SunGuide®:

Software Architecture Guidelines

SunGuide-SAG-2.0

Prepared for:

Florida Department of Transportation
Traffic Engineering and Operations Office
605 Suwannee Street, M.S. 90
Tallahassee, Florida 32399-0450
(850) 410-5600

December 5, 2012

Document Control Panel

File Name:	SunGuide-SAG.doc		
File Location:	SunGuide CM Repository		
CDRL:	n/a		
	Name	**Initial**	**Date**
Created By:	Steve Dellenback, SwRI	SWD	11/03/06
	Lynne A. Randolph, SwRI	LAR	11/03/06
Reviewed By:	Steve Dellenback, SwRI	SWD	11/30/06
	Steve Dellenback, SwRI	SWD	1/17/07
	Steve Dellenback, SwRI	SWD	2/16/07
	Roger Strain, SwRI	RLS	12/4/12
Modified By:	Steve Dellenback, SwRI	SWD	1/17/07
	Lynne Randolph	LAR	1/21/07
	Tucker Brown	TJB	12/02/12
Completed By:			

Table of Contents

List of Tables

List of Figures

List of Acronyms

C2C	Center-To-Center
DLL	Dynamic Link Library
DMS	Dynamic Message Sign
DOM	Document Object Model
EH	Executive Handler
FDOT	Florida Department of Transportation
ICD	Interface Control Document
IDE	Integrated Development Environment
IM	Incident Management
ITN	Invitation to Negotiate
ITS	Intelligent Transportation Systems
RMS	Ramp Metering Subsystem
SAG	Software Architecture Guidelines
SwRI	Southwest Research Institute®
XIM	XML Interface Manager
XML	Extensible Markup Language

Revision History

Revision	Date	Changes
1.0.0-Draft	November 30, 2006	Initial Release.
1.0.1-Draft	January 17, 2007	Updated with comments from FDOT CO
2.0	November 2, 2012	Updated to current standards

1.0 Scope

1.1 Document Identification

This document serves as the Software Architecture Guidelines (SAG) for the SunGuide software.

1.2 Project Overview

The Florida Department of Transportation (FDOT) SunGuide® Support, Maintenance, and Development Contract, contract number BDQ69, addresses the necessity of supporting, maintaining, and performing enhancement development to the SunGuide software. The SunGuide software was developed by FDOT through a contract from October 2003 and ongoing as of 2012. The SunGuide software is a set of intelligent transportation systems (ITS) software that allows control of roadway devices as well as information exchange across a variety of transportation agencies; it is deployed throughout the state of Florida. The SunGuide software is based on ITS software available from the state of Texas with significant customization and development of new software modules to meet FDOT's needs. Figure 1 provides a graphical view of the SunGuide software.

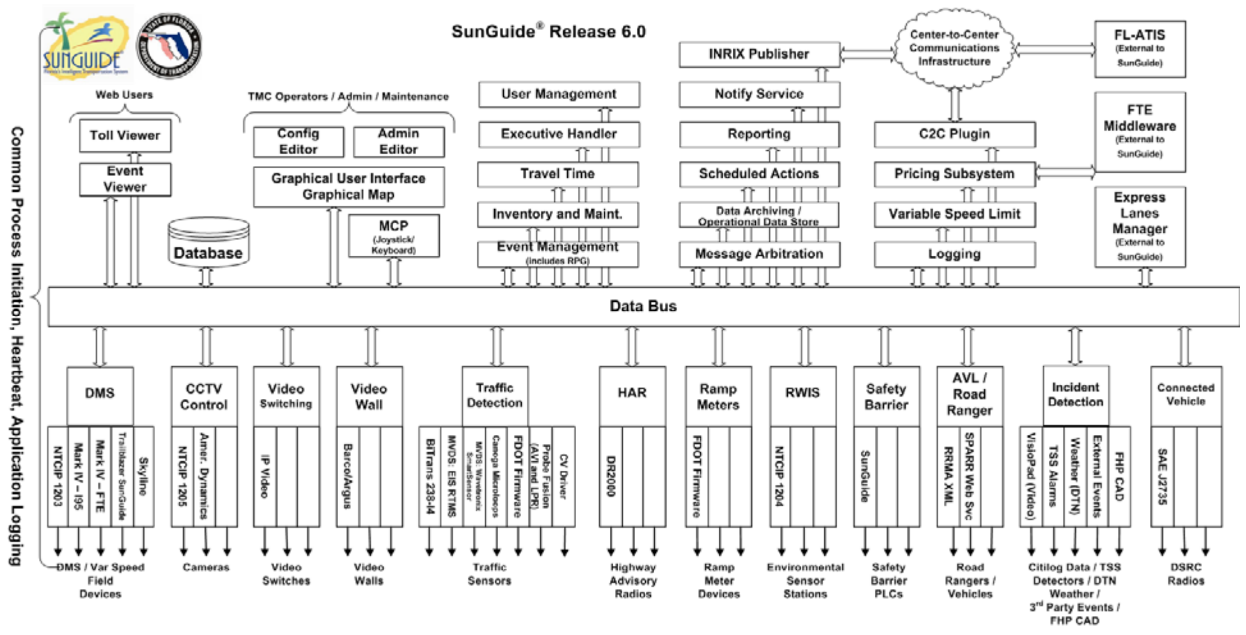


Figure 1-1 - High-Level Architectural Concept

The SunGuide software development effort began in October 2003; several major releases have been developed and this document addresses an incremental update of the most recent release. After development, the software will be deployed to a number of regional and local transportation management centers throughout Florida and support activities will be performed.

1.3 Related Documents

The following documents were used to develop this document:

- Statewide Transportation Management Center Software Library System: Scope of Services. Florida Department of Transportation, September 23, 2003. Contract BD826.

- These documents are available from the document library on the SunGuide project web site at http://sunguidesoftware.com.

1.4 Contacts

The following are contact persons for the SunGuide software project:

- Elizabeth Birriel, ITS Section, Traffic Engineering and Operations Office, elizabeth.birriel@dot.state.fl.us, 850-410-5606
- Arun Krishnamurthy, FDOT SunGuide Project Manager, arun.krishnamurthy@dot.state.fl.us, 850-410-5615
- Clay Packard, Atkins Project Manager, clay.packard@dot.state.fl.us, 850-410-5623
- David Chang, Atkins Project Advisor, David.Chang@dot.state.fl.us, 850-410-5622
- Robert Heller, SwRI Project Manager, rheller@swri.org, 210-522-3824
- Tucker Brown, SwRI Software Project Manager, tbrown@swri.com, 210-522-3035

2.0 Creating SunGuide® Software

The majority of processes incorporated in the SunGuide architecture were developed using C#.NET. As many subsystems and drivers were included in the initial release, a generic framework was developed to allow base functionality to be provided. This allowed subsystems and drivers to utilize the same code for functionality that was required of all processes. The library provides classes and methods for logging messages to the Status Logger, performing communication with the Executive Handler (EH), connecting to an Oracle and SQL Server database, and other common tasks.

Some of the processes, including the Dynamic Message Sign (DMS) processes, and the Ramp Metering Subsystem (RMS) were developed using the Java coding language. Java was used where an existing code base already existed for a particular subsystem. Table 2-1 **Error! Reference source not found.**shows the various tools used during development for both languages.

Table 2-1 - Language Specific Development Tools

Tool	C#.NET	Java
Compiler	Visual Studio C#	Sun Java Compiler
Integrated Development Environment (IDE)	Visual Studio .NET 2010	Intellij IDEA
Refactoring Productivity Tool	Jetbrains Resharper	Included in the IDE
Configuration Management	AccuRev	AccuRev
XML Schema development	XML Spy	XML Spy

The SunGuide architecture utilizes Extensible Markup Language (XML) ICDs to provide an easily understandable interface to the system. Figure 2-1 shows an overview of the capabilities included in the generic library. The shaded circles represent functionality that exists in both the generic subsystem and driver. The non-shaded circles contain functionality utilized solely by the generic subsystem. The following sections detail a summary of each functional area.

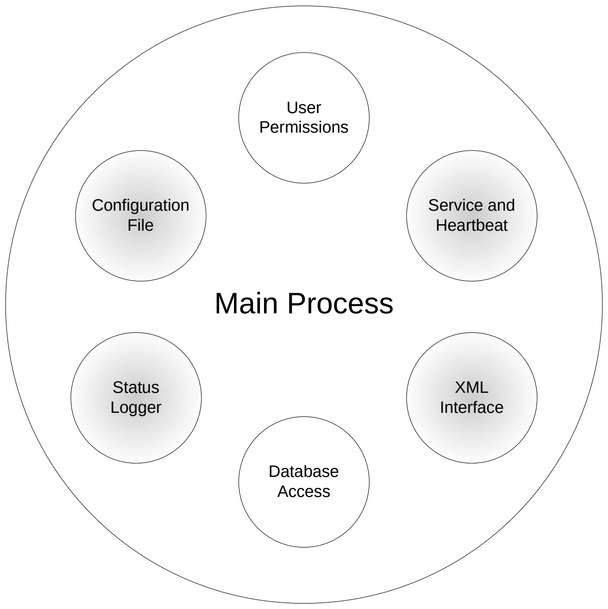


Figure 2-1 - Generic Capabilities

The benefits of creating software using the generic library are numerous.

- Saves time and effort during development; a working subsystem shell can be created in several hours.
- Handler framework allows new functionality to be self-contained.
- Basic functionality is provided and uses the same format as other systems. If a bug is found in common functionality, fixing it once fixes it everywhere. Reduces risk of forgetting basic functionality.
- Reduces learning curve for maintaining other processes. Once a developer understands how the framework functions, maintaining another process becomes much easier.

2.1 Configuration File

SunGuide subsystems and drivers extract information for startup, communication with other processes, and system parameters from an XML configuration file. This file is located on a shared drive and the same file is used for all SunGuide processes. The Data Bus utilizes this configuration file to determine what providers are available and what type of data will be stored and updated by each provider.

Both the generic subsystem and the driver contain the necessary code for reading values from the configuration file. These values include the common connectivity information such as host and port as well as other process specific values. The XML shown below is a configuration file entry for a sample subsystem. The subsystem name in this instance is “sb”. One of the command line parameters for a subsystem is the name of the process. This is used to retrieve the appropriate configuration file values for that process. A driver has an additional parameter for the subsystem to which it belongs, allowing it to be uniquely identified.

```
<sb>
   <host>129.162.101.113</host>
```

```
        <port>40008</port>
        <icdVersion>1.0</icdVersion>
        <maxConnections>20</maxConnections>
        <logLevel>slInfo</logLevel>
        <validation>false</validation>
        <commTolerance>3</commTolerance>
        <providerType>sbStation</providerType>
        <customTag>myValue</customTag>
        <handlers>
            <gov.its.sb.xml.SbRetrieveDataHandler/>
            <gov.its.sb.xml.SbSubscribeHandler/>
            <gov.its.sb.xml.SbStatusHandler/>
            <gov.its.sb.xml.SbBarrierEventHandler/>
            <gov.its.sb.xml.SbConfigurationHandler/>
            <gov.its.sb.xml.SbControlHandler/>
        </handlers>
        <subscriptions>
            …
        </subscriptions>
        <statusUpdates>
            …
        </statusUpdates>
        <drivers>
            <driver>
                <identifier>SBDriver</identifier>
                <host>129.162.101.113</host>
                <port>40088</port>
                <logLevel>slInfo</logLevel>
                <validation>false</validation>
                <icdVersion>1.0</icdVersion>
                <maxConnections>10</maxConnections>
                <packetTimeout>5000</packetTimeout>
                <packetRetries>1</packetRetries>
                <pulseRate>30</pulseRate>
                <handlers>
                    <gov.its.sb.driver.xml.SbDriverConfigHandler/>
                    <gov.its.sb.driver.xml.SbDriverControlHandler/>
                    <gov.its.sb.driver.xml.SbDriverUpdateHandler/>
                </handlers>
            </driver>
        </drivers>
    </sb>
```

Table 2-2**Error! Reference source not found.** shows a brief description of the various entries and how they are handled.

Table 2-2 - Sample Configuration File Values

Config File Tags	Handling by Generic
host port maxConnections	Used to create the listener socket. This is the socket to which clients will connect.
icdVersion	Saved and used to validate that XML being received is the appropriate ICD version.
logLevel	Determines the log level of the process used on startup. This may be changed during runtime by the EH.
validation	Whether XML should be validated against the schema. This is typically used during development only as validation slows down processing time of XML.
commTolerance	Used for device related subsystems. This determines the number of consecutive communications which must fail before the device is moved from Error to Failed status.
providerType	Used by Data Bus to determine what type of provider this subsystem is.
handlers	The list of handlers are XML processing classes that should be instantiated at runtime.
subscriptions statusUpdates	These are used by Data Bus and will be explained in Section 2.7.2. Generic does not read these configuration values.
packetTimeout	Used by the driver to determine how long to wait for a response from a device before timing out.
packetRetries	The number of times a packet will be retried before failing.
pulseRate	How frequently a device should be polled. This value may only be a default, depending on the particular driver.

The configuration file portion of the generic library may also be used to read custom tags from the file. An example of this in the XML above would be the tag, <customTag>, which may be read and used by a process utilizing the generic library.

2.2 Status Logger

The status logging portion of the generic library provides logging functionality. First, the connection to status logger is made using the status logger configuration values. Next, the status logger component uses the logLevel read from the configuration file to determine what level of logging is required. For example, if the logLevel is slInfo, no debug or detailed messages will be logged. If the EH is later used to change the log level to slDetail, the status logger component will accordingly adjust the level of detail logged. As shown in Figure 2-2, the subsystems use a status logger client library for communication with the service.

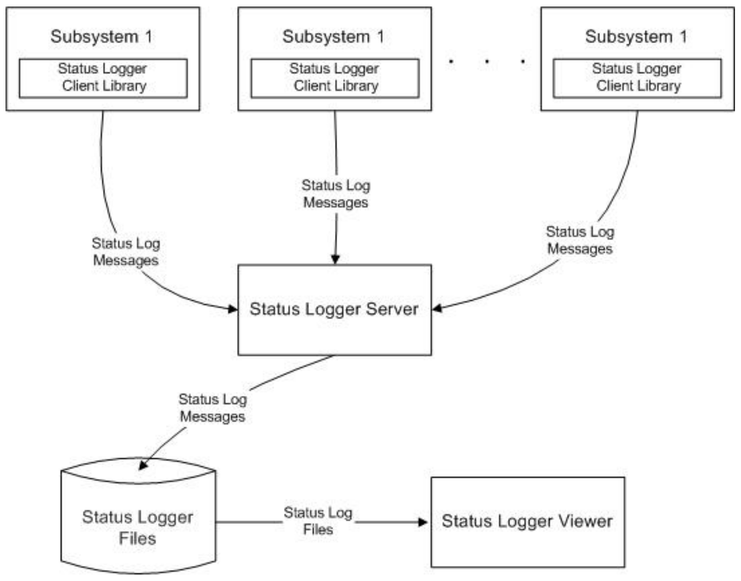


Figure 2-2 - Status Logger

Alternatively, if the status logger service cannot be reached, the status logger component will log to a file in the directory from which the process is running. If the status logger service later begins functioning, the logging will be moved to the process at that time. Example code for logging a message using the generic framework is described in Table 2-3.

Table 2-3 - Sample Log Messages

Sample Message	Field Description
`MainProcess.logMessage(`	The static method for logging a message, part of the main process.
`StatusLogger.level.slDebug,`	What type of message is to be logged (slError, slWarn, slInfo, slDebug, slDetail)
`(int)crm.GetObject("ADD_MSG_CODE"),`	The numeric code for this message, should be a constant, not hard coded.
`GetType().Name,`	The name of the class logging the message.
`MethodInfo.GetCurrentMethod().Name,`	The method name from which the log message is initiated.
`"Adding message to queue: " + device.getId() + ", msg: " + msg.getMsgText());`	The actual message to be logged, should contain as much detail as needed.

2.3 Service and Heartbeat

The generic library contains functionality for communicating with the EH server as shown in Figure 2-3. The EH viewer may be used to send commands to processes and view the current status. The generic library is responsible for sending heartbeat messages to the server at the appropriate interval. This communication is transparent to any process extending the generic framework.

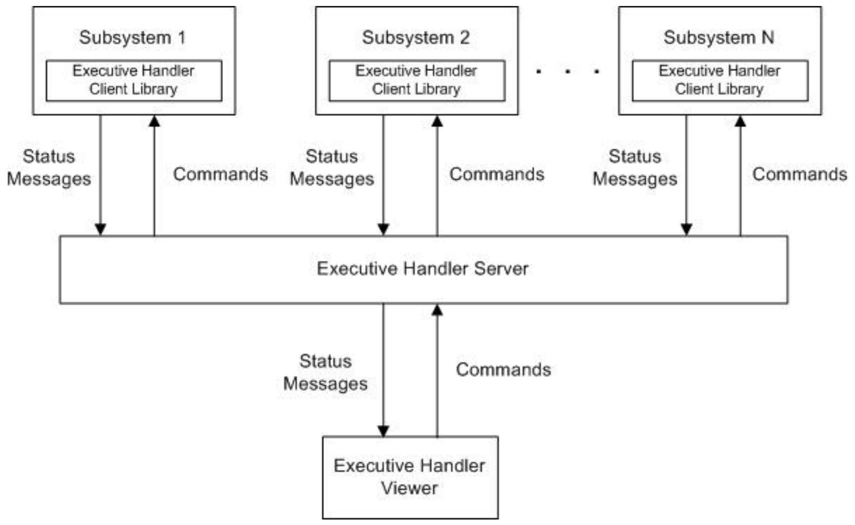


Figure 2-3 - Executive Handler

In addition to status, the EH viewer may be used to send commands to processes. These commands include changing the log level, starting and stopping the process. The generic library contains methods to perform each of those actions when requested by the EH viewer.

2.4 XML Interface

The XML interface portion of the generic library contains functionality for sending, receiving, and handling XML requests, messages, and responses. This component includes the generic main process which is responsible for creating the XML interface, XML handlers, and the communication classes. Figure 2-4 shows the classes which compose this component.

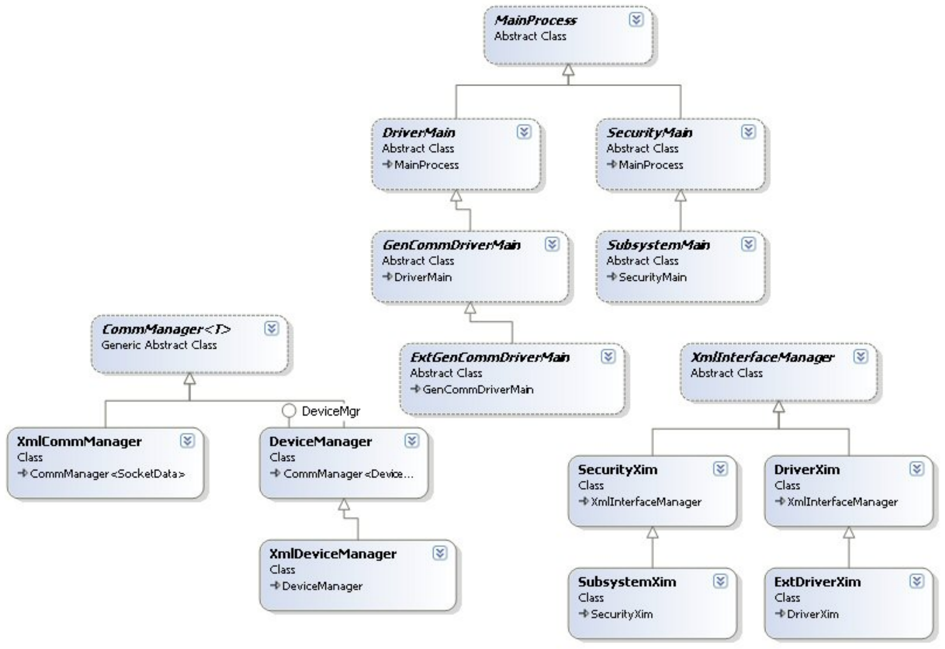


Figure 2-4 - XML Interface Classes

The generic process contains several handlers shown in Figure 2-5 which implementing processes utilize. The handlers provide the following functionality:

- SetPropertiesHandler—handles the setSystemPropertiesReq, which allows modifying the log level and whether XML validation should be used.
- UpdateSystemDataHandler—handles the updateSystemDataMsg, allowing a process to update cached data from the database.
- ClientDisconnectHandler—handles the clientDisconnectMsg, allowing a process to properly remove a connected client.
- DefaultHandler—provides a generic error response if a message or request is received for which no handler is registered.
- AuthenticateHandler—allows subsystems to handle an authenticateReq and log users into the system. A subsystem must only implement a User class to receive this functionality.

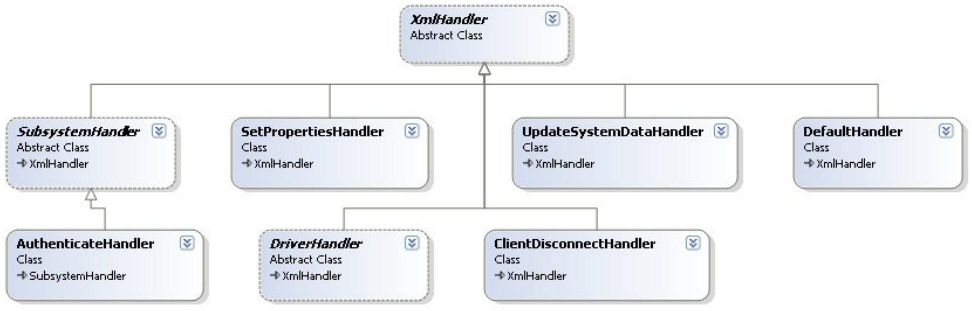


Figure 2-5 - Generic Handlers

One of the primary functions provided by the XML interface is handling XML requests, messages and responses. Figure 2-6 shows how requests are handled by the XML Interface Manager (XIM). When a client sends a request, the XIM checks which handler has registered to handle the request. The handler determines whether this is a request to be forwarded to a driver or can be handled immediately. The blue lines show the path when a request is sent to the driver. Once a response is received from the driver, the handler sends it to the appropriate client or clients. The red lines show the path when the request can be handled immediately and a response returned to the client.

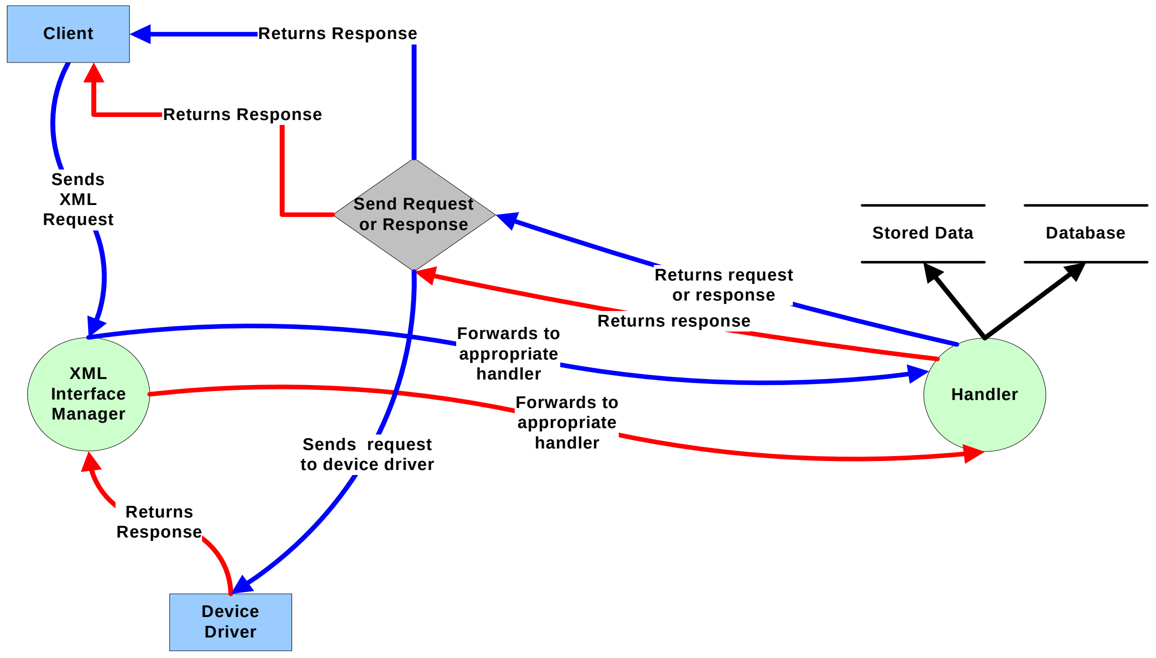


Figure 2-6 - XML Handling

All communications are asynchronous, permitting multiple requests to be handled without requiring the system to wait until a response is generated. Asynchronous communication is especially important for requests which cannot be completed immediately, allowing the process to continue to be responsive to clients.

2.5 User Permissions

For subsystems created using the generic library, a user permission component is included. This component utilizes the AdoDbUser class that is part of Database Access. When creating the AdoDbUser, the subsystem specifies the subsystem name for which permissions and users should be retrieved. The database tables (subsystems, subsystem_permissions, users and user_permissions) shown in Figure 2-7 are used to store the permissions for the various subsystems.

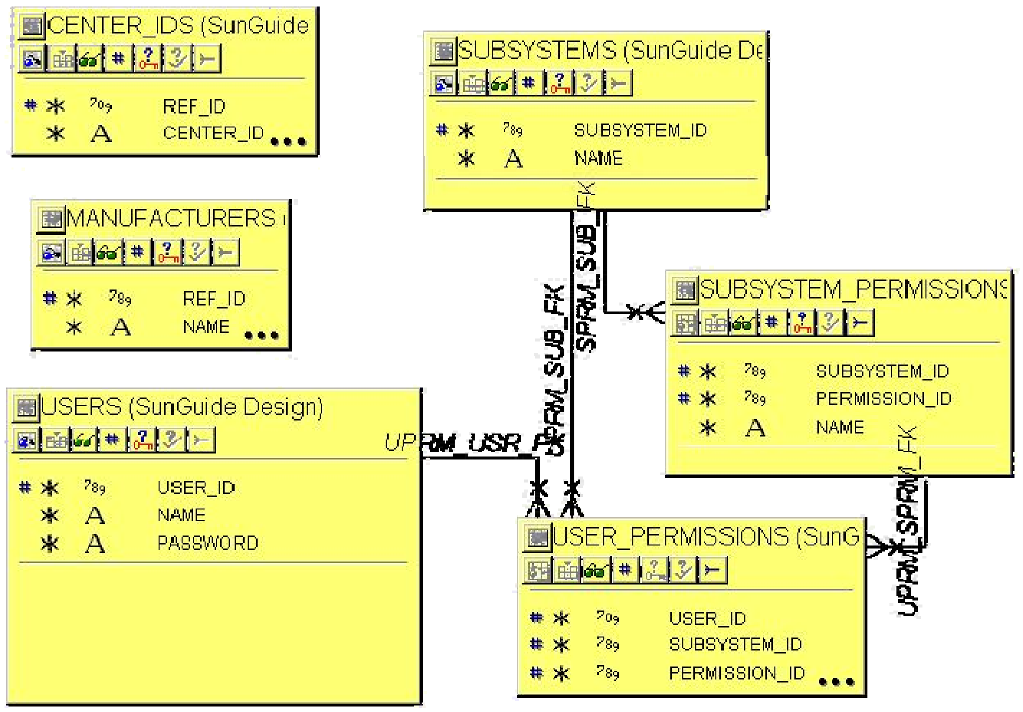


Figure 2-7 - User Permission Tables

When retrieving users from the AdoDbUser, only users with permissions for that subsystem will be retrieved. Once the users are obtained, a Security Controller is created with the list of users. The authenticate handler contained within the generic framework uses this Security Controller when an authenticate request is received. If the user can be logged into the Security Controller, a user with the appropriate permissions is used to verify permissions for subsequent requests. If the user cannot be logged in, an appropriate error response is returned to the requestor. The possible error messages are shown in Table 2-4.

Table 2-4 - Login Error Messages

Error Message	Description
"Bad user name or password."	The user does not exist or the password is incorrect.
"No user name or password provided."	Tried to login with an empty user name or password.
"User does not have system access permission"	Some subsystems have a “system access” permission required to login, and this user does not have that permission.
"No users exist with permissions to this subsystem."	This is a special error message, useful for new subsystems for which administrators might not have assigned permissions.
"There was an error logging out the user which was already connected."	The user was already logged in and could not be logged out.

2.6 Database Access

A library of database classes contained in the generic library allows database communication to be abstracted. First, as discussed in the user permissions section, the AdoDbUser retrieves users with subsystem permissions for validating logins. A AdoDbItem class exists which handles connecting to the database and performing the appropriate actions. Each class that handles database items should extend AdoDbItem to gain this functionality. The classes used by the Database Access component are shown in Figure 2-8.

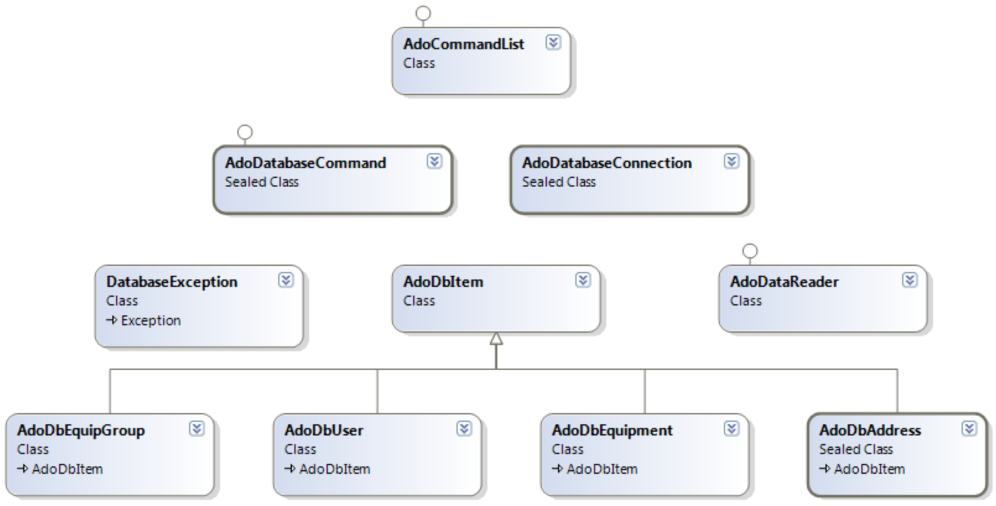


Figure 2-8 - Database Classes

The database access component includes helper classes for database methods. The AdoDatabaseCommand class handles ensuring a database connection exists, throwing exceptions if errors occur, and reconnecting to the database if needed. The AdoCommandList class allows a list of database commands to be executed using a transaction. Then, if any statements fail, the entire list of commands will be rolled back. When executing any commands, the connection must always be closed before returning from the method. This allows the database cursors to be released. If commands are not closed properly, the database may throw a “Too many open cursors” error. The sample code below shows the proper method for executing and closing a command:

```
updateOpstatusCommand.setParameter("1", opStatus);
updateOpstatusCommand.setParameter("2", harId);
int numRows = updateOpstatusCommand.executeCommand();
updateOpstatusCommand.closeConnection();
```

2.7 Interfaces

The following sections describe the use of SunGuide interfaces.

2.7.1 Interface Control Documents

The architecture utilizes XML ICDs to provide an easily understandable interface to the system. A general ICD exists which describes the byte ordering of the transactions and how the data is organized. This ICD also discusses that authenticate, subscribe and retrieve data should be implemented by all subsystems. The general ICD also outlines XML schema which are used by the generic library. These include objects, general transaction structure, and XML messages, requests and responses which are handled at the generic level.

Each subsystem/driver has a unique ICD which describes the communication between a client and the subsystem and the subsystem and the driver. In addition to outlining the XML schema for this subsystem, general system behavior is included in the ICD. Startup messages for a client-subsystem or subsystem-driver connection are shown. Each specific ICD contains two tables which describe how the schemas are used. These tables show the interface that the subsystem and driver are responsible for implementing—which schemas are sent and received by either the subsystem or the driver. Another table in the ICD would be used by a client to determine which types of data updates should be requested.

2.7.2 Data Bus

The Data Bus has two responsibilities. It distributes status updates to clients and routes commands from clients to subsystems and from subsystems to clients. Data Bus has two ICDs available, one is for client communication to Data Bus and the other is for creating a provider to which Data Bus can communicate. Figure 2-9 shows an overview of the Data Bus process. Commands are routed in both directions, while status is pushed into Data Bus from a provider and subsequently pushed to the client by Data Bus.

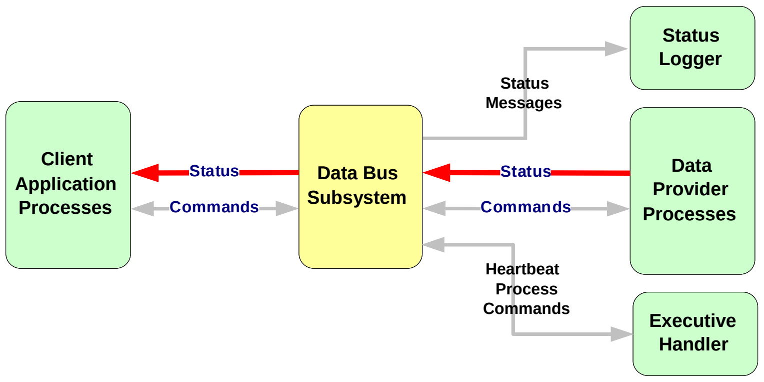


Figure 2-9 - Data Bus Overview

Data Bus was designed to provide a single portal to the system for a client. This allows one client connection from which status data of all types can be retrieved and a single portal for sending commands to multiple subsystems as shown in Figure 2-10. If a subsystem must be moved to a new machine, the only change that must be made is for Data Bus to know where the process has been moved.

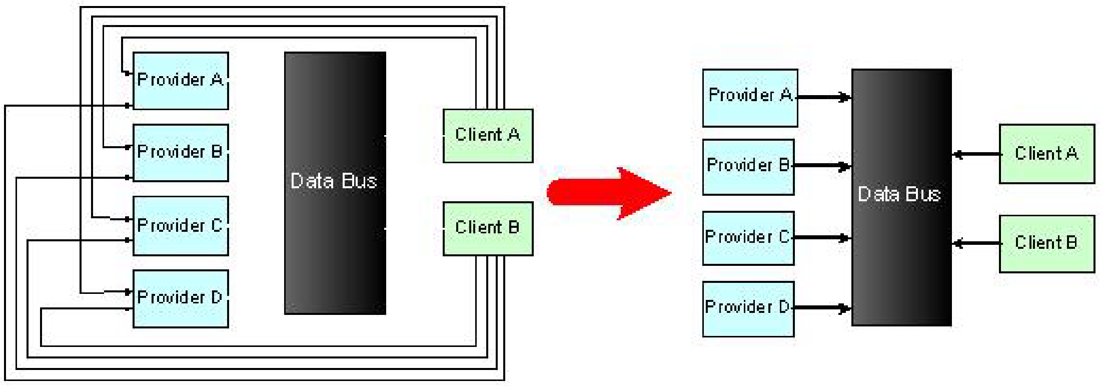


Figure 2-10 - Data Bus Connections

When creating a provider for Data Bus, the configuration file is used to provide information to Data Bus. This allows Data Bus to be expanded to store new types of status information without any code changes required. The XML below shows the relevant portions of the configuration file.

```
<dataProviders>
    <sb>
        …
        <handlers>…</handlers>
        <subscriptions>
            <stationData/>
            <statusData/>
            <eventData/>
        </subscriptions>
        <statusUpdates>
            <sbStation>
                <addSbResp action="add"/>
                <modifySbResp action="modify"/>
                <deleteSbResp action="delete"/>
                <onlineStatusMsg/>
                <setOnlineStatusMsg/>
                <setOnlineStatusResp/>
                <setStatusResp/>
                <statusResp/>
            </sbStation>
            <sbEvent>
                <barrierEventMsg action="add"/>
                <cancelBarrierEventMsg action="delete"/>
                <acknowledgeEventResp/>
            </sbEvent>
        </statusUpdates>
        <drivers>…</drivers>
    </sb>
</dataProviders>
```

Any subsystem listed under the dataProviders tag can provide status updates to Data Bus and receive commands routed through Data Bus. Data Bus requires the subsystem to have the following attributes:

- Implement authenticateReq
- Implement retrieveDataReq with a data type of "statusList". The status list must return any status values that should be stored in Data Bus.
- Implement a subscribeReq:
    - For any data types that will be updated or have items added, the appropriate subscriptions should be listed in the subscriptions tag of the configuration file.
    - For responses or messages which will update or add items to the Data Bus, these should be listed in the configuration file under the appropriate resource type. In the example above, two types of data are stored in Data Bus for the subsystem, sbStation and sbEvent. The actions shown are described in detail in the Data Bus Subsystem Provider Template ICD.

2.8 DataLibrary

As new subsystems are added to SunGuide and old subsystems are updated, DataLibrary is being used to manage XML serialization. To ensure consistent operation across the system, these objects should be created using the approach described below.

2.8.1 Overall Structure of the DataLibrary Project

The DataLibrary project should contain a top level folder for each included subsystem. These folders typically should match the folders in the project XSD schemas. Each folder will typically contain individual folders for schema objects, messages, requests, and responses, again based on the XSD schema folder structure. Responses typically include a response-specific data object; these should be segregated into a folder under the responses folder. (Some subsystems have this folder under objects; this is acceptable, but not preferred.)

2.8.2 Objects

All DataLibrary objects should be subclassed from the XmlBase object (or one of its subclasses). This provides a variety of functions which can be used by and on DataLibrary objects. These objects should each represent a single XSD object, such as a `Har`, `HarConfig`, `HarStatus`, `Privilege` set, etc. Each object should be placed in a file in the subsystem's objects folder. Each XML child element of the object should be represented as a public property with a get and set method. The properties typically have the same name as the element, only with an initial capital. These properties should be tagged using XML Serialization attributes to reflect their XSD naming, and in some cases content. A full discussion of XML Serialization attributes is beyond the scope of this document. Information on MSDN may be found at the following locations, among numerous others:
http://msdn.microsoft.com/en-us/library/83y7df3e%28v=vs.100%29.aspx
http://msdn.microsoft.com/en-us/library/2baksw0z%28v=vs.100%29.aspx

2.8.2.1 Nullable Primitive Data Properties

By default, the serializer will include an object such as a nullable integer (`int?`) even if it has no value. Typically, SunGuide schemas are designed so that if the field is not populated, the element should not be present. To simplify developing these type of fields, the `[AutoOptionals]` attribute may be applied to the class. When using this attribute, a special code region named AutoGeneratedXml must also be included in the file. A preprocessor application will run before a build session to generate the appropriate supplemental properties for any nullable fields to ensure the XML serializer works as desired.

2.8.3 Messages

XML messages for a subsystem should be placed in the messages folder of the subsystem. Each message should subclass from a subsystem-specific message base class, which itself extends the `Message` class. This subsystem-specific base class should implement the ProviderXsdFolder property to provide the folder name under which subsystem schemas may be found. Each specific message object should specify its custom child elements as normal. As these messages are not typically stored or updated, auto-properties and non-observable collections may be used for the message object's properties. Standard message attributes (i.e., provider name, provider type, ref ID, and ICD version) are already declared and serialized by the superclass objects. Each message should also specify an `[XmlRoot]` attribute to specify the root tag name of the request, such as "addHarMsg".

2.8.4 Requests

XML requests for a subsystem should be placed in the requests folder of the subsystem. Each request should subclass from a subsystem-specific request base class, which itself extends the `Request<TResponse>` class. This subsystem-specific base class should implement the ProviderXsdFolder property to provide the folder name under which subsystem schemas may be found. The `TResponse` generic type parameter of the class allows each request to specify the response which is created by the subsystem to reply to the request. This information is used in a variety of ways by lower layers of the system. Each specific request object should specify its custom child elements as normal. As these requests are not typically stored or updated, auto-properties and non-observable collections may be used for the message object's properties. Standard request elements and attributes (i.e., username, security token, provider name, provider type, ref ID, and ICD version) are already declared and serialized by the superclass objects. Each request should also specify an `[XmlRoot]` attribute to specify the root tag name of the request, such as "addHarReq".

2.8.5 Responses

XML responses for a subsystem should be placed in the responses folder of the subsystem. Each response should subclass from a subsystem-specific response base class, which itself extends the `Request<TResponseData>` class. This subsystem-specific base class should implement the ProviderXsdFolder property to provide the folder name under which subsystem schemas may be found. The `TResponseData` generic type parameter of the class allows each response to specify the response data object type which is contained by the response. Responses should not typically contain any properties themselves. The data for the response will be serialized by the base class. Each response should specify an `[XmlRoot]` attribute to specify the root tag name of the request, such as "addHarResp".

2.8.5.1 Response Data Objects

The main body of response objects is not defined by the response itself, but rather a matching response data object. These objects should typically be placed in the responseData folder under the responses folder for the subsystem. These objects must extend `ResponseData` and specify an `[XmlType (“typeName”)]` attribute which matches the XSD type name from the schema. It is acceptable for multiple responses to share a response data object. If different schema type names must be specified, a subclass of the custom response data object may be created with a different `[XmlType]` declared. As these response data objects are not typically stored or updated, auto-properties and non-observable collections may be used for the response data object’s properties.

2.8.5.2 Retrieve Data Response Objects

Response data objects for retrieveDataResp schemas should follow the standard response data architecture, but additionally should implement the `IUserList<TPrivileges>` interface. This requires that the response data object provide a `Users` property which returns a list of `User<TPrivilege>` objects. The `TPrivilege` type should be a type defined in the subsystem’s objects folder which extends `PrivilegeBase` and specifies user privileges as a set of Boolean attributes.

2.9 Subsystem Development

Creating a new subsystem using the generic library is generally a quick and easy process. Following the steps outlined below will get you up and running rapidly. Table 2-5 details tools that developers typically use to provide an efficient development environment.

Table 2-5 - Development Tools

Tool	Required?	Description
Microsoft Visual Studio 2010	Y	The C#.NET integrated development environment
Jetbrains Resharper	N	A C# refactoring tool used to provide increased productivity
AccuRev	N	A change management repository

2.9.1 Creating a Project

Create a new Visual Studio project. The easiest method is to create a solution with three projects, one for the library of functionality, one to create a console application, and one to create a service. This allows easy debugging during development without the concern that files will be out of synch with the service project.

Add the ITSGeneric Dynamic Link Library (DLL) reference for the generic library to each project. If desired for debugging purposes, the actual ITSGeneric project can be added instead of the DLL.

2.9.2 Steps to Start

The steps below will create a subsystem with limited functionality. Client connections will be accepted and authenticate requests handled.

2.9.2.1 Extend User class

The base `User` class contains a username and permission level for the user. Class responsibilities include creating a user, specifying the privilege names for given privilege values, and providing an XML node type representing the user object.

- Create an enumeration for privilege names. The value of each member of the enumeration corresponds to the bit in the permission level of the user.
  - **Important Note**: The order in which the privilege names are defined in the enumeration must be consistent with the respective permission_id stored in the subsystem_permissions database table. The first privilege is associated with the subsystem permission with permission_id = 0. The second privilege is associated with the subsystem permission where permission_id = 1, etc.
  - A subsystem must have at least as many permissions as subscriptions.
    - Subscriptions must match up in order with the respective permissions for that subscription.
    - Example: The following permissions {stationData, stationStatus, userData, configStations, setStatus} may be associated with the subscriptions {stationData, stationStatus, userUpdates}. The names do not have to match up, but the order must be the same. When a client subscribes to data, the subscriptions are ANDed with the client's permissions to determine whether the subscription should succeed.
- Define a static integer member, specifying the number of privileges defined in the privilege enumeration.
- Define a default constructor.
  - Initialize the base numberOfPrivileges member to the static integer member
  - Retrieve the resource manager using the main process static method.
  - If any other constructors are created, ensure that these steps are included or the default constructor is called.
- Define a public method to generate an XML node type for this user object. Typically, the method name generateXml is used for this purpose.
  - Log any exceptions that occur in this method.

2.9.2.2 Data Management Classes

A subsystem typically uses three data classes to manage ensuring the database and cached data are kept in synch.

2.9.2.2.1 Data Access

The class used for data access controls all database retrievals and updates, ensuring the database code is centralized. More code will be added to this class in Section 2.9.4. The steps below are for creating a preliminary data access class with the ability to retrieve users and passwords.

- Define a private AdoDbUser class member.
- Define a constructor for the class.
  - Instantiate the AdoDbUser class member.
- Provide a method for retrieving users.
- Provide a method for retrieving passwords.

2.9.2.2.2 System Data

The system data class controls updates to the cached data used by the system during runtime. More code will be added to this class in Section 2.9.4. The steps below are for creating a preliminary system data class with the ability to retrieve users and passwords.

- Define private class members for a list of users and passwords.
- Define a constructor for the class.
    - Instantiate the class members.
- Provide a method for retrieving users.
- Provide a method for retrieving passwords.

2.9.2.2.3 Data Manager

The data manager class is responsible for keeping the database and cached data synchronized. The class contains member variables for the data access and system data classes. Updates to the database should be performed and checked before cached data is updated. The steps below are for creating a preliminary class that can retrieve and update users and passwords. More code will be added to this class in Section 2.9.4.

- Define private member variables for the subsystem's data access and system data class objects.
- Define a constructor.
    - Initialize the subsystem's system data and data access classes.
- Implement the updateSystemData() method that will refresh locally cached data with current database data. This method is called on startup and whenever an updateSystemDataMsg is received.
    - Reset the system data users and passwords from the data access class.

2.9.2.3 Main Process

The main process contains startup methods for the subsystem and other methods for retrieving data.

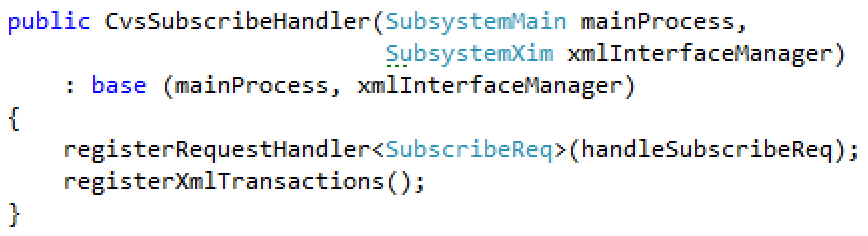
- Extends SubsystemMain to create a main process for this subsystem.
- Define private member variables.
    - Define a data manager
- Define a constructor with a parameter of a string array (for the command line arguments).
    - Call the base class constructor with the string array.
    - Initialize the data manager class member.
    - Implement all abstract functions including the following:
        - runLoop
            - Handle subsystem-specific startup procedures.
            - Call the base method.
            - Set startup completed flag (until this flag is set, no XML will be processed by the subsystem).
        - `setDataMgr`()
            - Verify the data manager is the appropriate class type.
            - Set the data manager class member.
        - `getDataMgr`()—return subsystem-specific data manager member.

- `createCustomizedResourceManager`()
    - This method, while still necessary, should not be used as a way to retrieve constants. Instead, create a “Common Constants” class that will have public static members that can be referenced throughout the project.
- `updateSystemData`()
    - Invoke the data manager’s updateSystemData method.
    - Update users and passwords members via the data manager member.
    - Invoke the security controller member’s updateUsers method, sending the updated user and password information.
    - If the subsystem uses devices, instantiate the statusMgr member. The status manager tracks the communication errors for devices.
- Provide schema location information for subsystem-specific requests, messages, and responses.
    - Add a method to setup the base schema location reference and store it as a member variable.
    - Override getReqSchemaLocation(), using the base schema location member to set the subsystem-specific requests schema location.
    - Override getRespSchemaLocation(), using the base schema location member to set the subsystem-specific responses schema location.
    - Override getMsgSchemaLocation(), using the base schema location member to set the subsystem-specific messages schema location.
- instantiateSecurityController() sets the users and passwords stored by the process.
- driverDisconnect() to process steps that need to be taken when a driver is no longer connected. In some instances, no actions are necessary.
    - Send change of device state messages to the client.
    - Stop polling or other processing.
- driverReconnect() to process steps that need to be taken when a driver reconnects to the system. Again, in some instances no actions are necessary.
    - Add devices to specific drivers.
    - Restart polling or other processing.
- If devices are used by the subsystem, send XML messages for online status changes to the appropriate driver(s) and process add/remove devices from driver as necessary.
    - addDevicesToDriver() to add one or more devices to the specified drivers.
    - removeDevicesFromDriver() to remove all devices from the drivers.
    - If using the status manager, override sendOnlineStatusMsg(). This method is used by the status manager to send online status

messages to subscribed clients when a device changes operational status.
- *initialize*() to add the supported ICD versions before calling the base initialize().
- *getProviderType*() to return the appropriate string value for the subsystem's provider type.

2.9.2.4 Initial XML Handlers

The following steps will create handlers for subscribing and retrieving initial data (users only). Once these are complete, other handlers may be added to perform additional functionality.



```
public CvsSubscribeHandler(SubsystemMain mainProcess,
                           SubsystemXim xmlInterfaceManager)
    : base (mainProcess, xmlInterfaceManager)
{
    registerRequestHandler<SubscribeReq>(handleSubscribeReq);
    registerXmlTransactions();
}
```

Figure 2-11 - Registering a request with the subsystem

- Extend the SubsystemHandler class to create an abstract class for this subsystem.
  - This class will be extended by the other subsystem-specific handlers.
  - Initialize subsystem-specific members.
    - Subsystem-specific main process, data manager, etc.
  - Provides common subsystem-specific functionality used by the other subsystem-specific handlers.
    - Can define methods to process sending requests down to the driver.
    - Base handler constructor should not assign any types of requests, messages, and responses supported.
- Extended subsystem-specific handlers—these steps should be followed for each handler that is created.
  - Each subsystem-specific handler should extend the base subsystem handler.
  - Handler constructors.
    - Assign the types of requests, responses, and messages supported by the handler by registering the DataLibrary type with the handler.
    - Handling of a specific request, response, or message
      - A method must be created with the request, response, or message as the only argument
      - This method must be then passed in as the argument to the registerRequestHandler method. This will tell the framework what to call when the handler receives a particular message.
    - Ensure the handler calls the registerXmlTransactions() method at the end of the constructor. This will confirm to the subsystem that the local cache of defined above must be used instead of an older registering method not documented in this document.

2.9.2.5 Subscribe Handler

The subscribe handler should be created as discussed in 2.9.2.4. The only message this handler should process in the subscribeReq. When returning a response, only the successful subscriptions should be returned.

2.9.2.6 Retrieve Data Handler

The retrieve data handler should be created as discussed in 2.9.2.4. The subsystem should retrieve, at a minimum, user data and a status list if it is a Data Bus provider. If a user requests user data but a permission prevents them from retrieving all user data, the subsystem should return the permissions for the user who requested the data.

2.9.3 Running the Subsystem

After completing the above steps, the subsystem can be started. The command line parameters are the name of the process, the configuration file location and an optional "-d" debug mode flag. The best method of debugging system startup is to have the logLevel flag in the configuration file set to "slDetail" and the –d command line flag set. Then, after starting the process, view the log file to see detailed messages of the startup sequence of events. If any errors occur, these will be logged.

Prior to logging into the subsystem, the following steps must be performed in the database:

- Add the subsystem name to the SUBSYSTEMS table.
- Add the appropriate permissions to the SUBSYSTEM_PERMISSIONS table.
- Add permissions for a user to the USER_PERMISSIONS table.

Once the process has started successfully, use a client tester program to send an authenticate request to the subsystem with the user whose permissions were added above. Once successfully authenticated, retrieve the user data. The data should contain permissions for the user.

2.9.4 Finishing the Subsystem

The rest of the steps for creating a subsystem will be specific to the subsystem being created. The following list contains details to consider.

- Examine what schema requests/responses/messages your subsystem must handle. Then try to combine schemas together to create a handler. For example, a configuration handler can be created that handles add, modify and delete requests and responses.
- Add appropriate classes for objects needed by the system.
  - If accessing from the database, extend AdoDbItem for database connectivity.
  - Modify the data management classes to retrieve and update the database objects.

2.10 Driver Development

Creating a new driver using the generic library is generally a quick and easy process. Following the steps outlined below will get you up and running rapidly. Table 2-5 details tools that developers typically use to provide an efficient development environment.

2.10.1 Creating a Project

Create a new Visual Studio project. The easiest method is to create a solution with three projects, one for the library of functionality, one to create a console application, and one to create a service. This allows easy debugging during development without the concern that files will be out of synch with the service project.

Add the ITSGeneric DLL reference for the generic library to each project. If desired for debugging purposes, the actual ITSGeneric project can be added instead of the DLL.

2.10.2 Steps to Start

Creating a driver is similar to creating a subsystem, with a few differences. There are no users in a driver, no database access, and no subscribe or retrieve data handlers. Most of the functionality is contained within the command classes.

2.10.2.1 Data Management

Other than data retrieved from the configuration file, a driver typically receives its data from the associated subsystem. Therefore only a system data class is needed to store cached data.

- DataLibrary objects should be created for all driver-specific items.
    - Driver specific messages and request/response pairs should be created
- Create a system data class to store driver-specific cached data.
    - Define a constructor, initializing driver-specific cached data.
    - Provide necessary members/methods to manage pertinent driver-specific cached data.
    - Provide getter methods for appropriate driver-specific items.
    - Define a method to reinitialize driver cache. This method should be called when the client process disconnects from the driver.

2.10.2.2 Main Process

Either the ExtGenCommDriverMain or GenCommDriverMain class should be extended to create a new driver. The ExtGenCommDriverMain allows asynchronous device communication while GenCommDriverMain performs communications synchronously.

- Define a system data member.
- Define a default constructor.
- Implement abstract and virtual functions.
    - instantiateDataMgr() to initialize the system data member responsible for caching driver data.
    - runLoop()
        - Handle driver-specific startup procedures.
        - Set the provider type.
        - Instantiate the member class responsible for managing cached driver data.
        - Call the base class runLoop().
    - getDataMgr() to return the class member responsible for manager cached driver data.
    - createCustomizedResourceManager() must be implemented however should not be used. Instead, create a "Common Constants" class that will have public static members that can be referenced throughout the project.
    - Provide schema location information for subsystem-specific requests, messages, and responses.

        - Add a method to setup the base schema location reference and store it as a member variable.
        - Override getReqSchemaLocation(), using the base schema location member to set the subsystem-specific requests schema location.
        - Override getRespSchemaLocation(), using the base schema location member to set the subsystem-specific responses schema location.
        - Override getMsgSchemaLocation(), using the base schema location member to set the subsystem-specific messages schema location.
    - clientDisconnect() to process steps that need to be taken when a client is no longer connected such as clearing driver table information and reinitializing the driver cache.
    - initialize() to add the supported ICD versions before calling the base.initialize().
    - getProviderType() to return the appropriate string value.

2.10.2.3 Handlers

See Section 2.9.2.4 for information on creating handlers for the driver. The only difference is that DriverHandler should be overridden rather than the SubsystemHandler.

2.10.2.4 Creating Commands

The driver device manager handles collecting data packets until a command packet is complete. A command should be created for each type of communication with a device (i.e., poll request, reset command).

- Define driver command classes for each type of device command supported by the driver, extending the DriverCommand class.
    - Override the following methods:
        - getCommandName()
        - receiveResponse()
        - executeNextStep()
        - callSuccessfulResponse()
        - callUnsuccessfuleResponse()
        - handleCommandTimeout()
    - When adding equipment to the driver's local cache, also add the appropriate address and protocol information to DriverMain's member tables, using methods addAddressToTable() and addProtocolToTable().
- Instantiate the appropriate driver-specific command for each command request sent to the driver from within the appropriate driver-specific XML handler, and send it to the driver device manager for processing.
    - Retrieve appropriate address and protocol information from the DriverMain's member tables, using methods `getAddressFromTable() and getProtocolFromTable().`
- When instantiating the command, these protocol and address items are listed as parameters to the command, as well as the calling XML handler object.
    - To send the command down to the driver device manager, call sendDriverCommandToDeviceMgr().
    - XML handlers instantiating driver commands must include methods invoked by the driver command to send successful/unsuccessful responses to the client. The

names of these methods are defined in the driver-specific command classes' callSuccessfulResponse() and callUnsuccessfulResponse() methods.

2.10.3 Running the Driver

After completing the above steps, the driver can be started. The command line parameters are the name of the process, the configuration file location, the name of the subsystem for this driver, and an optional "-d" debug mode flag. The best method of debugging system startup is to have the debugMode flag in the configuration file set to "slDetail". Then, after starting the process, view the log file to see detailed messages of the startup sequence of events. If any errors occur, these will be logged.

3.0 Notes

None.