FactoryLink 7

Version 7.0

. . . .

Concepts

Contents

Chapter 1

FactoryLink Concepts

FactoryLink Overview

FactoryLink is a SCADA (Supervisory Control and Data Acquisition) product built exclusively for Microsoft Operating systems. It is built on Microsoft's Distributed interNet Architecture (DNA) standard. FactoryLink lets you automate an industrial process and monitor and control that process. Using FactoryLink's graphics tool, you can build a user interface that graphically represents your process. The graphical objects within the interface are linked to real-world data points, which gives you control of the process in real time.

A process is any activity performed repeatedly, such as:

- Production of goods at a factory
- Movement of liquid or gas through a pipeline
- Periodic collection of data

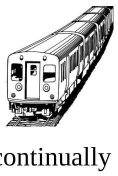
FactoryLink is not a single program, but a collection of programs or *tasks*. Each task is a separate program that performs a specialized function within FactoryLink. For example, functions such as alarming, data logging, graphics, etc. are controlled by separate tasks, but the tasks function together to produce a seamless application.

The uses for FactoryLink are unlimited. It is a tool to build a graphical application that mimics the process you want to automate.

Examples of FactoryLink Implementations

FactoryLink has been implemented in a wide variety of applications. Here are a few examples:

Dallas Area Rapid Transit



One application of FactoryLink is used in the management of a city's rapid-transit system. The Dallas (Texas) Area Rapid Transit (DART) uses FactoryLink to monitor the locations of its trains.

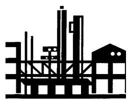
Three huge life-size monitors display the locations of all trains in the rapid-transit system. FactoryLink reads data from SLC 500s that control the trains. FactoryLink continually monitors the status of the rapid transit system for alarm conditions, such as trains too close together, or emergency conditions in the tunnels used by the trains.

Nanjing Lukou International Airport

At Nanjing (People's Republic of China) Lukou International Airport, FactoryLink monitors and controls the baggage handling system. The departures system feeds off three independent departure terminals, each with 20 check-in desks. FactoryLink automatically processes and x-rays each piece of luggage before it joins a collector conveyor, and then feeds it into an unloading carousel in the baggage hall. In the arrivals system, there are 8 infeed lines in turn feeding 4 inclined baggage-reclaim carousels.

FactoryLink's graphical user interface displays a fully animated screen, providing the airport's maintenance engineers with a visual check on the status of the entire system, including some 2,000 meters of conveyors, carousels and x-ray machines. Its highly sophisticated alarm-management system detects with pinpoint accuracy conveyor blockages, conveyor belt motor failure, or photo-cell alignment problems. The layout is color-coded to show green for Running, red for Stop, black for Ready, and yellow for Fault --allowing for differences in the languages.

British Steel



To lower costs and increase product flexibility, British Steel upgraded one of its steel works with new manufacturing equipment and a FactoryLink system. British Steel set three goals for the refurbishment:

- Higher-quality, more consistent products
- Lower production costs
- Increase in operational flexibility to produce more products and shorter runs

FactoryLink monitors and controls initial setup conditions and all plant operations. The application provides a window into the plant, with dynamic graphical displays showing up-to-the-second status of the plant and material. Diagrams indicate the position and status of individual steel sections. Real-time and historical trending graphs provide invaluable decision support. With FactoryLink's sophisticated alarm handling, operators can see and act on any alarm from any station. Both the alarm and its subsequent handling are logged in a historical file. In addition, operators can use the terminals to enter new set-points or other key data. All changeable parameters are protected by a multilevel password system.

To speed change-over between products, FactoryLink stores machine settings and other parameters relevant to every type of steel section. For example, operators can instantly retrieve any one of over 200 setup configurations for the RSMs (roller straightening machines). This capability has reduced RSM setup time from 15 minutes to a few seconds. The system controlling the saws is automated to the extent that operators only have to check the position of the saw before starting saw cuts.

One particular plant employs two FactoryLink systems. Because each FactoryLink system has the power to handle all of the application, the servers are configured to back up each other in case one fails or is damaged.

By upgrading its mill with FactoryLink, British Steel has realized several important benefits:

- By eliminating several manufacturing steps and controlling operations through FactoryLink, British Steel can now respond faster to customer demands and produce smaller quantities efficiently.
- Set-up time for the RSMs has dropped dramatically.
- Engineers and managers can now see their entire section of the plant on the FactoryLink graphical display, enabling them to spot problems immediately and fix them faster.
- With more information available faster, engineers can make better operational decisions.

Telecommunications Manufacturer

FactoryLink is used by the world's leading provider of corded and cordless phones and answering machines. This manufacturer designs, develops, manufactures, markets, and sells a complete range of personal communication products, including digital and analog cellular phones, PCS phones, and other cordless and corded telephones.

The Wireless Technology Production Center, an automated production facility that manufactures the new breed of cellular PCS phones, required a manufacturing execution system (MES) to control production and link the shop-floor system with its SAP enterprise system. The company selected FactoryLink to connect and control individual production machines, conveyor controls, test station controllers, and assembly-line controllers.

The solution had two goals:

- Make products that could be individually configured and still ship within 24 hours.
- Track product information closely and connect the design and ordering systems to the production systems.

These capabilities would allow the company to immediately correct nonconformities and quickly move new products from design to production.

FactoryLink provides real-time communications to different types of production equipment, conveyor controls, and test-station and assembly-station controllers. By connecting to several databases, FactoryLink allows production data to be moved directly into engineering- and production-control data storage. FactoryLink's direct integration with shop-floor systems allows manufacturing data reporting in real time.

The new system provides some major advantages:

- Integration of the enterprise system, FactoryLink, and the shop floor control system, seamlessly passing data among them to optimize production efficiency
- Managed execution of scheduled production runs; the system can reschedule nonconforming product, route parts to and from rework stations, and direct production lines to produce the required product
- Rapid line changes by downloading new process programs to equipment before product reaches the machine
- Real-time usage data uploaded automatically to purchasing and accounting systems in SAP, and real-time test data logged to the database for analysis
- Exception or nonconformance reporting, including data saved for off-line analysis by third-party packages

FactoryLink Environment

The FactoryLink environment is a client/server system based on Microsoft's DNA standard, which is a multi-tier architecture.

Client/Server

Client/server describes the relationship between two computer programs. A *client* is the requesting program, or user, which requests data from another program, the server. The *server* receives a request for data, processes the request, and returns the data to the client. A server program provides data to client programs in the same computer or in other computers. The computer that the server program runs on is also referred to as a server (though it may contain a number of server and client programs). For example, the user of a Web browser is effectively making client requests for pages from servers all over the Web. The browser itself is a client in its relationship with the computer that is receiving and returning the requested files. The computer handling the request and sending the Web pages back to the user is the server.

Multi-Tier

Multi-tier is a term that applies to a computer system where the software is layered, by function, among multiple computers. Generally, each of these layers resides on its own computer. The most common multi-tier architecture is the client/server system, which is a two-tier system; the user interface is on one tier (the client) and the business process and data storage are on the other tier (the server).

Windows DNA architecture supports three tiers, or layers:

- User interface
- Business process

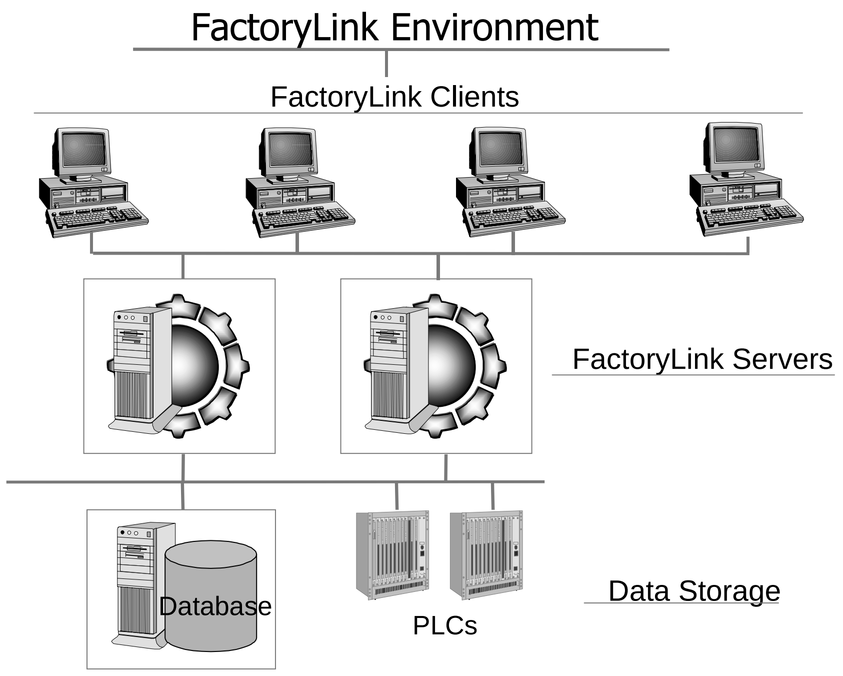
- Data storage

FactoryLink's Architecture

FactoryLink conforms to the Windows DNA three-tier standard:

- User interface: FactoryLink Client
- Business process (the application): FactoryLink Server
- Data storage: SQL Server, or other database product, and PLCs

The following diagram illustrates a FactoryLink environment:



FactoryLink Client

The FactoryLink Client provides two things:

- A connection to the server to build a FactoryLink application
- A graphical user interface into the application

FactoryLink's application-development tools reside on the client.

FactoryLink Server

The FactoryLink Server is where the FactoryLink application and the FactoryLink programs reside, so the server provides all data-processing functionality.

Data Storage

Because data required by the application can come from any external source, the data-storage tier can be a database product and/or PLCs.

OPC

The FactoryLink Client and Server communicate using an OPC interface. OLE for Process Control (OPC) is a communication standard developed by a group of software and hardware vendors, in cooperation with Microsoft, to standardize the way data is accessed. OPC is incorporated into Microsoft Windows DNA.

OPC allows client applications to access data consistently, regardless of the source (server). Hardware vendors generally provide OPC servers so that their data can be accessed without needing proprietary drivers. SCADA vendors generally provide OPC clients. Because of FactoryLink's client/server architecture, FactoryLink is unique in that it also provides an OPC server. Your FactoryLink application can not only collect data, it can also, using its OPC server, distribute that data to other applications throughout your organization.

FactoryLink's Benefits

FactoryLink's multi-tier architecture offers major benefits:

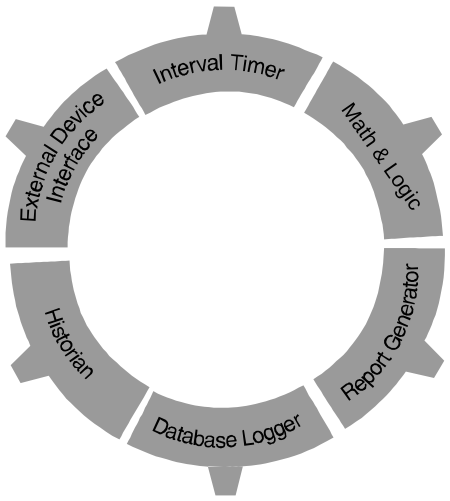
- Each server can have one or more application.
- The applications can be developed and maintained from any of the clients.
- Modifications made to any application are "pushed" to any clients looking at that application.
- A FactoryLink system can be scaled or expanded, based on the needs of the business. A small application can exist with all tiers on a single computer. A large application may require multiple servers. For example, the middle tier for the servers can have alarming on one server, trending on another, and so forth. And, because the application database is separate, the data-storage tier can be scaled to grow with the business.

FactoryLink Tasks

FactoryLink is a set of programs that perform a specific activity in the automation process, such as:

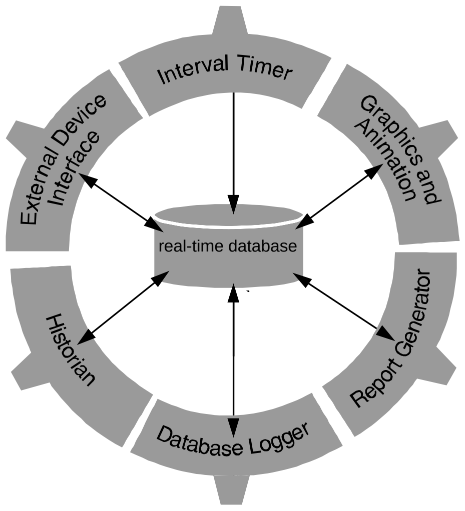
- Reading and writing of data to external devices (PLCs, RTUs)
- Collection and storage of data
- Alarming
- Generating reports

These programs are referred to as modules or *tasks* because they are independent programs that do only one specific job or task alone, but together make an application fully functional. You access the tasks through the Configuration Explorer, one of the FactoryLink application-development tools.



REAL-TIME DATABASE

USDATA's patented Open Software Bus architecture provides a global real-time database (RTDB) through which all FactoryLink tasks communicate.



The real-time database is a block of memory allocated by the system when the application starts. It serves two purposes:

- Stores application data
- Provides intertask communication

The real-time database stores data that has been:

- Collected from an external device, such as a PLC
- Computed by a FactoryLink task, such as Math & Logic
- Manually entered through a keyboard or a graphic screen

After data is stored in the real-time database, other tasks can access and manipulate the data. The tasks communicate with each other by reading values from and writing values to the database. Tasks do not communicate directly.

This architecture allows communication with other applications, such as an Oracle database or an Excel spreadsheet, and also allows you to develop your own tasks to integrate with FactoryLink.

Together, the real-time database and the tasks make up the FactoryLink application, which runs on the server.

Real-Time Database Elements (Tags)

The values in the real-time database are stored in memory locations known as real-time database elements or *tags*. This section explains:

- Concept of a tag
- How to define a tag

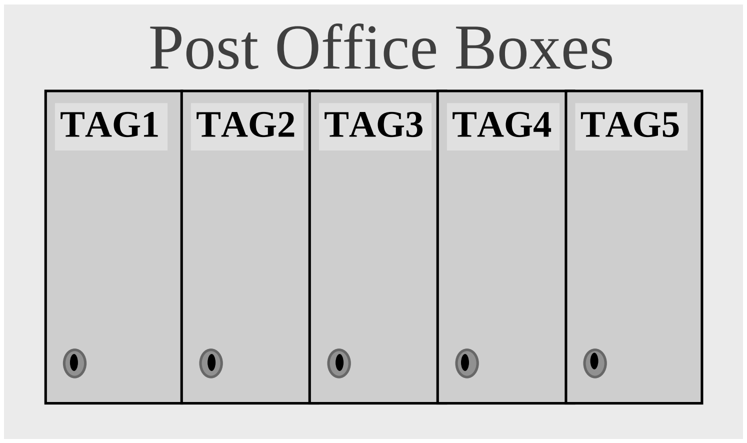
- Basic structure of a tag
- Tag types and their specific structure

What is a Tag?

A tag is nothing more than a memory location within the real-time database (specifically, a segment and offset). Any value that is written to the database is stored within a tag.

To use an analogy, the tag concept is similar to that of a post-office box. Each tag is like a P.O. box in that, just as a P.O. box stores mail, a tag stores information. Just as each P.O. box has a unique identification: its box number, a tag has a unique identification; its *tag name*. **A tag name is the name you assign to a tag when you define it.** Just as the contents of the P.O. box change constantly, so does the value of the tag change. Just as, to send something to the P.O. box, you must use the correct box number, to write a value to a tag, you must use the correct tag name. To read the contents of a P.O. box or a tag, you must go to the correct P.O. box or tag to retrieve its contents or value.

Tag = Element
Tagname = Name used to reference elements



- A tag is like a P.O. box.
- Each box or tag has a unique way to be identified.
- Each tag has a unique tag name.
- Each tag's contents may change.
- You can send or receive (write or read) values (contents) to or from a tag.

Defining and Naming Tags

Whenever you enter a new tag name in one of the FactoryLink application-development tools, Configuration Explorer or Client Builder, you are prompted to define a new tag.

Tag Naming Requirements

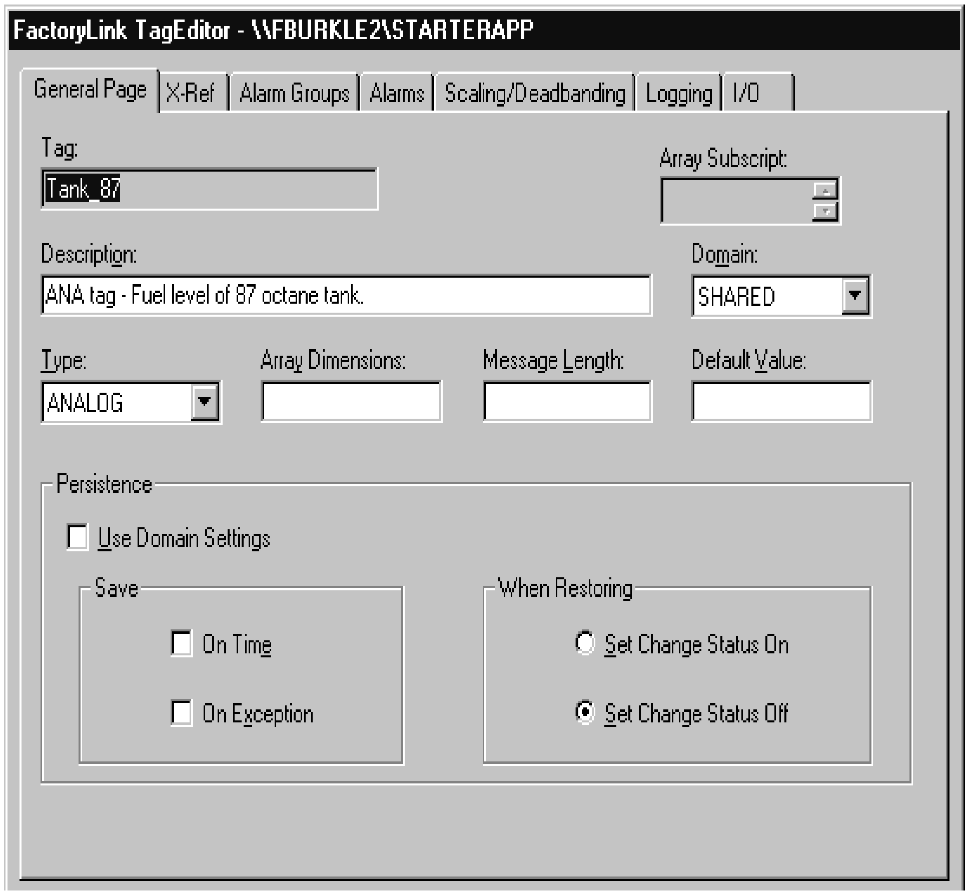
When naming tags, be aware of the following requirements:

- **Tags are case-sensitive.** Use capitalization consistently in tag names. For example, a tag named “TEST” and a tag named “Test” are two different tags to FactoryLink.
- Tags can have up to 32 alphanumeric characters.
- Tags cannot have spaces or periods.
- Tags cannot start with numbers.

Try to name tags in a way that describes their purpose. This is very helpful when you need to reference them in an application. After a tag is defined, any object in any Client Builder screen or any task may reference that tag.

Tag Editor

Entering a new tag name in a configuration table in the Explorer or in an animation dialog in Client Builder displays the FactoryLink Tag Editor. The Tag Editor prompts you for information about the new tag so the system can define it in the real-time database. The sample diagram below shows the definition of a tag named Tank_87.


FactoryLink TagEditor - \\FBURKLE2\STARTERAPP

General Page | X-Ref | Alarm Groups | Alarms | Scaling/Deadbanding | Logging | I/O

Tag: Tank_87

Array Subscript:

Description: ANA tag - Fuel level of 87 octane tank.

Domain: SHARED

Type: ANALOG

Array Dimensions:

Message Length:

Default Value:

Persistence

☐ Use Domain Settings

Save

☐ On Time

☐ On Exception

When Restoring

○ Set Change Status On

● Set Change Status Off

The Tag Editor requires that you specify a data type. The other information is optional, depending on how you intend to use the tag in the application. Tag descriptions are recommended because they are very useful in self-documentation of the application. For a full discussion of the Tag Editor, refer to the *FactoryLink Configuration Explorer User Guide*.

Tag Data Types

FactoryLink supports several different types of data in the real-time database. The tag type you select determines what data a tag can store.

Digital

A digital type is a one-bit binary number that can only be set to a value of 0 (OFF) or 1 (ON).

Analog

An analog type is a 16-bit signed integer. This data type can handle a number between negative 32,768 and positive 32,767. Remember, no internal checking is done for rollover of numbers. Rollover can cause problems by misrepresenting the value, much as a car odometer rolls over from 99,999 to 0. Though the odometer may read 00002, you know the car has gone 100,002 miles. You may not be aware of an analog tag's value rolling over, so ensure that the values are within a range that prevents this condition.

Long-Analog

A long-analog type (longana) is a 32-bit signed integer; its value can range between negative 2,147,483,648 and positive 2,147,483,647.

Floating-point

A floating-point (float) type is an IEEE double-precision number supporting up to 31 places to the right of the decimal with a value range from positive or negative 10 raised to the power of plus or minus 308.

Message

A message value can be any combination of alphanumeric characters. You set the length limit in the tag definition.

Mailbox

A mailbox is a unique data type that specific tasks use to communicate with each other. This is the only data type that can queue data rather than overwrite the previous value. The contents of this data type vary in length.

System Tags

Many time-related *system tags* are pre-defined by FactoryLink to provide system-wide time information to all FactoryLink tasks. For example, there are tags for date, time, number of minutes past the hour, current month, etc. These tags are updated as soon as FactoryLink starts and are continuously updated as the system runs.

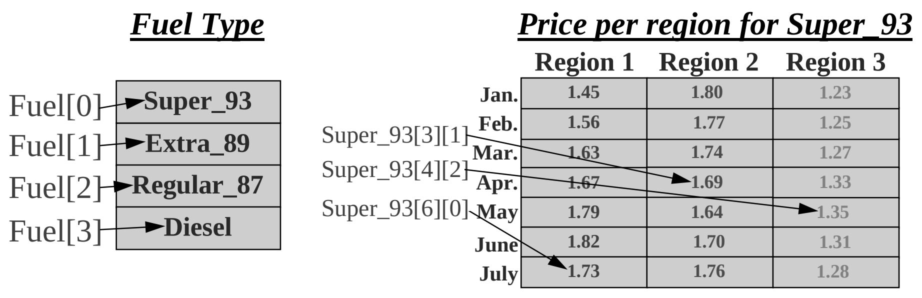
Arrays

A tag can also be defined as an array of elements. An array tag has the following format:

- **tagname[n]:** One-dimensional array (list of elements indexed in sequence)
- **tagname[n][n]:** Multi-dimensional array (matrix or table containing a fixed number of rows and columns. Each element in a two-dimensional array is distinguished by a pair of indexes. The first index gives the row and the second gives the column of the array the element is located in. A three-dimensional array is distinguished by three indexes, and so on.

The letter **n** is the number of elements in the array. The brackets **[ ]** around the array size are required when referencing an array tag.

The following figure illustrates single and multi-dimensional arrays and tag naming conventions used for each.



	Region 1	Region 2	Region 3
Jan.	1.45	1.80	1.23
Feb.	1.56	1.77	1.25
Mar.	1.63	1.74	1.27
Apr.	1.67	1.69	1.33
May	1.79	1.64	1.35
June	1.82	1.70	1.31
July	1.73	1.76	1.28

Single Dimensional (a list)

Multi-Dimensional (one or more tables)

An advantage of using arrays is that certain FactoryLink tasks, such as Math & Logic and Database Browser, can perform operations on an entire element array using only one reference to the array rather than using separate references to each element in the array.

Tag Structure

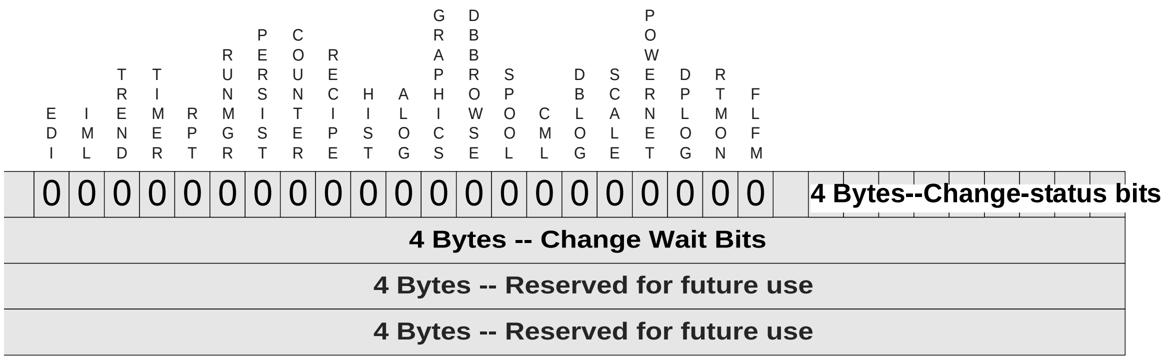
If it were possible to see a tag, you would see that it consists of a number of bits. Excluding the bits required for the tag's value, every tag, regardless of its data type, has the same basic bit structure.

Basic Tag Structure

All tags require 16 bytes just for the structure of the tag itself. The following illustration shows the basic bit structure for every tag in the real-time database.

The 16 bytes are designated as follows:

- 4 bytes that function as change-status bits
- 4 bytes that function as change-wait bits
- 8 bytes that are reserved for future use.



The change-status bits are very important to the FactoryLink system.

Change-Status Bits

The change-status bits enable FactoryLink to operate based on *exception processing*. Exception processing means that tasks do not access the database to read a tag's value UNLESS the tag's value has changed since the last time it was read.

Each FactoryLink task is assigned to a change-status bit. The task looks at the same bit in every tag that you define. (The diagram above is just an example. The order in which the tasks are assigned is irrelevant.) The value of the bit determines whether the tag's value has changed. The bit is set to either 1 (ON) or 0 (OFF). One (1) indicates to a task that the value of the tag has changed since the last time it was read, so the task will access the database again to read the tag's new value. Zero (0) indicates that the tag value has not changed since the last time it was read, so the task will not access the database unnecessarily to read a value that did not change.

When a task writes a value to a tag, FactoryLink automatically sets ALL the change-status bits to 1. When a task reads a tag, it sets just that task's individual change-status bit to 0. The change-status bits for tasks NOT configured to read a tag always have a value of 1; they do not affect other tasks or the application.

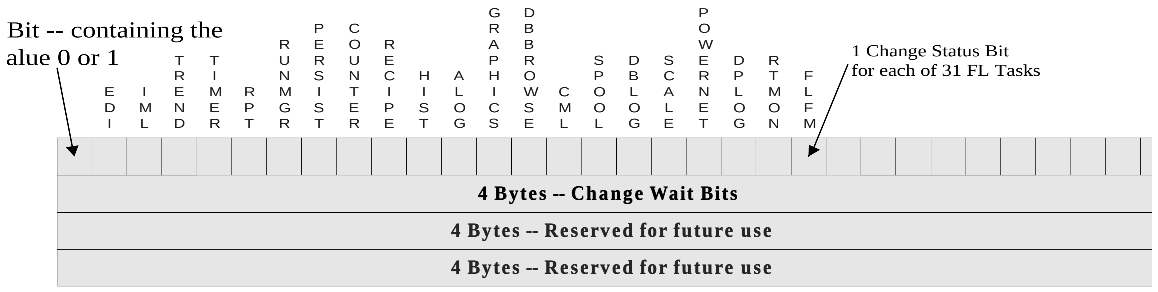
Start-up Values in Tags

Unless you define a default value for a tag, the start value of all the change-status bits is zero (0), indicating that no activity has taken place.

Digital Tag Structure

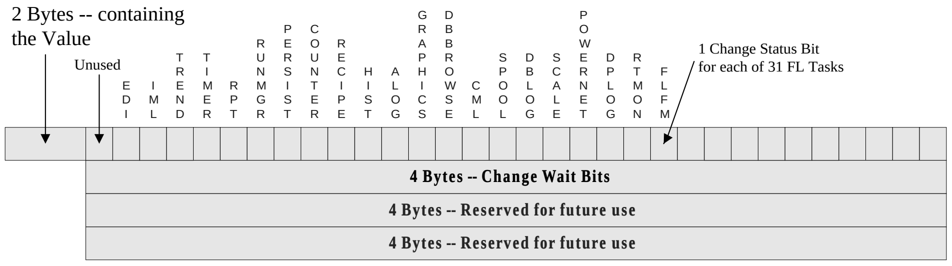
A digital tag contains 16 bytes:

- The first bit stores the tag value (either 0 or 1).
- The second bit through the thirty-second bit store the value for the change status of each task (31 tasks).
- The next four bytes are change-wait bits used to allow a task to "sleep" until a tag's value changes.
- The final eight bytes are reserved for future use.


Bit -- containing the alue 0 or 1
EDI
IML
TREND
TIMER
RPT
RUNMGR
PERSIST
COUNTER
RECIPE
HIST
ALOG
GRAPHICS
DBBROWSE
CML
SPOOL
DBLOG
SCALE
POWERNET
DPLOG
RTMON
FLFM
1 Change Status Bit for each of 31 FL Tasks
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use

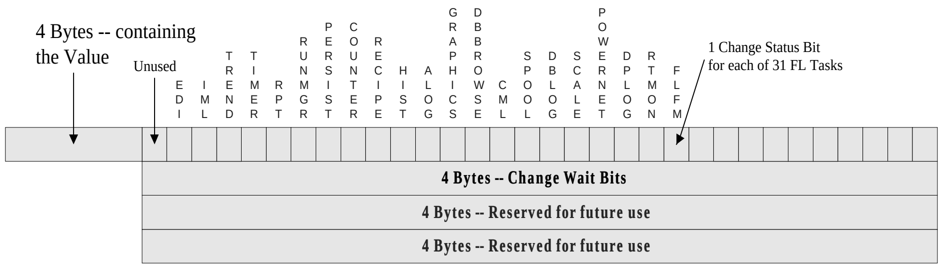
Analog Tag Structure

An analog tag contains 18 bytes. The structure of an analog tag is the same as that of a digital tag, but with two additional bytes to contain the value of the tag.


2 Bytes -- containing the Value
Unused
EDI
IML
TREND
TIMER
RPT
RUNMGR
PERSIST
COUNTER
RECIPE
HIST
ALOG
GRAPHICS
DBBROWSE
CML
SPOOL
DBLOG
SCALE
POWERNET
DPLOG
RTMON
FLFM
1 Change Status Bit for each of 31 FL Tasks
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use

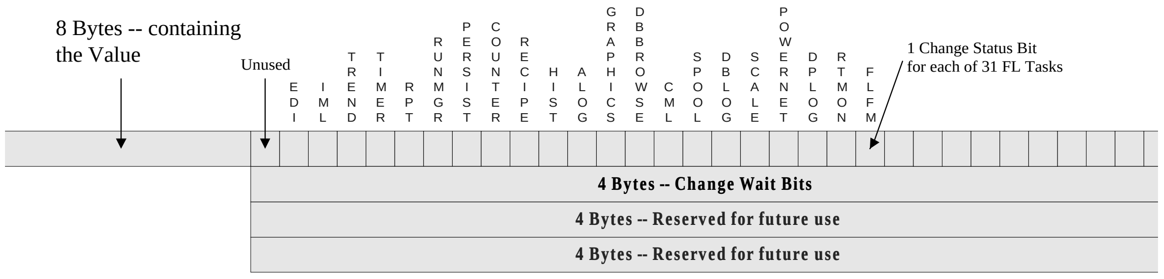
Long-Analog Tag Structure

A long-analog tag contains 20 bytes. The structure of a long-analog tag is the same as that of a digital tag, but with four additional bytes to contain the value of the tag.


4 Bytes -- containing the Value
Unused
EDI
IML
TREND
TIMER
RPT
RUNMGR
PERSIST
COUNTER
RECIPE
HIST
ALOG
GRAPHICS
DBBROWSE
CML
SPOOL
DBLOG
SCALE
POWERNET
DPLOG
RTMON
FLFM
1 Change Status Bit for each of 31 FL Tasks
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use

Floating-point Tag Structure

A floating-point tag is 24 bytes. The structure of a floating-point tag is the same as that of a digital tag, but with eight additional bytes to contain the value of the tag.


8 Bytes -- containing the Value
Unused
EDI
IML
TREND
TIMER
RPT
RUNMGR
PERSIST
COUNTER
RECIPE
HIST
ALOG
GRAPHICS
DBBROWSE
CML
SPOOL
DBLOG
SCALE
POWERNET
DPLOG
RTMON
FLFM
1 Change Status Bit for each of 31 FL Tasks
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use

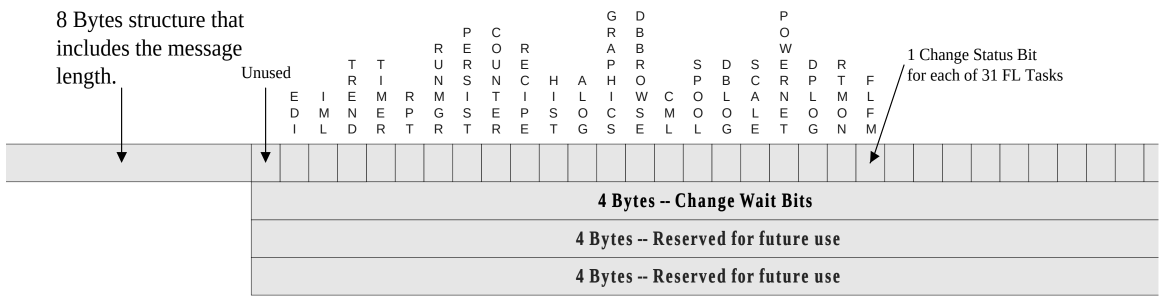
Message-Tag Structure

A message tag is 24 bytes plus bytes needed to store the message itself.

The structure of a message tag is the same as that of a digital tag, but with 8 additional bytes:

- 4 bytes to point to the memory location where the actual message is stored
- 2 bytes to specify the maximum message length
- 2 bytes to specify the current length of the message

plus the bytes required to store the message itself.


8 Bytes structure that includes the message length.
Unused
EDI
IML
TREND
TIMER
RPT
RUNMGR
PERSIST
COUNTER
RECIPE
HIST
ALOG
GRAPHICS
DBBROWSE
CML
SPOOL
DBLOG
SCALE
POWERNET
DPLOG
RTMON
FLFM
1 Change Status Bit for each of 31 FL Tasks
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use

Mailbox Tag Structure

Mailbox tags are different from other data types; their values do not get overwritten, because new values are appended (queued) to current values already stored.

A mailbox tag contains 24 bytes plus bytes for storing the length of the value and the number of values.

The 24 byte-structure of a mailbox tag is the same as the 24-byte structure of a message tag:

- The basic 16-byte tag structure
- 4 bytes point to the memory location where the actual message is stored
- 2 bytes specify the maximum message length
- 2 bytes specify the current length of the message

plus the bytes required to store the mailbox messages and the number of messages.

Real-Time Database Access: Reads and Writes

While knowledge of how tasks access the real-time database is not required to develop an application, it will help you understand what is going on as an application runs, and will help with troubleshooting.

Tasks access the real-time database to read values from tags and to write values to tags. When a task accesses the database, it is actually making a call to the FactoryLink *kernel*. The kernel is a part of the real-time database that manages the reads and writes to the database. Its function is transparent to you. There are two types of write calls and two types of read calls:

- Normal (conditional) write
- Forced (unconditional) write
- Change (conditional) read
- Normal (unconditional) read

You do not have to configure which write or read call a task makes. The type of call a task makes to the real-time database is programmed into the task; it is part of its code. The task makes the appropriate calls, depending on its purpose in life.

Normal (Conditional) Write

A normal write occurs ONLY if the new value being written to the tag is different from its current value. Thus, the term *conditional*: Is the new value different from the current value? The kernel checks this condition. If it is true, the kernel performs the write and sets the tag's change-status bits to 1; if not, the kernel does not perform the write, and the change-status bits remain 0.

For example, given the following information:

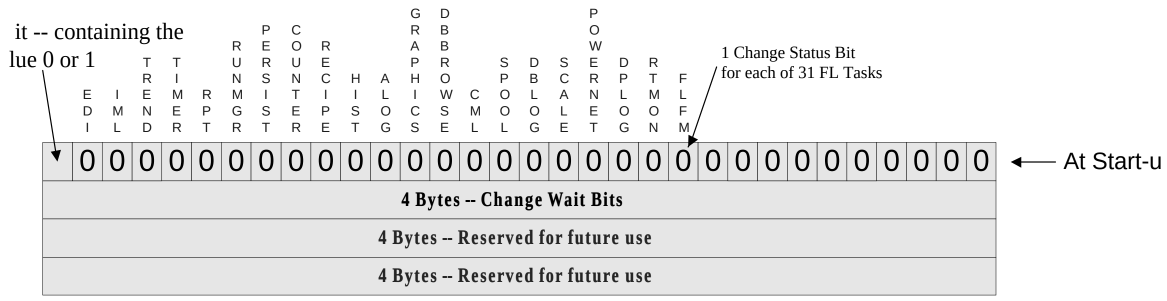
- A tag is named Tank1.
- Its current value is 10.
- The Interpreted Math & Logic task (IML) is configured to perform a calculation and assign the result to Tank1.

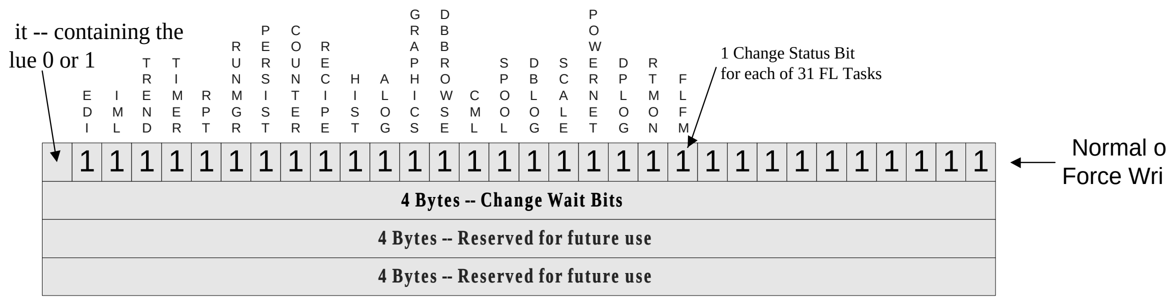
Example 1: IML performs its calculation and the result is 10. IML makes the normal write call to the database and the kernel compares the current value of Tank1 with the new value from IML. The current value is 10 and the new value is also 10. Because the old and new values are the same, the kernel does not continue with the write, and the current value remains in Tank1. Because the kernel does not perform the write, the change-status bits for Tank1 remain zero. Therefore, no tasks are notified to perform a read of Tank1.

Example 2: IML performs its calculation and the result is 20. IML makes the normal write call to the database, and the kernel compares the current value of Tank1, 10, with the new value from IML, 20. Because the old and new values are different, the kernel continues with the write and writes 20 to Tank1. Because the kernel performed the write, it sets the change-status bits for Tank1 to 1. Therefore, the kernel notifies any tasks waiting for change on Tank1 that they should read the new value.

Forced (Unconditional) Write

Unlike the normal write, the forced write updates the value, whether or not it is different from the old value. Thus, the term *unconditional:* regardless of the tag's status, its change-status bits are forced to 1.


it -- containing the
lue 0 or 1
EDI IML TREND TIMER RPT RUNMGR PERSIST COUNTER RECIPE HIST ALOG GRAPHICS DBBROWSE CML SPOOL DBLOG SCALE POWERNET DPLOG RTMON FLFM
1 Change Status Bit for each of 31 FL Tasks
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
At Start-u
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use


it -- containing the
lue 0 or 1
EDI IML TREND TIMER RPT RUNMGR PERSIST COUNTER RECIPE HIST ALOG GRAPHICS DBBROWSE CML SPOOL DBLOG SCALE POWERNET DPLOG RTMON FLFM
1 Change Status Bit for each of 31 FL Tasks
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Normal o
Force Wri
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use

Using the previous example:

- A tag named Tank1
- Its current value is 10.
- The Interpreted Math & Logic task (IML) is configured to perform a calculation and assign the result to Tank1.

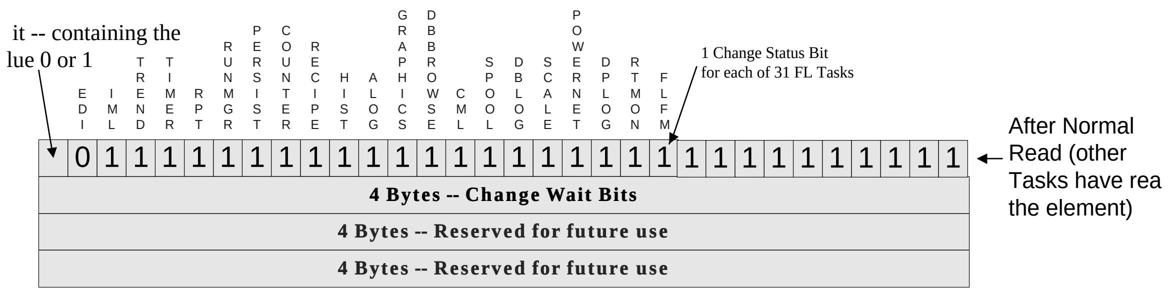
IML performs its calculation and the result is 10. IML makes the forced-write call to the database. The kernel writes the new value of 10 to Tank1. Because the kernel performed the write, it sets the change-status bits for Tank1 to 1. Therefore, the kernel notifies any tasks waiting for change on Tank1 that they should read the "new" value.

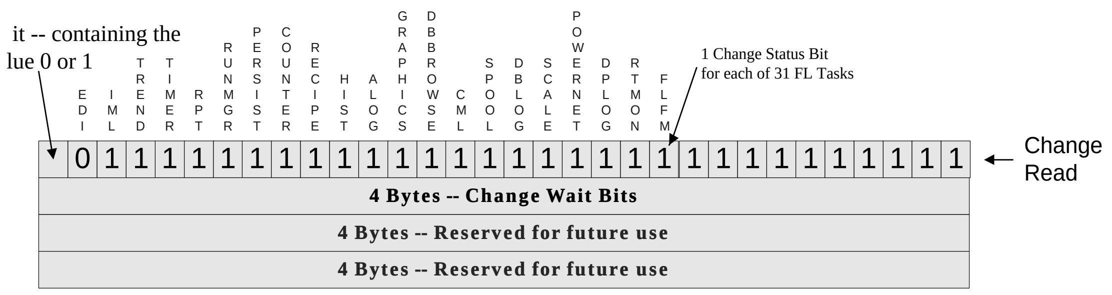
Change (Conditional) Read

A change-read call returns the value ONLY if the data has changed. The condition is: Are the change-status bits for the tag set to 1? If the change-status bits are 1, then the value must be different from the last read, so the kernel will notify waiting tasks to read it again. This type of read significantly optimizes performance.

Normal (Unconditional) Read

An unconditional-read call returns the value, regardless of whether the value has changed since the last read. It is a “forced read.”


it -- containing the lue 0 or 1
EDI
IML
TREND
TIMER
RPT
RUNMGR
PERSIST
COUNTER
RECIPE
HIST
ALOG
GRAPHICS
DBBROWSE
CML
SPOOL
DBLOG
SCALE
POWERNET
DPLOG
RTMON
FLFM
1 Change Status Bit for each of 31 FL Tasks
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
4 Bytes -- Change Wait Bits
4 Bytes -- Reserved for future use
4 Bytes -- Reserved for future use
After Normal Read (other Tasks have rea the element)



FactoryLink Triggers

The term *trigger* refers to a tag whose change in value causes another event to occur in the application. For example, a message tag **station_name** stores the name of a workstation. You could have Math & Logic run a script every time you change the name of the workstation. What happens when you enter a new workstation name? The kernel sets all the change-status bits for **station_name** to 1. What happens when change-status bits go to 1? The kernel notifies waiting tasks that their bit is set, and the tasks then read the new value and perform their job. In this example, Math & Logic runs its script. What you are saying, then, is that **station_name** triggers Math & Logic to run your script. So, for a tag to act as a trigger, its change-status bits must be set to 1. Any tag type can act as a trigger.

However, for digital tags to act as triggers, two conditions are required:

- The change-status bits must be 1, AND
- The tag’s value must be 1.

Remember that digital tags are either 1 or 0; ON or OFF. The reason for the two conditions above is that the change-status bits themselves, not the values of 1 or 0, function as the ON/OFF switch.

For example, given a digital tag at startup with its value as 0 and its change-status bits 0. You write a 1 to the tag. Writing a new value sets all the change-status bits to 1. Both conditions are true, so the tag is considered ON. Any task that is configured to read this digital tag is notified by the kernel to do so. Now, you want to turn it OFF. You do not write a 0 to the tag, because the change-status bits turn the tag OFF for you.

After a task reads the value of 1, it set its own change-status bit to 0. Both conditions are no longer true. The tag’s value is still 1; it never changed. But the reading task’s change- status bit is now 0. So, **for the reading task**, both conditions are no longer true, and the tag is considered OFF. Note that the tag is still ON for any other task that is configured to read it but has not done so.

Given that you do not write a 0 to a digital tag to turn it OFF, then its value remains a 1. So how do you turn it ON again if its value is already 1? Use the force-write call to write a 1 back into the tag again. A force-write call writes a 1 to the tag and forces the change-status bits to 1. Both conditions are now true again, so the tag is considered ON again.

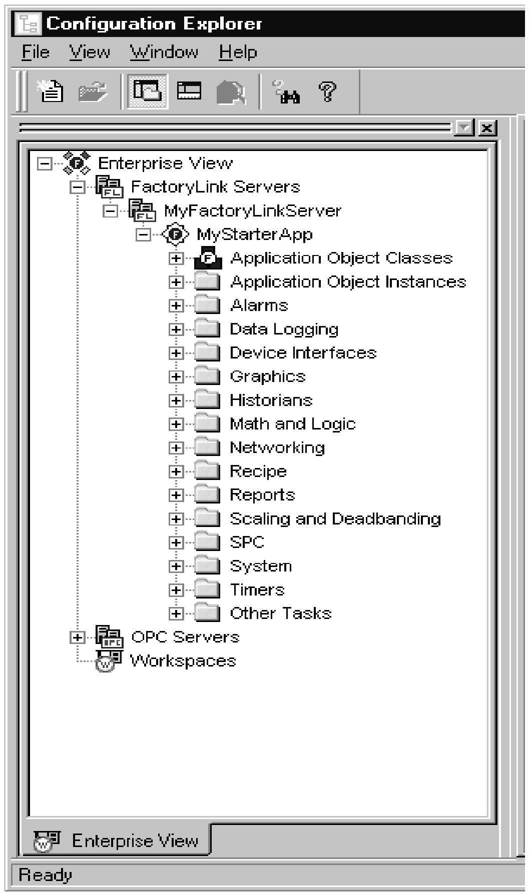
In general, the following rules apply for digital tags:

- A digital is ON when its value is 1 and its change-status bits are set to 1.
- Once the value is written to 1, it remains 1 for the duration of the application (unless you purposely write a 0 to it).
- The change-status bit, not the tag value, serves as the ON/OFF switch for the tag.
- A digital value is OFF for a task when the change-status bit for that task is set to 0 (when the task completes its read of the value).
- Use a forced write to turn a digital ON again.

FACTORYLINK TOOLS

FactoryLink includes two application-development tools: Configuration Explorer and Client Builder. Both reside on the FactoryLink client computer.

Configuration Explorer

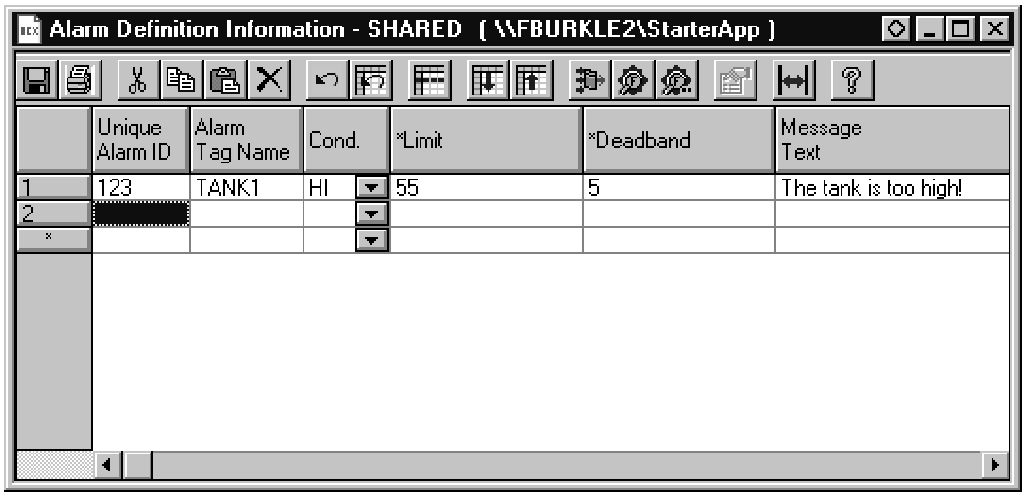


Configuration Explorer provides access to the FactoryLink tasks in a tree view much like the Windows Explorer. You use it to create the server application.

Using Configuration Explorer, you configure a task to make it part of the application. Each task has a specific job to do, such as logging data to a database, reading from or writing to a PLC or other external device, alarming, etc. A task must be configured before it will run.

You configure a task by completing its associated configuration table(s).

Using the Alarm task as an example, the diagram below shows a sample table.


Alarm Definition Information - SHARED (\\FBURKLE2\StarterApp)

	Unique Alarm ID	Alarm Tag Name	Cond.	*Limit	*Deadband	Message Text
1	123	TANK1	HI	55	5	The tank is too high!
2						
*						

Given that the general purpose of the Alarm task is to monitor values in the real-time database, compare these values against the criteria defined by the user in the Alarm table, and display an alarm message if the real-time database values meet the criteria; this sample configuration tells the Alarm task to do the following:

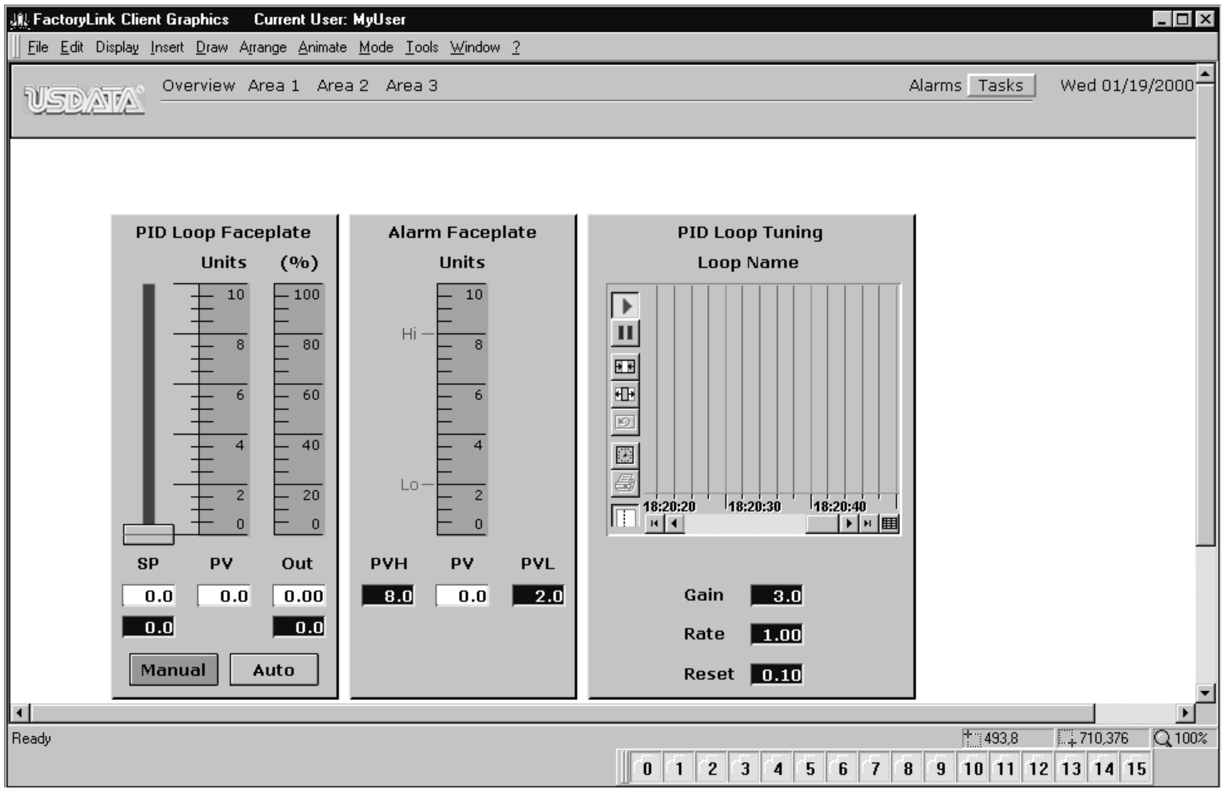
“Read the value of TANK1 and compare it to the alarm condition ‘greater than 55.’ If TANK1’s value is greater than 55, display the alarm message ‘The tank is too high!’ on the application screen.”

This is a simple example, but the idea is that a task will do nothing unless it has been told to do so by its configuration table. Every task has at least one configuration table, depending on the job the task performs in an application.

Configuration Explorer, including its ease-of-use features and the configuration table structure, is discussed in detail in the *FactoryLink Configuration Explorer User Guide*. Individual tasks are discussed in the *FactoryLink Task Configuration Reference*.

Client Builder

Client Builder is the tool you use to create the user interface for your application directory on the server. It is where you develop the graphic screens you see when you run the application. A sample application screen appears below:



Client Builder exchanges data with the server through its OPC connection. FactoryLink's starter application files provide you with many default screens that help you learn about the features in Client Builder, as well as many libraries of images to help you quickly create good-looking, user-friendly screens.

Chapter 2
Starter Application

During the FactoryLink 7.0 installation process, you have an option to install the Starter Application, which USDATA highly recommends. It is a baseline application with functionality common to most FactoryLink applications. Using the Starter Application, users can modify the existing components to meet their needs, therefore reducing the time and effort required to configure a new application.

Starter Application Overview

The FactoryLink Starter Application consists of two sets of components:

- A server application with an Application Object database and source file examples
- A Client Builder project

Server Application

The Server Application consists of a set of preconfigured tasks that are commonly used in industry, such as Alarm Logger, Database Logger, Historian, OPC Server, Math and Logic. These tasks provide examples of logging data for alarm, trend, and browse, as well as generate real-time data for graphic animation display. The data is accessed by the Client Builder graphics using the Alarm Server and the OPC Server running within the FactoryLink application and the Trend Server and Database Browser Control running from Client Builder.

Application Objects

The Application Objects configured in the **AOInstance.mdb** database are collections of Template Variables, Configuration Objects, and File Objects based on common FactoryLink task configuration tables. These Application Objects use the example input source files located in the **{FLAPP}\AppObj** directory to dynamically populate one or more FactoryLink task configuration tables.

As an Application Object is copied to an application, each Application Object's instance is written to the Instances database file (**AOInstance.mdb**). On successful completion of each instance, the configuration table's database record is written to that table's database (*.cdb) file.

Client Builder Project

The Client Builder project consists of three sets of screens that demonstrate each of the major functions that are available in the graphical development software.

- **The first set** demonstrates real-time data display for alarming, trending, and browsing, using the developed ActiveX components.
- **The second set** demonstrates standard animation features commonly used for color control, value input and output, and image display.
- **The third set** demonstrates advanced features used to add movement, visibility, script, and objects to the graphics.

Note: The FacotryLink Client installation installs a set of clip art images in the **USDATA\Client Builder\Shared Libraries** directory. To use the clip art graphics, you need to copy them into your new Client Builder project.

APPLICATION OBJECTS

The Application Objects examples are installed in Configuration Explorer. Each Application Object is a collection of template variables, configuration objects and file objects that is specifically configured to populate one or more configuration tables within the FactoryLink Server.

There are two main advantages for using Application Objects:

- Increased development productivity
  - Simultaneous development by users with different FactoryLink skill levels
  - Reusability of common variables and objects
  - Structured configuration methodology
- Easier application maintenance
  - Organized view of records grouped by function
  - Able to add, insert, and remove instances per changing specifications
  - Able to recalculate all the instances of an Application Object from revised data source files

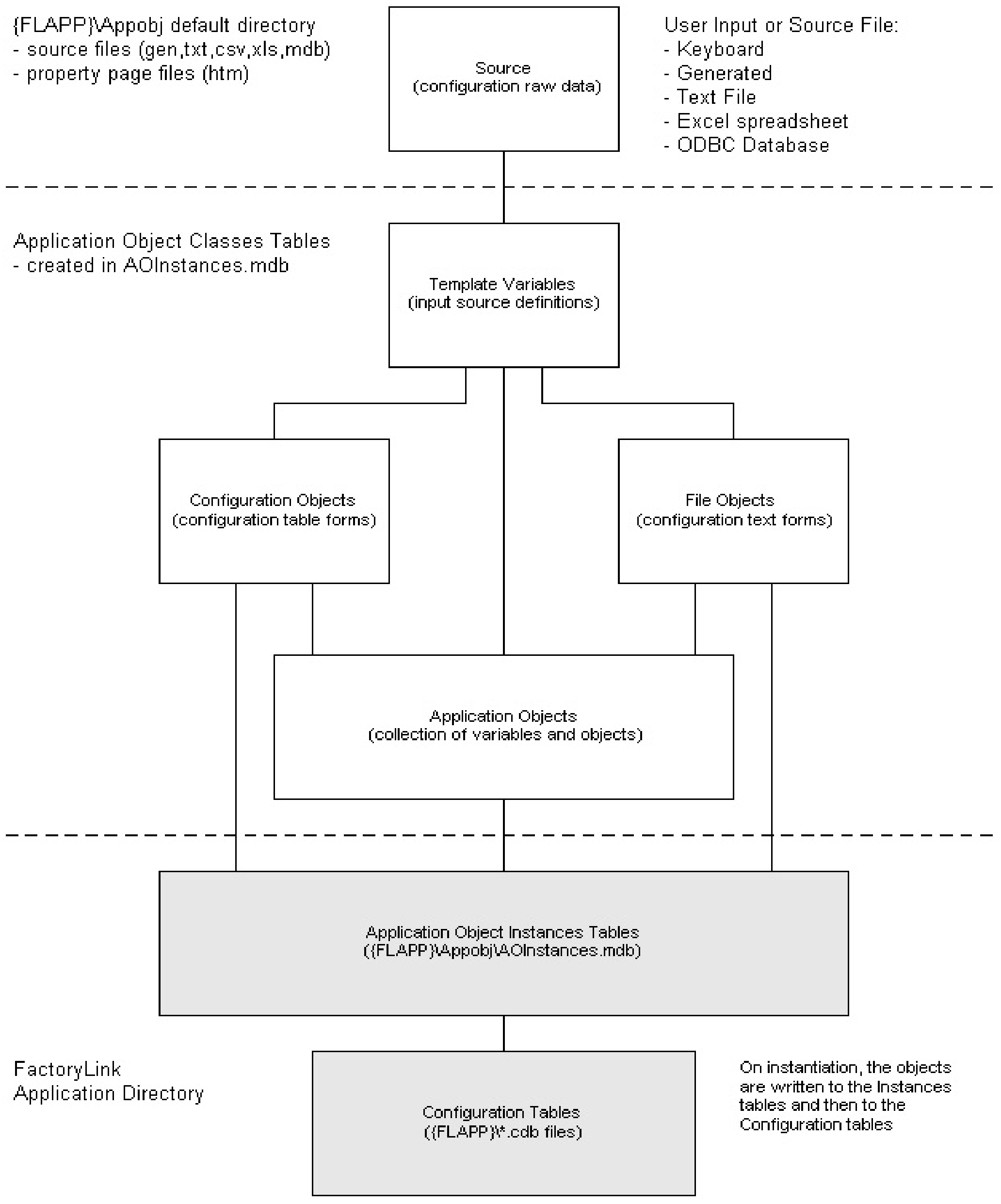
Application Object Overview Diagram

Figure 2-1 illustrates how data flows from the developer’s raw information to the configuration tables. There are three sections in this process:

- Where does the data come from?
  - Raw configuration data can be a user input or a source file.
  - User input can be entered by keyboard or by record generator panel.
  - Source file data types can be a text file, spreadsheet, or database.
- How the classes are defined?
  - Template variables define the input or source parameters.
  - Template variables can be used in configuration objects, file objects and Application Objects.
  - Configuration objects define the template variables or constants used in the configuration forms.
  - File objects define the template variables or text used in the text forms.
  - Application Objects define the collection of template variables and objects used to represent a functional equipment object.

- How are the Application Objects instantiated into configuration tables?
  - Application objects are instantiated using a copy and paste method from the Classes tables to the Instances tables within the Application Objects database.
  - Application Object Instances tables within the database are used to record and maintain the location of the instantiated objects within the FactoryLink application.
  - Each successful instantiation is then written as a record to the FactoryLink application configuration tables and files.

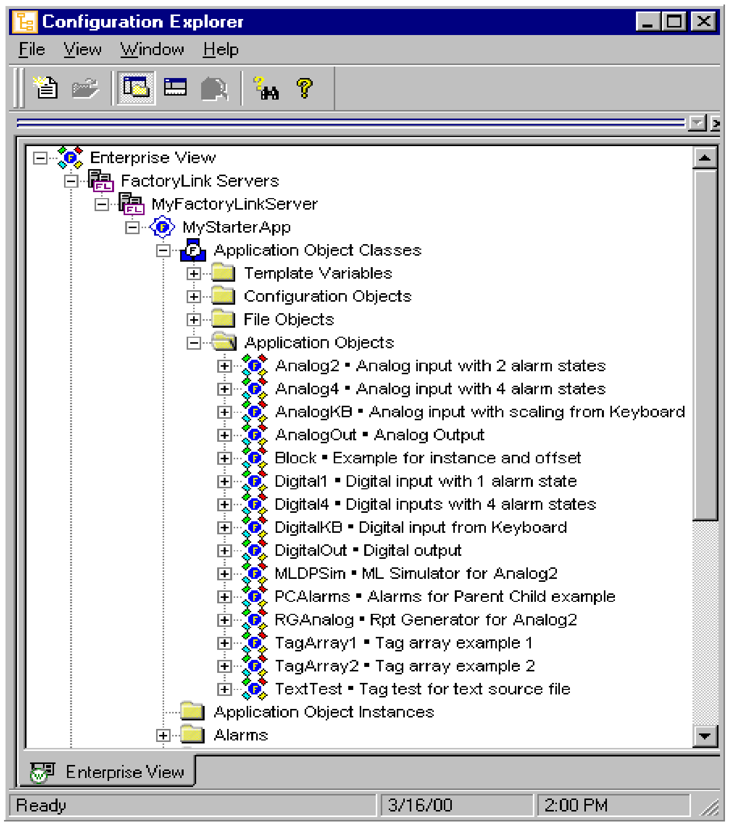
Figure 2-1 Application Objects Overview Diagram


{FLAPP}\Appobj default directory
- source files (gen,txt,csv,xls,mdb)
- property page files (htm)
Source
(configuration raw data)
User Input or Source File:
- Keyboard
- Generated
- Text File
- Excel spreadsheet
- ODBC Database
Application Object Classes Tables
- created in AOInstances.mdb
Template Variables
(input source definitions)
Configuration Objects
(configuration table forms)
File Objects
(configuration text forms)
Application Objects
(collection of variables and objects)
Application Object Instances Tables
({FLAPP}\Appobj\AOInstances.mdb)
FactoryLink
Application Directory
Configuration Tables
({FLAPP}*.cdb files)
On instantiation, the objects are written to the Instances tables and then to the Configuration tables

Examples of Application Objects

The Application Objects in the Classes folder are configured as examples of a typical function that would be required in a FactoryLink development project (see Figure 2-2). Each of these objects is described in general in the following subsections.

Figure 2-2 Application Objects in Configuration Explorer


Configuration Explorer
File View Window Help
Enterprise View
FactoryLink Servers
MyFactoryLinkServer
MyStarterApp
Application Object Classes
Template Variables
Configuration Objects
File Objects
Application Objects
Analog2 • Analog input with 2 alarm states
Analog4 • Analog input with 4 alarm states
AnalogKB • Analog input with scaling from Keyboard
AnalogOut • Analog Output
Block • Example for instance and offset
Digital1 • Digital input with 1 alarm state
Digital4 • Digital inputs with 4 alarm states
DigitalKB • Digital input from Keyboard
DigitalOut • Digital output
MLDPSim • ML Simulator for Analog2
PCAlarms • Alarms for Parent Child example
RGAnalog • Rpt Generator for Analog2
TagArray1 • Tag array example 1
TagArray2 • Tag array example 2
TextTest • Tag test for text source file
Application Object Instances
Alarms
Enterprise View
Ready
3/16/00
2:00 PM

Analog2 - Analog Input with Two Alarm States

Analog2 configures an analog input from an Excel spreadsheet in these task functions: read from an OPC server, scale to engineering units, log to a database, and alarm on high and low limits.

Analog4 - Analog Input with Four Alarm States

Analog4 configures an analog input from an Excel spreadsheet in these task functions: read from an OPC server, scale to engineering units, log to a database, and alarm on high-high, high, low, and low-low limits.

AnalogKB - Analog Input with Scaling from Keyboard Entry

AnalogKB configures an analog input from a keyboard manual entry in these task functions: read from an OPC server and scale to engineering units.

AnalogOut - Analog Output for Write

AnalogOut configures an analog output from an Excel spreadsheet for write to an OPC server.

Block - Example for Instance and Offset

Block configures a block of related tags from an Excel spreadsheet for Math and Logic variables.

Digital1 - Digital Input with One Alarm State

Digital1 configures a digital input from an Excel spreadsheet in these task functions: read from an OPC server and alarm for an ON condition.

Digital4 - Digital Inputs with Four Alarm States

Digital4 configures two digital inputs from an Excel spreadsheet in these task functions: read from an OPC server, calculate the four analog states of two digital inputs using Math and Logic, and alarm for the four analog conditions.

DigitalKB - Digital Input from Keyboard Entry

DigitalKB configures a digital input from a keyboard manual entry for read from an OPC server.

DigitalOut - Digital Output for Write

DigitalOutput configures a digital output from an Excel spreadsheet for write to an OPC server.

MLDPSim - Math and Logic Simulator for Analog2

MLDPSim configures tags from an Excel spreadsheet for Math and Logic trigger, variables and program for simulation of the analog inputs for the prior Analog2 object.

PCAlarms - Alarms for Parent Child Example

PCAlarms configures tags from an ODBC database for alarming with a Parent-Child dependency relationship.

RGAnalog - Report Generator for Analog2

RGAnalog configures tags from an Excel spreadsheet for a report generation of the analog inputs for the prior Analog2 object.

TagArray1 - Tag Array Example 1

TagArray1 configures array tags using one column for “tags[index]” from an Excel spreadsheet for Math and Logic variables.

TagArray2 - Tag Array Example 2

TagArray2 configures array tags using two columns for “tags” and “index” from an Excel spreadsheet for Math and Logic variables.

TextTest - Tag Test for Text Source File

TextTest configures tags from two types of text file extensions (Txt, Csv) for Math and Logic variables.

CLIENT BUILDER STARTER PROJECT GRAPHICS

You can view the Client Builder Starter Project examples by navigating the graphic template's menu bar. These examples are grouped under each of the area headings on the menu bar, where Area1 contains the common ActiveX controls, Area2 contains the standard animation functions, and Area3 contains the advanced animation features (see Figure 2-3).

Figure 2-3 USDATA Logo Graphic

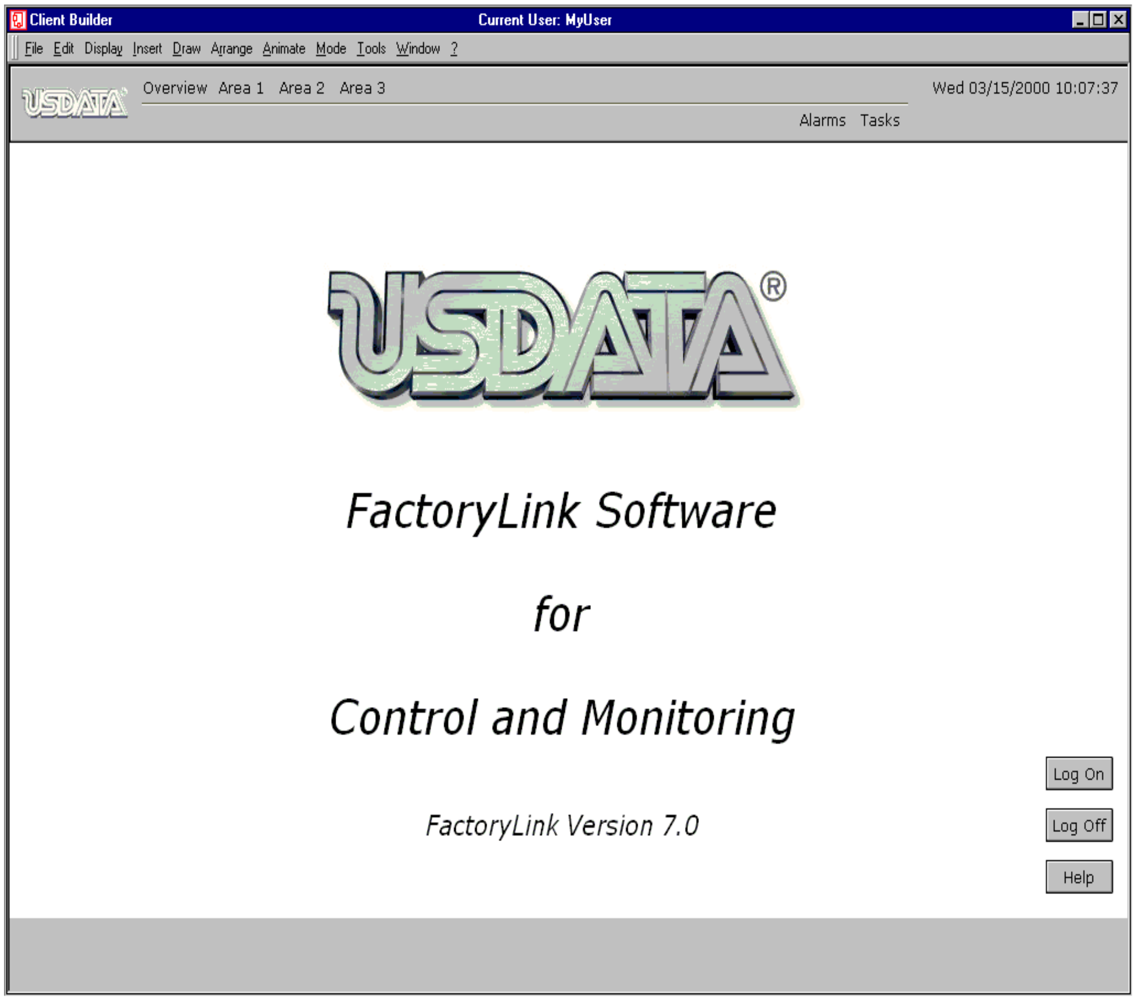
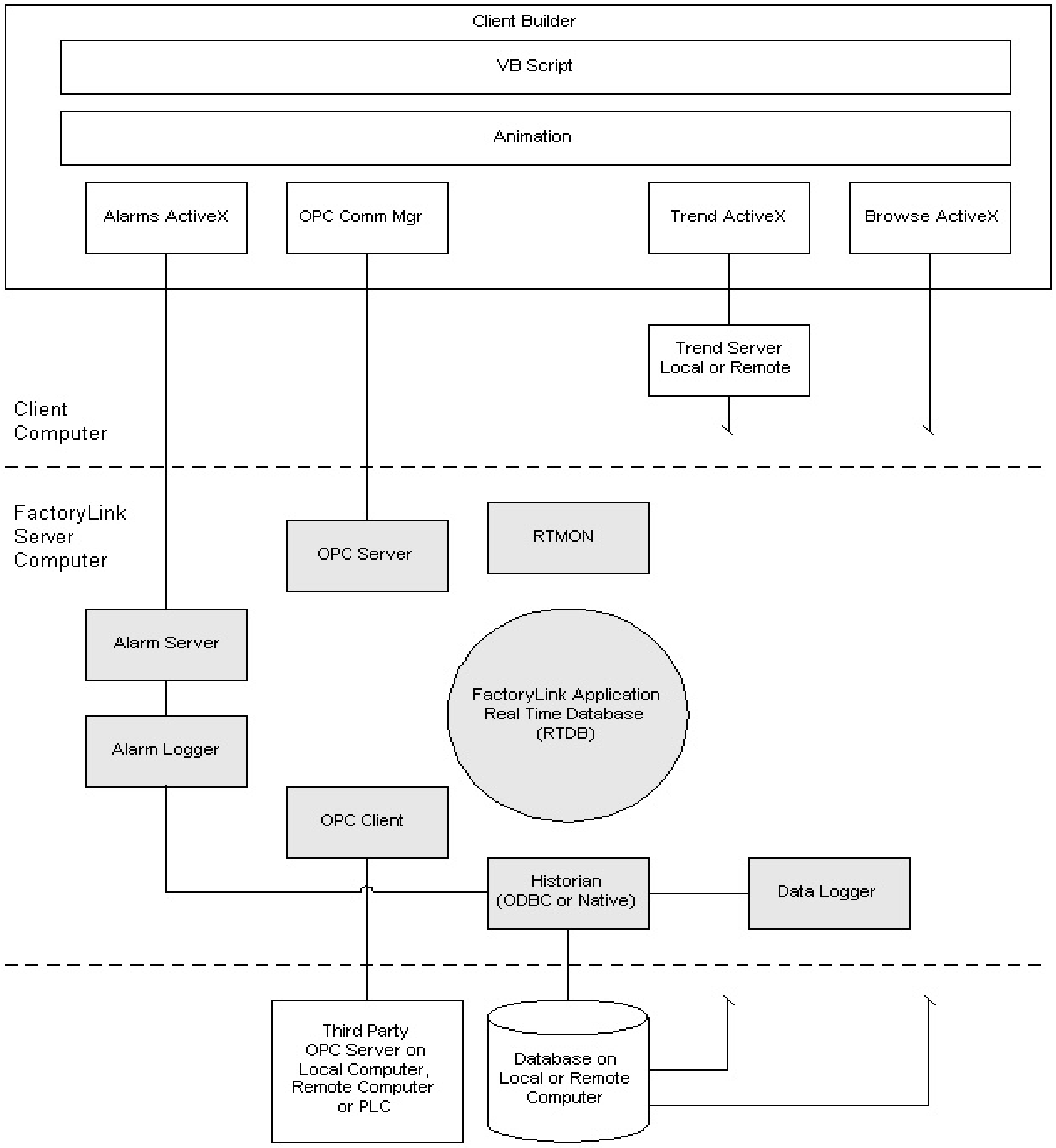


Figure 2-4 depicts the communication connections between the major components of the Client Builder Project, the FactoryLink Server, and the Database Server.

Figure 2-4 FactoryLink Components Communication Diagram


Client Builder
VB Script
Animation
Alarms ActiveX
OPC Comm Mgr
Trend ActiveX
Browse ActiveX
Trend Server Local or Remote
Client Computer
FactoryLink Server Computer
OPC Server
RTMON
Alarm Server
FactoryLink Application Real Time Database (RTDB)
Alarm Logger
OPC Client
Historian (ODBC or Native)
Data Logger
Third Party OPC Server on Local Computer, Remote Computer or PLC
Database on Local or Remote Computer

Animation Real-time Display

With the FactoryLink Starter Application running, the Client Builder Project Graphics connect to the FactoryLink Real-time Database (RTDB) using the OPC Communication Manager's client connection to the FactoryLink OPC Server. This communication allows any of the FactoryLink tag values to be read and written from the graphics. The standard and advanced animation features use those tag values as well as the graphics' local register variables for real-time display and control of the FactoryLink Server. The internal VB Script editor can be used to write custom code using each graphic object's exposed properties, methods, and events for control beyond the animation features and functions.

ActiveX Controls

There are three ActiveX controls in Client Builder:

- Alarms
- Trend
- Browser

For detailed information on the three ActiveX controls, refer to the *FactoryLink Task Configuration Reference Guide*.

Alarms

The Alarm Viewer Control is used to configure, display, sort, filter, and acknowledge the active alarms from the FactoryLink Server as communicated through the Alarm Server. The Alarm Viewer properties can be accessed for configuring the parameters for general control, colors/fonts, groups, and fields. The Alarm Logger uses an ODBC Historian task to write the alarm data to either a dBase IV or SQL Server database selected during installation.

Trend

The Trend Control is used to configure, display, and select data as communicated through the Trend Server from values logged to a database using the Data Logger tasks. Using this database method, the Trend Control can display the data in either a real-time or historical mode. The Trend Control properties can be accessed for configuring the parameters for graph, pens, and fonts. Within that panel, a Trend Editor is accessed for pen assignments to the database tables and columns. The Trend Server is used so a single Trend Control can connect to multiple databases or tables at the same time. The Trend Server makes a direct connection to the databases using an ODBC data source configuration.

Browse

The Browse Control is used to configure, display, and select data from values logged to the database using the Data Logger tasks. The Browse Control properties can be accessed for configuring the parameters for general control, database sources, columns, select statement, and sort order. The Browse Control makes a direct connection to a database using an ODBC data source configuration.

Standard Animation Features

The following types of animation are used for most of the graphic functions. Certain combinations of animation can be applied on a single object so that object could serve multiple purposes.

- **Color animation** color-fills any drawn object, bargraph, or legend by using a bit or numeric register value.
- **Text animation** displays messages, labels, or numeric values.
- **Symbol animation** selects predrawn objects from a library by using a bit or numeric register value.
- **Send animation** writes values to bit, numeric, or text registers.
- **Run animation** aunches an external program.
- **Link animation** opens or closes graphics, connects to a document or URL, and views or edits a text file as a note.

Advanced Animation Features

The following advanced animation features can be used to add movement, visibility, script, and symbols to the graphics:

- **Layers** are different sheets on the same graphic used to group objects that can be seen when the Layer Toolbar is used to select a layer number, or the Layers value is set by script.
- **Visibility bound** is an object property that defines object visibility based on zoom range.
- **Rotation** is an animation function that rotates an object through an angle range relative to a register value range.
- **Scaling** is an animation function that scales an object through a percentage range relative to a register value range.
- **One-axis positioning** is an animation function that moves an object relative to its original position in X and/or Y weighted directions using one register value linked to both axes.
- **Two-axis positioning** is an animation function that moves an object relative to its original position in X and Y weighted directions using a separate register value for each axis.

- **Free positioning** is an animation function that moves an object to absolute coordinates in X and Y directions using a separate register value for each axis.
- **VB script** is a single object or a grouped object script file that executes event-based subroutines.
- **Function objects** are examples of library symbols that can be created with local register variables for substitution with tags after inserting the symbol.

USING THE STARTER APPLICATION

This section discusses several issues on how to run the Starter Application:

- Computer location
- Help file
- Running the Starter Application

Computer Location

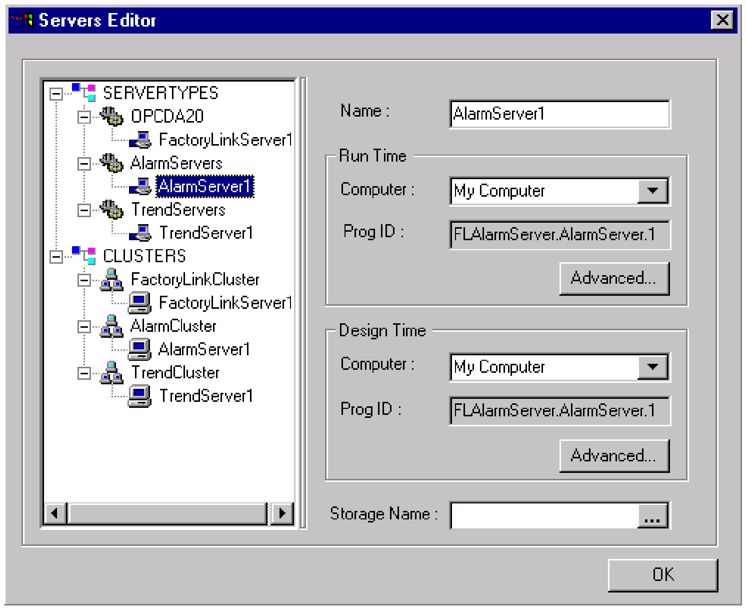
The Starter Application, by design, runs on a computer where both the server and the client are installed. However, for the Starter Application files to work with a server install on the local machine and the client install on a remote machine, you must change the information regarding the computer location in two places.

Perform the following steps to change the computer location information:

1. Double-click the Client Builder icon from your desktop. Open the **Starter** project in the **Open Project** dialog box.
2. Select the **Starter.fvp** file.
3. From the **Tools** menu, select **Servers** to open the **Servers Editor** dialog box.
4. Point the OPC and Alarm servers to the server computer name for Run Time or Design Time. The Communication Manager computer should point to the local host, **MyComputer** (see Figure 2-5**)**.

   The Trend Server should point to the local host so it will always start on the Client Builder's computer. You can do this from any remote computer's Client Builder program, referencing the server's Starter Project.

Figure 2-5 Servers Editor


Servers Editor

- SERVERTYPES
  - OPCDA20
    - FactoryLinkServer1
  - AlarmServers
    - AlarmServer1
  - TrendServers
    - TrendServer1
- CLUSTERS
  - FactoryLinkCluster
    - FactoryLinkServer1
  - AlarmCluster
    - AlarmServer1
  - TrendCluster
    - TrendServer1

Name : AlarmServer1

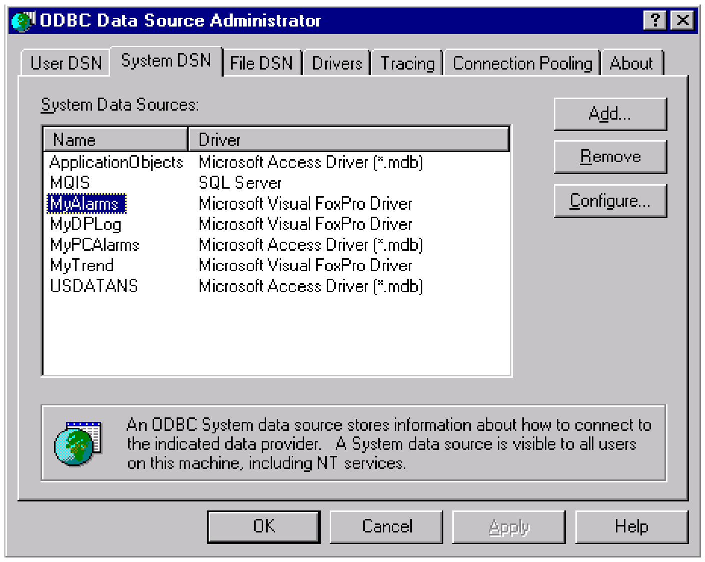
Run Time
Computer : My Computer
Prog ID : FLAlarmServer.AlarmServer.1

Design Time
Computer : My Computer
Prog ID : FLAlarmServer.AlarmServer.1

Storage Name :

5 Configure the remote computer’s ODBC System Data Source Names (DSNs) to point at the server’s database location (see Figure 2-6).

Figure 2-6 ODBC Data Source Administrator



System Data Sources:

Name	Driver
ApplicationObjects	Microsoft Access Driver (*.mdb)
MQIS	SQL Server
MyAlarms	Microsoft Visual FoxPro Driver
MyDPLog	Microsoft Visual FoxPro Driver
MyPCAlarms	Microsoft Access Driver (*.mdb)
MyTrend	Microsoft Visual FoxPro Driver
USDATANS	Microsoft Access Driver (*.mdb)

An ODBC System data source stores information about how to connect to the indicated data provider. A System data source is visible to all users on this machine, including NT services.

Help File

The Starter Application Help File, **USDATA\Client Builder\Project\Starter\Help Files\Starter.txt**, covers the basic requirements for configuring and running the application. Please review this file for any final notes that did not make it to this chapter. You can also access this file by clicking the **Help** button on the Client Builder Starter Project logo screen.

The Help File covers the following topics:

- Configuration steps
  - Set up the database
    - SQL Sever Flink database
    - dBase IV Flink database
    - Application Objects database
  - Set up and start the FactoryLink Server
  - Set up and start the Client Builder graphics
- Using the Application Objects
- Save and restore

Running the Starter Application

The FactoryLink installation program installs all the Starter Application components that are pre-configured. To view how the Starter Application operates, perform the following steps:

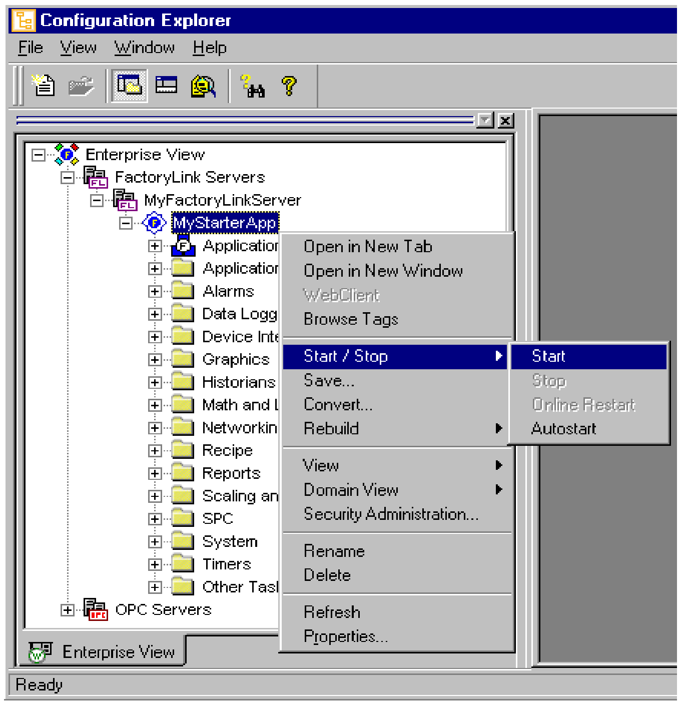
1 Start the Server Application “**MyStarterApp**” in Configuration Explorer.

2 Launch the Starter Project in Client Builder.

Step I: Start the Server Application

In the Configuration Explorer’s Enterprise View, right-click “MyStarterApp” under FactoryLink Application and select **Start/Stop>Start** in the pop-up menu (see Figure 2-7).

Figure 2-7 Configuration Explorer

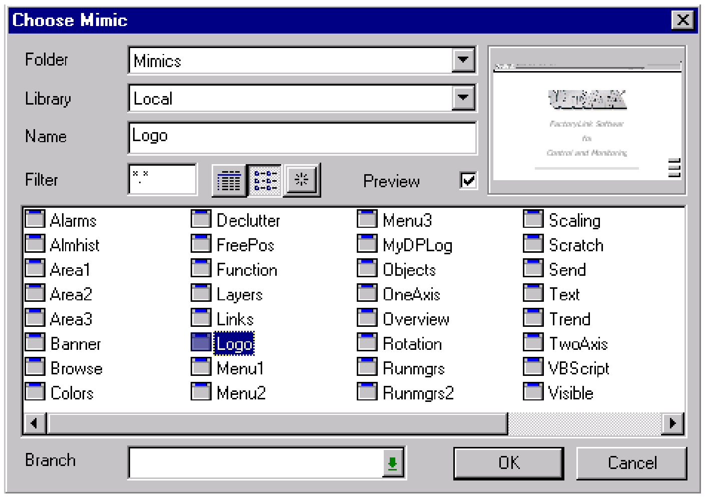


This application is configured to run only the Server tasks, which are the same as the legacy SHARED domain tasks. The Real-time Database Monitor (RTMON) is also started for debug using the tag input, the watch list, or the command input windows.

Step 2: Launch the Client Project

Double-click the Client Builder icon on your desktop, select the **Starter.fvp** file in the **Starter** project, and open the **Logo** graphic or mimic (see Figure 2-8).

Figure 2-8 Choose Mimic



Optional methods to launch the Starter Project are to configure a copy of the default Client Builder icon with the Logo graphic as the default mimic using the icon's shortcut properties per the following examples (see Figure 2-9 and Figure 2-10).

Option 1 (with menus)

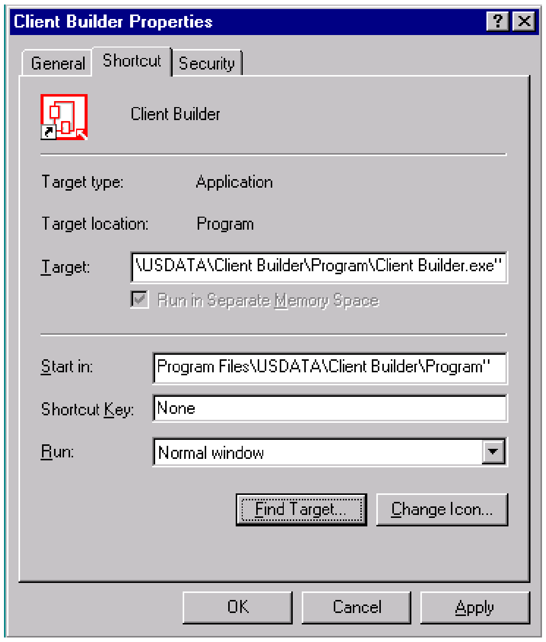
Target = "...\Client Builder\Program\ClientBuilder.exe" <space>

..\Project\Starter\Starter.fvp <space> /OpenWindow:"Logo"

Start In = "...\Client Builder\Program"

> **Note:** The two dots in front of "\Project" refers to the parent directory above Project.

Figure 2-9 Client Builder Properties



Client Builder Properties

General | Shortcut | Security

Client Builder

Target type: Application

Target location: Program

Target: \USDATA\Client Builder\Program\Client Builder.exe"

[✓] Run in Separate Memory Space

Start in: Program Files\USDATA\Client Builder\Program"

Shortcut Key: None

Run: Normal window

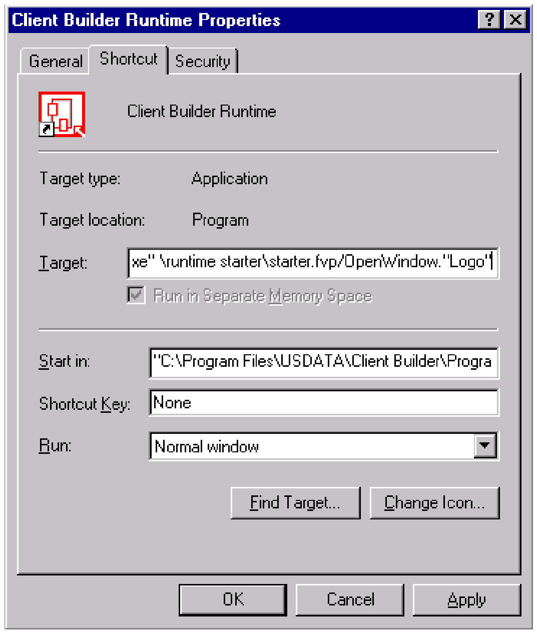
Option 2 (without menus)

Target = "...\Client Builder\Program\ClientBuilder.exe" <space>

/Runtime <space> ..\Project\Starter\Starter.fvp <space> /OpenWindow:"Logo"

Start In = "...\Client Builder\Program"

Figure 2-10 Client Builder Runtime Properties



Target type: Application

Target location: Program

Target: xe" \runtime starter\starter.fvp/OpenWindow."Logo"

Start in: "C:\Program Files\USDATA\Client Builder\Progra

Shortcut Key: None

Run: Normal window

- **STARTER APPLICATION**
- *Using the Starter Application*
-
-