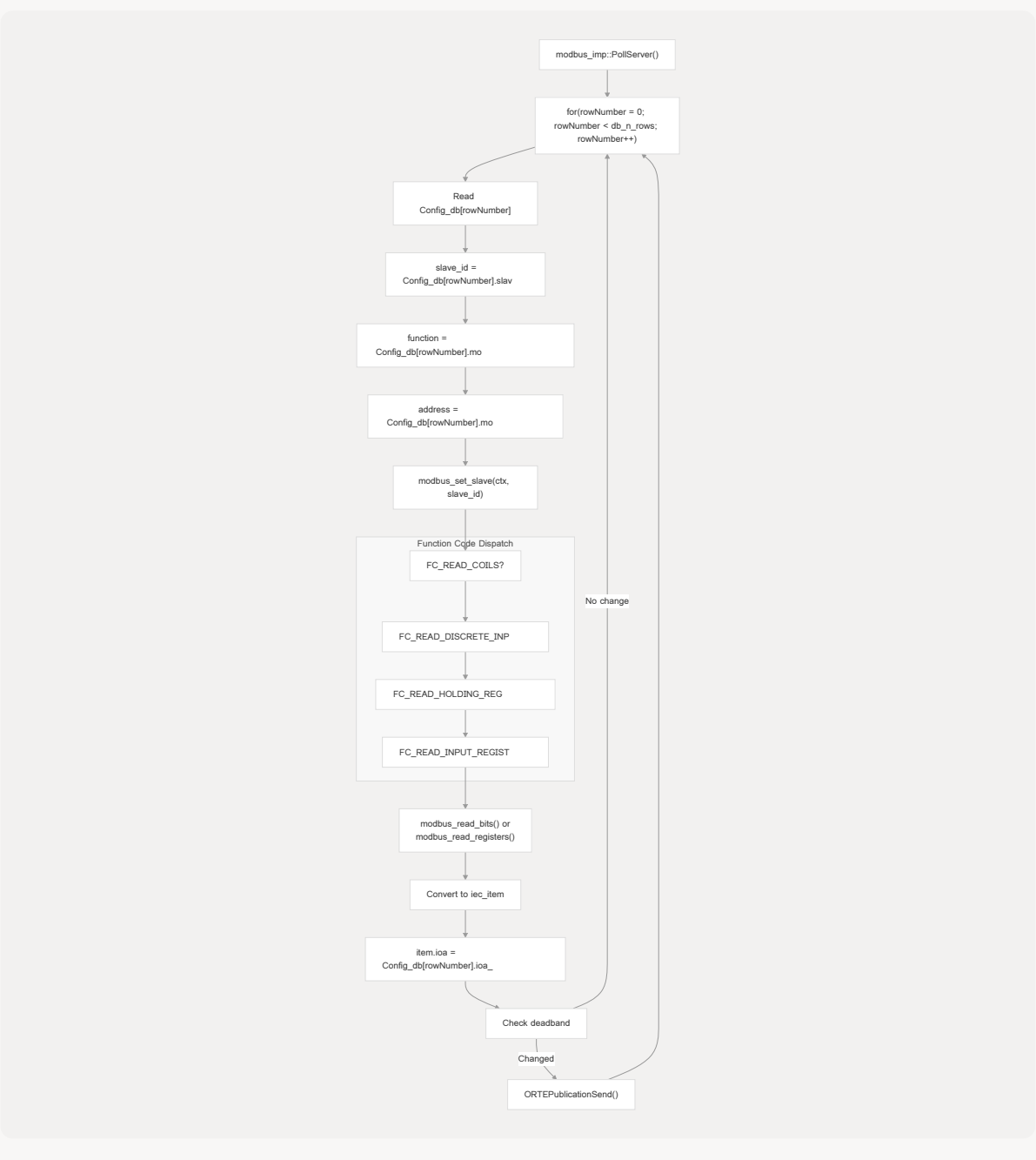
```
// From load_database_modbus.cpp:239-250
db_n_rows = gl_row_counter;
g_dwNumItems = db_n_rows;
```

Sources: `src/drivers/modbus/protocol/load_database_modbus.cpp` 171-222 `src/drivers/opc_ua/protocol/client/load_database_opc_ua.cpp` 82-134

Runtime Configuration Usage

Polling and Data Acquisition

Once loaded, the configuration database drives the polling loop:


modbus_imp::PollServer()
for(rowNumber = 0;
rowNumber < db_n_rows;
rowNumber++)
Read
Config_db[rowNumber]
slave_id =
Config_db[rowNumber].slav
function =
Config_db[rowNumber].mo
address =
Config_db[rowNumber].mo
modbus_set_slave(ctx,
slave_id)
Function Code Dispatch
FC_READ_COILS?
FC_READ_DISCRETE_INP
FC_READ_HOLDING_REG
FC_READ_INPUT_REGIST
modbus_read_bits() or
modbus_read_registers()
Convert to iec_item
item.ioa =
Config_db[rowNumber].ioa_
Check deadband
No change
Changed
ORTEPublicationSend()

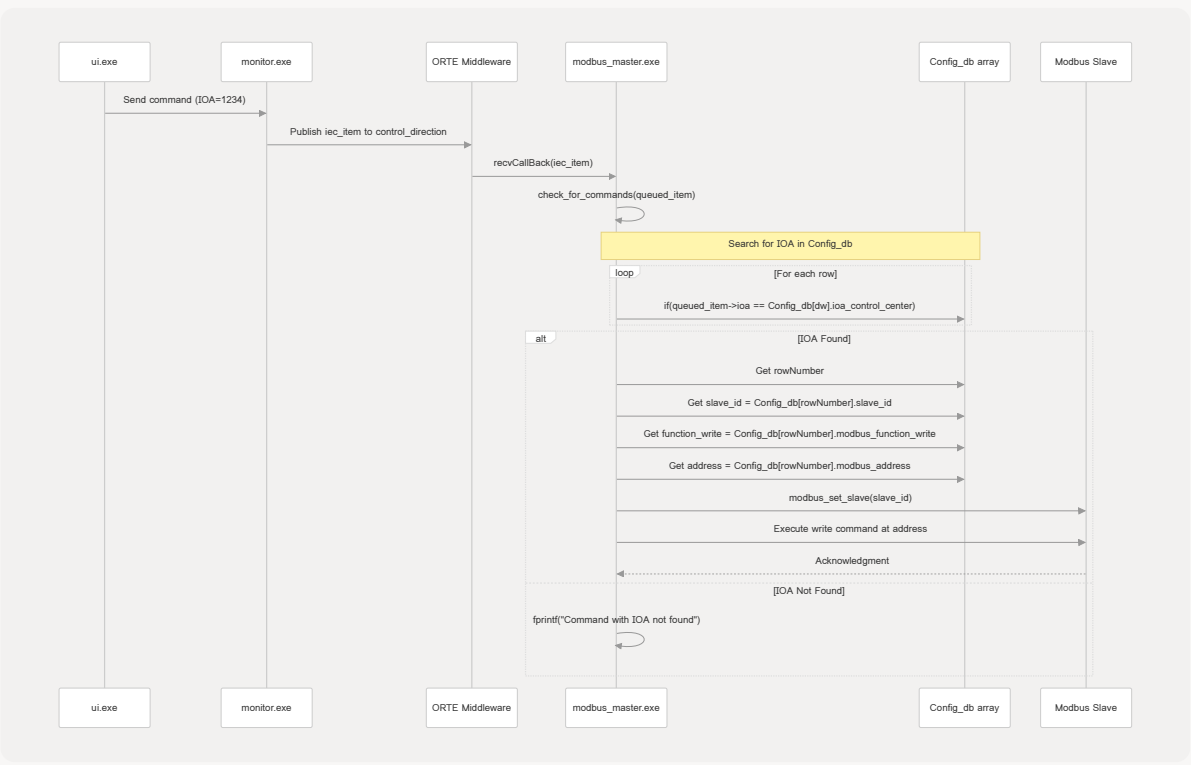
Example Implementation (Modbus):

```
// From modbus_imp.cpp:577-633
for(int rowNumber = 0; rowNumber < db_n_rows; rowNumber++) {
    int address = Config_db[rowNumber].modbus_address;
    modbus_set_slave(ctx, Config_db[rowNumber].slave_id);

    if(Config_db[rowNumber].modbus_function_read == FC_READ_COILS) {
        rc = modbus_read_bits(ctx, address, bit_size, tab_rp_bits);
        uint8_t value = tab_rp_bits[0];

        if(Config_db[rowNumber].last_value.a != value) {
            Config_db[rowNumber].last_value.a = value;
            send_item = true;
        }

        item_to_send.iec_obj.ioa = Config_db[rowNumber].ioa_control_center;
        item_to_send.iec_type = M_SP_TB_1;
        item_to_send.iec_obj.o.type30.sp = value;
    }
}
```

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 562-1228 `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 581-781

Command Processing with Configuration

Control commands from the HMI are matched against the configuration database to find the target device:


ui.exe
monitor.exe
ORTE Middleware
modbus_master.exe
Config_db array
Modbus Slave
Send command (IOA=1234)
Publish iec_item to control_direction
recvCallBack(iec_item)
check_for_commands(queued_item)
Search for IOA in Config_db
loop
[For each row]
if(queued_item->ioa == Config_db[dw].ioa_control_center)
alt
[IOA Found]
Get rowNumber
Get slave_id = Config_db[rowNumber].slave_id
Get function_write = Config_db[rowNumber].modbus_function_write
Get address = Config_db[rowNumber].modbus_address
modbus_set_slave(slave_id)
Execute write command at address
Acknowledgment
[IOA Not Found]
fprintf("Command with IOA not found")

Implementation:

```
// From modbus_imp.cpp:1366-1399
// Search for IOA in configuration
int found = 0;
DWORD rowNumber = -1;

for(int dw = 0; dw < db_n_rows; dw++) {
    if(queued_item->iec_obj.ioa == Config_db[dw].ioa_control_center) {
        found = 1;
        rowNumber = dw;
        break;
    }
}

if(found == 0) {
    fprintf(stderr, "Error: Command with IOA %d not found in I/O list\n",
            queued_item->iec_obj.ioa);
    return;
}

// Execute command using configuration
int slave = Config_db[rowNumber].slave_id;
int address = Config_db[rowNumber].modbus_address;
modbus_set_slave(ctx, slave);
```

Sources: `src/drivers/modbus/protocol/modbus_imp.cpp` 1343-1748 `src/drivers/opc_ua/protocol/client/opc_ua_imp.cpp` 896-1191

Protocol Configurator Tool

While the configuration files are SQLite databases, IndigoSCADA includes a dedicated configuration tool:

Tool Information

Attribute	Value
Executable	`protocol_configurator.exe`
Purpose	GUI tool for creating/editing protocol databases
Status	Mentioned in CHANGES.txt (line 99), not included in source tree
Alternative	Direct SQLite database editing with standard tools

From `src/CHANGES.txt` 99 :

```
05-May-2012
Added protocol configurator, based on SQLite 3 database files
```

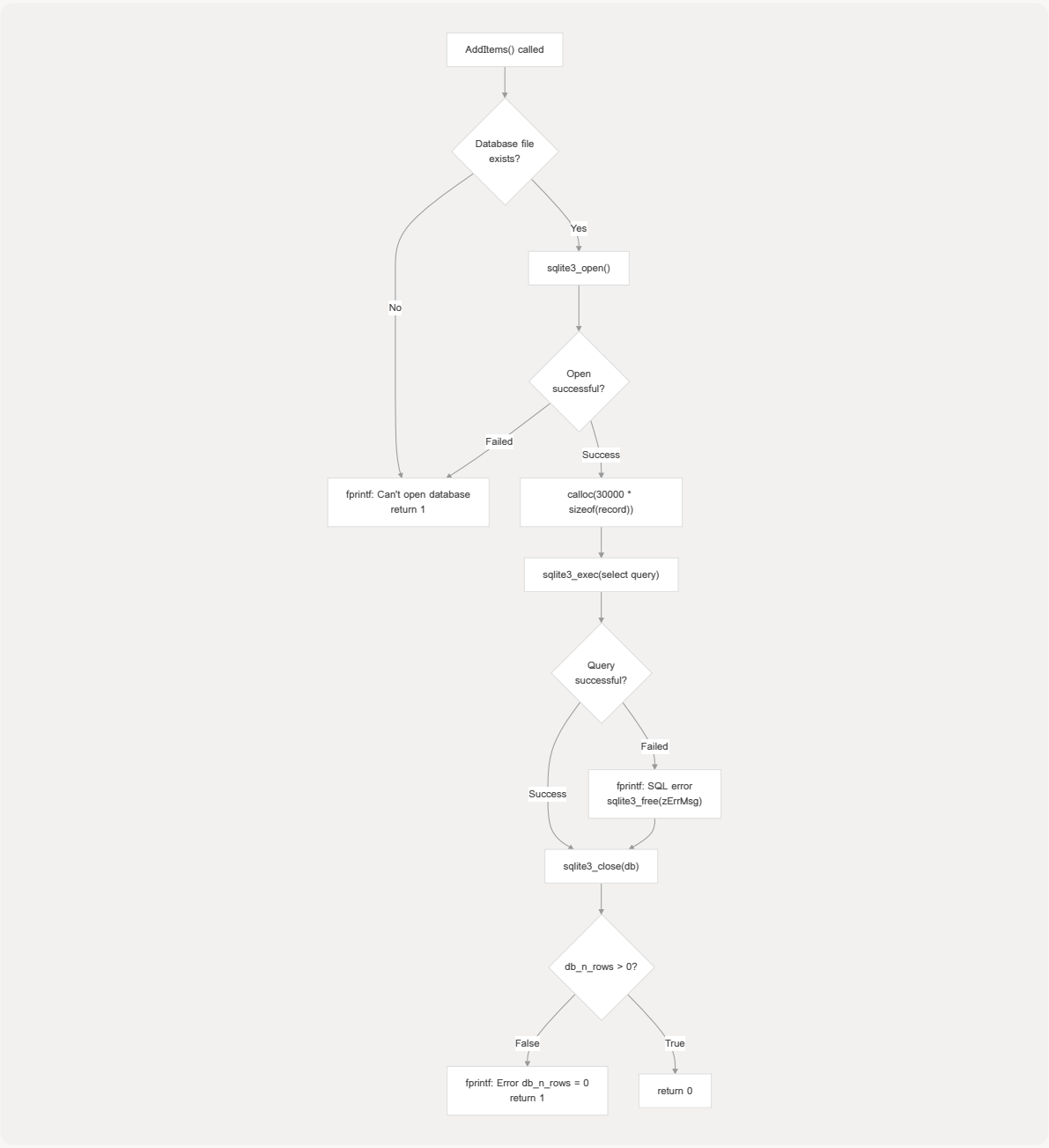
From `src/CHANGES.txt` 103 :

```
30-05-2012
Integration of the middleware (RTPS) with protocol processes
Implemented configurator for opc_client_da.exe and opc_client_ae.exe
```

Sources: `src/CHANGES.txt` 99-103

Error Handling During Configuration Load

Configuration loading includes error checking at multiple levels:


AddItems() called
Database file
exists?
Yes
No
sqlite3_open()
Open
successful?
Failed
Success
fprintf: Can't open database
return 1
calloc(30000 *
sizeof(record))
sqlite3_exec(select query)
Query
successful?
Failed
Success
fprintf: SQL error
sqlite3_free(zErrMsg)
sqlite3_close(db)
db_n_rows > 0?
False
True
fprintf: Error db_n_rows = 0
return 1
return 0

Implementation:

```
// From load_database_modbus.cpp:209-248
rc = sqlite3_open(database_name, &db);
if(rc) {
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
    fflush(stderr);
    sqlite3_close(db);
    return 1;
}

rc = sqlite3_exec(db, "select * from modbus_table;", db_callback, 0, &zErrMsg);
if(rc != SQLITE_OK) {
    fprintf(stderr, "SQL error: %s\n", zErrMsg);
    fflush(stderr);
    sqlite3_free(zErrMsg);
}

sqlite3_close(db);

if(db_n_rows == 0) {
    fprintf(stderr, "Error: db_n_rows = %d\n", db_n_rows);
    fflush(stderr);
    return 1;
}
```

Sources: `src/drivers/modbus/protocol/load_database_modbus.cpp` 209-248

Summary

Protocol Configuration Tools in IndigoSCADA provide a flexible, protocol-independent method for mapping field device data points to the SCADA system's internal addressing through:

1. **SQLite databases** - One database file per protocol driver instance (line)
2. **IOA mapping** - Central linking mechanism between protocol-specific addresses and SCADA sample points
3. **Runtime loading** - Configuration loaded once at driver startup
4. **Type conversion** - Protocol-native data types mapped to IEC 104 types
5. **Deadband support** - Reduces unnecessary data transmission for analog values
6. **Command routing** - Bidirectional IOA-based command dispatch

The configuration database serves as the "Rosetta Stone" that translates between the diverse addressing schemes of industrial protocols (Modbus addresses, OPC NodeIds, DNP3 points) and IndigoSCADA's unified IOA addressing system.

Sources: All files listed above

Configuration and Setup

› Relevant source files

Purpose and Scope

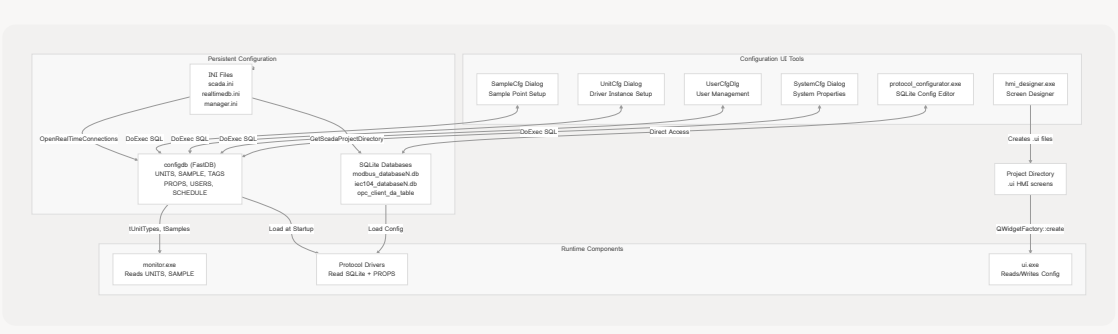
This document describes the configuration system of IndigoSCADA, including the configuration databases, INI files, user interface dialogs, and initial setup procedures. Configuration in IndigoSCADA is multi-layered, utilizing INI files for system-level settings, FastDB tables for runtime configuration, and SQLite databases for protocol-specific parameters.

For information about building and deploying the system, see Build System and Deployment. For protocol-specific driver configuration details, see Protocol Drivers.

Configuration Architecture Overview

IndigoSCADA's configuration system operates in distinct layers, from persistent storage through to runtime state management.

Configuration Data Flow


Persistent Configuration
INI Files scada.ini realtimedb.ini manager.ini
Configuration UI Tools
SampleCfg Dialog Sample Point Setup
UnitCfg Dialog Driver Instance Setup
UserCfgDlg User Management
SystemCfg Dialog System Properties
protocol_configurator.exe SQLite Config Editor
hmi_designer.exe Screen Designer
OpenRealTimeConnections
DoExec SQL
DoExec SQL
DoExec SQL
GetScadaProjectDirectory
DoExec SQL
Direct Access
Creates .ui files
configdb (FastDB) UNITS, SAMPLE, TAGS PROPS, USERS, SCHEDULE
SQLite Databases modbus_databaseN.db iec104_databaseN.db opc_client_da_table
Project Directory .ui HMI screens
IUnitTypes, ISamples
Load at Startup
Load Config
QWidgetFactory::create
Runtime Components
monitor.exe Reads UNITS, SAMPLE
Protocol Drivers Read SQLite + PROPS
ui.exe Reads/Writes Config

Sources: `src/database/realtimedb.cpp 137-244` `src/monitor/monitor.cpp 149-319` `src/ui/user.cpp 494-720` `src/drivers/driver.cpp 207-227`

Configuration File Locations

IndigoSCADA uses multiple configuration files located in standard directories. The project directory is determined at runtime from the executable path.

Directory Structure

Path	Purpose	Set By
`C:\scada\bin\`	Executables and DLLs	`GetScadaHomeDirectory()`
`C:\scada\project\`	Configuration files, databases	`GetScadaProjectDirectory()`
`C:\scada\project\*.ui`	HMI screen definitions	hmi_designer.exe
`C:\scada\project\*.db`	SQLite protocol configs	protocol_configurator.exe
`C:\scada\init_db\`	Database initialization schemas	Database setup scripts
`C:\scada\logs\`	Protocol logging output	Driver instances

Sources: `src/ui/main.cpp 67-68` `src/ui/user.cpp 393-408`

Primary INI Files

scada.ini

```
[path]
internet_browser = C:\Program Files\Internet Explorer\iexplore.exe

[log_to_file]
drivers = 1

[System]
AutoLogoffTime = 300
```

Located at `GetScadaProjectDirectory() + "\\scada.ini"`. Contains system-wide settings including logging preferences and UI behavior.

realtimedb.ini

```
[rtsqlserver]
user = admin
password = rainbow

[databases]
online = configdb currentdb resultsdb

[configdb]
address = localhost
port = 5101
is_low_freq = 1

[currentdb]
address = localhost
port = 5102
is_low_freq = 0

[resultsdb]
address = localhost
port = 5103
is_low_freq = 1
```

Defines database connection parameters. Parsed by `OpenRealTimeConnections()` to establish TCP connections to FastDB/GigaBASE servers.

manager.ini

```
[manager]
NumOfDrivers = 3

[driver1]
name = iec104master
command_line = iec104master.exe -a 192.168.1.100 -p 2404 -c 1 -l 1

[driver2]
name = modbus_master
command_line = modbus_master.exe -a 192.168.1.101 -p 502 -l 1
```

Parsed by `manager.exe` to start and monitor child processes. Each driver section specifies process name, command line arguments, and line number.

Sources: `src/database/realtimedb.cpp 143-244` `src/ui/user.cpp 994-996` `src/drivers/driver.cpp 219-226`

Configuration Database Schema

The `configdb` database (FastDB) stores all runtime configuration. It is memory-mapped for high-performance access and persists to disk.

Core Configuration Tables

UNITS Table

```
Columns: NAME, COMMENT, UNITTYPE, ENABLED, IPINDEX, PROPS
Purpose: Defines driver instances (e.g., "Modbus_Line1", "IEC104_RTU3")
```

Each row represents a driver instance. The `UNITTYPE` column references a driver DLL name (e.g., "modbus", "iec104"). The `ENABLED` flag determines if the unit is started by monitor.

SAMPLE Table

```
Columns: NAME, COMMENT, UNIT, QTYPE, IPINDEX, UNITS, ENABLED,
         FILEABLE, LOGMODE, RETRIGGER, ALARMTHRESHOLD
Purpose: Defines data acquisition points
```

Sample points are the fundamental data objects. Each is associated with a `UNIT` and has a quality type `QTYPE` (e.g., `TYPE_M_ME_NC_1` for analog measurements). The `IPINDEX` maps to driver-specific addressing (IOA, Modbus register, OPC ItemID).

TAGS Table

```
Columns: NAME, TAG, UPPERLIMIT, LOWERLIMIT, UPPERWARNING, LOWERWARNING,
         ALARMGROUP, COMMENT, RECEIPE, ENABLED, IOA, PARAMS, UNIT
Purpose: Alarm limits and IOA mapping per tag
```

Each sample point has one or more tags (typically just `VALUE_TAG`). The `IOA` column stores the Information Object Address used by IEC protocols to route data. Digital points may have tags for each state.

PROPS Table

```
Columns: SKEY, IKEY, DVAL
Purpose: Key-value property storage
```

Stores arbitrary configuration properties. Common keys:

- `SKEY='System', IKEY='Language'` - UI language setting
- `SKEY='SAMPLEPROPS', IKEY=<samplepoint_name>` - Sample point specific config (JSON/XML)
- `SKEY='Semaphore'` - Hardware semaphore properties

USERS Table

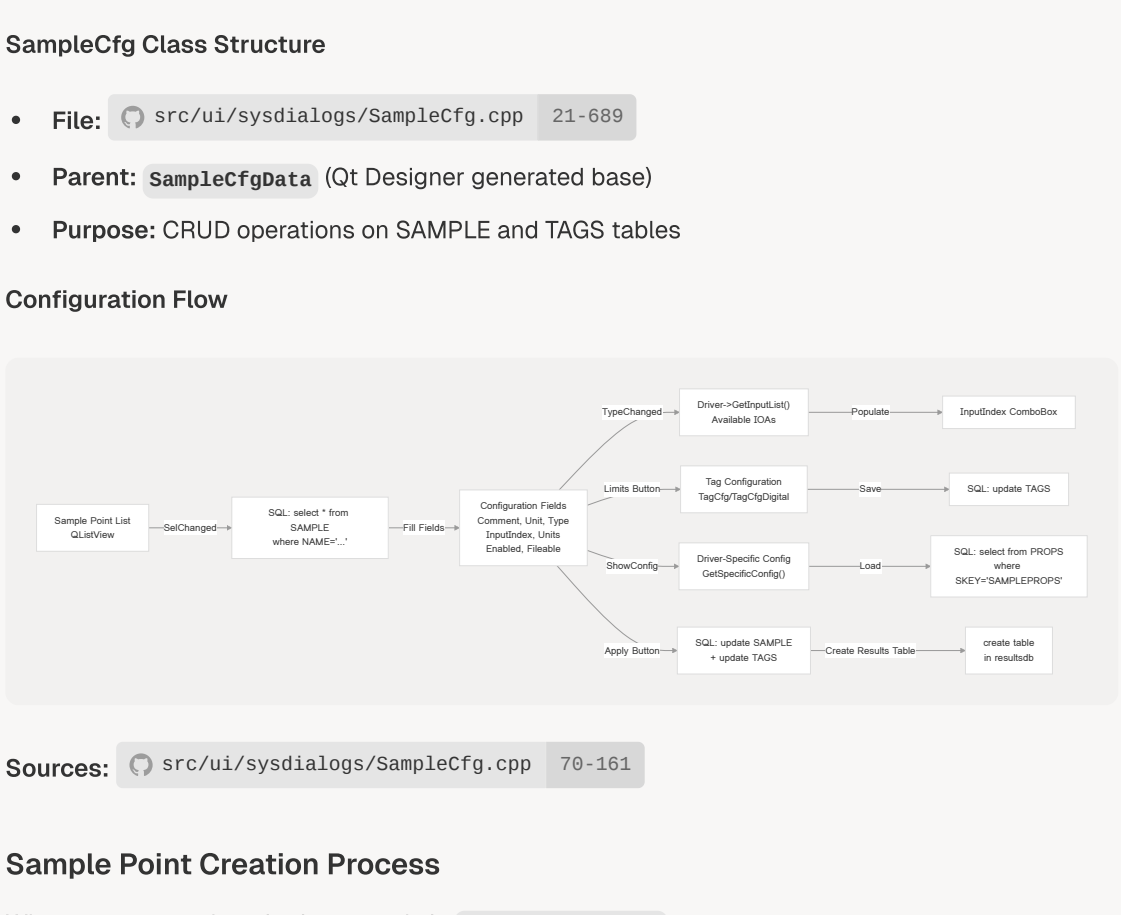
```
Columns: NAME, PASSWORD, PRIVS
Purpose: User authentication and authorization
```

Privilege levels:

- `PRIVS_ALL` - Full administrative access
- `PRIVS_FACTORY_ADMIN` - Factory floor supervisor
- `PRIVS_ACK_ALARMS` - Can acknowledge alarms
- `PRIVS_REPORTS` - Can generate reports
- `PRIVS_CAN_EXIT` - Can exit the application

Sources: `src/ui/sysdialogs/SampleCfg.cpp 21-689` `src/ui/sysdialogs/UnitCfg.cpp 18-238` `src/monitor/monitor.cpp 305-307`

Configuration Loading Sequence


manager.exe
monitor.exe
configdb
Protocol Driver
SQLite Config
System Startup Sequence
StartProcess("monitor.exe")
"select IKEY,DVAL from PROPS where SKEY='System'"
System properties
"select UNITTYPE from UNITS where ENABLED=1"
List of enabled unit types
FindDriver(unittype)
Load DLLs
"select * from SAMPLE where ENABLED=1"
Enabled sample points
Build Results::EnabledPoints
StartProcess("iec104master.exe -l 1")
"select * from iec104_database1"
IOA mappings
"select IKEY,DVAL from PROPS where SKEY='SAMPLEPROPS'"
Sample point properties
Initialize protocol stack
Configuration loaded
Start() all drivers
manager.exe
monitor.exe
configdb
Protocol Driver
SQLite Config

Sources: `src/monitor/monitor.cpp 305-319` `src/monitor/monitor.cpp 913-992` `src/drivers/iec104/iec104driverthread.cpp 22-143`

Sample Point Configuration

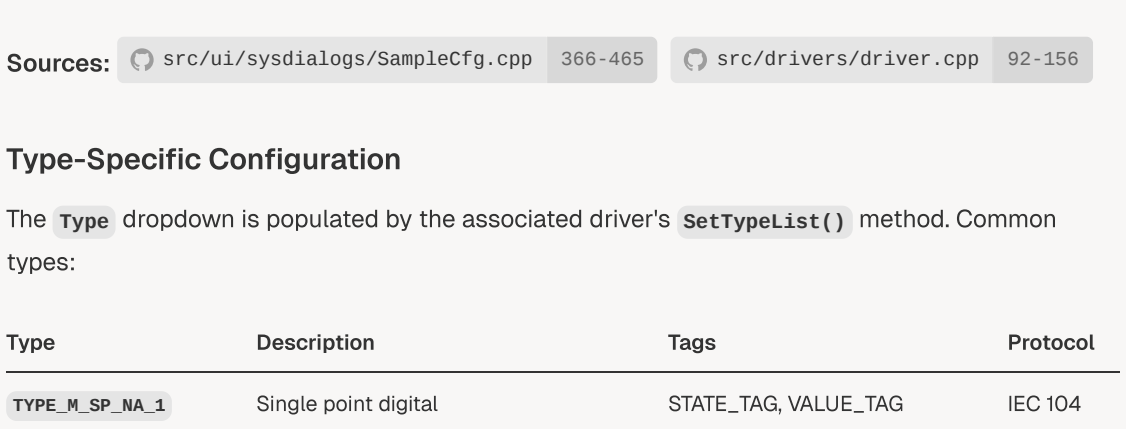
Sample points are configured through the `SampleCfg` dialog, accessed via **Configure → Configure Sample Points** menu.

Sample Point Configuration Dialog

SampleCfg Class Structure

- **File:** `src/ui/sysdialogs/SampleCfg.cpp 21-689`
- **Parent:** `SampleCfgData` (Qt Designer generated base)
- **Purpose:** CRUD operations on SAMPLE and TAGS tables

Configuration Flow


Sample Point List QListView
SelChanged
SQL: select * from SAMPLE where NAME='...'
Fill Fields
Configuration Fields Comment, Unit, Type InputIndex, Units Enabled, Fileable
TypeChanged
Driver->GetInputList() Available IOAs
Populate
InputIndex ComboBox
Limits Button
Tag Configuration TagCfg/TagCfgDigital
Save
SQL: update TAGS
ShowConfig
Driver-Specific Config GetSpecificConfig()
Load
SQL: select from PROPS where SKEY='SAMPLEPROPS'
Apply Button
SQL: update SAMPLE + update TAGS
Create Results Table
create table in resultsdb

Sources: `src/ui/sysdialogs/SampleCfg.cpp 70-161`

Sample Point Creation Process

When a new sample point is created via `SampleCfg::New()`:

1. **User Input:** NewDlg prompts for unique sample point name
2. **Database Records Created:**
   - `insert into SAMPLE` - Main configuration record
   - `insert into TAGS` - One row per tag (usually just VALUE_TAG)
   - `insert into TAGS_DB` - Current value placeholder in currentdb
   - `insert into CVAL_DB` - Aggregate current value
   - `insert into PROPS` - Empty properties placeholder
3. **Results Table Created:**

```
create table <SamplePointName> (
    TIMEDATE <DATE_TIME_COL_TYPE>,
    STATE int4,
    VALUE_TAG real8
);
create index on <SamplePointName>.TIMEDATE;
```

4. **UI Updates:** Added to list view, selected for immediate configuration

Sources: `src/ui/sysdialogs/SampleCfg.cpp 366-465` `src/drivers/driver.cpp 92-156`

Type-Specific Configuration

The `Type` dropdown is populated by the associated driver's `SetTypeList()` method. Common types:

Type	Description	Tags	Protocol
`TYPE_M_SP_NA_1`	Single point digital	STATE_TAG, VALUE_TAG	IEC 104
`TYPE_M_DP_NA_1`	Double point digital	STATE_TAG, VALUE_TAG	IEC 104
`TYPE_M_ME_NC_1`	Short floating point analog	VALUE_TAG	IEC 104
`TYPE_M_ME_TF_1`	Floating point with timestamp	VALUE_TAG	IEC 104

When `Type` changes, `SampleCfg::TypeChanged()` calls the driver's `GetInputList()` to populate available input indices (IOAs, Modbus addresses, OPC ItemIDs).

Driver-Specific Configuration Widget:

If a driver implements `GetSpecificConfig()`, a custom QWidget is embedded in the `ConfigFrame`. For example, IEC 104 drivers may show IOA mapping tables; Modbus drivers may show register address configuration.

Sources: `src/ui/sysdialogs/SampleCfg.cpp 83-101` `src/ui/sysdialogs/SampleCfg.cpp 113-142` `src/ui/sysdialogs/SampleCfg.cpp 143-162`

Unit Configuration

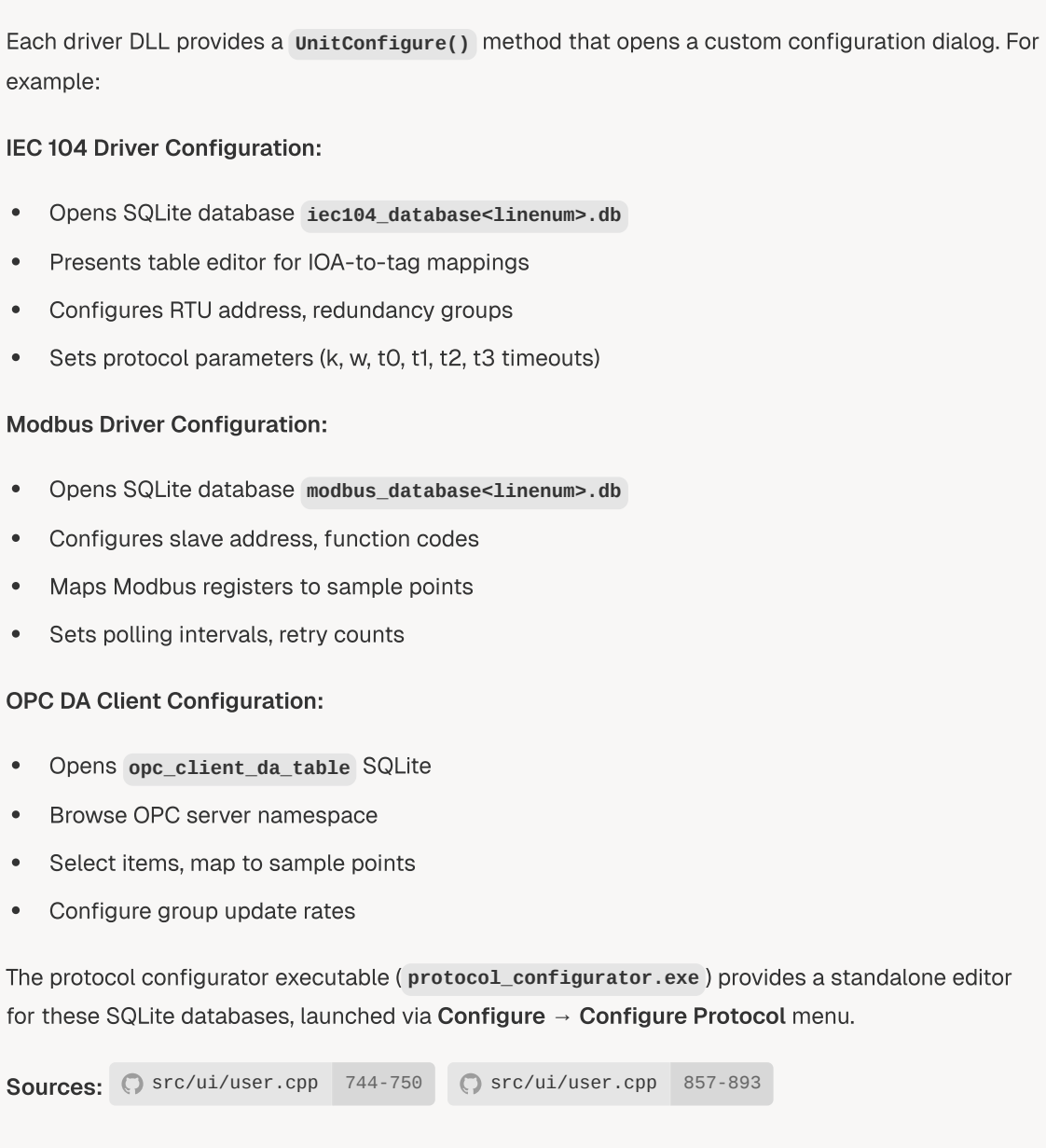
Units (driver instances) are configured through the `UnitCfg` dialog, accessed via **Configure → Configure Units** menu.

Unit Configuration Dialog

UnitCfg Class Structure

- **File:** `src/ui/sysdialogs/UnitCfg.cpp 18-238`
- **Purpose:** Manage driver instances, invoke driver-specific configurators

Configuration Workflow


User Opens UnitCfg
SQL: select NAME from UNITS
Display Unit List
User Selects Unit
SQL: select * from UNITS where NAME='...'
Fill Fields: Comment, UnitType, Enabled
Action Buttons
Configure Button
New Button
Apply Button
Delete Button
FindDriver(UnitType)
NewDlg: Get Unit Name
SQL: update UNITS
Confirmation Dialog
driver->UnitConfigure() Driver-specific dialog
Select UnitType from DLL list
SQL: delete from UNITS + driver->DropOneSpecTable()
Edit SQLite database (protocol mapping tables)
SQL: insert into UNITS
Auto-invoke Configure

Sources: `src/ui/sysdialogs/UnitCfg.cpp 58-238`

Driver-Specific Configuration

Each driver DLL provides a `UnitConfigure()` method that opens a custom configuration dialog. For example:

IEC 104 Driver Configuration:

- Opens SQLite database `iec104_database<linenum>.db`
- Presents table editor for IOA-to-tag mappings
- Configures RTU address, redundancy groups
- Sets protocol parameters (k, w, t0, t1, t2, t3 timeouts)

Modbus Driver Configuration:

- Opens SQLite database `modbus_database<linenum>.db`
- Configures slave address, function codes
- Maps Modbus registers to sample points
- Sets polling intervals, retry counts

OPC DA Client Configuration:

- Opens `opc_client_da_table` SQLite
- Browse OPC server namespace
- Select items, map to sample points
- Configure group update rates

The protocol configurator executable (`protocol_configurator.exe`) provides a standalone editor for these SQLite databases, launched via **Configure → Configure Protocol** menu.

Sources: `src/ui/user.cpp 744-750` `src/ui/user.cpp 857-893`

Protocol Configuration Tools

protocol_configurator.exe

Standalone application for editing protocol-specific SQLite databases without restarting the monitor process.

Launch: Via **Configure → Configure Protocol** menu or directly from `C:\scada\bin\`

Functionality:

- Opens SQLite databases in `C:\scada\project\`
- Provides table editor for IOA mappings, register addresses, OPC items
- Validates configuration before saving
- No database server connection required (SQLite is file-based)

Typical SQLite Schema (IEC 104):

```
CREATE TABLE iec104_database1 (
    ID INTEGER PRIMARY KEY,
    IOA INTEGER UNIQUE NOT NULL,
    CASDU INTEGER,
    TYPE INTEGER,
    SAMPLEPOINT TEXT,
    TAG TEXT,
    DESCRIPTION TEXT
);
```

Sources: `src/ui/user.cpp 857-893`

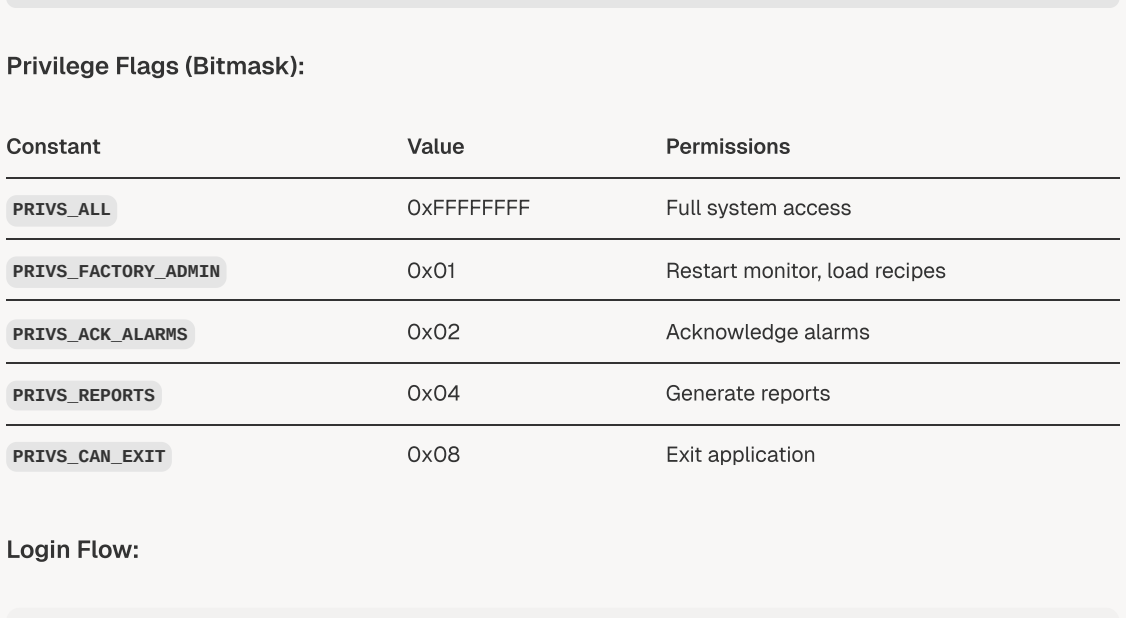
hmi_designer.exe

Qt Designer-based tool for creating HMI screens, launched via **Configure → Configure HMI** menu.

Features:

- Drag-drop custom SCADA widgets (LEDs, tanks, thermometers, switches)
- Saves `.ui` files to `C:\scada\project\`
- Widget names map to sample points (truncated at underscore)
- Loaded dynamically by `ui.exe` at startup via `QWidgetFactory::create()`

HMI Screen Loading:


ui.exe
File System
QWidgetFactory
HMI_manager
QTabWidget
FindFirstFile("C:\scada\project\*.ui")
loop
[For each .ui file]
DescriptoreF.cFileName
hmi_database.insert(filename, new HMI_manager)
QWidgetFactory::create(map_path, hmi_mng)
QDialog* designerHMI
setParent(designerHMI)
addTab(designerHMI, filename)
All HMI screens loaded as tabs
ui.exe
File System
QWidgetFactory
HMI_manager
QTabWidget

Sources: `src/ui/user.cpp 812-848` `src/ui/user.cpp 390-476`

User and Security Configuration

User authentication and role-based access control are configured via the `UserCfgDlg` dialog.

User Management Dialog

Access: Configure → Configure Users (requires `PRIVS_ALL`)

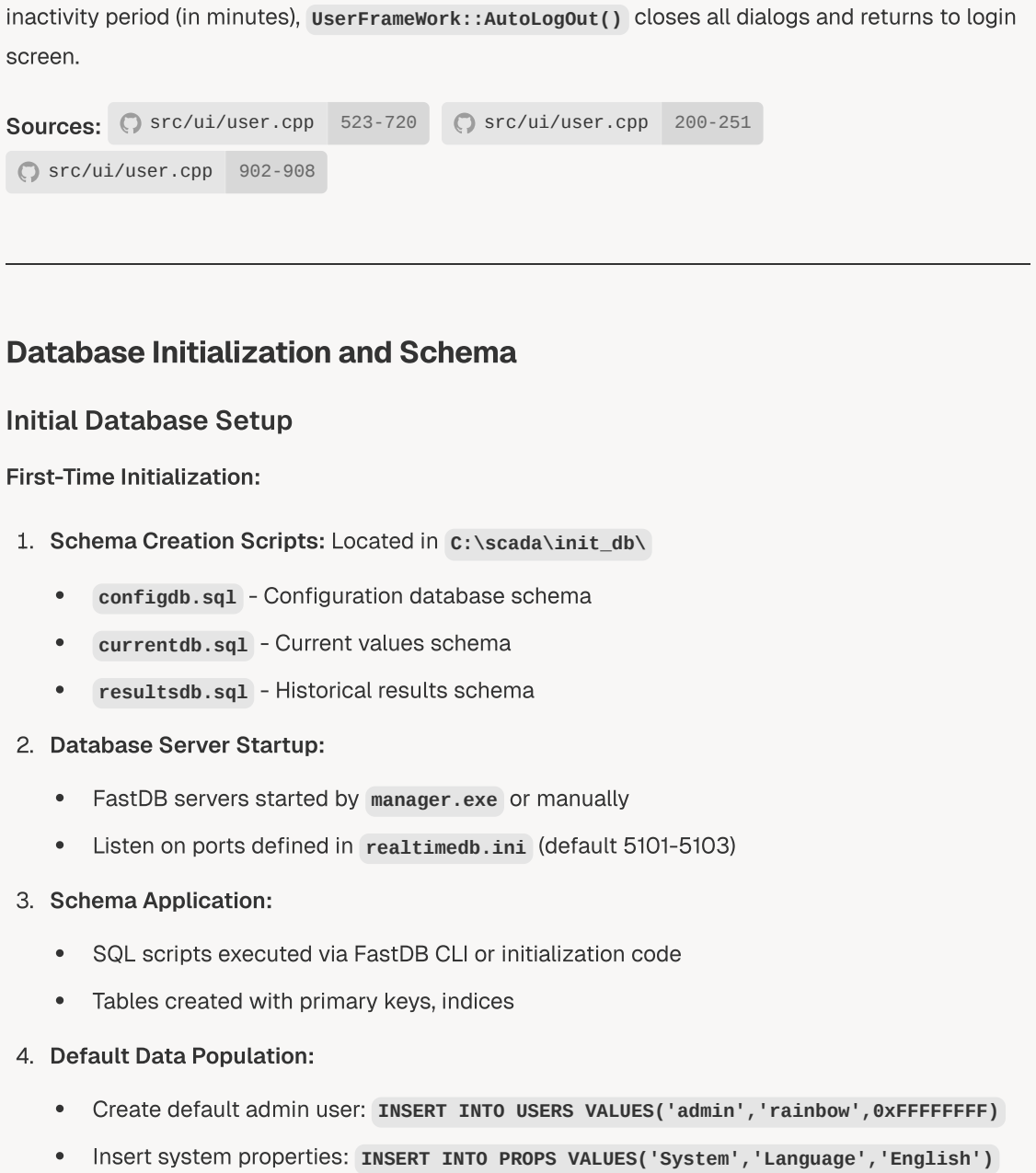
User Table Schema:

```
CREATE TABLE USERS (
    NAME TEXT PRIMARY KEY,
    PASSWORD TEXT,
    PRIVS INTEGER
);
```

Privilege Flags (Bitmask):

Constant	Value	Permissions
`PRIVS_ALL`	0xFFFFFFFF	Full system access
`PRIVS_FACTORY_ADMIN`	0x01	Restart monitor, load recipes
`PRIVS_ACK_ALARMS`	0x02	Acknowledge alarms
`PRIVS_REPORTS`	0x04	Generate reports
`PRIVS_CAN_EXIT`	0x08	Exit application

Login Flow:


Operator
LoginDlg
configdb
UserFrameWork
Menu Bar
Login() called
Prompt username/password
Credentials
"select * from USERS where NAME='...' and PASSWORD='...'"
alt
[Authentication Success]
User record
Set GetUserDetails().privs
Build menu based on privileges
opt
[Has PRIVS_REPORTS]
Add Reports menu
opt
[Has PRIVS_ALL]
Add Configure menu
Add System Control menu
opt
[Has PRIVS_ACK_ALARMS]
Add Ack All Alarms button
Start auto-logout timer
[Authentication Failure]
Error message
Retry or exit
Operator
LoginDlg
configdb
UserFrameWork
Menu Bar

Auto-Logout:

If `System.AutoLogoffTime` property is set, the system monitors keyboard/mouse activity. After inactivity period (in minutes), `UserFrameWork::AutoLogOut()` closes all dialogs and returns to login screen.

Sources: `src/ui/user.cpp 523-720` `src/ui/user.cpp 200-251` `src/ui/user.cpp 902-908`

Database Initialization and Schema

Initial Database Setup

First-Time Initialization:

1. **Schema Creation Scripts:** Located in `C:\scada\init_db\`
   - `configdb.sql` - Configuration database schema
   - `currentdb.sql` - Current values schema
   - `resultsdb.sql` - Historical results schema
2. **Database Server Startup:**
   - FastDB servers started by `manager.exe` or manually
   - Listen on ports defined in `realtimedb.ini` (default 5101-5103)
3. **Schema Application:**
   - SQL scripts executed via FastDB CLI or initialization code
   - Tables created with primary keys, indices
4. **Default Data Population:**
   - Create default admin user: `INSERT INTO USERS VALUES('admin','rainbow',0xFFFFFFFF)`
   - Insert system properties: `INSERT INTO PROPS VALUES('System','Language','English')`

Configuration Database Schema Details

Complete SAMPLE Table Definition:

```
NAME TEXT PRIMARY KEY         - Unique sample point identifier
COMMENT TEXT                  - User description
UNIT TEXT                     - Associated driver instance name
QTYPE TEXT                    - Quality type (IEC type)
IPINDEX TEXT                  - Input index (IOA, address, ItemID)
UNITS TEXT                    - Engineering units (e.g., "degC", "kPa")
ENABLED BOOLEAN               - Include in monitoring
FILEABLE BOOLEAN              - Store to historical database
LOGMODE BOOLEAN               - Continuous or change-of-value logging
RETRIGGER BOOLEAN             - Retrigger alarms on each violation
ALARMTHRESHOLD INTEGER        - Hysteresis (seconds) before alarm
```

TAGS Table Extensions:

```
NAME TEXT                     - Sample point reference
TAG TEXT                      - Tag name (VALUE_TAG, STATE_TAG, etc.)
UPPERLIMIT REAL               - High alarm limit
LOWERLIMIT REAL               - Low alarm limit
UPPERWARNING REAL             - High warning limit
LOWERWARNING REAL             - Low warning limit
ALARMGROUP TEXT               - Alarm group membership
COMMENT TEXT                  - Tag-specific comment
RECEIPE TEXT                  - Recipe association
ENABLED BOOLEAN               - Tag enabled flag
IOA INTEGER                   - Information Object Address
PARAMS TEXT                   - Additional parameters (JSON/XML)
UNIT TEXT                     - Associated driver instance
```

Sources: `src/drivers/driver.cpp 92-156` `src/ui/sysdialogs/SampleCfg.cpp 396-456`

Database Backup and Version Control

Configuration Versioning:

On Linux systems, `monitor.exe` can invoke a version control script on startup:

```
#!/bin/bash
# version.sh - Called by monitor.exe on startup
cd $SCADA_HOME/configdb
cvs commit -m "$1"  # Commit message from monitor
```

This provides CFR 21 Part 11 compliance by tracking all configuration changes in CVS. On Windows, this functionality is commented out but can be implemented with Git or similar tools.

Manual Backup:

FastDB databases can be backed up by copying the memory-mapped files:

- `configdb.fdb`
- `currentdb.fdb`
- `resultsdb.dbs`

Sources: `src/monitor/monitor.cpp 833-858`

Configuration Change Propagation

When configuration changes are saved via UI dialogs, the changes must be propagated to running processes.

Runtime Configuration Updates

Restart Required:

Most configuration changes (units, sample points, alarm limits) require a monitor restart via **System Control → Restart Monitor**. This is handled by `UserFrameWork::restart()`:

1. Sends `CMD_SHUTDOWN` to monitor.exe via dispatcher
2. Monitor stops all drivers via `Stop()`
3. Monitor reloads configuration from configdb
4. Monitor restarts all enabled drivers via `Start()`
5. Broadcasts `MONITOR_STARTED_NOTIFY` when ready

Hot Reconfiguration:

Some properties can be changed without restart:

- User privileges (takes effect on next login)
- Alarm acknowledgment
- System properties (language, auto-logout time)

Sources: `src/ui/user.cpp 1196-1226` `src/monitor/monitor.cpp 803-865` `src/monitor/monitor.cpp 873-898`

Configuration Best Practices

Initial System Setup Checklist

1. **Install IndigoSCADA:** Copy files to `C:\scada\`
2. **Configure Databases:**
   - Edit `realtimedb.ini` with server addresses
   - Start FastDB servers on specified ports
   - Initialize schemas from `init_db/*.sql`
3. **Create Admin User:**

```
INSERT INTO USERS VALUES('admin','<password>',4294967295);
```

4. **Configure System Properties:**
   - Launch `ui.exe`, login as admin
   - **Configure → Configure System**
   - Set language, auto-logout, timezone
5. **Define Units:**
   - **Configure → Configure Units**
   - Create driver instances for each RTU/PLC
   - Configure protocol-specific parameters via driver dialogs
6. **Define Sample Points:**
   - **Configure → Configure Sample Points**
   - Create sample points for each monitored variable
   - Set alarm limits, engineering units
   - Map to unit input indices (IOAs, addresses)
7. **Configure HMI:**
   - Launch `hmi_designer.exe`
   - Create `.ui` screens with SCADA widgets
   - Name widgets to match sample points
   - Save to project directory
8. **Test Configuration:**
   - **System Control → Restart Monitor**
   - Verify data acquisition in Alarms tab
   - Check sample point values update
   - Test control commands

Common Configuration Issues

Problem	Cause	Solution
Sample point not updating	IOA mismatch between TAGS and driver config	Verify IOA in both TAGS table and SQLite protocol DB
Driver fails to start	Missing SQLite config or connection error	Check protocol_configurator DB exists and has valid entries
"Lock System Manager" error	Previous crash left lock set	**System Control → Clear System Manager Lock**
HMI screen not loading	Syntax error in .ui file	Check qwt plugin loaded, verify .ui in project directory
Cannot login	Database connection failed	Verify realtimedb.ini, check FastDB servers running

Sources: `src/ui/user.cpp 917-925` `src/monitor/monitor.cpp 305-319`

Sources for this entire document: `src/ui/user.cpp 1-2029` `src/ui/sysdialogs/SampleCfg.cpp 1-689` `src/ui/sysdialogs/UnitCfg.cpp 1-238`

src/monitor/monitor.cpp 1-1715
src/database/realtimedb.cpp 1-375
src/drivers/driver.cpp 1-736
src/ui/main.cpp 1-138

Build System and Deployment

› Relevant source files

This page documents the build system, compilation process, and deployment procedures for IndigoSCADA. It covers the Visual Studio workspace structure, project dependencies, third-party library integration, and installation layout.

For information about configuring databases after installation, see page 6.5. For details about the process manager that orchestrates the runtime system, see page 2.4.

Build System Overview

IndigoSCADA uses **Microsoft Visual Studio 6.0** as its primary build environment. The entire system is organized as a single workspace containing multiple interdependent projects.

Workspace Structure

The main workspace file is `daq.dsw`, located at `src/daq/daq.dsw 1-1047` This workspace contains **over 40 individual projects** organized into several categories:

Category	Projects	Purpose
Core Processes	`monitor`, `manager`, `ui`	Main system executables
Protocol Drivers	`iec104driver`, `modbus_driver`, `dnp3driver`, `opc_client_da_driver`, etc.	Driver DLLs loaded by monitor
Protocol Masters	`iec104master`, `modbus_master`, `dnp3master`, `opc_client_da`, etc.	Child processes for protocol communication
Databases	`fastdb`, `gigabase`, `realtimesqlserver`, `sqlserver`	Database engines
Middleware	`librtps`, `rtps_mngr`, `fifo`	Inter-process communication
Utilities	`utilities`, `scada`	Shared libraries and common code
HMI Components	`designer`, `designerlib`, `qui`, `qwt`, `qwtplugin`	User interface and design tools
Servers	`opc_server_da`, `opc_ua_server`, `dspserver`	Data export services

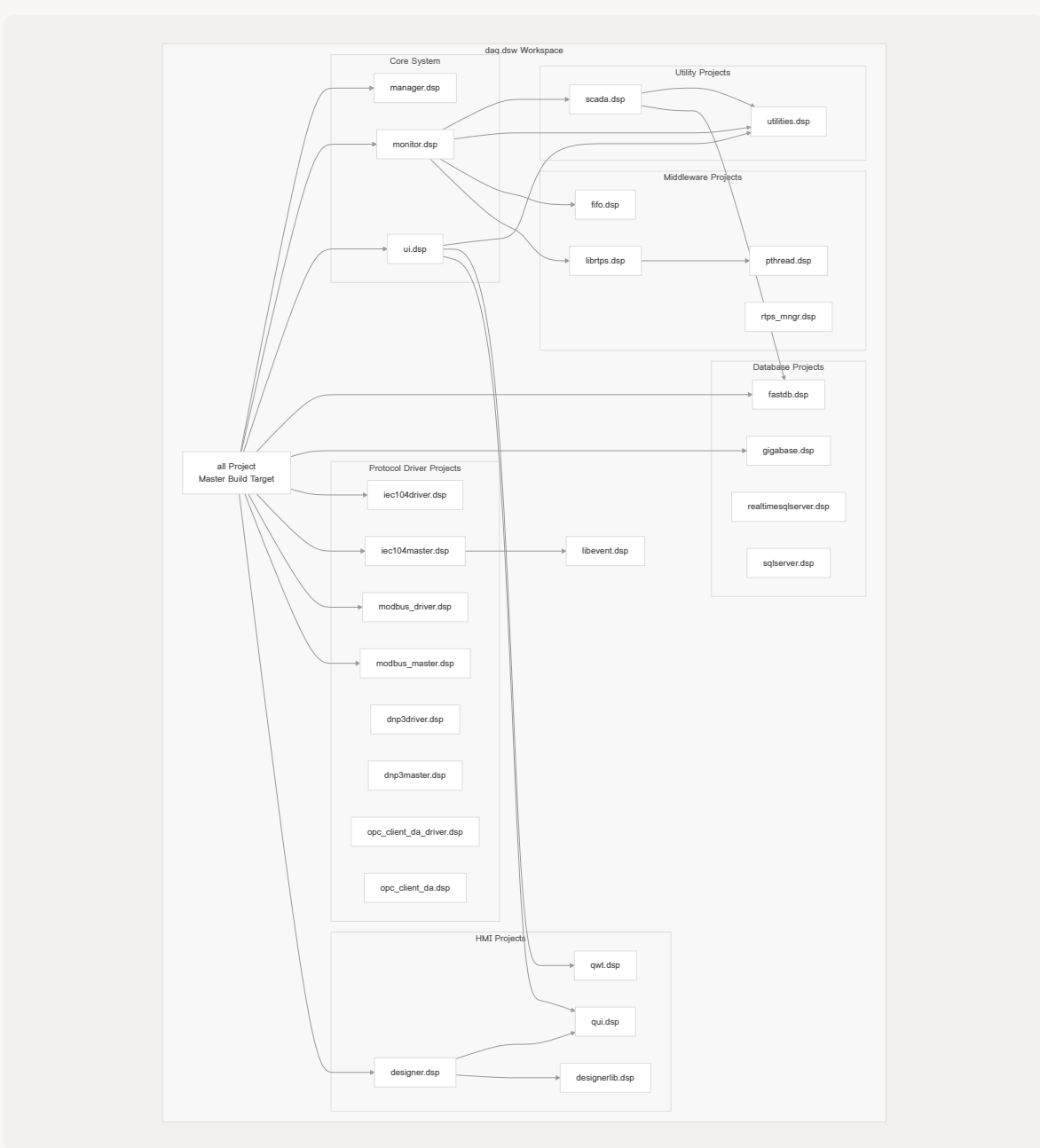


Diagram: Visual Studio Workspace Project Organization

The `all` project serves as the master build target with dependencies on all other projects, ensuring correct build order.

Sources: `src/daq/daq.dsw 51-227`

Build Configuration

Each project in the workspace defines two build configurations: **Release** and **Debug**.

Compiler Settings

Release Configuration:

- Compiler: `/nologo /MD /W3 /GX /O2`
- Optimization: Full optimization (`/O2`)
- Runtime: Multithreaded DLL (`/MD`)
- Output: `c:\scada\bin`
- Intermediate: `Release` directory

Debug Configuration:

- Compiler: `/nologo /MDd /W3 /Gm /GX /Zi /Od`
- Optimization: Disabled (`/Od`)
- Runtime: Multithreaded Debug DLL (`/MDd`)
- Debug Info: Full debug info (`/Zi`)
- Output: `c:\scada\bin`
- Intermediate: `Debug` directory

Include Paths

Most projects share common include directories:

```
$(QTDIR)\include
.
..
..\..
..\..\common
..\..\database
..\..\utilities
..\..\configurator\sqlite
..\..\middleware\rtps\include
..\..\middleware\ripc\inc
..\..\fifo
```

These paths enable projects to access shared headers and cross-component interfaces.

Linker Configuration

Common libraries linked:

- Qt: `qt-mt3.lib`, `qtmain.lib`
- Windows: `kernel32.lib`, `user32.lib`, `ws2_32.lib`, `ole32.lib`, `oleaut32.lib`
- System: `utilities.lib` (Debug: `utilitiesd.lib`)
- Middleware: `rtps.lib`, `fifo.lib`
- Database: `fastdb.lib`, `gigabase.lib`

Example from modbus_master project:

`src/drivers/modbus/protocol/modbus_master.dsp 54-56` (Release):

```
LINK32 ... ws2_32.lib winmm.lib rtps.lib utilities.lib /nologo /subsystem:console
/machine:I386 /out:"c:\scada\bin/modbus_master.exe" /libpath:"c:\scada\lib"
```

`src/drivers/modbus/protocol/modbus_master.dsp 84` (Debug):

```
LINK32 ... ws2_32.lib winmm.lib fifo.lib rtps.lib utilitiesd.lib /nologo
/subsystem:console /incremental:no /debug /machine:I386
/out:"c:\scada\bin/modbus_master.exe" /libpath:"c:\scada\lib"
```

Sources: `src/drivers/modbus/protocol/modbus_master.dsp 31-86` `src/drivers/iec103/protocol/iec103master.dsp 31-84`

Project Dependencies

The workspace defines explicit project dependencies to ensure correct build order. The dependency graph is managed through `Project_Dep_Name` entries.

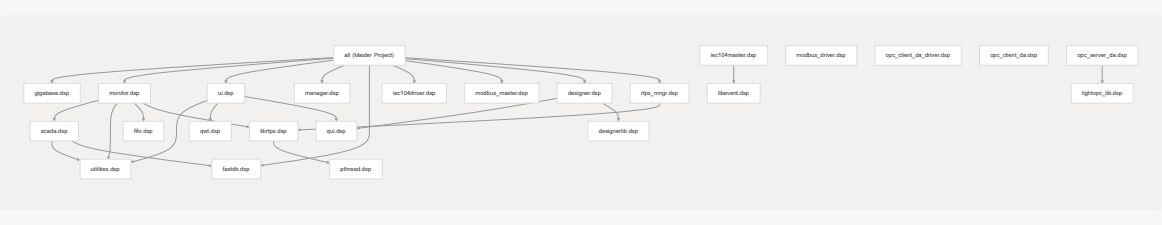


Diagram: Critical Build Dependencies

Key dependency patterns:

- **monitor.exe** depends on: `scada`, `utilities`, `fifo`, `librtps`
- **ui.exe** depends on: `utilities`, `qwt`, `qui`
- **Protocol masters** (e.g., `iec104master.exe`) depend on: middleware and utilities
- **Middleware** (`librtps`) depends on: `pthread`

These dependencies ensure libraries are built before the executables that use them.

Sources: `src/daq/daq.dsw 639-660` `src/daq/daq.dsw 999-1019` `src/daq/daq.dsw 948-959`

Third-Party Library Integration

IndigoSCADA integrates several third-party libraries, some included in the source tree and others required as external dependencies.

Included Dependencies

Library	Version	Purpose	Location in Source Tree
FastDB	3.56	In-memory database for configuration and real-time data	`src/fastdb`
GigaBASE	3.37	Object-oriented database for historical data	`src/gigabase`
EiC	4.4.2	Embedded C interpreter for calculated driver	Embedded in calculated driver
qwt	4.2.0	Qt Widgets for Technical Applications (HMI components)	`src/ui/qwt-4.2.0`
pthreads-w32	2.8.0	POSIX threads for Windows	`src/common/pthreads-w32-2-8-0`
SQLite	3.x	Protocol configuration databases	`src/configurator/sqlite`
libevent	1.4.13	Event notification library for IEC 104	`src/drivers/iec104/protocol/libevent-1.4.13-stable`
LightOPC	0.888	OPC DA 2.0 server library	`src/drivers/opc_server_da/lo-0.888`

External Dependencies

Qt Development Framework 3.3.8 is the primary external dependency. Projects reference Qt through the `$(QTDIR)` environment variable:

- Include path: `$(QTDIR)\include`
- Library path: `$(QTDIR)\lib`
- Required libraries: `qt-mt3.lib`, `qtmain.lib`

All projects use the **multithreaded, DLL version** of Qt (`qt-mt3.lib`), indicated by the `/MD` (Release) and `/MDd` (Debug) compiler flags.

Compiler Defines for Qt:

- `QT_DLL`
- `QT_THREAD_SUPPORT`
- `QT_ACCESSIBILITY_SUPPORT`
- `QT_TABLET_SUPPORT`
- `QT_NO_DEBUG` (Release only)

Sources: `src/README.txt 1-30` `src/CREDITS.txt 1-7` `src/hmi_designer/designer/app/designer.dsp 45`

Compilation Process

Building from Source

According to the build instructions:

1. **Prerequisites:**
   - Windows operating system
   - Visual Studio C++ 6.0 or later
   - Qt 3.3.8 for Windows installed with `QTDIR` environment variable set
2. **Build Steps:**

```
1. Navigate to C:\scada\src\daq directory
2. Open daq.dsw with Visual Studio C++ 6.0
3. Set the "all" project as the active project
4. Press F7 to compile all projects
```

3. **Build Output:**
   - Executables and DLLs: `C:\scada\bin`
   - Static libraries: `C:\scada\lib`
   - Intermediate object files: `Release` or `Debug` subdirectories in each project folder

Build Order

The Visual Studio workspace automatically resolves the dependency graph and builds projects in the correct order. The `all` project, with dependencies on all other projects, ensures proper sequencing.

Typical build sequence:

1. Utility libraries (`utilities`, `pthread`)
2. Database engines (`fastdb`, `gigabase`)
3. Middleware (`librtps`, `fifo`)
4. Core shared library (`scada`)
5. Protocol implementations and drivers
6. Main executables (`monitor`, `manager`, `ui`)
7. HMI components (`designer`, `qwt`, `qui`)

Sources: `src/README.txt 9-16` `src/daq/daq.dsw 51-227`

Installation and Directory Structure

Installation Procedure

Binary Installation:

From `src/USAGE.txt 3-12`:

```
1. Download indigo-scada-binaries-day-month-year-hour-min.exe
2. Remove C:\scada directory if upgrading from previous release
3. Run installer and install to C:\ disk
4. Run C:\scada\bin\manager.exe
```

Source Installation:

From `src/USAGE.txt 17-22`:

```
1. Download indigo-scada-src-day-month-year-hour-min.tar.gz
2. Copy \src directory to C:\scada to have both runtime and development environment
```

Directory Layout

The installation creates the following directory structure:

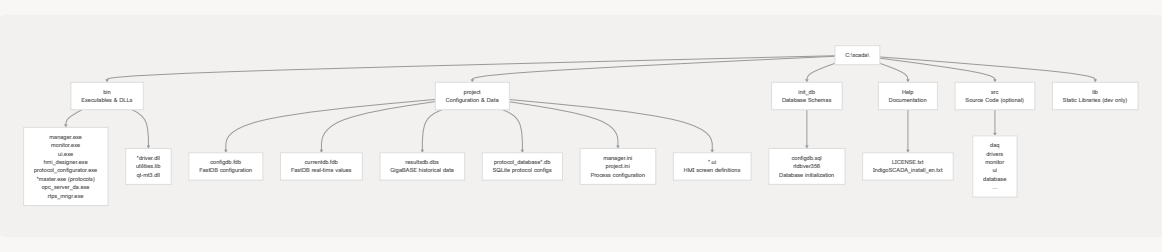


Diagram: IndigoSCADA Directory Structure

Directory Purposes

Directory	Content	Purpose
`C:\scada\bin`	Executables, DLLs, Qt libraries	Runtime binaries
`C:\scada\project`	Configuration databases, INI files, UI files	Project-specific configuration and data (configurable via `project.ini`)
`C:\scada\init_db`	SQL schema files	Database initialization scripts
`C:\scada\Help`	License, installation guides	Documentation
`C:\scada\lib`	Static libraries (`.lib` files)	Development-only, for linking
`C:\scada\src`	Source code tree	Development-only, optional

Home Directory Detection

The system automatically determines its installation directory at runtime using `src/common/common.cpp 43-70`:

```
void SetScadaHomeDirectory(const QString &s)
{
    #ifdef WIN32
    char path[_MAX_PATH];
    path[0] = '\0';
    if(GetModuleFileName(NULL, path, _MAX_PATH))
    {
        *(strrchr(path, '\\')) = '\0';   // Strip \filename.exe
        *(strrchr(path, '\\')) = '\0';   // Strip \bin
        HomeDirectory = path;            // Results in C:\scada
    }
    #endif
}
```

This function is called at process startup to establish `HomeDirectory`, which other components use to locate databases, configuration files, and libraries.

Project Directory Configuration

Starting from version noted in `src/CHANGES.txt 161` (21-10-2013), the project directory became configurable. The system reads `C:\scada\bin\project.ini` to determine the location of project files:

From `src/common/common.cpp 72-149`:

```
void SetScadaProjectDirectory(const QString &s)
{
    QString ini_file = QString(path) + "\\bin\\project.ini";
    Inifile iniFile((const char*)ini_file);

    if(iniFile.find("path","project_directory"))
    {
        QString dir;
        dir = iniFile.find("path","project_directory");
        ProjectDirectory = dir;
    }
    else
    {
        // Initialize default directory in project.ini
        FILE * fp = fopen((const char*)ini_file,"w+");
        fprintf(fp, "[project_directory]\n");
        fprintf(fp, "path=%s\\project\n", path);
        fclose(fp);
    }
}
```

This allows multiple project configurations on the same machine.

Sources: `src/USAGE.txt 1-24` `src/common/common.cpp 43-149` `src/CHANGES.txt 161`

Platform Support and Porting

Windows Platform (Primary)

IndigoSCADA is **actively maintained for Windows** with full Visual Studio 6.0 build support. The codebase extensively uses Windows-specific APIs:

- Win32 API for process management, IPC (named pipes), file operations
- COM/DCOM for OPC client and server implementations
- Windows sockets (Winsock2) for network protocols
- Windows threading and synchronization primitives

Windows-specific code is guarded by `#ifdef WIN32` preprocessor directives throughout the codebase.

Linux Platform (Legacy)

According to `src/HISTORY.txt 1-12`:

```
The codebase origin goes back to 2000, when it was developed under Linux OS.
Then in 2001 it was ported to Windows 2000 Pro.
Since 2001 we had no time and requests to port the codebase back to Linux again.
So the codebase now requires a lot of #ifdef WIN32 or #ifdef UNIX
to compile again on Linux.

I am looking for volunteers interested in porting it back to Linux.
```

Porting Challenges:

1. **No Linux Makefiles:** `src/README.txt 22-24` states: "At the moment no makefiles are available for compilation under Linux. You need to create your own makefiles."
2. **Pervasive Windows Code:** Conditional compilation directives like `#ifdef WIN32` appear throughout the codebase in:
   - Process management (CreateProcess vs fork/exec)
   - IPC mechanisms (named pipes vs Unix domain sockets)
   - DLL loading (LoadLibrary vs dlopen)
   - File paths (backslash vs forward slash)
3. **Platform-Specific Dependencies:**
   - Windows: COM/OLE for OPC, Win32 threading
   - Linux: Would require pthreads, replacement for OPC functionality

Example of platform-specific code from `src/common/common.cpp 184-200`:

```
handle =
#ifdef UNIX
    dlopen((const char *)dllName, RTLD_GLOBAL); // Linux
#endif
#ifdef WIN32
    LoadLibrary((const char *)dllName);         // Windows
#endif
```

Platform Evolution

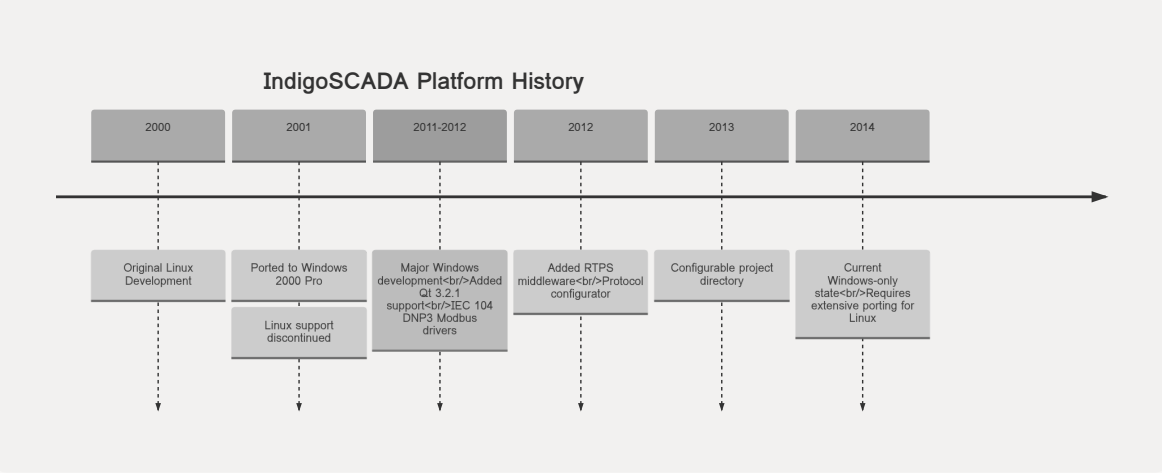

IndigoSCADA Platform History
2000
2001
2011-2012
2012
2013
2014
Original Linux Development
Ported to Windows 2000 Pro
Linux support discontinued
Major Windows development<br/>Added Qt 3.2.1 support<br/>IEC 104 DNP3 Modbus drivers
Added RTPS middleware<br/>Protocol configurator
Configurable project directory
Current Windows-only state<br/>Requires extensive porting for Linux

Diagram: Platform Support Timeline

Sources: `src/HISTORY.txt 1-12` `src/README.txt 21-29` `src/common/common.cpp 184-221` `src/CHANGES.txt 29`

Deployment Packages

IndigoSCADA provides two types of distribution packages:

Binary Distribution

Package Name: `indigo-scada-binaries-day-month-year-hour-min.exe`

Contents:

- Pre-compiled executables in `C:\scada\bin`
- Required DLLs (including Qt libraries)
- Empty project directory structure in `C:\scada\project`
- Database initialization files in `C:\scada\init_db`
- Documentation in `C:\scada\Help`

Target Users: End users and system operators who want to run IndigoSCADA without development tools.

Deployment Steps:

1. Run self-extracting installer
2. Install to `C:\` drive (creates `C:\scada` directory tree)
3. Initialize databases from `init_db` schemas if needed
4. Configure `manager.ini` for specific deployment
5. Launch `C:\scada\bin\manager.exe`

Source Distribution

Package Name: `indigo-scada-src-day-month-year-hour-min.tar.gz`

Contents:

- Complete source code tree
- Visual Studio workspace and project files
- Third-party library sources (FastDB, GigaBASE, qwt, etc.)
- Build scripts and configuration files

Target Users: Developers who want to modify, extend, or study the system.

Development Setup:

1. Install binary distribution first (for runtime environment)
2. Extract source tarball to `C:\scada\src`
3. Install Qt 3.3.8 SDK with `QTDIR` environment variable
4. Install Visual Studio C++ 6.0 or later
5. Open `C:\scada\src\daq\daq.dsw`
6. Build using instructions in `Compilation Process` section

Version Identification

Distribution packages use timestamp-based versioning: `day-month-year-hour-min`. This indicates the build date and time, allowing users to identify the exact version.

Change Log: The `src/CHANGES.txt 1-167` file documents major changes by date, providing a development history that corresponds to these versioned releases.

Sources: `src/USAGE.txt 1-24` `src/README.txt 1-30`

Build System Evolution

IndigoSCADA's build system has evolved significantly over its development history:

Date	Change	Impact
2000-2001	Initial Linux development	Makefiles, gcc, Linux-specific code
2001	Windows port	Visual Studio 6.0 workspace created
16-09-2011	Qt upgrade	Ported to `qt-win-noncommercial-msvc-3.2.1.exe`
02-May-2012	RTPS middleware	Added RTPS projects to workspace
15-Jul-2012	HMI expansion	Added `pthreads-w32-2-8-0`, `qwt-4.2.0`, HMI designer
06-07-2014	RIPC integration	Removed ORTE, integrated RIPC middleware

The workspace file `src/daq/daq.dsw 1-1047` contains all these integrated components as separate projects with managed dependencies.

Sources: `src/CHANGES.txt 1-167` `src/HISTORY.txt 1-12` `src/daq/daq.dsw 1-1047`

Known Issues and Limitations

Build Environment Constraints

1. **Visual Studio Version:** The `.dsp` project files are Visual Studio 6.0 format. Newer Visual Studio versions will prompt to convert projects. The conversion generally succeeds but may require adjustment of project settings.
2. **Qt Version Lock:** The system is tightly coupled to **Qt 3.3.8**. Migration to Qt 4.x or Qt 5.x would require significant code changes due to API incompatibilities.
3. **32-bit Only:** All projects target 32-bit Windows (`Win32` platform). 64-bit compilation would require:
   - Platform configuration changes
   - Testing of all COM/OLE interop code
   - Verification of database format compatibility

Deployment Limitations

1. **Fixed Installation Path:** Many components assume the default `C:\scada` installation directory. While the project directory is configurable via `project.ini`, some hard-coded paths may remain.
2. **Administrator Rights:** Installation to `C:\` typically requires administrator privileges on Windows Vista and later.
3. **No Installer Customization:** The binary distribution installer does not support custom installation paths beyond the drive letter.

Linux Status

As documented in TODO.txt, Linux support is listed as a priority but remains unimplemented:

From `src/TODO.txt 7`:

```
- Porting to current Linux distributions
```

The effort required includes:

- Creating Makefiles or CMake build system
- Replacing Windows-specific APIs
- Porting or replacing OPC functionality (requires DCOM)
- Testing and debugging on Linux distributions

Sources: `src/TODO.txt 1-14` `src/HISTORY.txt 1-12`

Summary

IndigoSCADA's build system is a **Visual Studio 6.0 workspace** with over 40 interdependent projects organized into core processes, protocol drivers, databases, middleware, and HMI components. The system:

- Uses **two-stage compilation** (Debug and Release configurations)
- Outputs binaries to **C:\scada\bin** directory
- Depends on **Qt 3.3.8** as primary external library
- Includes embedded third-party libraries (FastDB, GigaBASE, qwt, EiC, SQLite)
- Provides **binary and source distributions** with installer packages
- Is **actively maintained for Windows only** since 2001
- Requires significant porting effort for Linux support

The installation creates a structured directory tree under `C:\scada` with separate directories for binaries, project data, database schemas, and documentation. The project directory location is configurable via `project.ini` to support multiple configurations.

Sources: src/daq/daq.dsw 1-1047 src/README.txt 1-30 src/USAGE.txt 1-24 src/HISTORY.txt 1-12 src/CHANGES.txt 1-167

Installation and Directory Structure

Relevant source files

This page documents the installation procedures for IndigoSCADA and the file system organization used by the platform. It covers the default directory layout, binary and source installations, path resolution mechanisms, and platform-specific considerations.

For information about building from source, see Build Configuration. For platform support details, see Platform Support and Porting.

Overview

IndigoSCADA uses a fixed directory structure rooted at `C:\scada` on Windows systems. The system automatically determines installation paths at runtime using the executable location, with support for configurable project directories through the `project.ini` configuration file. The directory structure separates executables, project files, database initialization scripts, logs, and documentation into distinct folders.

Sources: `src/manager/buttongroups.cpp 29-138` `src/USAGE.txt 1-24`

Default Installation Root

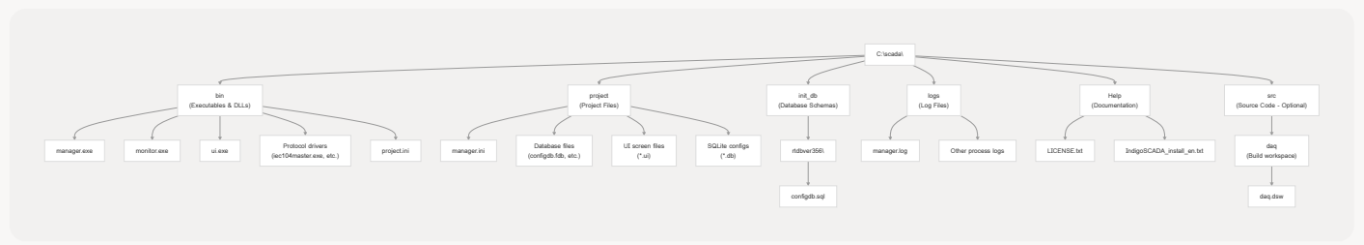
IndigoSCADA's standard installation root is:

```
C:\scada\
```

This location contains all system components including executables, configuration files, databases, and documentation. The installation root is determined at runtime by the `SetScadaHomeDirectory()` function.

Sources: `src/USAGE.txt 6-10` `src/manager/buttongroups.cpp 29-56`

Directory Structure


C:\scada\
bin
(Executables & DLLs)
project
(Project Files)
init_db
(Database Schemas)
logs
(Log Files)
Help
(Documentation)
src
(Source Code - Optional)
manager.exe
monitor.exe
ui.exe
Protocol drivers
(iec104master.exe, etc.)
project.ini
manager.ini
Database files
(configdb.fdb, etc.)
UI screen files
(*.ui)
SQLite configs
(*.db)
rtdbver356\
configdb.sql
manager.log
Other process logs
LICENSE.txt
IndigoSCADA_install_en.txt
daq
(Build workspace)
daq.dsw

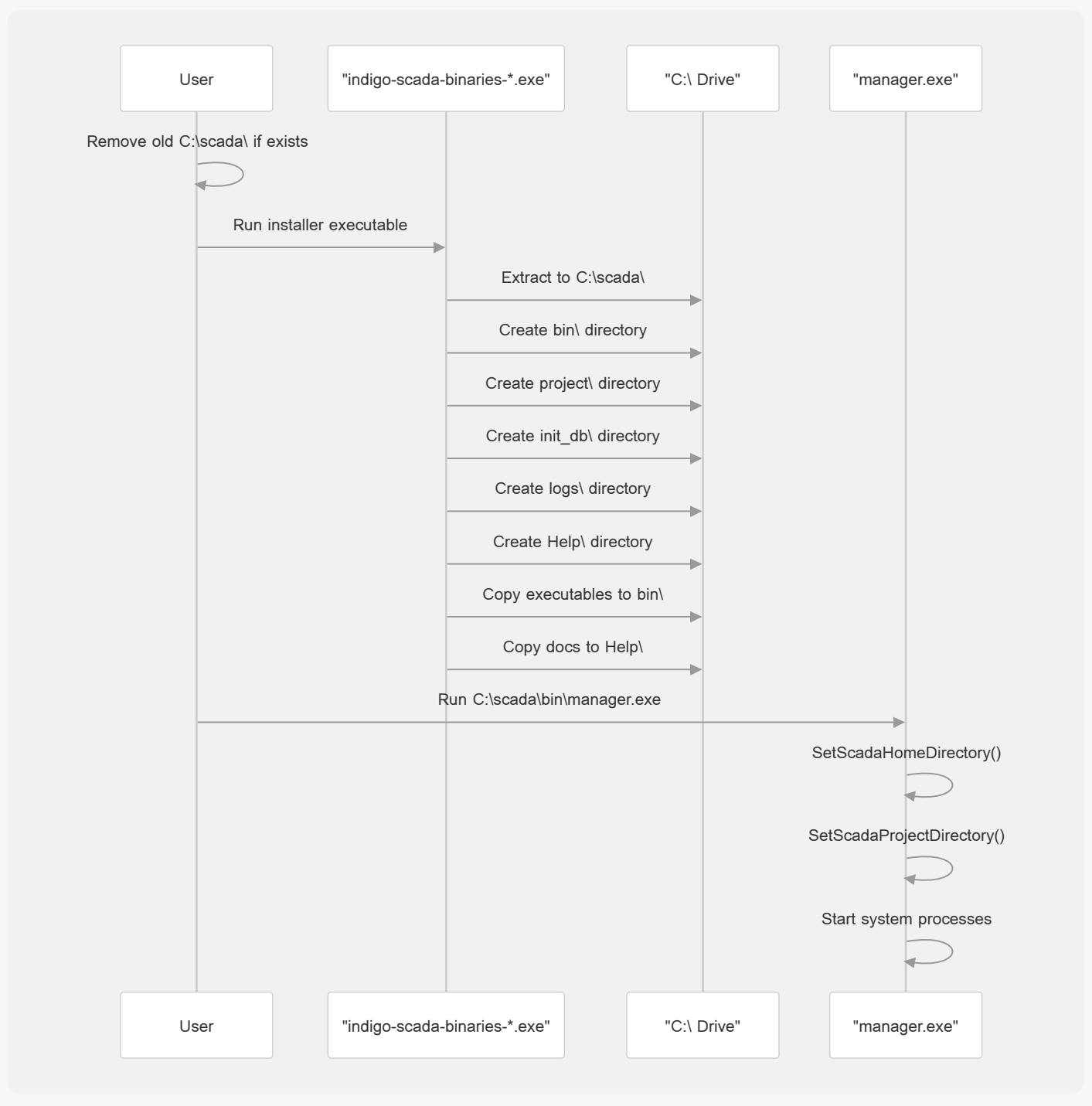
Directory Descriptions

Directory	Purpose	Key Contents
`bin\`	Compiled executables and dynamic libraries	All `.exe` and `.dll` files, `project.ini`
`project\`	Runtime project configuration and data	`manager.ini`, database files (`.fdb`, `.dbs`, `.db`), UI screens (`.ui`)
`init_db\`	Database initialization schemas	SQL scripts for creating database tables
`logs\`	System and process log files	`manager.log` and driver-specific logs
`Help\`	System documentation	License file, installation guides
`src\`	Source code (optional)	Complete source tree for development

Sources: `src/manager/buttongroups.cpp 230-233` `src/manager/buttongroups.cpp 498-512` `src/CHANGES.txt 53` `src/USAGE.txt 1-24`

Binary Installation

Standard Binary Installation Procedure


User
"indigo-scada-binaries-*.exe"
"C:\ Drive"
"manager.exe"
Remove old C:\scada\ if exists
Run installer executable
Extract to C:\scada\
Create bin\ directory
Create project\ directory
Create init_db\ directory
Create logs\ directory
Create Help\ directory
Copy executables to bin\
Copy docs to Help\
Run C:\scada\bin\manager.exe
SetScadaHomeDirectory()
SetScadaProjectDirectory()
Start system processes
User
"indigo-scada-binaries-*.exe"
"C:\ Drive"
"manager.exe"

Installation Steps

1. **Remove Previous Installation**
   - Delete `C:\scada\` directory if it exists from a previous release
2. **Run Installer**
   - Execute `indigo-scada-binaries-day-month-year-hour-min.exe`
   - Install to `C:\` root drive
3. **Launch Manager**
   - Run `C:\scada\bin\manager.exe` to start the system
4. **Review Documentation**
   - License: `C:\scada\Help\LICENSE.txt`
   - Installation guide: `C:\scada\Help\IndigoSCADA_install_en.txt`

Sources: `src/USAGE.txt 1-14`

Source Code Installation

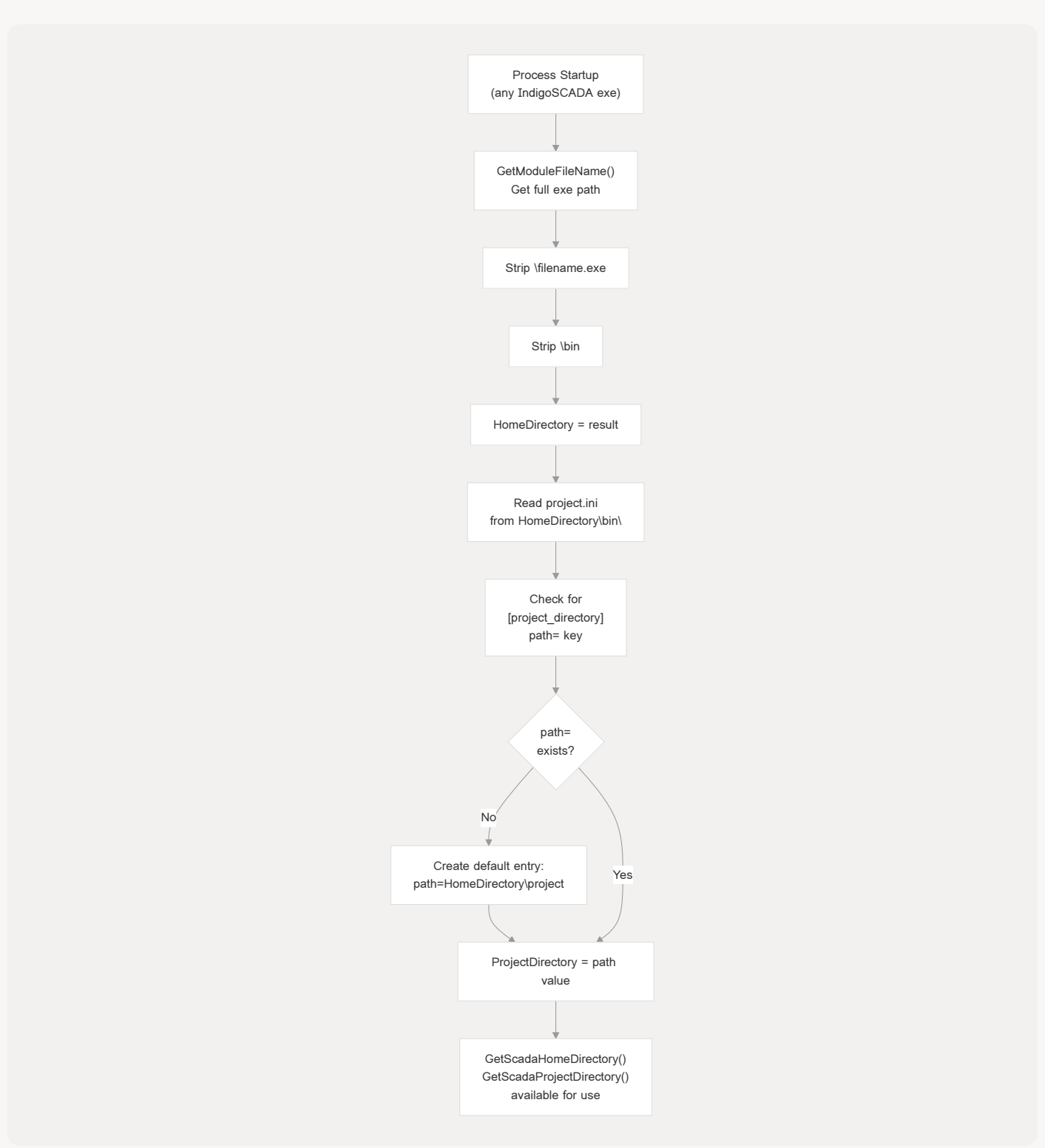
For development environments, source code can be installed alongside the binary runtime:

1. **Download Source Archive**
   - Obtain `indigo-scada-src-day-month-year-hour-min.tar.gz`
2. **Extract to Installation Root**
   - Copy the `\src` directory from the tarball to `C:\scada\`
   - Final structure: `C:\scada\src\`
3. **Build from Source**
   - Navigate to `C:\scada\src\daq`
   - Open `daq.dsw` with Visual Studio C++ 6.0 or later
   - Set "all" project as active
   - Press F7 to compile all projects

Sources: `src/USAGE.txt 17-23` `src/README.txt 9-16`

Path Resolution Mechanism

IndigoSCADA uses dynamic path resolution at runtime to locate system directories. The following diagram illustrates the code flow:


Process Startup
(any IndigoSCADA exe)
GetModuleFileName()
Get full exe path
Strip \filename.exe
Strip \bin
HomeDirectory = result
Read project.ini
from HomeDirectory\bin\
Check for
[project_directory]
path= key
path=
exists?
No
Yes
Create default entry:
path=HomeDirectory\project
ProjectDirectory = path
value
GetScadaHomeDirectory()
GetScadaProjectDirectory()
available for use

SetScadaHomeDirectory Implementation

The home directory is determined by stripping the executable name and `\bin` directory from the module path:

```
Example:
Module path: C:\scada\bin\manager.exe
After stripping \manager.exe: C:\scada\bin
After stripping \bin: C:\scada
Result: HomeDirectory = "C:\scada"
```

Implementation: `src/manager/buttongroups.cpp 29-56`

SetScadaProjectDirectory Implementation

The project directory is configurable via `project.ini` located in the `bin\` directory:

Configuration file: `C:\scada\bin\project.ini`

```
[project_directory]
path=C:\scada\project
```

If `project.ini` does not exist or the key is missing, the system creates it with a default value of `HomeDirectory\project`.

Implementation: `src/manager/buttongroups.cpp 58-135`

Usage in Code

Both directories are accessed via getter functions:

Function	Returns	Usage
`GetScadaHomeDirectory()`	`const QString&`	Base installation path
`GetScadaProjectDirectory()`	`const QString&`	Project files location

Sources: `src/manager/buttongroups.cpp 137-138`

Configuration File Locations

System Configuration Files

The following table maps configuration files to their absolute paths:

File	Path	Purpose
`project.ini`	`C:\scada\bin\project.ini`	Project directory configuration
`manager.ini`	`C:\scada\project\manager.ini`	Process manager configuration
`*.db` (SQLite)	`C:\scada\project\*.db`	Protocol-specific configurations
`configdb.fdb`	`C:\scada\project\configdb.fdb`	Configuration database (FastDB)
`currentdb.fdb`	`C:\scada\project\currentdb.fdb`	Real-time values database (FastDB)
`resultsdb.dbs`	`C:\scada\project\resultsdb.dbs`	Historical database (GigaBASE)

Configuration Path Construction

Example code showing how `manager.ini` path is constructed:

```
strcpy(pInitFile, (const char*)GetScadaProjectDirectory());
strcat(pInitFile, "\\manager.ini");
// Result: C:\scada\project\manager.ini
```

Implementation: `src/manager/buttongroups.cpp 498-500`

Sources: `src/manager/buttongroups.cpp 498-512` `src/CHANGES.txt 53`

Log File Management

Log Directory Structure

All system logs are written to the `logs\` subdirectory:

```
C:\scada\logs\
    manager.log         (Process manager log)
    iec104master1.log   (IEC 104 driver line 1)
    modbus_master2.log  (Modbus driver line 2)
    ...
```

Log Path Construction

The manager log path is constructed as:

```
char pLogFile[_MAX_PATH];
if(GetModuleFileName(NULL, pLogFile, _MAX_PATH))
{
    *(strrchr(pLogFile, '\\')) = '\0';  // Strip \filename.exe
    *(strrchr(pLogFile, '\\')) = '\0';  // Strip \bin
}
strcat(pLogFile, "\\logs\\manager.log");
// Result: C:\scada\logs\manager.log
```

Implementation: `src/manager/buttongroups.cpp 504-512`

Log Writing Function

The `ButtonsGroups::WriteLog()` function writes timestamped entries:

```
Format: MM/DD/YYYY, HH:MM:SS
        Message text
```

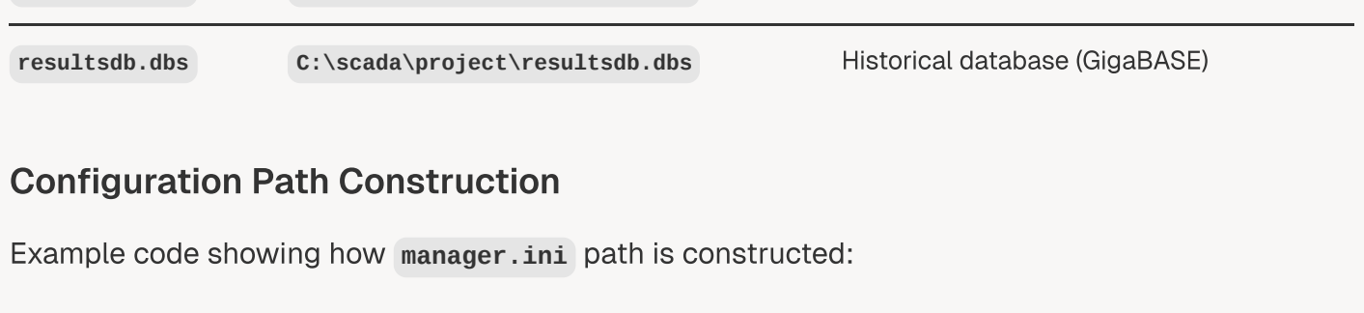
Implementation: `src/manager/buttongroups.cpp 140-153`

Sources: `src/manager/buttongroups.cpp 140-153` `src/manager/buttongroups.cpp 504-512`

Process Executable Paths

StartProcess Path Resolution

The `ButtonsGroups::StartProcess()` function constructs executable paths for child processes:


GetScadaHomeDirectory()
+ '\\bin\\'
+ ProcessName
Full executable path
Example:
C:\scada + \bin\ +
iec104master.exe
=
C:\scada\bin\iec104master.

Code implementation:

```
QString hm_dir = GetScadaHomeDirectory();
strcpy(pCommandLine, (const char*)hm_dir);
strcat(pCommandLine, "\\bin\\");
strcat(pCommandLine, pProcessName);
```

Implementation: `src/manager/buttongroups.cpp 230-233`

Sources: `src/manager/buttongroups.cpp 157-286`

Database Initialization Files

Schema Location

Database initialization SQL scripts are stored in version-specific subdirectories:

```
C:\scada\init_db\
    rtdbver356\
        configdb.sql    (FastDB 3.56 schema for configuration database)
```

The version number in the directory name (`rtdbver356`) corresponds to the FastDB version (3.56).

Schema Usage

When creating a new `configdb.fdb` database, the SQL schema from `init_db\rtdbver356\configdb.sql` is used. This file contains CREATE TABLE statements for all configuration tables including `UNITS`, `SAMPLE`, `TAGS`, `PROPS`, `USERS`, `RECEIPES`, and `SCHEDULE`.

Sources: `src/CHANGES.txt 50-74`

Platform-Specific Considerations

Windows (Primary Target)

IndigoSCADA is designed primarily for Windows environments:

- Uses Windows-specific path separators (`\`)
- Uses `GetModuleFileName()` Win32 API for path resolution
- Requires Visual Studio C++ 6.0 or later for compilation
- Qt 3.3.8 for Windows is required

Conditional compilation: `src/manager/buttongroups.cpp 31-42`

Linux (Legacy Support)

Historical Linux support exists but requires porting:

- Original development occurred on Linux in 2000
- Ported to Windows 2000 Pro in 2001
- Extensive `#ifdef WIN32` conditionals throughout codebase
- No current makefiles for Linux compilation

The path resolution functions include Unix-style alternatives, but they have not been maintained since 2001:

Unix path handling: `src/manager/buttongroups.cpp 44-55`

Note: Linux compilation would require creating makefiles and addressing platform-specific code throughout the system.

Sources: `src/HISTORY.txt 1-12` `src/README.txt 21-28` `src/manager/buttongroups.cpp 31-55`

Build Dependencies and Installation

Required Components for Compilation

Component	Version	Purpose
Visual Studio C++	6.0 or later	Primary compiler
Qt Development Frameworks	3.3.8	GUI framework and utilities
FastDB	3.56	Real-time database (included)
GigaBASE	3.37	Historical database (included)
EiC	4.4.2	C interpreter for calculated driver (included)

Included Dependencies

The following dependencies are bundled with the source distribution and do not require separate installation:

- **GigaBASE 3.37:** Historical time-series database
- **FastDB 3.56:** In-memory real-time database
- **EiC-4.4.2:** Embedded C interpreter

Sources: `src/README.txt 1-30` `src/CREDITS.txt 1-7`

Installation Verification

After installation, verify the following directory structure exists:

```
C:\scada\
├── bin\
│   ├── manager.exe
│   ├── monitor.exe
│   ├── ui.exe
│   ├── project.ini
│   └── (protocol drivers)
├── project\
│   └── manager.ini
├── init_db\
│   └── rtdbver356\
│       └── configdb.sql
├── logs\
└── Help\
    └── LICENSE.txt
```

Verify the system starts by running:

```
C:\scada\bin\manager.exe
```

The manager process should:

1. Successfully resolve home and project directories
2. Create log file at `C:\scada\logs\manager.log`
3. Load `C:\scada\project\manager.ini`
4. Display the task manager control panel

Ask Devin about jonathanxavier/IndigoSCADA

Fast

Sources: src/manager/buttongroups.cpp 387-543 src/USAGE.txt 1-14