Satlinx Programmer's Manual

V1.0

Table of Contents

1 Architectural Overview

1.1 Overview

Satlinx is an object-oriented framework. Careful design of the class hierarchies locates common functionality in base classes, much as libraries provide in a procedural implementation. It is intended that the standard base classes are rich in function, and perform much of the core functionality of the system. Application level classes only need implement that functionality that is specific to the instrument or operation in question. This radically reduces the size of the code base to maintain and the effort involved in adding new instruments and functions to the system.

The framework is built on the industry standard Common Object Request Broker Architecture (CORBA). This provides all the low-level mechanisms for cross platform distributed inter-object communications. Once the objects are CORBA-enabled, methods and attributes on remote objects may be accessed as if they were being accessed through a local pointer. To make the objects CORBA enabled their interface must be defined in the CORBA Interface Definition Language (IDL), much as C++ classes are defined, and the object class must inherit from the base class generated by the IDL compiler.

In addition, many of the generic services used by the system are supplied with the CORBA implementation.

The overall framework is comprised of components that fall into one of the following categories:

- Virtual Function Objects
- Virtual Task Objects
- Virtual Instrument Objects
- Physical Instrument Objects
- Communications Adapter Objects

Other special objects provide services to the framework or allow user access to the components comprising the framework. These include:

- User Interface Modules
- Authentication Server
- File Manager
- System Logger

- Configuration Manager

The details of this framework will be described in the succeeding sections.

1.2 CORBA

At the most basic level, CORBA is a standard for distributed objects. CORBA allows an application to request an operation to be performed by a distributed object and for the results of the operation to be returned to the application making the request. The application communicates with the distributed object that is actually performing the operation. This is basic client/server functionality, where a client issues a request to a server and the server responds back to the client. Data can pass from the client to the server and is associated with a particular operation on a particular object. Data is then returned to the client in the form of a response.

The advantages of CORBA over other competing technologies are as follows:

- CORBA supports many existing languages. CORBA also supports mixing these languages within a single distributed application. CORBA uses a standard interface definition language (IDL) to define the distributed objects. The same IDL is used in all language mappings. Since different pieces of the entire station control software will be written in different languages, primarily C++ and Java, the use of CORBA IDL ensures that all modules, regardless of implementation language, will be compatible.

- CORBA supports both distribution and Object Orientation.

- CORBA is an industry standard. This creates competition among vendors and ensures that quality implementations exist. The use of the CORBA standard also provides the developer with a certain degree of portability between implementations. Several ORBs exist that are available as open source. If additional features are required that are not available in the current ORB it will be possible to switch to another ORB without requiring extensive code changes. If care is taken to adhere closely to the language mappings, no code changes will be necessary.

- CORBA provides a high degree of interoperability. This insures that distributed objects built on top of different CORBA products can communicate. Different ORBs may be used for different pieces of the system. For example, new instruments may be developed that contain an embedded operating system. This operating system may require the use of a different ORB from the one used

by the rest of the software. However, the interoperability features of CORBA guarantee compatibility.

1.2.1 CORBA Services

The Object Management Group (OMG) has defined a number of services to provide additional capabilities to basic CORBA mechanisms. These are as follows:

- Event - a simple messaging mechanism for decoupling objects
- Naming - an object directory that allows objects to be located by a name
- Trading - another object directory that uses object properties in searches
- Notification - a more sophisticated messaging system, that supports guaranteed delivery and event filtering
- Life Cycle - provides support for the object lifecycle
- Property - allows values (properties) to be associated with an object at runtime
- Collection - supports the manipulation and management of collections of objects
- Concurrency - a mechanism for managing the concurrent access to system resources
- Relationship - allows networks of objects to be created and navigated.
- Time - a service for providing consistent time across a distributed system.

The services of particular relevance to Satlinx are described below.

1.2.1.1 Naming Service

The OMG Naming Service provides a mapping from names to object references. Given the name, the service returns the object reference stored under that name. This is much the same as when a DNS server maps system names to IP addresses. Objects register with the name server and thus become public. The names used are arbitrary but must be unique.

The Naming Service maps names to object references. A name-to-reference association is called a *name binding*. The same object reference can be stored several times under different names, but each name identifies exactly one reference. A *naming context* is an object that stores name bindings. The naming context implements a table that maps

names to object references. A name in the table can denote either an object reference to an application object or another context object in the Naming Service. This means that, like a file system, contexts can be connected to form hierarchies: contexts correspond to directories that store names either to directories (other contexts) or to files (application objects).

This hierarchical naming structure lends itself very well to the clustering requirements of Satlinx. Similar groupings of instruments can be assembled and given the same names, but belong to different naming contexts, just as files in a file system can have the same name once they are in different directories.

Objects seeking to establish a connection to a published object issue a name resolution request to the name server and are returned a unique reference to the object. The object reference acts as the equivalent of a pointer.

1.2.1.2 Event Service

Basic CORBA uses a synchronous request model. The CORBA Event Service provides asynchronous communications. CORBA objects are either suppliers or consumers of events. The Event Service connects suppliers and consumers through the mechanism of event channels. Suppliers can push data into the event channel or wait for it to be pulled. Similarly, consumers can wait for data to be pushed to them from the event channel or pull it from the event channel. Event channels can support multiple suppliers and consumers, and suppliers can be both push and pull suppliers and consumers can be both push and pull consumers, on the same event channel. This mechanism supports a very flexible configuration of communication between CORBA objects. The same data can be pushed from the supplier and pushed onward to a push consumer and buffered for data collection by a pull consumer.

Applications for the Event Service within the Satlinx architecture include the interface between the user interface modules and the virtual instruments and virtual functions displayed, the fault manager module and the fault generating modules in the system, and various modules providing repetitive data and status and their upstream clients.

Most of these applications are implemented with push suppliers and push consumers. Other configurations may be used as appropriate.

1.3 Satlinx Components

Satlinx objects share the following attributes:

- Self-contained, named objects.
- Self-configuring.
- Self-referencing.
- Similar objects export a standard interface.
- Fully distributable on disparate platforms.

A system can be built from as many or few Satlinx objects as required. Since all objects are self-referencing and self-configuring, the entire system will self-configure once the list of objects and all the interconnections are defined. Specific support for individual instruments need not and should not be coded in other objects. If variations exist and must be handled, the objects can exchange capabilities and requirements and effectively negotiate a working interface. Due to the fine partitioning of the system such negotiation should be minimal and in respect to very narrow feature sets.

1.3.1 Virtual Function Objects

Virtual functions implement specific functions within the system. Some are persistent and are present as long as the system is active. Others are transient, and are instantiated only when required.

Virtual functions use the services provided by other virtual functions and virtual instruments. This isolates the functions from the specifics of the actual devices used in the installation.

Virtual functions may vary from the simplistic to the highly complex, and may connect to just a few other virtual devices or many. Where possible, highly complex operations should be divided between a number of virtual functions with a master controlling function over all. This highly modular, compartmentalized approach will reduce the maintenance demands of the system.

1.3.2 Virtual Task Objects

Virtual task objects implement functions within the system that are subject to scheduled control. They implement an interface that allows them to be loaded, configured and started at a preset time, and include functionality such as track managers, and various test modules.

1.3.3 Virtual Instrument Objects

Virtual instruments objects provide a standardized functional interface for the various types of instruments used to build a complete station control system. The specific instruments used may depend upon performance, price, availability, or customer preference. The functionality, however, is similar.

Virtual instruments implement the complete functionality for a generic system component and export a single interface. Some functionality may be optional, in which case virtual instrument should maintain a capability profile. Virtual functions will thus be able to tailor themselves accordingly.

Virtual instruments map the generic interface to the specifics of the physical instrument objects. Some virtual instruments are implemented by several physical instruments. Others use just a subset of the capabilities of the physical instrument, or correspond to single components of a redundant physical instrument subsystem. Yet others are mapped one-to-one to physical instruments.

Virtual instruments serve as a reductive adaptation layer, in that they reduce the instrument functionality to its simplest form. The command set exported by a virtual instrument will most likely be smaller but more ideal than the physical device it represents. This also serves to reduce complexity in the system by separating out unrelated capabilities of the physical device for use by a different virtual instrument. It also reduces the variations in the abstract functional layers of the system, thus minimizing the effects of adding or changing generic functionality and making it simpler to share such improvements among differing installations.

Where there are several combinations of instrument personalities that may be used to implement the functionality of a virtual instrument, the virtual instrument object will use the CORBA _narrow() method to identify the exact type of the subordinate module. A smart virtual instrument will be able to tailor itself to make appropriate use of the subordinate modules to implement the required functions and features.

1.3.4 Physical Instrument Objects

Physical instrument objects encapsulate the capabilities of a specific device. Thus, there is a one-to-one mapping between a physical instrument object and the device it controls. At the same time it exports a standard interface to the virtual instrument objects, according to functional personalities. Some capabilities of the standard interface may be fully supported while others may be emulated within the physical instrument.

The physical instrument object thus serves as an additive adaptation layer. For devices with fewer capabilities than the standard interface exports, the physical instrument object implements a safe buffer for these calls. Virtual instruments therefore need less negotiation to connect safely to the physical instrument. In addition, where devices of similar function use different metrics to represent data, the physical instrument will provide conversion facilities to a standard metric used by the interface. An example of this would be that of recorders, where tape position may be represented as time, inches, centimetres, or some other arbitrary unit.

A major purpose of the physical instrument object is to decouple the actual device from the rest of the system. A device may be limited as to I/O bandwidth, or particularly state-conscious in such a manner as to limit certain operations at some times. The physical instrument object will provide unrestricted, unconstrained access to the device's functions from the rest of the system, while enforcing whatever system and device specific limitations must be placed on these functions.

Physical instrument objects, like the devices they embody, may be comprised of multiple functional personalities. Some of these are related, while others, such as the discrete I/O contacts on the SA3860 ACU, can be used for completely unrelated operations. To support this the physical instrument can export multiple personality interfaces, each of which may be used by a different virtual instrument.

Physical instrument objects communicate with the devices they embody via virtual circuits provided by communications adapters. All the physical instrument needs to know is the name of the communications adapter to use, which it gets from the configuration manager.

1.3.5 Communications Adapter Objects

Communications adapter objects serve as I/O handlers. Each adapter manages one group of connections. These will typically be an IEEE488 bus, a collection of TCP/IP socket connections, a multi-dropped serial link using RS-485, a single point-to-point serial connection, or some other medium or protocol. Some devices may be implemented as PCI cards in a server PC.

The interface exported by a communications adapter is common to all transport types. This makes it transparent to the physical instrument as to which bus or protocol it is using. The communications adapter obtains the list of connected devices from the configuration manager and matches device names to protocol addresses. It then creates a unique virtual circuit object that is used by the physical instrument object to communicate with the device.

In the case of bus-connected devices, a custom communications adapter must be written to translate I/O commands from the physical instrument object to device driver invocations.

1.3.6 User Interface Modules

User interface modules can access exported data and controls in all components: virtual functions, virtual tasks, virtual instruments, physical instruments and communications adapters. Access is subject to authentication.

The interface modules themselves have no specific knowledge of the significance of various controls and data values. They are just user interface engines. Each module exports data and control properties by name. The user interface loads screen definitions that associate named properties from various modules with screen controls and screen layouts.

In addition to aggregated screen definitions, each module can maintain its own set of detailed screen layouts for all its properties. This allows standardized interfaces to each module, while allowing highly customized top-level screen and custom aggregated screens.

1.3.7 Authentication Server

The authentication server validates external client access to various objects within the system. Each object supports two levels of access: monitor and control. These correspond to read only and read/write file access. When a client attempts to access an object it will make a request and receive one of the following: full access, monitor access, or no access. Due to the requirements of multiple points of control a client's full access may be revoked at any time and reduced to monitor access.

1.3.8 System Logger

Built into the base classes inherited by all objects are logging methods that filter log calls by type and severity. Exported methods set the type mask and levels. Log calls that do not pass the filter requirements are effectively ignored with minimal system overhead. Calls that pass are sent to the system logger and are saved in files. Within the logger, messages are directed to various files based on the source module, log type, and log level, based on a set of filters. Messages are logged to as many files as the filters permit. (This approach is similar to the Unix *syslog* facility.)

The logging method implemented within the System Logger is essentially one of a filtered push. If the filters are correctly written, standard log output from various modules will be available in separate files. The nature of the logged data and the rate at which it is logged has to be coded into the individual modules. Exported methods allow system control of the fitering of logged data by level, category and source module.

1.3.9 Configuration Manager

The configuration manager acts as a repository for the system configuration. The system configuration is created as a text file at present, although a future enhancement is for it to be generated by a system configuration tool based on information provided by the object.

When objects initialize, they interrogate the configuration manager for names of connected objects, and any other startup parameters presented to the configuration tool by the object.

The configuration manager is also responsible for creating the initial objects required by the system. The CORBA framework differentiates between objects and the servants that incarnate them. Objects can be created and published by the Naming Service without the processes that incarnate the objects existing. Activation of the objects and incarnation of the servant processes can be deferred until clients connect to them.

The configuration manager also implements a watchdog function that pings static objects to make sure they are still alive. It will also keep track of transient objects and ping them during their lifetime to verify their existence.

1.4 Module interconnections

There are two types of interconnection between modules in the Satlinx system: static and dynamic.

1.4.1 Static Interconnections

Static interconnections are designed into the system when it is configured, and reflect the hard wiring of the various physical devices in the system. Object references required for static interconnections are obtained by querying the configuration manager for the name of the connected object and the name server for its object reference. Examples of static interconnections are:

- physical instrument objects and communications adapters,
- virtual instrument objects and physical instrument objects,
- virtual instrument objects (such as switches and multiplexers) and the virtual instrument objects they switch.

1.4.2 Dynamic Interconnections

Dynamic interconnections are created while the system is running. Object references required for dynamic interconnections are obtained by querying other modules that can specify which objects are appropriate for the context. Examples of dynamic interconnections are:

- virtual function objects and virtual instrument objects, such as for test and tracking operations,
- the node controller virtual function object and virtual function objects, virtual instrument objects, physical instrument objects, or communications adapters, for aggregated control and monitoring of the node,
- virtual instrument objects and other virtual instrument objects, where settings need to be shared,
- user interface modules and virtual function objects, virtual instrument objects, physical instrument objects, or communications adapters, depending on the nature of the user interface modules.

2 Development Environment

2.1 Operating System

The Satlinx project has been developed on a mix of Windows-NT and Windows 2000 machines. The mix of revisions is as follows:

- Windows-NT 4.0 SP4
- Windows-NT 4.0 SP5
- Windows-NT 4.0 SP6
- Windows 2000 SP1
- Windows 2000 SP2

I am guessing at the NT variants here. This needs to be verified with the rest of the group, if possible.

No coding changes have been necessary to support both operating systems, and no functional or performance differences have been noted.

Deployment of the product will most likely take place on Windows XP. No development or testing has been done to date on this platform, and no comment can be made as to what changes, if any, will be required to support it.

The code has been developed to be easy to port to another operating system. Linux has always been in mind, although no work has been done on Linux, either to build or run the application. Some portions of code have been coded to use Windows API calls, and these will need to be rewritten for another OS. Such sections are highly localized, however, and will be self-evident whenever a port begins. The greatest part of any porting effort will be that of switching to another compiler and build environment.

2.2 Tools and Libraries

2.2.1 Visual Studio

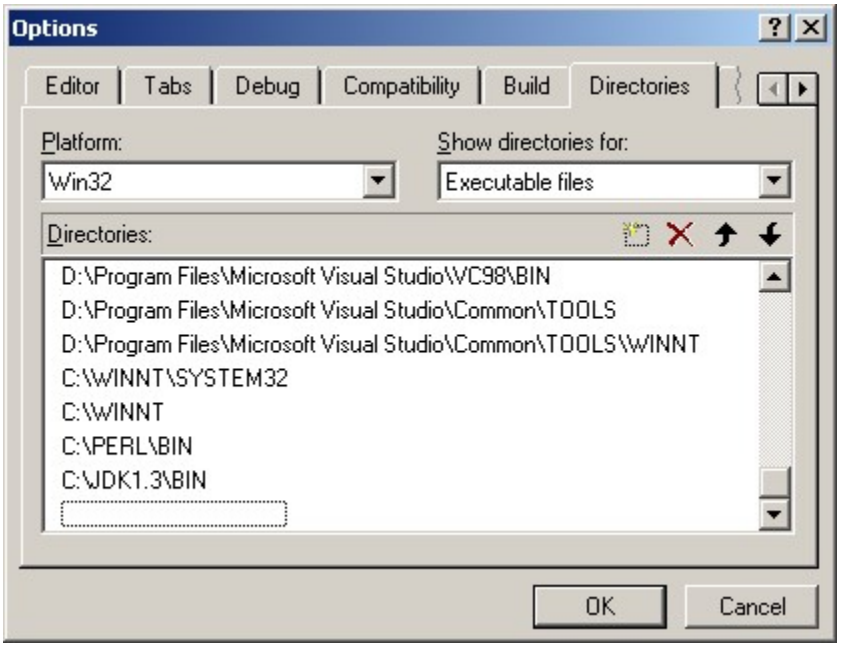
The code was built with Microsoft Visual C++ 6.0 with Service Pack 4. The C++ standard template library that ships with MSVC++ was not used, but the Dinkumware SXL libraries were used instead. This is achieved by adding the SXL include directory to the Visual Studio project paths. This is handled in the project (.DSP) files, not in the workspace.

Once installed, and with service packs applied, executable paths to the Java Development Kit (JDK) and Perl must be added. From the workspace select the menus **Tools→Options** and the select the **Directories** tab. From the **Show directories for:** drop down box select **Executable files**.

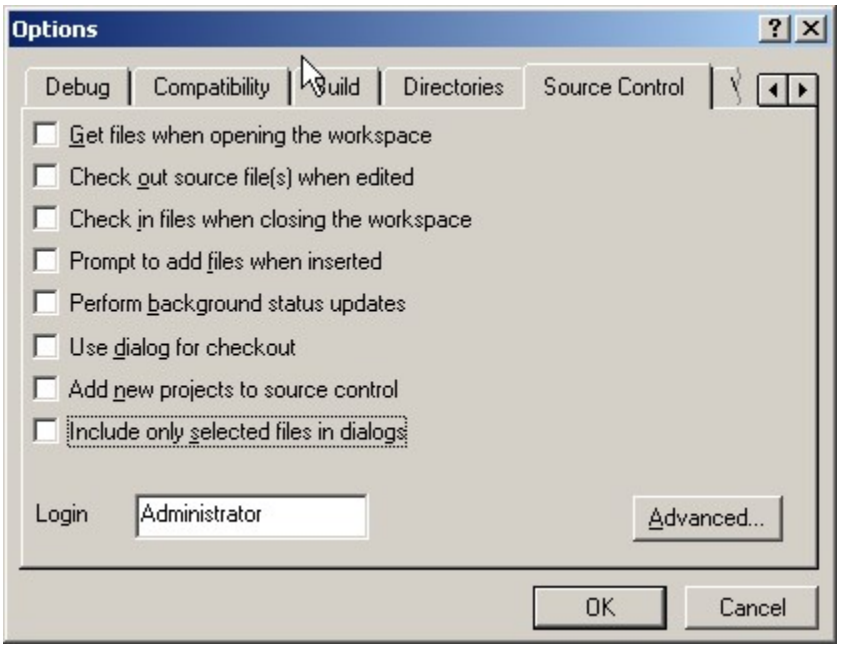
Select a new line in the directory box and enter the path to the Perl executables, typically C:\Perl\bin. Alternatively you may browse to find it.

Select a new line in the directory box and enter the path to the JDK executables, typically C:\JDK1.3\bin for version 1.3 of the JDK. Alternatively you may browse to find it.

The final result should appear similar to the following screenshot, with the Perl and JDK paths added to the executable files directory list.



Another recommended change for Studio is to remove its automatic association with SourceSafe. To do this select the **Source Control** tab, and deselect all the options. The final result should appear similar to the following screenshot.



2.2.2 Orbacus

The Orbacus libraries are required for Java, C++ debug mode, and C++ release mode. The Java libraries are available as pre-built JAR files, and may simply be installed in the correct directory. The C++ libraries must be built.

Orbacus is frequently updated, so it is necessary to keep track of the various releases. Rather than maintain a single source tree for all versions in SourceSafe, it is simpler to build the libraries outside of the project directory tree, and then copy the built libraries and executables into the tree. Versions used to date are

- 4.0.1
- 4.0.3
- 4.0.4
- 4.0.5
- 4.1.0

In SourceSafe there is a directory for each, with the extension indicating the version. Only the current version has no extension, and this is the version that will be linked by the Visual Studio project files.

2.2.2.1 Building and Installing a New Version of Orbacus for C++

Two versions of the libraries must be built. The sequence of building these versions is significant, since the executables are named the same for both, and only release versions of the executables are required. The libraries are named uniquely for debug and release. For this reason, it is necessary to build and install the debug versions first, and then the release versions.

Prior to version 4.1.0 of Orbacus it was necessary to build the JThreads libraries in addition to the Orbacus libraries. If you are building an older version of Orbacus from source be sure to build the relevant version of JThreads first. The associations are:

Orbacus version	JThreads version
4.0.1	1.0.12
4.0.3	1.0.12
4.0.4	1.0.14
4.0.5	1.0.14
4.1.0	2.0.0 (included)

The Orbacus and JThreads sources are packages in ZIP archives. Unzip the distibution archive into a staging directory, or to the root directory of the drive. A main directory will be created in the staging directory named for the product and release:

OB-xxxx	Orbacus for C++
JOB-xxxx	Orbacus for Java
JTC-xxxx	JThreads

2.2.2.1.1 JThreads

N.B. This section only applies to older releases, where JThreads must be built separately.

2.2.2.1.1.1 Setting the Build Options

The build settings for the JThreads package reside in a file named *Make.rules.mak* in <*JTC root*>/*config*. The top section of the file contains the customizable options, and is reproduced with annotations below:

```
# ********************************************************************
#
# Copyright (c) 2000
# Object Oriented Concepts, Inc.
# Billerica, MA, USA
#
# All Rights Reserved
#
# ********************************************************************

# --------------------------------------------------------------------
# Use the following lines to set your installation directories
# --------------------------------------------------------------------

prefix       = C:\OOC            <<<  This must be updated to reflect the installation path. This may
bindir       = $(prefix)\bin          either be the subdirectory in the Satlinx tree, or a staging
libdir       = $(prefix)\lib          directory to be copied into the Satlinx tree
includedir   = $(prefix)\include

# --------------------------------------------------------------------
# Either define OPTIMIZE_SPEED as "yes" to build with speed
# optimization turned on, or define OPTIMIZE_SIZE as "yes" to optimize
# for size, or define DEBUG as "yes" to build with debug
# information. Note that all these settings are mutual exclusive.
#
# NOTE: If you change these options later, you need to completely
# rebuild this distribution. (Using "nmake /f Makefile.mak clean", and
# "nmake /f Makefile.mak" after that.)
# --------------------------------------------------------------------

#OPTIMIZE_SPEED=yes          <<<  This is how these options must be set to build the debug
#OPTIMIZE_SIZE=yes           <<<  libraries. For the release libraries comment out 'DEBUG=yes'
DEBUG=yes                   <<<  and uncomment 'OPTIMIZE_SPEED=yes'.

# --------------------------------------------------------------------
# Define OLD_IOSTREAM as "yes" if you want to use the old iostream
# library, i.e., if you want to use '#include <iostream.h>" instead of
# "#include <iostream>".
#
# NOTE: If you want to check your application with "purify", please
# use the old iostream library instead of the new one. If the new one
# is used, purify will detect lots of non-existent memory problems.
# --------------------------------------------------------------------

#OLD_IOSTREAM=yes            <<<  Make sure this option is commented out.

# --------------------------------------------------------------------
# If you want to make use of a Dynamic Link Library (DLL) instead of a
# static library in the packages where this is supported, uncomment
# the following line to define DLL as "yes". Note that in this case
# you need "perl" for the compilation process.
# --------------------------------------------------------------------

DLL=yes                      <<<  Make sure this option is set to yes.

# --------------------------------------------------------------------
# If you want support for the stop, suspend and resume operations,
# uncomment the following line to define WITH_JTC_STOP as "yes".
#
# NOTE: Don't uncomment the following line for best performance.
# --------------------------------------------------------------------

#WITH_JTC_STOP=yes           <<<  Make sure this option is commented out.
```

```
# ------------------------------------------------------------------------
# If you are compiling JThreads/C++, and you want no iostream support,
# uncomment the following line to define WITH_JTC_NO_IOSTREAM as "yes".
#
# NOTE: This is intended for use with ORBacus/E. When compiled without
# iostream support, JThreads/C++ and ORBacus/E will not depend on the
# iostream library, allowing for smaller executables.
# ------------------------------------------------------------------------

#WITH_JTC_NO_IOSTREAM=yes          ←←←  Make sure this option is commented out.

# ------------------------------------------------------------------------
# Experts only: You can modify the lines below, for example to change
# the settings for code optimizations.
# ------------------------------------------------------------------------
```

2.2.2.1.1.2 Build and Install

The build process uses makefiles and Microsoft NMAKE. Visual C++ must be installed, and the path to the command-line tools included in the system path.

Once the library is built for either debug or release it must be installed, and then the source tree cleaned before modifying the settings and rebuild the other version. From the root of the JThreads source package enter the following sequence of commands:

```
nmake /f Makefile.mak
nmake /f Makefile.mak install
nmake /f Makefile.mak clean
```

This will build the library, copy the needed files to the installation directories, and then clean up all the object and intermediate files.

In summary, the steps to build the JThreads libraries are as follows:

1. Configure the build options for creating a debug library
2. Run *`nmake /f Makefile.mak`* to build the library
3. Run *nmake /f Makefile.mak* install to install the library
4. Run *`nmake /f Makefile.mak`* `clean` to delete the intermediate files
5. Reonfigure the build options for creating a release library
6. Run *`nmake /f Makefile.mak`* to build the library
7. Run *nmake /f Makefile.mak* install to install the library
8. Run *`nmake /f Makefile.mak`* `clean` to delete the intermediate files

2.2.2.1.2 Orbacus

2.2.2.1.2.1 Setting the Build Options

The build settings for the Orbacus package reside in a file named *Make.rules.mak* in *<OB root>/config*. The top section of the file contains the customizable options, and is reproduced with annotations below:

```
# ********************************************************************
#
# Copyright (c) 2000
# Object Oriented Concepts, Inc.
# Billerica, MA, USA
#
# All Rights Reserved
#
# ********************************************************************

# --------------------------------------------------------------------
# Use the following lines to set your installation directories
# --------------------------------------------------------------------

prefix       = C:\OOC            <<<  This must be updated to reflect the installation path. This may
bindir       = $(prefix)\bin     <<<  either be the subdirectory in the Satlinx tree, or a staging
libdir       = $(prefix)\lib     <<<  directory to be copied into the Satlinx tree
includedir   = $(prefix)\include
idldir       = $(prefix)\idl

# --------------------------------------------------------------------
# Either define OPTIMIZE_SPEED as "yes" to build with speed
# optimization turned on, or define OPTIMIZE_SIZE as "yes" to optimize
# for size, or define DEBUG as "yes" to build with debug
# information. Note that all these settings are mutual exclusive.
#
# NOTE: If you change these options later, you need to completely
# rebuild this distribution. (Using "nmake /f Makefile.mak clean", and
# "nmake /f Makefile.mak" after that.)
# --------------------------------------------------------------------

#OPTIMIZE_SPEED=yes          <<<  This is how these options must be set to build the debug
#OPTIMIZE_SIZE=yes           <<<  libraries. For the release libraries comment out 'DEBUG=yes'
DEBUG=yes                   <<<  and uncomment 'OPTIMIZE_SPEED=yes'.

# --------------------------------------------------------------------
# Define OLD_IOSTREAM as "yes" if you want to use the old iostream
# library, i.e., if you want to use '#include <iostream.h>" instead of
# "#include <iostream>".
#
# NOTE: If you want to check your application with "purify", please
# use the old iostream library instead of the new one. If the new one
# is used, purify will detect lots of non-existent memory problems.
# --------------------------------------------------------------------

#OLD_IOSTREAM=yes            <<<  Make sure this option is commented out.
```

```
# ------------------------------------------------------------------------
# If you want to make use of a Dynamic Link Library (DLL) instead of a
# static library in the packages where this is supported, uncomment
# the following line to define DLL as "yes". Note that in this case
# you need "perl" for the compilation process.
#
# NOTE: If you want to compile ORBacus as a DLL you must also compile
# JThreads/C++ as a DLL.  When using the DLL version of ORBacus or
# JThreads/C++, you must not specify a statically linked runtime
# library with the /MT, /MTd, /ML or /MLd compiler flags.
# ------------------------------------------------------------------------

DLL=yes                                 ←←←  Make sure this option is set to yes.
```

The section below is only for older versions of Orbacus, before 4.1.0
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

```
# ------------------------------------------------------------------------
# If you want to compile ORBacus with JThreads/C++ support, uncomment
# the following line to define WITH_JTC as "yes".
#
# NOTE: JThreads/C++ is required for the Event Service and the
# IMR. JThreads/C++ is a separate software package, and must be
# installed before ORBacus.
# ------------------------------------------------------------------------

WITH_JTC=yes

# ------------------------------------------------------------------------
# If JThreads/C++ support is enabled, use the following line to set
# the JThreads/C++ installation path.
# ------------------------------------------------------------------------

JTC_DIR=$(prefix)
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
End of old Orbacus version options

```
# ------------------------------------------------------------------------
# Experts only: You can modify the lines below, for example to change
# the settings for code optimizations.
# ------------------------------------------------------------------------
```

2.2.2.1.2.2 Build and Install

The build process uses makefiles and Microsoft NMAKE. Visual C++ must be installed, and the path to the command-line tools included in the system path.

Once the library is built for either debug or release it must be installed, and then the source tree cleaned before modifying the settings and rebuild the other version. From the root of the Orbacus source package enter the following sequence of commands:

```
nmake /f Makefile.mak
nmake /f Makefile.mak install
nmake /f Makefile.mak clean
```

This will build the libraries and executables, copy the needed files to the installation directories, and then clean up all the object and intermediate files.

In summary, the steps to build the Orbacus libraries are as follows:

1. Configure the build options for creating the debug libraries
2. Run *`nmake /f Makefile.mak`* to build the libraries and executables
3. Run *`nmake /f Makefile.mak`* `install` to install the libraries and executables
4. Run *`nmake /f Makefile.mak`* `clean` to delete the intermediate files
5. Reonfigure the build options for creating the release libraries
6. Run *`nmake /f Makefile.mak`* to build the libraries and executables
7. Run *`nmake /f Makefile.mak`* `install` to install the libraries and executables. Note that the release versions of the executables will replace the debug versions. This is the desired effect.
8. Run *`nmake /f Makefile.mak`* `clean` to delete the intermediate files

2.2.2.2 Installing a New Version of Orbacus for Java

The Orbacus Java libraries are supplied both as source and prebuilt and packaged in JAR files. The prebuilt versions are compatible with our development environment, so it is not necessary to build them for deployment.

To install the JAR files unzip the distribution archive The IDL files included with the JAR files duplicate the IDL supplied with the C++ source, so it is not necessary to copy these over. Copy the contents of the LIB subdirectory into a new subdirectory of the rest of the Orbacus distribution. The build process for the Java modules looks for the JAR files in *<devroot>\OOC\jars*, so this is where they must be copied.

2.2.3 Xerces XML Parser

2.2.3.1 Xerces for C++

The Xerces XML parser libraries are supplied both as source and prebuilt for Windows platforms. The prebuilt versions are compatible with our development environment, so it is not necessary to build them for deployment.

Over the life of the Satlinx project we have used the following versions of the Xerces libraries:

- 1.1.0
- 1.3.0
- 1.5.1

Rather than maintain a single source tree for all versions in SourceSafe, it is simpler to use the pre-built libraries and copy them into the tree. In SourceSafe there is a directory for each, with the extension indicating the version. Only the current version has no extension, and this is the version that will be linked by the Visual Studio project files.

2.2.3.1.1 Building a new version

Should it prove necessary to build a version from source, unzip the source archive into a staging directory. The entire package is defined in a Visual Studio workspace file found in the directory *<Xerces source root>\Projects\Win32\VC6*. This should be loaded, and the required pieces built. The only library from the whole package we use is the SAX parser, although some of the sample programs may be useful if extensive changes or enhancements to the XML subsystem are required.

2.2.3.1.2 Installing the Xerces libraries

To install the Xerces XML parser libraries unzip the Windows binary distribution archive into a staging directory. The entire package will be extracted under a single distribution root directory named xerces-cX_X_X-win32, where X_X_X corresponds to the version number. Rename this directory to XML, and then copy or move it to the development root directory of the Satlinx project. It is assumed that the previous version of the XML libraries has been deleted or renamed before copying over the new release.

2.2.3.2 Xerces for Java

The Xerces for Java package is supplied as a zipfile containing documentation, samples, and the 'xerces.jar' archive containing the actual library. The zipfile may be unzipped anywhere desired, but the jarfile must be placed in *<Xerces source root>/jars*.

2.2.4 Dinkumware SXL Visual C++ Libraries

The Dinkumware SXL Visual C++ Libraries are fixed versions of the standard libraries that ship with MSVC++. These were adopted when there were some obscure problems occurring and they went away when these libraries were used. Since then some of the

problems recurred, but not deterministically enough to tie down. It is possible that there is a floating memory corruption occurring that moves as code is altered. Most of the time it is benign, but occasionally it causes aberrant behaviour.

Whatever the reason for adopting the SXL libraries, they are currently part of the system. Occasionally they are updated. Versions used to date are:

- 2.33
- 3.08

As with the other third party libraries, rather than maintain a single source tree for all versions in SourceSafe, it is simpler to build the libraries outside of the project directory tree, and then copy the built libraries and executables into the tree.

In SourceSafe there is a directory for each, with the extension indicating the version. Only the current version has no extension, and this is the version that will be linked by the Visual Studio project files.

2.2.4.1 The SXL Distribution Package

The library is supplied as a self-extracting archive. Execute the distribution code and it will prompt for the name of the root directory for the tree. This may be a staging directory, or directly into the Satlinx development tree. It is probably preferable to use a staging directory since debug versions of the library need to be built.

The library comes with three versions pre-built: the statically linked single-threaded library (sxl.lib), the statically linked multi-threaded library (sxlmt.lib), and the dynamically linked multi-threaded library (sxlrt.lib and sxlrt308.dll). The only one of these required for the Satlinx project is the dynamically linked multi-threaded library. The debug versions of these libraries must be built as described below.

The directory tree contains several subdirectories:

`bin`	holds the library DLLs as well as batch scripts that can be executed in any directory (v50setup or v60setup adds bin to PATH)
`html`	holds the HTML pages for the on-line documentation, rooted in index.html
`include`	holds the include files, additions or replacements to the standard headers needed to implement this library atop the existing MS include files

`lib`	holds sxl.lib (the statically linked single-threaded library), sxlmt.lib (the statically linked multi-threaded library), and sxlrt.lib (the import library for the DLL implementation of the library), the additions to the object library needed to implement this library atop the existing MS libraries
`sxl`	holds library source files that replace or supplement existing MS object library source files (supplies object files for lib\sxl.lib)
`test`	a scratch directory, with one or two tiny test files

2.2.4.2 Building a new version

To rebuild the release version of all of the libraries, execute *`sxlbuild.bat`* in the root of the directory tree. The file *`sxlbuild.log`* captures any compiler, linker or lib reports.

To create the debug builds of the three library versions, execute *`sxldbuild.bat`* in the root of the directory tree. The file *`sxlbuild.log`* captures any compiler, linker, or lib reports.

A successful build produces three debug versions of the library: the statically linked single-threaded library (sxld.lib), the statically linked multi-threaded library (sxlmtd.lib), and the dynamically linked library (sxlrtd.lib and sxlrt308d.dll). The only one of these required for the Satlinx project is the dynamically linked multi-threaded library.

2.2.4.3 Installing the SXL libraries

The Satlinx project files look for the SXL libraries and include files in the SXL subdirectory of the Satlinx development tree. After the required libraries have been built, copy the following subdirectories from the staging directory to the SXL directory in the Satlinx development tree:

- `bin` (only sxlrt308.dll and sxlrt308d.dll are required)
- include
- `lib` (only sxlrt.lib and sxlrtd.lib are required)
- `html` (optional, for documentation only)

If this is an update from an earlier version, the old SXL directory should be deleted or renamed before copying over the new release.

2.2.5 Java Development Kit

The Java SDK is supplied as a self-installing archive. Simply run the installation executable to install it. It is best to remove all prior versions of the SDK before installing a new one.

The path to the bin directory of the SDK tree must be added to Visual Studio, as described above. The path must also be added to the system environment, as only the path necessary to execute java code is added to the path by default, not the parts needed to build java code.

2.2.6 Other required packages

2.2.6.1 PERL

Perl is supplied as a Windows Installer Package. Simply execute it to install the tool.

The path to the bin directory of the Perl tree must be added to Visual Studio, as described above.

If you are developing under Windows NT, you need to install the Windows Installer tool prior to installing Perl. This can be found in the Perl subdirectory of the Satlinx development tree.

2.3 Project Structure

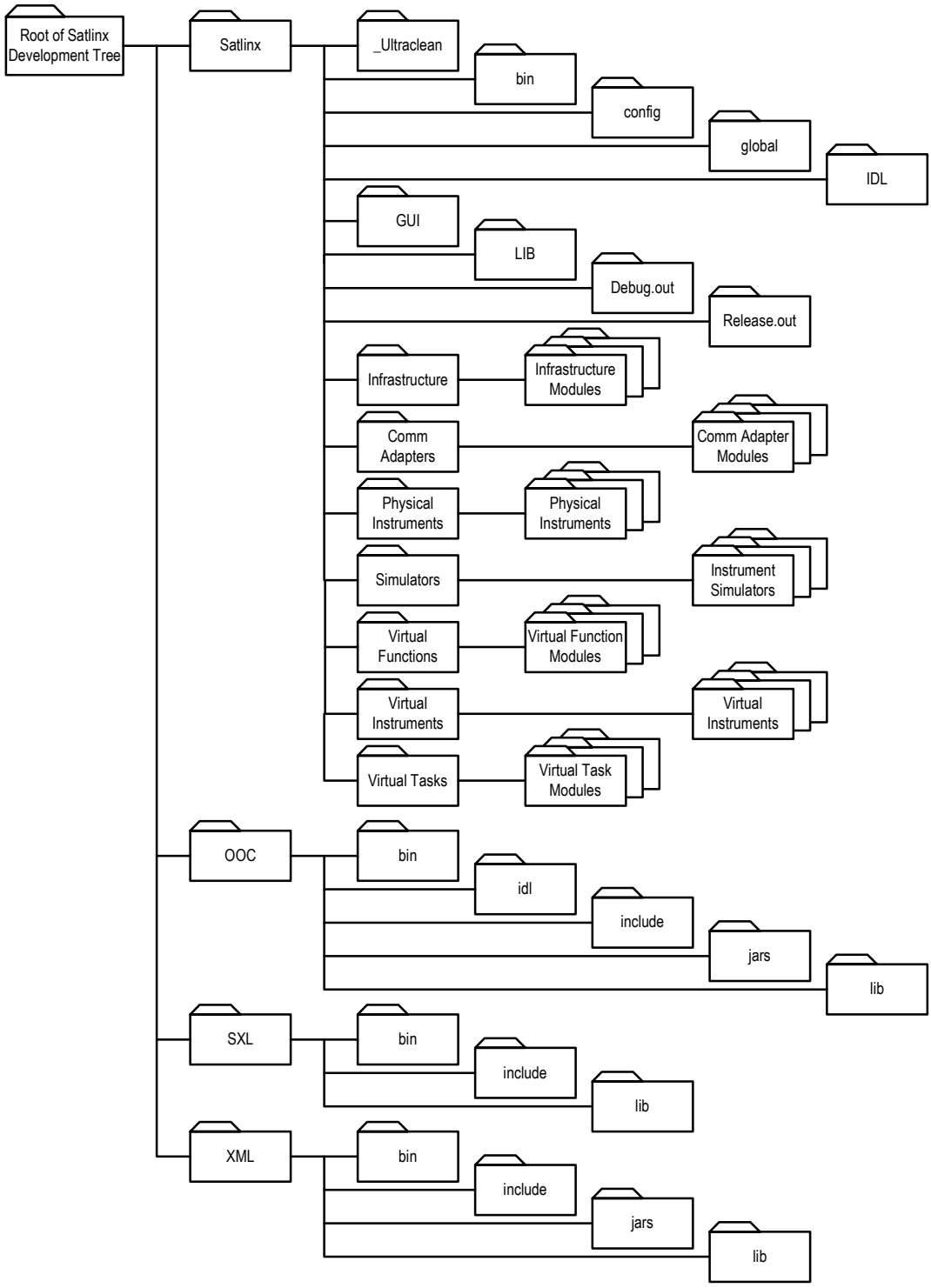
2.3.1 Source Control Layout

The source control directory tree and methodologies are discussed in detail in chapter 6.

2.3.2 Working Directories

2.3.2.1 Source

The Satlinx development tree is represented below:



This layout depicts the basic minimum directory layout for Satlinx development. Additional directories may be added for other subsystems or module groupings. If additional third party libraries are incorporated they too would need their own subdirectories in the tree to ensure that relative paths can be used for the build process.

The three third party directories — OOC, SXL, and XML — have been discussed above. Each of the main subdirectories of the Satlinx tree will now be considered.

`_Ultraclean`	This is a special project for inclusion in the Visual Studio Satlinx workspace. Building this project deletes all generated files from the source tree.
`bin`	This directory contains scripts and executables required by the build process, both Studio projects and makefiles.
`CommAdapters`	This contains the various communication adapter module sources.
`config`	This directory contains configuration files required by the build process.
`Debug.out`	This is the output directory for all intermediate files generated by a debug build. Under Visual Studio the executables and DLLs are built into target directories, also in the Satlinx development tree, although not shown in the above diagram. See the section on *Staging Directories* for details.
`global`	This is the home for all include files that need global visibility. This includes infrastructure class header files and system constants and data structure definitions, as well as headers for home-grown utility libraries.
`GUI`	This is the home of the GUI subsystem source.

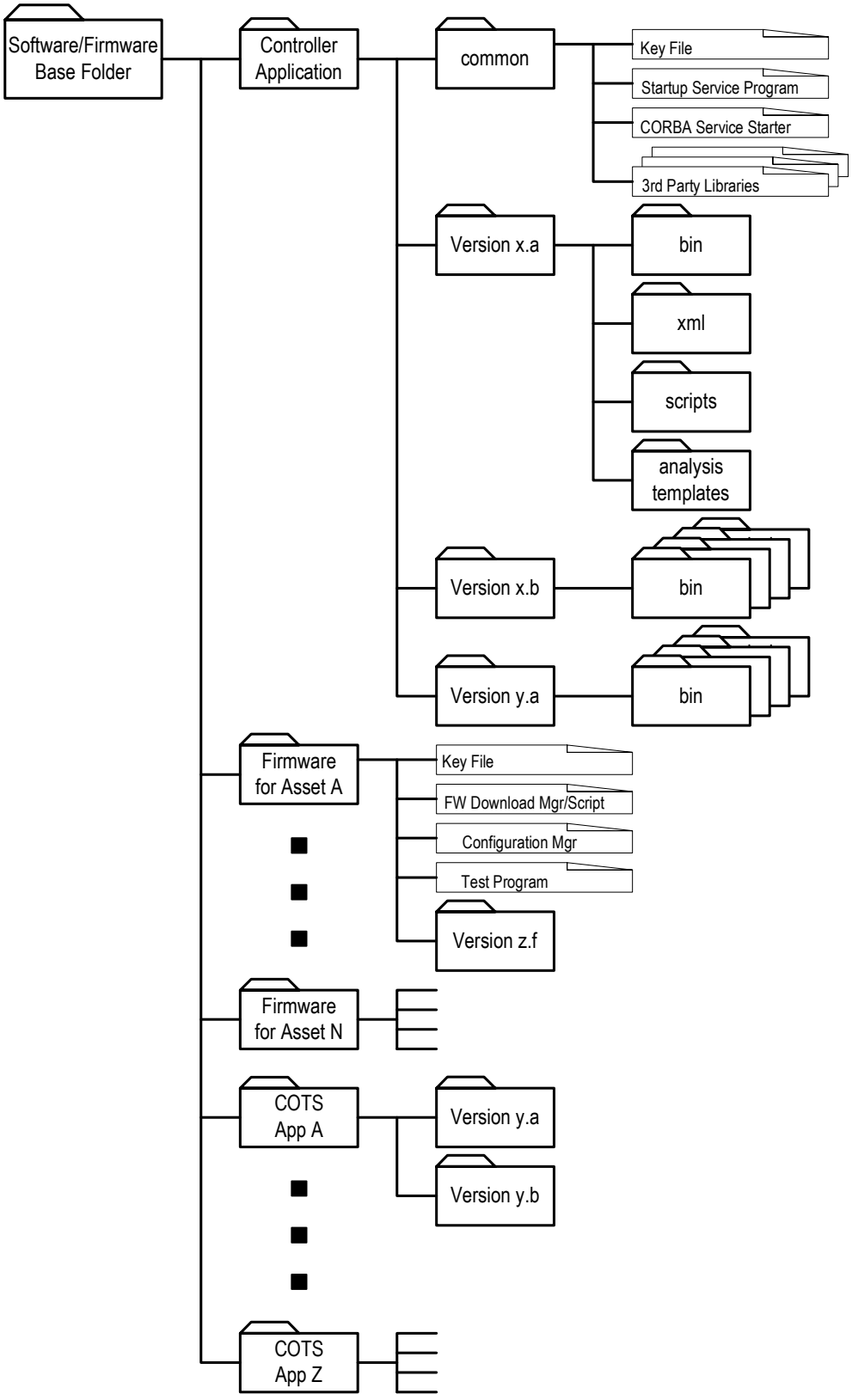
`IDL`	This contains all IDL files with global scope. This includes all the infrastructure interfaces, as well as personality IDL for instruments and task and function IDL. Under this directory is placed all the generated files. Global IDL files have their stubs and skeletons generated into the CPP subdirectory, and mezzanine IDL files have their stubs and skeletons generated into the CPP/MEZZANINE subdirectory. This approach keeps generated C++ code from getting mixed up with developed source.
`Infrastructure`	This contains the various infrastructure module sources. It is also the appropriate location to place home-grown utility libraries.
`LIB`	This is the output directory for DLL import libraries for those base classes that are intended to be implicitly linked. This is primarily the DLLs containing the implementation of base classes.
`PhysicalInstruments`	This contains the various physical instrument module sources.
`Release.out`	This is the output directory for all intermediate files generated by a release build. Unlike the debug build, the executables and DLLs are also built into this directory tree and moved to a staging directory if one is specified. See the section on *Staging Directories* for details.
`Simulators`	This contains the various instrument simulator module sources.
`VirtualFunctions`	This contains the various virtual function module sources.
`VirtualInstruments`	This contains the various virtual instrument module sources.
`VirtualTasks`	This contains the various virtual task module sources.
`1.0`	Staging directory (see below)
`Bin`	Staging directory (see below)

2.3.2.2 Staging Directories

When Satlinx is built it must be staged. All executables and DLLs must be in the correct relative locations. The Studio project files stage the debug version into the directories *<development root>/bin* and *<development root>/1.0/bin*. These are the defaults for the two target directories for Satlinx binaries, which we can refer to as the common binaries, and the version specific binaries.

The software is designed to operate from the common binaries' directory without the paths being added to the path environment variables.

The release version is not staged by default, but can be staged if the environment variable `STOVOKOR_STAGE` is set. This would point to the controller application directory subtree of the Satlinx software tree, as shown below:


Software/Firmware Base Folder
Controller Application
common
Key File
Startup Service Program
CORBA Service Starter
3rd Party Libraries
Version x.a
bin
xml
scripts
analysis templates
Version x.b
bin
Version y.a
bin
Firmware for Asset A
Key File
FW Download Mgr/Script
Configuration Mgr
Test Program
Version z.f
Firmware for Asset N
COTS App A
Version y.a
Version y.b
COTS App Z

2.3.2.2.1 Common Binaries

The common binaries directory — *<controller application>/common* —contains those files that do not change from version to version. At the present this includes all third party libraries and the following Satlinx executables:

- CorbaService.exe
- Stovokor.exe
- Satlinx.jar
- SatBeans.jar

It is likely that the Java binaries will be version dependent, and will need to relocate to the version specific tree. It is also possible that at least the Orbacus libraries, which are updated quite frequently, will also need to be relocated to the version specific tree, although this may require adjustments to the startup program to ensure that the libraries are in the default search path for the executables.

2.3.2.2.2 Version Specific Binaries

The version specific binaries live in the directory *<controller application>/<version >/bin*, and include the rest of the executables and DLLs making up the Satlinx system.

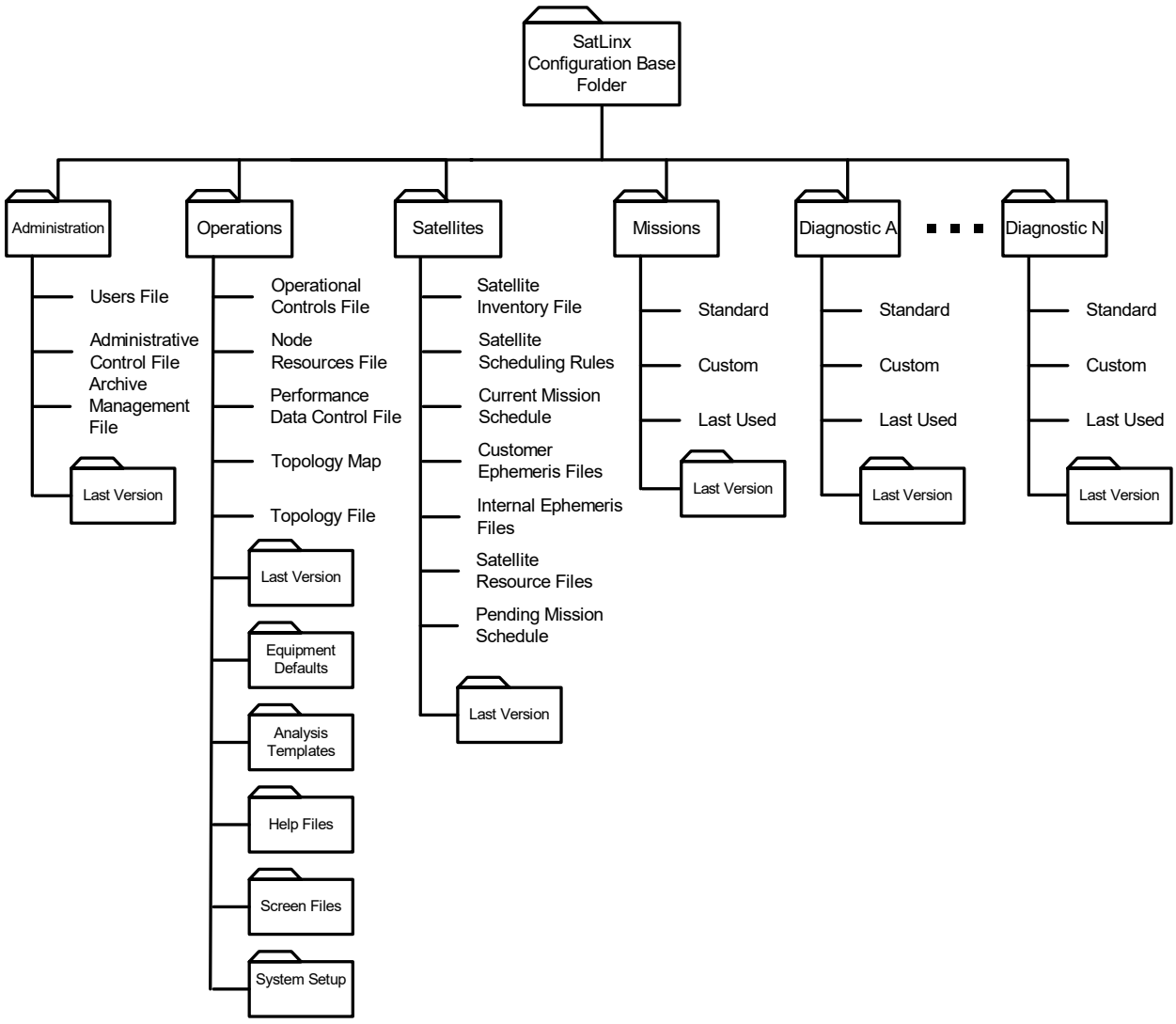
2.3.2.2.3 Other Version Specific Files

The remaining subdirectories of the version subtree are for deployment of the software. When shipped, the factory versions of all configuration files, scripts and templates are included under the software tree, and copied to the configuration tree during the installation process. The directories and their contents are as follows:

`xml`	This is another subtree, and contains the default and instance configuration files and the screen definition files.
`scripts`	Any shell scripts, batch files, Perl or Python files are stored here.
`Analysis Templates`	All Excel templates used by the Report Generator are stored here, along with template files for other applications that might be added to the system.

2.3.2.3 Configuration

The Satlinx system configuration directory tree is shown below:



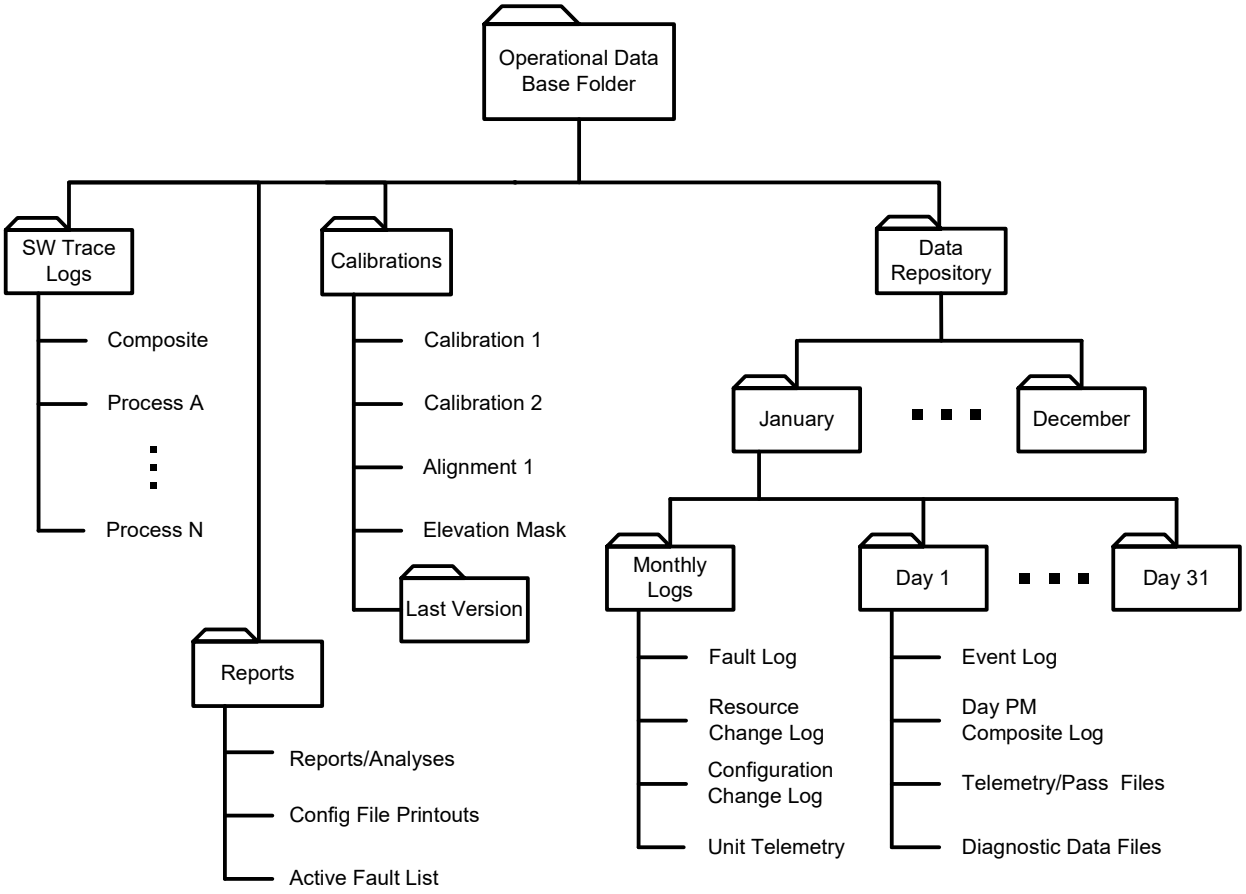
The installation process builds this tree, and copies factory shipped configuration files into the appropriate directories. The factory default files shipped with the software and their target directories are:

Module default configuration XML	➔	Operations/Equipment Defaults
Instance configuration templates	➔	Operations/System Setup
Excel report templates	➔	Operations/Analysis Templates
Screen definition XML	➔	Operations/Screens

Satellite resource files	➔	Satellites
Standard mission profiles	➔	Missions
Operating Scripts (ie dialer)	➔	Operations/Scripts
Help files	➔	Operations/Help

2.3.2.4 Data Repository

The data repository contains files generated by the Satlinx modules. The layout of this tree is reproduced below:



No data is deployed in this tree, but helper methods in the base classes facilitate placement of data files in the appropriate part of the tree, and the configuration manager creates this tree when necessary.

2.3.3 Visual Studio Workspace and Project Files

2.3.3.1 The Visual Studio Workspace

The Visual Studio Workspace file contains the names of projects and project interdependencies. This is a simple text file and is most easily edited by hand. Visual Studio has a nasty habit of trying to re-engineer the system if allowed to edit project and workspace files itself. This file is *LEAST* subject to this problem and may be edited by the IDE if desired. If the IDE is used to change it, remember to close the workspace or exit from Visual Studio before checking the file back into Visual SourceSafe, since the IDE doesn't store the file until it is closed. This is a subset of the file:

```
Microsoft Developer Studio Workspace File, Format Version 6.00
# WARNING: DO NOT EDIT OR DELETE THIS WORKSPACE FILE!

###############################################################################

Project: "ACU_PI_386x"=.\PhysicalInstruments\ACU_PI_386x\ACU_PI_386x.dsp - Package Owner=<4>

Package=<5>
{{{
}}}

Package=<4>
{{{
    Begin Project Dependency
    Project_Dep_Name PhysicalInstrument
    End Project Dependency
    Begin Project Dependency
    Project_Dep_Name SComponent
    End Project Dependency
}}}

###############################################################################

Project: "ACU_VI"=.\VirtualInstruments\ACU_VI\ACU_VI.dsp - Package Owner=<4>

Package=<5>
{{{
}}}

Package=<4>
{{{
    Begin Project Dependency
    Project_Dep_Name SComponent
    End Project Dependency
}}}

###############################################################################
```

Care must also be taken with the automatic source control features of Visual Studio. It has a nasty habit of insinuating itself into the workspace and project files. If this happens, delete all text between the '{{{' and }}} in Package <5>.

2.3.3.2 The Project Files

The Visual Studio Project (DSP) files contain the individual project file and build settings. They are **VERY** subject to damage by allowing the IDE to change the settings. They have been set up and standardized to have the same build settings. Please edit them by hand and only do so to add files to the project.

The top section of the file contains project name information, followed by build information for the release and debug versions. After that are the file inclusions, followed by custom build sections where the commands to build the IDL reside. This last section is superfluous and may be removed **if the makefile has been properly implemented**, since this function is handled by the makefile architecture before the DSP file is executed.

To create a new project, copy an existing DSP file from a project of similar type and configuration. Bring the file up in a text editor and begin. Change the necessary project name instances throughout the top section. Change the target names in the two build sections. You should avoid changing anything else in the build sections unless absolutely necessary. If you chose the source well, all the libraries and DLLs should be the same. Next, find the section titled '# Begin Group "Source Files"' and add or remove file names from the next section. Last, find the section titled '# Begin Group "IDL Files"' which will be after the end of the 'Source Files' group. If you have built a proper makefile, you may remove the custom build section if it exists. It is STRONGLY suggested that a makefile be created for each module, with the proper IDL references in it. This is a convenience now, but will be a necessity when building releases from source control. It also helps to resolve dependencies, which relying on DSP files cannot do. The following is an example of a DSP file used to build the Miteq LNA controller PI:

```
# Microsoft Developer Studio Project File - Name="LNAC_PI_Miteq" - Package Owner=<4>
# Microsoft Developer Studio Generated Build File, Format Version 6.00
# ** DO NOT EDIT **

# TARGTYPE "Win32 (x86) Dynamic-Link Library" 0x0102

CFG=LNAC_PI_Miteq - Win32 Debug
!MESSAGE This is not a valid makefile. To build this project using NMAKE,
!MESSAGE use the Export Makefile command and run
!MESSAGE
!MESSAGE NMAKE /f "LNAC_PI_Miteq.mak".
!MESSAGE
!MESSAGE You can specify a configuration when running NMAKE
!MESSAGE by defining the macro CFG on the command line. For example:
!MESSAGE
!MESSAGE NMAKE /f "LNAC_PI_Miteq.mak" CFG="LNAC_PI_Miteq - Win32 Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "LNAC_PI_Miteq - Win32 Release" (based on "Win32 (x86) Dynamic-Link Library")
```

```
!MESSAGE "LNAC_PI_Miteq - Win32 Debug" (based on "Win32 (x86) Dynamic-Link Library")
!MESSAGE

# Begin Project
# PROP AllowPerConfigDependencies 0
# PROP Scc_ProjName ""
# PROP Scc_LocalPath ""
CPP=cl.exe
RSC=rc.exe

!IF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Release"

# PROP BASE Use_MFC 0
# PROP BASE Use_Debug_Libraries 0
# PROP BASE Output_Dir "Release"
# PROP BASE Intermediate_Dir "Release"
# PROP BASE Target_Dir ""
# PROP Use_MFC 0
# PROP Use_Debug_Libraries 0
# PROP Output_Dir "..\..\Release.out\LNAC_PI_Miteq"
# PROP Intermediate_Dir "..\..\Release.out\LNAC_PI_Miteq"
# PROP Ignore_Export_Lib 0
# PROP Target_Dir ""
# ADD BASE CPP /nologo /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_MBCS" /FD /c
# ADD CPP /nologo /MD /W3 /WX /GR /GX /O2 /I "." /I "..\..\IDL\CPP" /I
"..\..\IDL\CPP\Mezzanine" /I "..\..\global" /I "..\..\..\sxl\include" /I
"..\..\..\xml\include" /I "..\..\..\ooc\include" /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D
"_MBCS" /FD /Zm200 /c
# ADD BASE RSC /l 0x409 /d "NDEBUG"
# ADD RSC /l 0x409 /d "NDEBUG"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
LINK32=link.exe
# ADD BASE LINK32 kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib /nologo /dll
/machine:I386
# ADD LINK32 SComponent.lib CommAdapt.lib PhysicalInstrument.lib sxlrt.lib ob.lib
CosNaming.lib CosEvent.lib jtc.lib xerces-c_1.lib /nologo /dll /machine:I386
/libpath:"..\..\lib" /libpath:"..\..\..\sxl\lib" /libpath:"..\..\..\xml\lib"
/libpath:"..\..\..\ooc\lib"
# Begin Special Build Tool
PostBuild_Cmds=..\..\bin\scp $(OutDir) LNAC_PI_Miteq.dll
# End Special Build Tool

!ELSEIF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Debug"

# PROP BASE Use_MFC 0
# PROP BASE Use_Debug_Libraries 1
# PROP BASE Output_Dir "Debug"
# PROP BASE Intermediate_Dir "Debug"
# PROP BASE Target_Dir ""
# PROP Use_MFC 0
# PROP Use_Debug_Libraries 1
# PROP Output_Dir "..\..\..\1.0\bin"
# PROP Intermediate_Dir "..\..\Debug.out\LNAC_PI_Miteq"
# PROP Ignore_Export_Lib 0
# PROP Target_Dir ""
# ADD BASE CPP /nologo /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D "_MBCS"
/FD /GZ /c
# ADD CPP /nologo /MDd /W3 /Gm /GR /GX /ZI /Od /I "." /I "..\..\IDL\CPP" /I
"..\..\IDL\CPP\Mezzanine" /I "..\..\global" /I "..\..\..\sxl\include" /I
"..\..\..\xml\include" /I "..\..\..\ooc\include" /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D
"_MBCS" /FD /GZ /Zm200 /c
```

```
# ADD BASE RSC /l 0x409 /d "_DEBUG"
# ADD RSC /l 0x409 /d "_DEBUG"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
LINK32=link.exe
# ADD BASE LINK32 kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib /nologo /dll /debug
/machine:I386 /pdbtype:sept
# ADD LINK32 SComponentd.lib CommAdaptd.lib PhysicalInstrumentd.lib sxlrtd.lib obd.lib
CosNamingd.lib CosEventd.lib jtcd.lib xerces-c_1d.lib /nologo /dll /debug /machine:I386
/pdbtype:sept /libpath:"..\..\lib" /libpath:"..\..\..\sxl\lib" /libpath:"..\..\..\xml\lib"
/libpath:"..\..\..\ooc\lib"
# Begin Special Build Tool
PostBuild_Cmds=..\..\bin\scp $(OutDir) LNAC_PI_Miteq.dll
# End Special Build Tool

!ENDIF

# Begin Target

# Name "LNAC_PI_Miteq - Win32 Release"
# Name "LNAC_PI_Miteq - Win32 Debug"
# Begin Group "Source Files"

# PROP Default_Filter "cpp;c;cxx;rc;def;r;odl;idl;hpj;bat"
# Begin Source File

SOURCE=..\..\IDL\CPP\LNASwitch.cpp
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\LNASwitch_skel.cpp
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\LNA_Controller.cpp
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\LNA_Controller_skel.cpp
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\Mezzanine\LNAC_PI_Miteq.cpp
# End Source File
# Begin Source File

SOURCE=.\LNAC_PI_Miteq_impl.cpp
# End Source File
# Begin Source File

SOURCE=.\LNAC_PI_Miteq_main.cpp
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\Mezzanine\LNAC_PI_Miteq_skel.cpp
# End Source File
# End Group
# Begin Group "Header Files"

# PROP Default_Filter "h;hpp;hxx;hm;inl"
# Begin Source File
```

```
SOURCE=..\..\IDL\CPP\LNASwitch.h
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\LNASwitch_skel.h
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\LNA_Controller.h
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\LNA_Controller_skel.h
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\Mezzanine\LNAC_PI_Miteq.h
# End Source File
# Begin Source File

SOURCE=.\LNAC_PI_Miteq_impl.h
# End Source File
# Begin Source File

SOURCE=..\..\IDL\CPP\Mezzanine\LNAC_PI_Miteq_skel.h
# End Source File
# End Group
# Begin Group "Resource Files"

# PROP Default_Filter "ico;cur;bmp;dlg;rc2;rct;bin;rgs;gif;jpg;jpeg;jpe"
# End Group
# Begin Group "IDL Files"

# PROP Default_Filter "*.idl"
# Begin Source File

SOURCE=..\..\IDL\LNASwitch.idl

!IF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Release"

# PROP Ignore_Default_Tool 1
# Begin Custom Build - Compiling IDL File
InputDir=\Orbacus\stovokor\IDL
InputPath=..\..\IDL\LNASwitch.idl
InputName=LNASwitch

BuildCmds= \
    ..\..\..\ooc\bin\idl -I$(InputDir) -I..\..\..\ooc\idl --output-dir $(InputDir)\CPP
$(InputPath)

"$(InputDir)\CPP\$(InputName).h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName).cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)
# End Custom Build

!ELSEIF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Debug"
```

```
# PROP Ignore_Default_Tool 1
# Begin Custom Build - Compiling IDL File
InputDir=\Orbacus\stovokor\IDL
InputPath=..\..\IDL\LNASwitch.idl
InputName=LNASwitch

BuildCmds= \
    ..\..\..\ooc\bin\idl -I$(InputDir) -I..\..\..\ooc\idl --output-dir $(InputDir)\CPP
$(InputPath)

"$(InputDir)\CPP\$(InputName).h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName).cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)
# End Custom Build

!ENDIF

# End Source File
# Begin Source File

SOURCE=..\..\IDL\LNA_Controller.idl

!IF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Release"

# PROP Ignore_Default_Tool 1
# Begin Custom Build - Compiling IDL File
InputDir=\Orbacus\stovokor\IDL
InputPath=..\..\IDL\LNA_Controller.idl
InputName=LNA_Controller

BuildCmds= \
    ..\..\..\ooc\bin\idl -I$(InputDir) -I..\..\..\ooc\idl --output-dir $(InputDir)\CPP
$(InputPath)

"$(InputDir)\CPP\$(InputName).h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName).cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)
# End Custom Build

!ELSEIF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Debug"

# PROP Ignore_Default_Tool 1
# Begin Custom Build - Compiling IDL File
InputDir=\Orbacus\stovokor\IDL
InputPath=..\..\IDL\LNA_Controller.idl
InputName=LNA_Controller
```

```
BuildCmds= \
    ..\..\..\ooc\bin\idl -I$(InputDir) -I..\..\..\ooc\idl --output-dir $(InputDir)\CPP
$(InputPath)

"$(InputDir)\CPP\$(InputName).h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName).cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"$(InputDir)\CPP\$(InputName)_skel.cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)
# End Custom Build

!ENDIF

# End Source File
# Begin Source File

SOURCE=LNAC_PI_Miteq.idl

!IF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Release"

# PROP Ignore_Default_Tool 1
# Begin Custom Build - Compiling IDL File
InputDir=.
InputPath=LNAC_PI_Miteq.idl
InputName=LNAC_PI_Miteq

BuildCmds= \
    ..\..\..\ooc\bin\idl -I..\..\IDL -I..\..\..\ooc\idl --output-dir ..\..\IDL\CPP\Mezzanine
$(InputPath)

"..\..\IDL\CPP\Mezzanine\$(InputName).h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"..\..\IDL\CPP\Mezzanine\$(InputName).cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"..\..\IDL\CPP\Mezzanine\$(InputName)_skel.h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"..\..\IDL\CPP\Mezzanine\$(InputName)_skel.cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)
# End Custom Build

!ELSEIF  "$(CFG)" == "LNAC_PI_Miteq - Win32 Debug"

# PROP Ignore_Default_Tool 1
# Begin Custom Build - Compiling IDL File
InputDir=.
InputPath=LNAC_PI_Miteq.idl
InputName=LNAC_PI_Miteq

BuildCmds= \
    ..\..\..\ooc\bin\idl -I..\..\IDL -I..\..\..\ooc\idl --output-dir ..\..\IDL\CPP\Mezzanine
$(InputPath)

"..\..\IDL\CPP\Mezzanine\$(InputName).h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)
```

```
"..\..\IDL\CPP\Mezzanine\$(InputName).cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"..\..\IDL\CPP\Mezzanine\$(InputName)_skel.h" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)

"..\..\IDL\CPP\Mezzanine\$(InputName)_skel.cpp" : $(SOURCE) "$(INTDIR)" "$(OUTDIR)"
   $(BuildCmds)
# End Custom Build

!ENDIF

# End Source File
# End Group
# End Target
# End Project
```

2.3.4 Makefile Architecture

The makefile architecture is now required to be implemented for all modules, even if they are being built from within Visual Studio. One of the targets for the makefiles is ***idl***, which builds all the C++ stubs and skeletons from the IDL files. This provides a more reliable set of dependency checking rules compared with Visual Studio, whose support of CORBA is non-existant, and clashes with COM.

As an alternative to the Visual Studio project files, the makefile architecture also simplifies the build and installation process for staged builds. This is especially useful with release versions. As an extension to the makefiles, a checkout option would allow builds to be staged directly out of source control. This feature has not been implemented yet.

The makefile system is comprised of several components. These are described below with their locations relative to the root of the Satlinx application source subtree:

`Makefile(.mak)`	The top-level makefile in the root of the Satlinx application source subtree.
`manifest.list`	The list of modules to be built.
`config/Make.rules`	The make rules, containing all the dependency requirements for the various file types.
`config/Make.dirs`	The directory definitions, containing the paths for third-party libraries and for the targets.
`config/subsys.mak`	The subsystem make rules.

config/*.mkx	The internal library dependency fragments, containing all the dependency definitions for Satlinx library subsystems. These are assembled by the make process into a single set of dependencies for all Satlinx libraries
<*subsystem*>/Makefile.mak	The subsystem makefiles.
<*subsystem*>/<*target*>/<*target*>.mak	The target module makefiles.

Note that the makefiles are written for use with Microsoft Nmake, and use features that are unique to its syntax. While it would be preferable to use a more platform independent make, such as GNU Make, this would not have some of the unique features of nmake that compensate for the lack of a proper shell and shell scripting under Windows. It may still be possible to create a GNU Make compatible build system, but this would require more time to research and complete.

2.3.4.1 The Top-Level Makefile

The top-level makefile is reproduced in part below:

```
# ****************************************************************************
# Stovokor Master Makefile
# Author: Andrew Manison
# Date:   9/27/2000
# Rev:
# ****************************************************************************

MODULE_NAME = Top Level

# ****************************************************************************
# Include the directory definitions
# ****************************************************************************
STOVOKOR_TOP=$(MAKEDIR)
!include $(STOVOKOR_TOP)\config\Make.dirs

# ****************************************************************************
# Include the build rules
# ****************************************************************************
!include $(STOVOKOR_TOP)\config\Make.rules

SUBDIRS = Infrastructure      \
          CommAdapters        \
          VirtualTasks        \
          VirtualFunctions    \
          VirtualInstruments  \
          PhysicalInstruments \
          Simulators          \
          GUI
```

The subdirectories listed above are the various subtrees of the application source tree. Most of the dependency lines below iterate through this list to perform the operations on all subtrees. Any subdirectory may be commented out with the makefile comment character '#' as the first character on the line, and that subtree will be ignored.

```
checkout::
        @for %i in ( $(SUBDIRS) ) do \
            @if not exist %i \
                @echo "Creating %i..." & \
                cmd /c "md %i"

        @for %i in ( $(SUBDIRS) ) do \
            @if not exist %i\Makefile.mak \
                @echo "Checking out %i\Makefile.mak... (not really! Just a touch)" & \
                @$(TOOLDIR)/touch %i\%i.mak

        @for %i in ( $(SUBDIRS) ) do \
            @if exist %i \
                @echo "Running BUILD make in %i ..." & \
                cmd /c "cd %i & $(MAKE) /nologo /f Makefile.mak $@"
```

The checkout target is in place but not implemented yet. It would iteratively pull sources from CM for all active subdirectories.

```
install:: obinstall
```

The install target is used in all the module files also to copy the individual module binaries into the staging tree. Note the use of the "::" to indicate a multiply defined target.

```
obinstall:
        @if not exist $(bindir) \
            @echo "Creating $(bindir)..." & \
            mkdir $(bindir)

        @if not exist $(cfgdir) \
            @echo "Creating $(cfgdir)..." & \
            mkdir $(cfgdir)

        @if not exist $(vbindir) \
            @echo "Creating $(vbindir)..." & \
            mkdir $(vbindir)

        @if not exist $(vcfgdir) \
            @echo "Creating $(vcfgdir)..." & \
            mkdir $(vcfgdir)

        @echo Copying $(OBLIB)\ob$(OBVERSION).dll to $(bindir)\ob$(OBVERSION).dll
        -@copy $(OBLIB)\ob$(OBVERSION).dll $(bindir) > nul
        @echo Copying $(OBLIB)\jtc$(JTCVERSION).dll to $(bindir)\jtc$(JTCVERSION).dll
        -@copy $(OBLIB)\jtc$(JTCVERSION).dll $(bindir) > nul

        .
        .       Many more files copied into the staging tree
        .
```

```
        @echo Copying $(XMLJAR)\xerces.jar to $(bindir)\xerces.jar
        @copy $(XMLJAR)\xerces.jar $(bindir) > nul

manifest::
        -@del $(STOVOKOR_TOP)\manifest.list > nul

manifest $(EVERYTHING)::
        @for %i in ( $(SUBDIRS) ) do \
            @if exist %i \
                @cmd /c "cd %i & $(MAKE) /nologo /f Makefile.mak STOVOKOR_TOP=$(STOVOKOR_TOP)
DEBUG=$(DEBUG) $@"

clean::
        @del $(STOVOKOR_TOP)\config\libs.mak

!if [ $(TOOLDIR)\cat $(STOVOKOR_TOP)\config\*.mkx > $(STOVOKOR_TOP)\config\libs.mak ]
!message Error creating libs.mak
!endif
```

The file libs.mak is a concatenation of the library dependency fragments.

```
!include $(STOVOKOR_TOP)\config\libs.mak

# ************************************************************************
# End of Makefile
# ************************************************************************
```

The main targets defined for the top-level makefile are:

`all`	This builds all modules
`install`	This installs all binaries into the staging tree.
`mostlyclean`	This cleans up the binaries and intermediate files generated during the build process.
`clean`	In addition to those files removed by the mostlyclean target, this deletes the dependency files, Visual Studio configuration files, and link definition files, effectively returning the source tree to the state it was after the files were staged for building out of source control.
`idl`	This target compiles all the IDL into C++ stubs and skeletons.
`depend`	This builds dependencies for all modules into include files for the individual module makefiles. These are not needed for staged builds, or when building from within Visual Studio, but are necessary when developing and building with the makefiles.
`manifest`	This target creates a new, comprehensive software manifest that can be edited to reflect the actual system configuration.

2.3.4.2 The Module Manifest

The manifest is an automatically generated list of all modules comprising the Satlinx system. If no manifest is found the build is comprehensive. If a manifest exists, only those modules listed will be built. A minimal manifest for a skeletal system including required infrastructure and a few modules appears below:

```
Infrastructure/authent
Infrastructure/configmgr
Infrastructure/corbaservice
Infrastructure/factory
Infrastructure/FileMgr
Infrastructure/logger
Infrastructure/SComponent
Infrastructure/stovokor
CommAdapters/CommBase
CommAdapters/Socket
VirtualFunctions/FaultMgr
VirtualFunctions/NodeControl
VirtualFunctions/UserListEditor
VirtualInstruments/NTS_VI
```

Such a list would be culled from the master list generated by '`make manifest`'.

2.3.4.3 The Make Rules

A detailed discussion of the make rules file is beyond the scope of this document, but is reproduced below. The sections are commented. In addition to specifying compiler and linker flags for release and debug version, the rules file selects the libraries to link with the module, and standard dependencies and build rules for the various source file types.

```
# ------------------------------------------------------------------------------#
# These messages display the settings of the configuration options. Uncomment  #
# them if there is a need to verify settings, otherwise leave them commented.  #
# ------------------------------------------------------------------------------#
#!message $(MODULE_NAME) ($(TARGETS)): DEBUG=$(DEBUG) IS_SCOMPONENT=$(IS_SCOMPONENT)
IS_COMMADAPTER=$(IS_COMMADAPTER) IS_DLL=$(IS_DLL)

# ------------------------------------------------------------------------------#
# Experts only: You can modify the lines below to change the settings for code #
# optimizations.                                                               #
# ------------------------------------------------------------------------------#

!if "$(OPTIMIZE)" == "yes"

#
# These options are for building with optimizations turned on.
# See Microsoft's compiler documentation for more information on the
# /O flag. Please note that debug information is automatically
# disabled by the Microsoft compiler when optimization is enabled.
#
OPT_CXXFLAGS    = /Ox
OPT_LINKFLAGS   =

!endif
```

```
# ============================================================================
# ----------------------------------------------------------------------------#
# MAKE GURUS ONLY BELOW THIS POINT:-                                          #
# If you modify the lines below you either know EXACTLY what you are doing, or #
# you want to sabotage the project!  These few lines were hard wrought over    #
# many weeks, and serve to keep the individual module makefiles very simple.   #
# If you REALLY, REALLY, REALLY need to change something here, confer with the #
# makefile maintainer first.                                                  #
# ----------------------------------------------------------------------------#

!if "$(DEBUG)" == "yes"

# ----------------------------------------------------------------------------
# These options are for building with debug information, and no optimization.
# ----------------------------------------------------------------------------
OPT_CXXFLAGS  = /ZI /Gi /Od /D "_DEBUG" /GZ
OPT_CPPFLAGS  = $(OPT_CPPFLAGS)
OPT_LINKFLAGS = /debug /pdbtype:sept $(OPT_LINKFLAGS)
CXX_RTL       = /MDd

!else

# ----------------------------------------------------------------------------
# These options are for building with no debug information. Optimization flags
# are set above.
# ----------------------------------------------------------------------------
OPT_CXXFLAGS  = $(OPT_CXXFLAGS) /D "NDEBUG"
OPT_CPPFLAGS  = $(OPT_CPPFLAGS) /WX
OPT_LINKFLAGS = $(OPT_LINKFLAGS) /incremental:no /release
CXX_RTL       = /MD

!endif

# ----------------------------------------------------------------------------
# These are the standard libraries required:
# SXL C++ standard library, Orbacus core, Orbacus naming service support, and
# Orbacus event service support.
# ----------------------------------------------------------------------------
LIBS          = sxlrt$(DD).lib ob$(DD).lib CosNaming$(DD).lib CosEvent$(DD).lib jtc$(DD).lib
shell32.lib
LIBDEPS       = $(SXLLIB)\sxlrt$(DD).lib $(OBLIB)\ob$(DD).lib $(OBLIB)\CosNaming$(DD).lib
$(OBLIB)\CosEvent$(DD).lib $(OBLIB)\jtc$(DD).lib

!if "$(IS_COMMADAPTER)" == "yes"
# ----------------------------------------------------------------------------
# These define the Comm Adapter library and force other required options.
# ----------------------------------------------------------------------------
IS_SCOMPONENT=yes
LIBS          = CommAdapt$(DD).lib $(LIBS)
LIBDEPS       = $(LIBDIR)\CommAdapt$(DD).lib $(LIBDEPS)
!endif

!if "$(IS_SIMULATOR)" == "yes"
# ----------------------------------------------------------------------------
# These define the Comm Adapter library and force other required options.
# ----------------------------------------------------------------------------
IS_SCOMPONENT=yes
LIBS          = CommAdapt$(DD).lib $(LIBS)
LIBDEPS       = $(LIBDIR)\CommAdapt$(DD).lib $(LIBDEPS)
!endif
```

```
!if "$(IS_TASK)" == "yes"
# ------------------------------------------------------------------------------
# These define the Comm Adapter library and force other required options.
# ------------------------------------------------------------------------------
IS_SCOMPONENT=yes
TASKIDLSRC = $(IDLDIR)\Task.idl

TASKIDLGEN = $(IDLCPP)\Task.h $(IDLCPP)\Task.cpp \
             $(IDLCPP)\Task_skel.h $(IDLCPP)\Task_skel.cpp

OBJECTS    = $(OBJECTS)              \
             $(INTDIR)\Task_skel.obj \
             $(INTDIR)\Task.obj      \
             $(INTDIR)\Scheduler_IF.obj

IDLSRC  = $(IDLSRC) $(TASKIDLSRC)

IDLGEN  = $(IDLGEN) $(TASKIDLGEN)

!endif

!if "$(IS_PHYSICALINSTRUMENT)" == "yes"
# ------------------------------------------------------------------------------
# These define the Comm Adapter library and force other required options.
# ------------------------------------------------------------------------------
IS_SCOMPONENT=yes
LIBS          = PhysicalInstrument$(DD).lib CommAdapt$(DD).lib $(LIBS)
LIBDEPS       = $(LIBDIR)\PhysicalInstrument$(DD).lib $(LIBDIR)\CommAdapt$(DD).lib $(LIBDEPS)
!endif

!if "$(IS_SCOMPONENT)" == "yes"
# ------------------------------------------------------------------------------
# These define the SComponent library and force other required options.
# ------------------------------------------------------------------------------
PARSES_XML=yes
USES_SCOMPONENT=no
LIBS          = SComponent$(DD).lib $(LIBS)
LIBDEPS       = $(LIBDIR)\SComponent$(DD).lib $(LIBDEPS)
!endif

!if "$(IS_SBASE)" == "yes"
!message $(TARGETS) - IS_SBASE: This is an out of date setting.
!endif

!if "$(PARSES_XML)" == "yes"
# ------------------------------------------------------------------------------
# These are the added options to include the XML parser.
# ------------------------------------------------------------------------------
LIBS            = xerces-c_1$(DD).lib $(LIBS)
OPT_LINKFLAGS   = /libpath:"$(XMLLIB)" $(OPT_LINKFLAGS)
CPPINCLUDE      = $(CPPINCLUDE) /I $(XMLINCLUDE)
MKDEPINCLUDE    = $(MKDEPINCLUDE) -I$(XMLINCLUDE)
!endif

!if "$(RHAPSODY_BASED)" == "yes"
# ------------------------------------------------------------------------------
# These are the added options to include the Rhapsody libraries.
# ------------------------------------------------------------------------------
LIBS            = MSaomanim.lib MStomtrace.lib MSaomtrace.lib MSomComAppl.lib MSoxf.lib
MSoxfinst.lib $(LIBS)
OPT_LINKFLAGS   = /libpath:"$(RHAPSODYLIB)" $(OPT_LINKFLAGS)
CPPINCLUDE      = $(CPPINCLUDE) /I $(RHAPSODYINCLUDE)
MKDEPINCLUDE    = $(MKDEPINCLUDE) -I$(RHAPSODYINCLUDE)
!endif
```

```
!if "$(IS_DLL)" == "yes"
# ------------------------------------------------------------------------------
# These are the added options to build the module as a DLL.
# ------------------------------------------------------------------------------
OPT_LINKFLAGS   = $(OPT_LINKFLAGS) /dll
!ifdef DLLOBJS
OPT_CPPFLAGS    = /GD /D "$(MODULE_NAME)_IMPORT=/**/" $(OPT_CPPFLAGS)
DEF_FILE        = $(INTDIR)\$(MODULE_NAME).def
OPT_LINKFLAGS   = $(OPT_LINKFLAGS) /def:$(DEF_FILE)
MAKE_EXPORTS    = perl $(TOOLDIR)\makedef.pl $(DEF_FILE) $(MODULE_NAME)$(DD) $(DLLOBJS)
IDLFLAGS        = $(IDLFLAGS) --dll-import $(MODULE_NAME)_IMPORT
!endif
!endif

!if "$(USES_SBASE)" == "yes"
!message USES_SBASE: This is an out of date setting.
!endif

!if "$(USES_SCOMPONENT)" == "yes"
# ------------------------------------------------------------------------------
# This defines the SComponent import macro so IDL generated stubs can be linked
# with this module.
# ------------------------------------------------------------------------------
!message USES_SCOMPONENT: This is an out of date setting.
OPT_CPPFLAGS    = /D "SCOMPONENT_IMPORT=/**/" $(OPT_CPPFLAGS)
!endif

OPT_LINKFLAGS   = $(OPT_LINKFLAGS) /libpath:"$(SXLLIB)" /libpath:"$(OBLIB)"
/libpath:"$(LIBDIR)"

# ------------------------------------------------------------------------------
# These are the command macro definitions for compiler, linker and IDL compiler.
# ------------------------------------------------------------------------------
CXX             = cl.exe
CXXFLAGS        = /nologo $(CXX_RTL) /W3 /GR /GX /Gy $(OPT_CXXFLAGS) /Zm200 /Fo"$(INTDIR)\\"
/Fd"$(INTDIR)\\"
CXXCPP          = cl.exe /nologo /E
CPPFLAGS        = /D "WIN32" $(OPT_CPPFLAGS)
CPPINCLUDE      = $(CPPINCLUDE) /I $(SXLINCLUDE) /I $(OBINCLUDE) /I . /I $(IDLCPP) /I
$(IDLMEZZ)
CXXINCLUDE      = /I $(STOVOKOR_TOP)\global
MKDEPINCLUDE    = $(MKDEPINCLUDE) -I$(SXLINCLUDE) -I$(OBINCLUDE) -I. -I$(IDLCPP) -I$(IDLMEZZ)
-I$(STOVOKOR_TOP)\global
GUILIBS         =
DLLLIBS         =
LINK            = link.exe
LINKFLAGS       = /nologo $(OPT_LINKFLAGS) /subsystem:console /out:$@
LINKGUIFLAGS    = /nologo $(OPT_LINKFLAGS) /subsystem:windows /out:$@
AR              = lib.exe
ARFLAGS         = /nologo
IDL             = $(OBEXE)\idl.exe
IDLINCLUDE      = -I$(IDLDIR) -I$(OBIDL)
IDLFLAGS        = $(IDLFLAGS) $(IDLINCLUDE) --output-dir $(IDLCPP)
CO              = $(MSDevDir)\..\VSS\Win32\ss.exe
```

```
# ------------------------------------------------------------------------
# Some miscellaneous directory handling.
# ------------------------------------------------------------------------

!if "$(OUTDIR)" == ""
OUTDIR=.
!endif

!if "$(INTDIR)" == ""
INTDIR=.
!endif

!ifndef INCLUDE
!error The INCLUDE environment variable needs to be set.
!endif
!ifndef LIB
!error The LIB environment variable needs to be set.
!endif

# ------------------------------------------------------------------------
# The main targets.
# ------------------------------------------------------------------------

EVERYTHING = all install mostlyclean clean idl depend

.SUFFIXES:
.SUFFIXES:   .cpp .obj .idl

!if "$(TARGETS)" != ""

$(TARGETS) : idl $(OUTDIR) $(INTDIR) $(LIBDEPS)

{$(IDLCPP)}.cpp{$(INTDIR)}.obj:
        @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo idl"
        @$(CXX) /c $(CPPFLAGS) $(CXXFLAGS) $(CPPINCLUDE) $<

{$(IDLMEZZ)}.cpp{$(INTDIR)}.obj:
        @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo idl"
        @$(CXX) /c $(CPPFLAGS) $(CXXFLAGS) $(CPPINCLUDE) $<

{}.cpp{$(INTDIR)}.obj:
        @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo idl"
        @$(CXX) /c $(CPPFLAGS) $(CXXFLAGS) $(CPPINCLUDE) $(CXXINCLUDE) $<

all:: $(INTDIR) $(OUTDIR) $(SOURCES) $(TARGETS)

!else

all:: $(SOURCES) $(TARGETS)

!endif
```

```
# ************************************************************************
# Dependency generation
# ************************************************************************

#SYSINCLUDE = $(INCLUDE)
#SYSINCLUDE = $(MSVCDIR)\Include
#!message MSVCDIR = $(MSVCDIR)
#!message SYSINCLUDE = $(SYSINCLUDE)
#!message INCLUDE = $(INCLUDE)

!if "$(TARGETS)" != ""
depend:: idl $(SOURCES) $(IDLGEN)
        @$(TOOLDIR)\touch .depend
        @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo idl"
        @$(TOOLDIR)\makedepend -X -Y$(SYSINCLUDE) -f.depend -p$(INTDIR)/ -DHAVE_SSTREAM
$(MKDEPINCLUDE) $(SOURCES)
        @$(TOOLDIR)\makedepend -f.depend -a -p$(IDLCPP)/ -o.h $(IDLINCLUDE) $(IDLSRC)
        @$(TOOLDIR)\makedepend -f.depend -a -p$(IDLCPP)/ -o.cpp $(IDLINCLUDE) $(IDLSRC)
        @$(TOOLDIR)\makedepend -f.depend -a -p$(IDLCPP)/ -o_skel.h $(IDLINCLUDE) $(IDLSRC)
        @$(TOOLDIR)\makedepend -f.depend -a -p$(IDLCPP)/ -o_skel.cpp $(IDLINCLUDE) $(IDLSRC)
!endif

# ************************************************************************
# IDL dependencies
# ************************************************************************

$(IDLCPP):
        @mkdir $(IDLCPP)

$(IDLMEZZ): $(IDLCPP)
        @mkdir $(IDLMEZZ)

idl:: $(IDLCPP) $(IDLMEZZ) $(IDLGEN)

{$(IDLDIR)}.idl{$(IDLCPP)}.cpp:
        @echo $(<F)
        @$(IDL) $(IDLFLAGS) --output-dir $(IDLCPP) $<

{$(IDLDIR)}.idl{$(IDLCPP)}.h:
        @echo $(<F)
        @$(IDL) $(IDLFLAGS) --output-dir $(IDLCPP) $<

{}.idl{$(IDLMEZZ)}.cpp:
        @echo $(<F)
        @$(IDL) $(IDLINCLUDE) --output-dir $(IDLMEZZ) $<

{}.idl{$(IDLMEZZ)}.h:
        @echo $(<F)
        @$(IDL) $(IDLINCLUDE) --output-dir $(IDLMEZZ) $<
```

```
# ************************************************************************
# Cleanup targets
# ************************************************************************

mostlyclean::
        -@del *.obj *.bak *.flc *.pdb *.pch > nul 2>&1
        -@del $(INTDIR)\*.obj $(INTDIR)\*.bak $(INTDIR)\*.lib > nul 2>&1
        -@del $(INTDIR)\*.def $(INTDIR)\*.exp $(INTDIR)\*.flc  > nul 2>&1
        -@del $(INTDIR)\*.pdb $(INTDIR)\*.idb $(INTDIR)\*.sbr  > nul 2>&1
        -@del $(INTDIR)\*.pch $(INTDIR)\*.ilk > nul 2>&1

clean:: mostlyclean
        -@del .depend > nul 2>&1
        -@del *.ncb *.opt *.plg *.def > nul 2>&1

!if "$(IDLGEN)" != ""
clean::
        @for %i in ( $(IDLGEN) ) do \
            @if exist %i del %i
!endif

!if "$(IDLGENX)" != ""
clean::
        @for %i in ( $(IDLGENX) ) do \
            @if exist %i del %i
!endif

!if "$(TARGETS)" != ""
mostlyclean::
        @for %i in ( $(TARGETS) ) do \
            @if exist %i del %i
!endif

!if "$(PDB_FILES)" != ""
mostlyclean::
        @for %i in ( $(PDB_FILES) ) do \
            @if exist %i del %i
!endif

# ************************************************************************
# Library dependencies
# ************************************************************************

$(LIBDEPS) :
        @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo $@"
```

```
# ************************************************************************
# Build directory dependencies
# ************************************************************************

!if "$(TARGETS)" != ""
!if "$(OUTDIR)" != "."
$(OUTDIR) :
        @echo Creating $(OUTDIR)
        -@mkdir $(OUTDIR)
!endif

!if "$(INTDIR)" != "."
!if "$(INTDIR)" != "$(OUTDIR)"
$(INTDIR) :
        @echo Creating $(INTDIR)
        -@mkdir $(INTDIR)
!endif
!endif
!endif

# ************************************************************************
# End of Make Rules
# ************************************************************************
```

It goes without saying that the rules file is not trivial, and is the heart of the effectiveness of the system. The sophistication of the rules allows the individual module makefiles to be very simple, as can be seen below.

2.3.4.4 The Library Dependency Fragments

To facilitate the addition of standard libraries to the system, as new base classes or utility classes and libraries are added, the library dependencies are not written in to the top-level makefile or make rules, but assembled from fragments and included in the top-level makefile. The fragments are found in the `config` subdirectory of the Satlinx application source subtree, and have the extension "`.mkx`".

An example fragment is shown below:

```
$(LIBDIR)\SComponent$(DD).lib:
        @cmd /c "cd $(STOVOKOR_TOP)\Infrastructure\SComponent & \
            $(MAKE) /nologo /f SComponent.mak \
            STOVOKOR_TOP=$(STOVOKOR_TOP) DEBUG=$(DEBUG) $@"
```

This is a make dependency for the library, according to standard makefile syntax. It simply invokes make on the library's own makefile.

2.3.4.5 The Make Directories

The directory definitions, containing the paths for third-party libraries and for the targets, are split out into a separate file from the make rules. The main build configuration parameters set in this file are:

- Debug and optimization settings

- Installation directories
- Third party library versions

The directory definitions file is reproduced below:

```
# ----------------------------------------------------------------------
# Just some informational messages as we go along.
# ----------------------------------------------------------------------
#!message Building $(MODULE_NAME) ...

# ----------------------------------------------------------------------
# Use the following lines to set build options.
# ----------------------------------------------------------------------
#OPTIMIZE=yes
#DEBUG=no

# ----------------------------------------------------------------------
# The following lines set the version for the installation directories.
# ----------------------------------------------------------------------

VERSIONFILE = $(STOVOKOR_TOP)\config\version

!if !exist ( $(VERSIONFILE) )
!if [ echo VERSION=1.0 > $(VERSIONFILE) ]
!message Cannot create version file.
!endif
!endif

!include $(VERSIONFILE)

# ----------------------------------------------------------------------
# Use the following lines to set your installation directories
# ----------------------------------------------------------------------

prefix          = $(STOVOKOR_TOP)\..
bindir          = $(prefix)\bin
cfgdir          = $(prefix)\cfg
vbindir         = $(prefix)\$(VERSION)\bin
vcfgdir         = $(prefix)\$(VERSION)\cfg

SYSINCLUDE      = "C:\Program Files\Microsoft Visual Studio\VC98\Include"

!if "$(OBVERSION)" == ""
OBVERSION       = 410
!endif

!if "$(JTCVERSION)" == ""
JTCVERSION      = 200
!endif

!if "$(SXLVERSION)" == ""
SXLVERSION      = 308
!endif

!if "$(XMLVERSION)" == ""
XMLVERSION      = 1_3
!endif
```

```
OBINCLUDE       = $(STOVOKOR_TOP)\..\ooc\include
OBIDL           = $(STOVOKOR_TOP)\..\ooc\idl
OBLIB           = $(STOVOKOR_TOP)\..\ooc\lib
OBEXE           = $(STOVOKOR_TOP)\..\ooc\bin
OBJAR           = $(STOVOKOR_TOP)\..\ooc\jars

SXLINCLUDE      = $(STOVOKOR_TOP)\..\sxl\include
SXLLIB          = $(STOVOKOR_TOP)\..\sxl\lib
SXLBIN          = $(STOVOKOR_TOP)\..\sxl\bin

XMLINCLUDE      = $(STOVOKOR_TOP)\..\xml\include
XMLLIB          = $(STOVOKOR_TOP)\..\xml\lib
XMLBIN          = $(STOVOKOR_TOP)\..\xml\bin
XMLJAR          = $(STOVOKOR_TOP)\..\xml\jars

RHAPSODYINCLUDE = $(STOVOKOR_TOP)\..\rhapsody
RHAPSODYLIB     = $(STOVOKOR_TOP)\..\rhapsody\lib

CFGDIR          = $(STOVOKOR_TOP)\config
LIBDIR          = $(STOVOKOR_TOP)\LIB
TOOLDIR         = $(STOVOKOR_TOP)\BIN
IDLDIR          = $(STOVOKOR_TOP)\IDL
IDLCPP          = $(STOVOKOR_TOP)\IDL\CPP
IDLMEZZ         = $(STOVOKOR_TOP)\IDL\CPP\MEZZANINE

# ----------------------------------------------------------------------
# Don't change these unless you know what you are doing!
# ----------------------------------------------------------------------

!if "$(DEBUG)" == "yes"
OUTDIR=$(STOVOKOR_TOP)\Debug.out\$(MODULE_NAME)
INTDIR=$(STOVOKOR_TOP)\Debug.out\$(MODULE_NAME)
DD = d
!else
OUTDIR = $(STOVOKOR_TOP)\Release.out\$(MODULE_NAME)
INTDIR = $(STOVOKOR_TOP)\Release.out\$(MODULE_NAME)
DD =
!endif
```

2.3.4.6 The Subsystem Make Rules

The Satlinx source tree is split into subsystems, based on the types of modules. Each subsystem has its own makefile, but the build procedure is the same for each. Basically, it must call the module makefile for each subdirectory in the subtree. For this reason the common portion of the makefile is split off into a generic include file, reproduced below:

```
# ****************************************************************
# Stovokor Generic Subsystem Makefile
# Author: Andrew Manison
# Date:   9/27/2000
# Rev:
# ****************************************************************
```

```
# ************************************************************************
# Check for required macros definitions.
# ************************************************************************
!ifndef STOVOKOR_TOP
!error STOVOKOR_TOP not set in the environment or on the command line.
!endif
!if "$(MODULE_NAME)" == ""
!error MODULE_NAME not set.
!endif

# ************************************************************************
# Include the directory definitions
# ************************************************************************
!include $(STOVOKOR_TOP)\config\Make.dirs

# ************************************************************************
# Include the build rules
# ************************************************************************
!include $(CFGDIR)\Make.rules

checkout::
!ifdef SUBDIRS
!if "$(SUBDIRS)" != ""
       @for %i in ( $(SUBDIRS) ) do \
           @if not exist %i \
               @echo "Creating %i..." & \
               cmd /c "md %i"

       @for %i in ( $(SUBDIRS) ) do \
           @if not exist %i\%i.mak \
               @echo "Checking out %i\%i.mak... (not really! Just a touch)" & \
               @$(TOOLDIR)/touch %i\%i.mak

       @for %i in ( $(SUBDIRS) ) do \
           @if exist %i cmd /c "cd %i & $(MAKE) /nologo STOVOKOR_TOP=$(STOVOKOR_TOP)
DEBUG=$(DEBUG) /f %i.mak $@"
!endif
!endif

manifest::
!ifdef SUBDIRS
!if "$(SUBDIRS)" != ""
       @for %i in ( $(SUBDIRS) ) do \
           @if exist %i cmd /c "echo $(MODULE_NAME)/%i >> $(STOVOKOR_TOP)\manifest.list"
!endif
!endif

$(EVERYTHING)::
!ifdef SUBDIRS
!if "$(SUBDIRS)" != ""
       @for %i in ( $(SUBDIRS) ) do \
           @if exist %i cmd /c "$(TOOLDIR)/cmake $(MODULE_NAME)/%i $(STOVOKOR_TOP) $(TOOLDIR)
$(MAKE) %i $@ $(DEBUG)"
!endif
!endif
```

2.3.4.7 The Subsystem Makefiles

As noted above, the Satlinx source tree is split into subsystems, based on the types of modules. Each subsystem has its own makefile that simply lists the subdirectories in its subtree. An example of one is reproduced below:

```
# ************************************************************************
# Stovokor Infrastructure Subsystem Makefile
# Author: Andrew Manison
# Date:   9/27/2000
# Rev:
# ************************************************************************

# ************************************************************************
# Define module name. This is used elsewhere for module unique
# parameters so make sure there is no clash with any other module.
# ************************************************************************
MODULE_NAME = Infrastructure

# ************************************************************************
# List of module directories in the subsystem. Add new ones as required.
# ************************************************************************
SUBDIRS = authent      \
          configmgr    \
          corbaservice \
          ctl          \
          factory      \
          FileMgr      \
          logger       \
          NSFederator  \
          SComponent   \
          stovokor

# ************************************************************************
# Include the subsystem makefile.
# ************************************************************************
!include $(STOVOKOR_TOP)\config\subsys.mak
```

2.3.4.8 The Module Makefiles

The target module makefiles are named for their directory, thus:

<subsystem>/*<target>*/*<target>*.mak

Creation of a module makefile is best accomplished by taking an existing one and modifying it. The alterations consist of modifying the following variables:

MODULE_NAME	A unique name for the module, used for creating module specific directories and variables.
IS_DLL	Set to 'yes' if the module is deployed as a DLL. By default the module will be built as an executable.
IS_SIMULATOR	Set to 'yes' if the module is an instrument simulator.
IS_TASK	Set to 'yes' if the module implements the Satlinx IDL `Task` interface.
IS_SCOMPONENT	Set to 'yes' if the module is derived from the `SBase_impl` or `SComponent_impl` base classes.

`IS_PHYSICALINSTRUMENT`	Set to 'yes' if the module is derived from the `PhysicalInstrument` base class.
`IS_COMMADAPTER`	Set to 'yes' if the module is derived from the `CommAdapter_impl` base class.
`RHAPSODY_BASED`	Set to 'yes' if the module is developed as a Rhapsody module.
`SOURCES`	A list of the original C++ source files comprising the module. This list does not include the generated source files.
`IDLSRC`	A list of the IDL files defining interfaces implemented by the module. This list does not include the IDL files defining interfaces this module uses as a client only.
`IDLGEN`	A list of the generated C++ source files created by the IDL compiler.
`TARGETS`	A list of the final binaries produced by the makefile. In most cases this will be a single executable or DLL, but makefiles creating base or utility class DLLs or libraries will have both a DLL and a link library target.
`OBJECTS`	A list of the object files created from the C++ source files comprising the module. This list does include the objects created from the IDL generated source files, both for client access and for a servant class implementation.

In addition, the install dependency will need to be edited to copy the target files into the correct staging directory, and display the correct message.

Here follows a simple module makefile for an executable target.

```
# ****************************************************************************
# Stovokor Module Makefile
# Module: AUTHENT
# Author: Andrew Manison
# Date:   9/27/2000
# Rev:
# ****************************************************************************

# ****************************************************************************
# Define module name. This is used elsewhere for module unique
# parameters so make sure there is no clash with any other module.
# ****************************************************************************
MODULE_NAME = AUTHENT

# ****************************************************************************
# Check for required macros definitions.
# ****************************************************************************
!ifndef STOVOKOR_TOP
!error STOVOKOR_TOP not set in the environment or on the command line.
!endif
!if "$(MODULE_NAME)" == ""
!error MODULE_NAME not set.
!endif

# ****************************************************************************
# Define subsystem dependencies.
# ****************************************************************************

#DEBUG=yes
#IS_DLL=yes
IS_SCOMPONENT=yes

# ****************************************************************************
# Include the directory definitions
# ****************************************************************************
!include $(STOVOKOR_TOP)\config\Make.dirs

# ****************************************************************************
# List the sources and intermediate sources used comprising the module.
# ****************************************************************************
SOURCES = Authent_main.cpp

IDLSRC  = $(IDLDIR)\Authent.idl

IDLGEN  = $(IDLCPP)\Authent.h $(IDLCPP)\Authent.cpp \
          $(IDLCPP)\Authent_skel.h $(IDLCPP)\Authent_skel.cpp

TARGETS = $(OUTDIR)\Authent.exe

# ****************************************************************************
# Define the objects to be linked.
# ****************************************************************************
OBJECTS = $(INTDIR)\Authent_impl.obj \
          $(INTDIR)\Authent_main.obj \
          $(INTDIR)\Authent_skel.obj

# ****************************************************************************
# Include the build rules
# ****************************************************************************
!include $(CFGDIR)\Make.rules
```

```
# ************************************************************************
# Target (link) dependencies
# ************************************************************************

$(TARGETS) : $(OBJECTS)
       @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo $(LIBDEPS)" > nul
       @$(LINK) @<<
  $(LINKFLAGS) $(LINK32_FLAGS) $(OBJECTS) $(LIBS)
<<

# ************************************************************************
# Installation dependencies
# ************************************************************************

install::
       @echo Copying $(TARGETS) to $(vbindir)\Authent.exe
       @copy $(TARGETS) $(vbindir) > nul

# ************************************************************************
# Include the generated dependencies
# ************************************************************************
!IF EXISTS(".depend")
!INCLUDE ".depend"
!ELSEIF "$(DEBUG)" == "yes"
!MESSAGE Warning: cannot find ".depend" for $(MODULE_NAME)
!ENDIF
```

Now a more complex makefile for a DLL module. This one is about as complex as they come. The module has multiple mix-in IDL interfaces, and uses mezzanine IDL to combine them.

Note that the OBJECTS macro includes Pedestal.obj, the CORBA stubs used for client access to a Pedestal object. This is the only reference to the Pedestal stubs required in the makefile. All dependency issues are handled in the make rules.

```
# ************************************************************************
# Stovokor Module Makefile
# Module: ACU_PI_386x
# Author: Andrew Manison
# Date:   9/27/2000
# Rev:
# ************************************************************************

# ************************************************************************
# Define module name. This is used elsewhere for module unique
# parameters so make sure there is no clash with any other module.
# ************************************************************************
MODULE_NAME = ACU_PI_386x

# ************************************************************************
# Check for required macros definitions.
# ************************************************************************
!ifndef STOVOKOR_TOP
!error STOVOKOR_TOP not set in the environment or on the command line.
!endif
!if "$(MODULE_NAME)" == ""
!error MODULE_NAME not set.
!endif
```

```
# ************************************************************************
# Define subsystem dependencies.
# ************************************************************************

#DEBUG=yes
IS_DLL=yes
IS_SCOMPONENT=yes
IS_PHYSICALINSTRUMENT=yes

# ************************************************************************
# Include the directory definitions
# ************************************************************************
!include $(STOVOKOR_TOP)\config\Make.dirs

# ************************************************************************
# List the sources and intermediate sources used comprising the module.
# ************************************************************************
SOURCES = ACU_PI_386x_impl.cpp \
          ACU_PI_386x_main.cpp

IDLSRC1 = ACU_PI_386x.idl

IDLGEN1 = $(IDLMEZZ)\ACU_PI_386x.h $(IDLMEZZ)\ACU_PI_386x.cpp \
          $(IDLMEZZ)\ACU_PI_386x_skel.h $(IDLMEZZ)\ACU_PI_386x_skel.cpp

IDLSRC2 = $(IDLDIR)\GenericACU.idl

IDLGEN2 = $(IDLCPP)\GenericACU.h $(IDLCPP)\GenericACU.cpp \
          $(IDLCPP)\GenericACU_skel.h $(IDLCPP)\GenericACU_skel.cpp

IDLSRC3 = $(IDLDIR)\diolines.idl

IDLGEN3 = $(IDLCPP)\diolines.h $(IDLCPP)\diolines.cpp \
          $(IDLCPP)\diolines_skel.h $(IDLCPP)\diolines_skel.cpp

IDLSRC  = $(IDLSRC1) $(IDLSRC2) $(IDLSRC3)

IDLGEN  = $(IDLGEN1) $(IDLGEN2) $(IDLGEN3)

TARGETS = $(OUTDIR)\ACU_PI_386x.dll

# ************************************************************************
# Define the objects to be linked.
# ************************************************************************
OBJECTS = $(INTDIR)\GenericACU.obj       \
          $(INTDIR)\GenericACU_skel.obj  \
          $(INTDIR)\diolines.obj         \
          $(INTDIR)\diolines_skel.obj    \
          $(INTDIR)\Pedestal.obj         \
          $(INTDIR)\ACU_PI_386x.obj      \
          $(INTDIR)\ACU_PI_386x_skel.obj \
          $(INTDIR)\ACU_PI_386x_impl.obj \
          $(INTDIR)\ACU_PI_386x_main.obj

# ************************************************************************
# Include the build rules
# ************************************************************************
!include $(CFGDIR)\Make.rules
```

```
# ************************************************************************
# Target (link) dependencies
# ************************************************************************

$(TARGETS) : $(OBJECTS)
       @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo idl"
       @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo $(LIBDEPS)" > nul
       @$(LINK) @<<
  $(LINKFLAGS) $(LINK32_FLAGS) $(OBJECTS) $(LIBS)
<<

# ************************************************************************
# Installation dependencies
# ************************************************************************

install::
       @echo Copying $(OUTDIR)\ACU_PI_386x.dll to $(vbindir)\ACU_PI_386x.dll
       @copy $(OUTDIR)\ACU_PI_386x.dll $(vbindir) > nul

# ************************************************************************
# Include the generated dependencies
# ************************************************************************
!IF EXISTS(".depend")
!INCLUDE ".depend"
!ELSEIF "$(DEBUG)" == "yes"
!MESSAGE Warning: cannot find ".depend" for $(MODULE_NAME)
!ENDIF
```

3 Developing a Satlinx CORBA Object

3.1 Conventions

All objects in the system are instantiated by name and connections are made by name. All objects are located through the CORBA Naming Service. As such, object and file naming conventions are needed and have been specified. The file names used in this section conform to the naming convention, and while it is not essential that the files conform to this convention for correct operation, it is desirable to ensure ease of management and maintenance of the code base.

3.1.1 Object Naming

The driving factor for object naming is the IDL for the object, since it is the first set of code produced. The IDL is named for the functionality of the interface the object will be implementing. Each CORBA servant can only provide the implementation for a single IDL interface, although those interfaces themselves can be multiply inherited. VI's and PI's use mezzanine, or intermediate, IDL's at the top level of inheritance to resolve parentage issues and provide a single interface. The example below uses the IFR Model 2023 Signal Generator PI and the Signal Generator function VI as an example set. The PI is named for the device class, a signal generator, and the specific model of instrument, since it is the equivalent of a device driver for that specific device. The VI is named solely for the device class, since its purpose is to provide the virtualization of that specific function without regard to how it is implemented by the device or what other functions that device may provide. A case in point would be the ACU PI that does antenna control tasks and digital switch type tasks, and hence exports two different interfaces, while the ACU VI is only concerned with antenna control tasks. Another VI, a digital switch or DIO VI also connects to the ACU PI, but to a different interface. Neither interface has methods which cross over into the other. They behave as if their functionality is the only one that device provides. Multiple instances of any individual module type may need to be instantiated, such as multiple receivers. In such a case the convention dictates that each would require a suffix of '_n' where 'n' is a sequential number. If the device has a unique task, rather than being one of a pool of devices, it can be named for the task. For example, in a system with seven IFR2023 signal generators, only one is the external LO and can therefore be named as such.

3.1.2 Object Factory Names and Object Grouping

Each object is created by a factory object, which is itself a named object. Any number of objects can be created within the context of a single factory. If the factory is to support one object it should be named for that object with a suffix of '_FACTORY'. If it is to support more than one, then the factory should be logically named for the functionality of the grouping, also with a suffix of '_FACTORY'. Grouping several objects within a single factory should typically be limited to those objects in a similar function group, for example all receiver control PI's and VI's or the ACU PI/VI group. This will enhance the performance of object creation and inter-object communications as well as reducing the memory footprint of the system. This feature should not be overused to the extent of compromising the distributed object-oriented nature of the system or the protection afforded by having different processes run in their own context.

The grouping of objects within a single factory does not affect the development process at all, or make special demands upon the object implementation. It is purely a deployment issue.

3.1.3 File Naming

3.1.3.1 IDL Source Files

These files should be named for the interface they provide, such as *SigGen.idl* in the example described above, and should reside in the 'IDL' subdirectory. Mezzanine IDL files should be named for the actual module they support, *SigGen_PI_IFR2023.idl* in our current example, and should be present in each module's own directory.

3.1.3.2 Generated Source Files

The Makefile system will automatically compile the interface IDL and generate the supporting C++ files into a subdirectory of the 'IDL' directory called 'CPP'. They would be:

- SigGen.h
- SigGen.cpp
- SigGen_skel.h
- SigGen_skel.cpp

These files should never be modified as they are regenerated every time the IDL compiler is run. The Makefile system will also generate any necessary mezzanine IDL

into a subdirectory of 'IDL\CPP' called 'MEZZANINE'. Their generated names follow the convention:

- SigGen_PI_IFR2023.h
- SigGen_PI_IFR2023.cpp
- SigGen_PI_IFR2023_skel.h
- SigGen_PI_IFR2023_skel.cpp

Again, these files should never be modified. These files are important to the function of the CORBA interface. Those files with the '_skel' suffix provide the server-side functions and are referred to as skeletons. The ones with no suffix are called stubs and contain the code necessary for the client side to make a connection to the skeletons. The generated skeleton files contain pure virtual functions, corresponding to the IDL interface methods, which must be overridden by the implementation files described below.

3.1.3.3 Non-CORBA Related Source Files

Files should be logically named for their function or place in the system, but must not be named directly for the interface or function that is defined in the IDL file. This would eclipse the stub files and prevent the project from building properly.

The file that contains the module definition information and entry point code should be named for the module with a suffix of '_main', so in our continuing example it would be *SigGen_PI_IFR2023_main.cpp*. One field in the module definition structure is the object type that is passed to the factory to determine which DLL to load and what to create. By convention, this would be the same as the top-level IDL filename, or `SigGen_PI_IFR2023` in our example PI.

3.1.3.4 CORBA Object Servant Implementation Files

These files (usually one source and one header) provide the actual implementation of the CORBA interface and are denoted by the suffix '_impl':

- SigGen_VI_impl.h
- SigGen_VI_impl.cpp

IDL compilers usually provide a function to generate these files in "empty shell" form. This can be disastrous if not handled carefully and should only ever be done once under the best of circumstances.

3.1.3.5 Object Default Settings

The default settings for each class of module are stored in an XML file named for the class of object, not the individual instance. The object class is that defined as the type in the object details structure, so in our example the class defaults would be in the file *SigGen_PI_IFR2023.XML*. These settings are the first to be applied to each instance of this type of object and contain the most basic default settings and ranges for the specific type of device.

3.1.3.6 Object Instance Settings

The instance settings for each module are stored in an XML file named for the individual instance, as described above in section 3.1.1. In the development process and the base configuration project, the instance settings file has the same name as the class defaults file, although it is placed in a different subdirectory of the configuration tree. These settings are the second to be applied to each instance of this type of object and contain specific individual settings and range restrictions for the specific device. Ranges can be made more restrictive than the default, but not less.

3.1.3.7 Object Interface Screen Definitions

The screen definitions are stored in XML files either named for the class of object, if all instances work the same, or for specific task of one or more instances if different instances of the same class do different jobs.

3.1.4 File Locations

All Satlinx files are located in one of three standard tree structures. The ones that are of significance to the development process are the development tree, the root of which will be referred to as <*devroot*>, and the configuration tree, the root of which will be referred to as <*cfgroot*>. These trees are described in more detail in section 2.3.

3.1.4.1 Global Header Files

These files contain system information required by most or all of the modules. This includes the implementation headers for all the base classes. These files reside in '<*devroot*>/Satlinx/global'. This list of files is as follows:

CommAdapt_impl.h	Communications Adapter implementation header
Connector.h	Connector template header
DataItem.h	Data Item templates and definitions

dllmap.h	DLL loading information for the factories
OTThread.h	Definition for "OneTimeThread" object. Used for asynchronous tasks
PhysicalInstrument.h	Intermediate helper class for PI's
ReportManager.h	ReportManager
SatLinx.h	Systemwide enums for faults, events, etc.
SBase_impl.h	Implementation header for system base class.
SComponent_impl.h	Implementation header for the SComponent class
SControl_impl.h	Implementation for the SControl mix-in class
SDataTypes.h	header for SMutex, SCFrequency, and SCTime helpers
sgp4Class.h	Header for orbital propagator
StrX.h	Header for converting to and from XMLStrings
WorkQueue.h	Threadsafe, serialized queue for message control
XMLParser.h	Header for XML mapping class

3.1.4.2 IDL Files

These are the interface files for all of the devices and classes used by the system, not the mezzanine files which are used only by their own module. These files reside in '<*devroot*>/Satlinx/IDL'.

3.1.4.3 Object Default Settings

The XML files which provide the basic class-default information reside in '<*cfgroot*>/Operations/Equipment Defaults'.

3.1.4.4 Object Instance Settings

The XML which controls the individual object's setup resides in '<*cfgroot*>/Operations/System Setup'.

3.1.4.5 Object Interface Screen Definitions

The XML files which contain all of the screen definitions reside in '<*cfgroot*>/Config/Operations/Screens'.

3.2 IDL Definition

IDL files define the CORBA interface for a module so that other modules will be able to make method calls on it. These files look similar to C++ class definitions except that they do not contain data. IDL interfaces contain the methods that a remote client can call, the data structures that can be passed over those calls, and enumerations that are specific to information passing over this interface. These enumerated values, unlike in C/C++, are not guaranteed to have any rational value at all, so they must only be compared against themselves in a Boolean operation. Interfaces can also contain 'attributes' which look like data but no space is declared. They simply declare some accessor methods to get to data in the implementation class. We don't use 'attributes'. Remember that IDL is an interface specification, not a language operation. It is language independent. As such, it does not allow for any method overloading or overriding. Methods and structures must be uniquely named within the inheritance tree where they're used. If two interfaces will never be aggregated together in the same servant, they can have like-named methods.

These files are compiled during the build process, generating the stubs and skeletons in the 'CPP' or 'CPP\MEZZANINE' subdirectories of the 'IDL' main directory.

3.2.1 Standard Objects

"Standard Objects" are those that have a function and need to be accessible to others, but are not device controllers. These inherit directly from the SComponent base class. An example of this would be the Scheduler, which is shown here with the comments stripped out:

```
interface Scheduler : SComponent
{
   const string personality = "Scheduler";

   short abort();
   short done ( in long jobNumber);
   short insert ( in string objectName, in string jobName, in string jobType,
                  in string profileName, in STOVOKOR::STime startTime,
                  in long durationTime, out long jobNumber);
   short putNewPassSchedule ( in string passScheduleProvider,
                              in string passScheduleLocation);
   short start();
};
```

The '`const string personality`' is a standard we use to define the name of the interface for use in the object information structure that the factory uses. It declares a string that is locally, but not remotely, accessible. All interfaces should use this construct. The rest of the interface is standard CORBA and can be researched in a CORBA text.

3.2.2 Objects with Mix-in Personalities

Device control interfaces do not inherit directly from SComponent and are used as mix-ins. This means that it adds functionality into a class hierarchy but is not in the direct line of derivation. A 'mix-in' has no parent class. Operational classes are derived from the baseclass hierarchy and one or more mix-ins. This requires that there be a 'mezzanine' or intermediate interface defined in an IDL file that inherits from both SComponent and the mix-in(s), but otherwise adds nothing else. It exists solely to resolve issues of parentage and inheritance for the implementation class, called the Servant.

One feature of this method is that a pointer of a mix-in type can access members of the instantiated object that are implemented by this class without giving access to or being concerned by methods and members of the base class. A 'mix-in' where all the members were declared pure-virtual would be analogous to Interface Inheritance in Java.

An example of a mix-in interface would be the LNASwitch, which is shown here with the comments stripped out:

```
interface LNASwitch
{
   const string personality = "LNASwitch";

   struct lna_current_data {
      short low_limit;
      short high_limit;
      short actual;
   };

   struct LNASwtelemetry {
      STOVOKOR::STime   data_time;
      string            swname;
      boolean           swvalue;
      lna_current_data  lna_current;
      boolean           amp_alarm;
      boolean           sw_alarm;
      boolean           ps_alarm;
      boolean           bad_telemetry;
      };

   typedef sequence <LNASwtelemetry> swTelem;

   short setLNA( in boolean block, in string lnaname, in boolean lna_enable );
   short getSetting( in boolean block, in string lnaname, out boolean lna_enable );
   short getLNATelemetry( in boolean block, in string lnaname,
                          out LNASwtelemetry telemetry );
   short getGain( in boolean block, in string lnaname, out short gain );
};
```

Notice that all the methods which will result is a command or query to the device have a parameter called 'block' which allows the client to specify whether the command should be completed during the context of this call or should return immediately and let the command proceed asynchronously.

3.2.2.1 Single Mix-ins

A single mix-in mezzanine would be declared thus:

```
interface LNAC_VI : LNAController, SComponent
{
};
```

3.2.2.2 Multiple Mix-ins

A multiple mix-in would be declare thus:

```
interface LNAC_PI_Miteq : LNAController, LNASwitch, SComponent
{
};
```

Be aware that mix-in interfaces that are combined in this manner must not have methods of the same name. Otherwise there would only be one method that would be passed to the implementation class, with all the ambiguity such a scenario would accord.

3.3 Communications Objects

The CommAdapters (CA) are used to establish the physical connection to the device or simulator. CA's are usually created as one per interface type and will manage all communications on that interface type. The CA's create VirtualCircuits (VC) which handle point-to-point communications within the context of the CA. The objects requesting the connection have no knowledge of the mechanism by which the communication takes place. They just know what CA to connect to. The object gets a reference to the CA and calls `OpenCircuit()`, passing its own name and a reference to itself as a parameter. The XML for the CA contains a list of object names which will connect to it and the address or port number to connect to when they call. The CA will match the incoming name with the address and try to open that port or address. Upon success, the CA creates a VC object to manage the individual link and returns a reference to the caller. The caller will then interact solely with the VC unless closing the circuit. If the circuit is broken, the VC becomes invalid and will advise the caller of this before destroying itself. The caller may attempt to reestablish the link by asking the CA to open a new VC.

The caller makes direct calls to the VC to send data to the device. These calls return immediately without waiting for a response. The VC is always listening for communications from the device and, when something is received, will make a direct call back to the calling object via the reference which was received during the `OpenCircuit()`. The bi-directional communications is completely asynchronous.

CAs given addresses or ports on opposite side of the same communications link can connect to each other. As an example, one socket CA would be given an address of '127.0.0.1:9999', which means 'open a socket connection to myself on port 9999 if anyone's there'. The other socket CA would simply be given an address of '9999' which means 'just listen on port 9999 in case anyone's connecting'. Once these two CAs are connected, their respective callers can converse, which will be discussed in section 3.4.3.

3.4 Instrument Objects

3.4.1 Physical Instruments

Physical Instrument (PI) objects are the lowest level in the device control hierarchy. They are driver files for the specific model of device that they control. They connect to the device through CA, so they don't have any concept of HOW the conversation is taking place, but they are command and data specific to the make and model of box they support. They virtualize this information for the next level up the food chain. PIs

have one or more device interfaces defined in IDL and mixed in with SComponent in a Mezzanine IDL file. The PI implementation class is derived from this mezzanine and from the `PhysicalInstrument` helper class. The PI must have DataItems for all of the device specific variables and settings that can be preset by XML or updated by the GUI. It must have all of the device specific faults, but not the communications faults, which are supplied and managed by `PhysicalInstrument`. It must have all of the SCI functions (see Chapter 5) necessary for the GUI to update the specified fields. It must have implementations of all of the IDL interface methods. These methods will formulate and send the actual commands to the device and block if told to. If a method call is specified as blocking, a tag must be used when sending the message out to the device. This tag should be unique among all the update functions but reused by this function. If a data query is specified as non-blocking, then the query will be sent to the device, but the PI will return the current value for the requested data to the caller without waiting for the device to respond. The module must also implement the following standard methods:

`ECPush()`	used only to receive SystemEvent messages.
`processDevMsg()`	receives messages from the device via the CA. This method must parse the message, update the necessary data structures and call `PIC_SendDataDone()` with the appropriate tag to release any blocked call that is waiting for this particular response.
`updateTelemetry(bool Bad)`	called on every tick of the heartbeat timer. The parameter tells whether the PI is connected to a device (false) or not (true). The method is where all necessary data is aggregated into telemetry structures and pushed onto the event channel for the benefit of any listeners. Then non-blocking telemetry queries are sent to the device and we wait for the next timer tick.
`Synchronize()`	used to query all settings from the device.
`Reset()`	used to reset the appropriate settings back to their default values.
`Exit()`	stop the state machine timer and shutdown

`configDone()`	do what setups are necessary and start the state machine.

3.4.2 Virtual Instruments

Virtual Instrument (VI) objects are the second level in the device control hierarchy. They can be thought of as driver files for a specific function rather than device. They connect to the PI for the device that provides the functionality they require. They are the virtual interface to a specific job, such as antenna control. VI's have (usually) one device interface defined in IDL and mixed in with SComponent in a Mezzanine IDL file. The VI implementation class is derived from this mezzanine and from the 'SComponent' base class. The VI must have DataItems for all of the variables and settings that can be preset by XML or updated by the GUI. It receives all device specific faults from the PI and will transparently pass them through, but they can define their own which will override the ones they receive. In some cases the fault codes can be rebased. When a PI controls multiples of a particular interface, the fault codes for each will be in a different range. In this instance, there would be one VI for each instance exported by the PI, so the VI has an XML control called 'FAULTBASE' which allows the VI fault system to shift the fault code down to the proper range for itself. All of this works automatically. The VI must have all of the SCI functions (see Chapter 5) necessary for the GUI to update the specified fields. It must have implementations of all of the IDL interface methods. These methods will usually simply pass through to the PI connection to the same interface method exported by the PI, then update the internal data with whatever is returned. The module must also implement the following standard methods:

`execConfig()`	this is called by the 'Configurator' object to tell the VI to issue the necessary commands to the PI to set up for a mission. The data will already have been sent in an XML block and automatically parsed into the internal datastores.
`Exit()`	close the PI connection and shutdown.
`ECPush()`	receive all telemetry and data structures that were pushed onto the event channel by the PI. Also receives faults, PI object status, PI connector feedback and SystemEvents.
`processRoleChange()`	receive role change request (Active,Backup,NotUsed) and act on them or pass them through to the PI.

`onHeartbeatChange()`	received when the heartbeat has been updated by the GUI. Send it down to the PI and reset the local value to whatever is returned.
`configDone()`	initiate the connection to the PI.
`heartbeatRate()`	external CORBA interface to set the heartbeat. Send it down to the PI and reset the local value to whatever is returned.
`Synchronize()`	pass through to the PI.
`Reset()`	pass through to the PI.

3.4.3 Instrument Simulators

Instrument simulators (SIMs) are code modules that simulate the behavior of the actual device for which the other code modules are being developed. Two CAs are configured to connect to each other, either on the same machine or two different ones, and the SIM is attached to one and the PI to the other. The SIM is usually a simple, reactive object that connects directly to the CA and has no process loop, rather than using the PhysicalInstrument helper class which does both. The SIM's job is simply to sit and wait for commands or queries, and then respond as the device would. Some complex device, such as those capable of autonomous operation or those that emit unsolicited status messages, will require an internal process loop. There is almost never a reason to derive a SIM from the PhysicalInstrument class because of the extra baggage that it entails.

3.5 Function Objects

Function objects are those that have their own purpose, rather than controlling a device. They have an interface that pertains to their own functionality, rather than that of the device they control (since they don't). As such, their IDL uses direct inheritance from the 'SComponent' interface, rather than mix-in inheritance.

3.5.1 Virtual Functions

Virtual Functions (VF) are objects that can (but not always) be higher level control objects and can use VIs to get their job done. They remain in the system at all times and their operation is usually continuous, such as the Asset Manager. VF's will usually maintain connections to multiple VI's to accomplish their task. Unlike PI's and VI's,

VF's don't have a predefined set of methods which must be implemented, other than the standard 'Exit()' and 'configDone()'.

3.5.2 Virtual Tasks

Virtual Tasks (VT) are higher level objects which are always controlled by objects higher up the food chain than they are and implement the IDL `Task` interface:

```
interface Task : SComponent
{
   const string personality = "Task";

   enum eHaltType
   {
      eABORT,  // Stop the job, put all equipment into stand-by and non-active mode
               // of operation.
      eLEAVE,  // Stop the job, leave the equipment as is.
      eSAFETY, // Stop the job, put all equipment into stand-by and non-active mode
               // of operation.
      ePROFILE // Use what profile says to do.
   };

   short   Startup ( in string profileName, in string jobType, in long jobNumber,
                     in STime startTime, in STime stopTime );

   short   Shutdown ( in eHaltType method, in long jobNumber );
   short   GetStatus ();
   string  GetReqPersonalities ();
}; // end of interface
```

They use VIs to accomplish their jobs and may remain in the system at all times, but have a defined start and stop to their activities. Examples would be the operational tests. When a controlling entity like the scheduler is directed to activate a task, it obtains a reference to that task object and calls `Startup(...)`. This includes the name of the profile that the task will use to configure its VI's, some job control information, and the true start and stop time of the operation. The task is responsible for accurate adherence to these times, as the controlling objects don't have direct control and cannot give the task precisely timed start or stop instructions. The `Shutdown(...)` instruction contains a halt type and job number. The task will match the job number against what is currently running, to ensure that the correct operation is being halted. The halt type is used to determine whether it's an orderly shutdown, a panic stop, or just 'hands-off', letting things continue as they are but relinquishing control. The `GetStatus()` method returns whether the operation is running or not. The `GetReqPersonalities()` method is used to request a list of IDL interface names corresponding to the VIs that the task will use to accomplish its job. The personalities are used rather than explicit VI names because another type of object may be supplying that particular behaviour.

3.5.3 Persistent and Non-persistent Objects

Persistent and non-persistent objects are different from CORBA persistent references. All of our objects use CORBA persistent references. This ensures that when objects stop or die and are restarted, references held by other objects are still valid and operations can resume as before.

The persistence we discuss here is within the SatLinx system. Some objects are persistent because the operation of the system depends on their presence. All of the PI's, VI's, and manager VF's are persistent because they are required at all times. If they fail for some reason, they are restarted and resume their position. Non-persistent objects, however, are registered at startup but don't reside in the system at all times. They are started at need and then stopped when done. Examples of this would be the editors, tests, and other tasks, which are only run when specifically called upon and then can be stopped.

Initialization of non-persistent objects is handled in the config manager, as is described in section 4.4.2.6. It starts deferred objects when needed without any special coding requirements for the module. If activation is built into the connector this overhead can be removed from the calling module.

Basically, the difference for the module itself is in the termination handling. The task must deregister with the config manager and then terminate. This will reset the module's state within the config manager to be deferred rather than active, and it can be restarted again in the normal manner.

Deregistration is something that all modules should do upon exit, although this has not been observed until now. If this is added to the base class there will be one less difference between persistent and non-persistent modules. The remaining consideration is when the module should exit. This is the aspect that needs more work. The simple case is when a module is idle and there are no active connection to it. How this is evaluated remains undocumented or unexplored.

4 Satlinx Infrastructure

4.1 Stovokor Global Data and Type Definitions

4.1.1 File Manager File Types

```
typedef enum { kFMConfigFile, kFMScreenFile, kFMMissionFile }
             FMFileType;
```

`FMFileType` is used when loading and saving files via the file manager. This value tells the file manager what the category of the file is, and thus determines its location in the file hierarchy.

4.1.2 Configuration Parameter Descriptor Structure

```
typedef struct { bool isPublic; bool isInstance; string cfgText; }
               ConfigItem;
```

`ConfigItem` is used when accessing configuration data. All configuration data are represented as strings. The `isPublic` flag, which determines whether this ConfigItem is available to outside processes (ie. the ProfileEditor), is set by an attribute in the XML file. The `isInstance` flag, which describes whether this ConfigItem was initialized by a default file or an instance file, is set based on the file from which it is set.

4.1.3 SCEventSink Mapping Constants

```
const int kSysEvents     = 1;
const int kFaultEvents   = 2;
const int kUserEventBase = 100;
```

SCEventSink objects are used to receive data from and Event Channel, and are discussed more fully in section 4.2.1.8.1. These constants are used when constructing SCEventSink objects.

4.1.4 Date Specific Data Repository Directory Name Arrays

```
extern char * monthDirs[];
extern char * dayDirs[];
```

Some modules produce telemetry or logging data for storage on the local drive. All such data is stored in the data repository tree. Some data is stored in monthly directories, others in daily directories. These two arrays contain the names of the date specific directories. For the most part, they should not need to be referenced directly, since helper methods in the base classes can generate complete path names from data structures.

4.2 Stovokor Global Functions

4.2.1 CORBA Object Name Resolution Methods

```
CORBA::Object_ptr resolveName ( const char * pName );
CORBA::Object_ptr resolveName ( const char * pName,
                                CosNaming::NamingContext_ptr nc );
```

These are global versions of the name resolution methods. The first form returns an object reference to the an object bound in the local naming context. The second form returns an object reference to the an object bound in the specified naming context.

4.2.2 SFrequency Conversion Methods

```
SFrequency& toHz ( SFrequency& freq );
SFrequency& tokHz ( SFrequency& freq );
SFrequency& toMHz ( SFrequency& freq );
SFrequency& toGHz ( SFrequency& freq );
```

These methods convert the passed SFrequency objects range representation.

4.2.3 STime Arithemetic Operator Methods

```
SCTime operator- ( const STime&, const STime& );
SCTime operator- ( const STime&, int );
int operator- ( int, const STime& );
SCTime operator+ ( const STime&, const STime& );
SCTime operator+ ( const STime&, int );
int operator+ ( int, const STime& );
```

These are the arithmetic operator overloads for the SCTime data class. The arguments and return values are either STime objects or seconds.

4.2.4 Fault Record Relational Operator Methods

```
typedef struct { SBase::eFaultStatus severity; short code; } FaultEntry;
```

This structure is used to combine fault record fields into a sortable entity.

```
bool operator>  ( SBase::eFaultStatus f1, SBase::eFaultStatus f2 );
bool operator<  ( SBase::eFaultStatus f1, SBase::eFaultStatus f2 );
bool operator>= ( SBase::eFaultStatus f1, SBase::eFaultStatus f2 );
bool operator<= ( SBase::eFaultStatus f1, SBase::eFaultStatus f2 );
```

These are the relational operator methods for the fault severity IDL enumerates.

```
bool operator>  ( FaultEntry f1, FaultEntry f2 );
bool operator<  ( FaultEntry f1, FaultEntry f2 );
bool operator>= ( FaultEntry f1, FaultEntry f2 );
bool operator<= ( FaultEntry f1, FaultEntry f2 );
bool operator== ( FaultEntry f1, FaultEntry f2 );
bool operator!= ( FaultEntry f1, FaultEntry f2 );
```

These are the relational operator methods for the fault entry combinatorial structure.

4.3 Object Base Classes

4.3.1 Class SBase_impl

4.3.1.1 Overview

This is the base class for all Stovokor modules. It implements logging, fault management, configuration and system administration functionality required for all objects that are part of the architecture.

All generic functionality that is unrelated to the event channel publish/subscribe model or the GUI interface should be found in this class, or its related classes. As new standard functionality is added consideration should be given to placing it in the SBase implementation, rather than allowing it to be multiply implemented in modules. The intent has always been to make the base classes rich in function to simplify the development of working modules.

4.3.1.2 CORBA Interface

```
interface SBase
```

This is the base IDL interface for all Stovokor modules. It is inherited by the SComponent interface, and defines logging and system administration methods that are either implemented in the base class implementation classes or must be implemented by all objects that are part of the architecture.

4.3.1.2.1 Constants

```
const short kLogTypeAll       = 7;  // All
const short kLogTypeCondition = 4;  // Something has happened
const short kLogTypeData      = 2;  // Logging a data value
const short kLogTypeFlow      = 1;  // Just tracking logic flow
const short kLogTypeNone      = 0;  // No category
```

Log type constants. These are arranged as bit flags, so that a log message may be categorized as belonging to several types, and will thus be logged in all log files to which it applies.

```
const string personality = "SBase";
```

This is the standard personality string defined for all IDL interfaces.

4.3.1.2.2 Enums

```
enum eFaultStatus { FaultCleared, FaultTransient, FaultWarning,
                    FaultMinor,   FaultMajor,   FaultCritical,
                    FaultIndeterminate, FaultEvent };
```

Fault levels. The order above is in ascending order of severity. FaultCleared indicates no fault, and FaultTransient is a passing fault event that does not persist. The other levels remain set until explicitly cleared.

```
enum eLevels { None, Failure, Error, Warning, Info, Debug };
```

Log levels. The order above is in descending order of severity. The module can be configured to pass to the logger only those matching a specified level and above.

4.3.1.2.3 Structs

struct FaultReport

```
struct FaultReport
{
    short seqnumber;
    eFaultStatus severity;
    string source;
    eModuleType sourceType;
    STime eventTime;
    short eventType;
    short probableCause;
    short problemCode;
    string<127> affectedItem;
    string<127> dataFieldOne;
    string<127> dataFieldTwo;
};
```

This is the structure used to propagate faults and events over the system fault channel. The values are set by the FaultObject that is doing the sending (section 4.3.1.8.2).

struct LevelSet

```
struct LevelSet
{
    eLevels level;
    eLevels sync;
    short type;
    string modName;
};
```

This is the structure used to change logging levels. It is sent by the logger over the system event channel, and the affected modules will update their logging filter parameters based on the fields. This processing is handled in the SBase class implementation.

4.3.1.2.4 Attributes

attribute eLevels LogLevel;

This defines the lowest severity log level the object will pass to the system logger.

attribute eLevels SyncLogLevel;

This defines the lowest severity log level the object will pass to the system logger as a synchronous log message. A synchronous log message is one that causes the system logger to flush its log files to disk to ensure that they are recorded. The synchronous log level should be reserved for more severe conditions where it is more important to have the message written out than to avoid the overhead of the I/O.

```
attribute short TypeFilter;
```

This is the combined type flags, as described above.

4.3.1.2.5 Operations

```
oneway void Exit();
```

This method is defined as a pure virtual function in the SBase skeleton class and is not implemented in the SBase_impl servant class. Intermediate classes will not overload this, so each real class implementation must provide an implementation. The method is called when the system is closing down and the object must terminate. Any persistent data must be saved at this time, and the object should call `SBase_impl::shutdown()` to shutdown the ORB.

```
StringSeq GetPersonalities();
```

This method returns a sequence of strings that identify all of the different CORBA interfaces a module exports.

```
void Ping(in string name, in SBase sb);
```

This is the function called to check that a module is still alive. The default behaviour implemented in the SBase implementation is to reply to the caller by calling its `Ping()` with the called object's own name and IOR. Derived object will not normally overload this method. The configuration manager object overloads this behaviour to register the replies in a map of active modules. Modules that fail to respond to several pings in a row will be marked as dead and restarted.

Parameters:

`name`	The published name of the object
`sb`	The object reference of the calling object.

```
void TraceDump();
```

This dumps the contents of the trace buffer. At present the tracing system is not completely implemented.

```
eFaultStatus getFaultStatus(in short cmdType);
```

This method returns the current fault status of the object, based on its most severe active fault object.

```
boolean isReady();
```

This checks to see if the remote object is configured and happy. A default implementation of this method is provided in the SBase implementation class, but this could be overridden by derived classes. The default implementation only cares that the modules has been basically configured and can take CORBA calls. The developer may want to override if there are operation concerns.

```
void propagateFaults();
```

This method causes all fault objects to post their status on the fault channel.

4.3.1.3 Static Member Functions

4.3.1.3.1 ORB Reference Accessor

```
static CORBA::ORB_ptr orb();
```

This method returns a reference to the module's ORB.

4.3.1.3.2 POA Reference Methods and Accessors

```
static PortableServer::POA_var basePOA();
```

This method returns a reference to the base POA for the executable. This may not be the POA on which the object was activated. That POA can be obtained with the Orbacus defined `_getDefaultPOA()` method.

```
static void addPOA ( char *psz,  PortableServer::POA_var poa );
static void addPOA ( string s, PortableServer::POA_var poa );
```

These methods associate a specific POA reference with a module name, and are used by the factory object.

If other modules fill a factory role and incarnate SBase-derived CORBA objects with unique POA requirements on a POA other than the base POA or the module's own POA, they should follow the following steps:

1. Create the new POA with the required properties.
2. Add the association for the module to be created to the POA by calling `addPOA ( module_name, POA_reference ).`
3. Create the CORBA object's servant object.

The SBase constructor will then activate the object on the correct POA.

N.B. If none of the above makes any sense to you, then clearly you are not sufficiently expert in advanced CORBA programming to even think about using these methods! Please acquire and study "Advanced CORBA Programming with C++" by Henning & Vinoski.

4.3.1.1.3.3 Common Object Name Accessors and Mutators

```
static const char * nodeName();
static void nodeName(string& s);
```

These methods return or set the name of the node of which this module is a part. In practice the name is set when the executable is started, and once set cannot be changed.

```
static const char * nodeCtl();
static void nodeCtl (string& s);
```

These methods return or set the name of the node controller for the node of which this module is a part. In practice the name is set when the executable is started, and once set cannot be changed.

```
static const char * logName();
static void logName (string& s);
```

These methods return or set the name of the system logger for the node of which this module is a part. In practice the name is set when the executable is started, and once set cannot be changed.

```
static const char * revName();
static void revName (string& s);
```

These methods return or set the name of the software revision manager for the node of which this module is a part. In practice the name is set when the executable is started, and once set cannot be changed.

```
static const char * cfgName();
static void cfgName (string& s);
```

These methods return or set the name of the configuration manager for the node of which this module is a part. In practice the name is set when the executable is started, and once set cannot be changed.

```
static const char * FMName();
static void FMName (string& s);
```

These methods return or set the name of the file manager for the node of which this module is a part. In practice the name is set when the executable is started, and once set cannot be changed.

```
static const char * nodeInfo();
static void nodeInfo(string& s);
```

These methods return or set the name of the node information manager for the node of which this module is a part. In practice the name is set when the first module in the executable is registered, and once set cannot be changed.

```
static const char * appName();
static void appName (string& s);
```

These methods return or set the name of the startup application for the node of which this module is a part. In practice the name is set when the first module in the executable is registered, and once set cannot be changed.

```
static const char * SWPath();
static void SWPath  (string& s);
```

These methods return or set the software path name for the node of which this module is a part. In practice the name is set when the first module in the executable is registered, and once set cannot be changed.

```
static const char * cfgPath();
static void cfgPath (string& s);
```

These methods return or set the configuration path name for the node of which this module is a part. In practice the name is set when the first module in the executable is registered, and once set cannot be changed.

```
static const char * dataPath();
static void dataPath(string& s);
```

These methods return or set the data repository path name for the node of which this module is a part. In practice the name is set when the first module in the executable is registered, and once set cannot be changed.

4.3.1.1.3.4 Naming Context Reference Accessors and Mutators

```
static CosNaming::NamingContext_ptr getNC();
static void setNC( CosNaming::NamingContext_ptr nc );
```

These methods return or set the object reference for the naming context in which the module is registered (or bound). This is the naming context of the individual node of which this module is a part. In practice the reference is set when the executable is started, and should not need to be changed within a module's implementation.

```
static CosNaming::NamingContext_ptr getRootNC();
static void setRootNC( CosNaming::NamingContext_ptr nc );
```

These methods return or set the object reference for the naming context in which the node of which this module is a part is registered (or bound). In a cluster of nodes, or an installation with a two tier control hierarchy, this is the naming context of the cluster. See the section on the Naming Service Federator (4.5.2) for more details. In practice the reference is set when the executable is started, and should not need to be changed within a module's implementation.

4.3.1.3.5 Startup Parameter Parser and Accessors

```
static void getArgs ( int & argc, char **argv );
```

This method processes the command line arguments passed to the factory or non-factory module executable and saves them for access as static data members. The arguments that are recognized as SBase class parameters are removed from the argument list before the function returns, and the value of `argc` is adjusted to reflect the remaining parameters. No command line arguments can be passed to a module other than those the ConfigMgr already knows about. Those that might be retrieved by higher level application code would likely have no meaning.

```
static int  orbThreads();
```

This method returns the number of threads defined for the ORB thread pool for the executable. It is set from the command line arguments, and is used by the factory and all non-factory loaded module startup code.

```
static bool useDebugConsole();
```

This method returns a Boolean indicating whether the executable is to have a console window. It is set from the command line arguments, and is used by the factory and all non-factory loaded module startup code.

4.3.1.3.6 Debug Console Creator

```
static void createDebugConsole ( const char * name );
```

This method creates Windows console for the executable and attaches it to the standard output stream, thus allowing `cout` and `printf` data to be displayed.

4.3.1.4 Public Member Functions

4.3.1.4.1 Module Shutdown Method

```
virtual void shutdown();
```

This method must be called in the implementation of the exit method, and in the destructor, of all SBase derived classes. It removes the binding for its name from the naming service, informs the configuration manager that it is going away, and then deactivates the object. If it is the only CORBA object active in the process the ORB is shutdown and the process terminates.

4.3.1.4.2 Module Name Mutators and Accessor

```
const char * modName();
void         modName ( const char * objName );
```

This method returns the name of the module, as it is registered with the Naming Service. Now you can know who you really are!

```
const char * fuzzyName();
void         fuzzyName ( const char * WFname );
```

This method returns the human-friendly name of the module. This is set in the `Register()` method from the GUINAME setup parameter. If the parameter does not exist the fuzzy name will be the same as the module name.

4.3.1.4.3 Module Type and Personality Methods

```
void addPersonality ( const char * pPers );
```

This method adds a personality string to the list of personalities implemented by the module. Personality strings are defined in the IDL definitions for all interfaces. The objects details structure for each module should contain a list of personalities based on the IDL inheritance. This method is called by the factory or non-factory module executable. The list of personalities is returned by the SBase CORBA interface `GetPersonalities()` method as described above.

```
STOVOKOR::eModuleType modType();
void modType ( eModuleType type );
```

Each module has a certain type, based on the role it plays in the system. These types are defined in the Stovokor IDL module, and are as follows:

`StovokorUnknown` Undefined module type.

`StovokorPI`	Physical instrument object.
`StovokorVI`	Virtual instrument object.
`StovokorVF`	Virtual function. These are modules that are not associated with a specific instrument and serve a specific function in the system. Some are full-time functions, such as loggers and managers, and are started when the system is started and only stop when the system is shut down. Other modules are transient by nature, such as editors, and are started on demand.
`StovokorCA`	Communications adapter.
`StovokorSIM`	Instrument simulator
`StovokorOps`	Node operations manager. This is a special case, and in all respects is a full-time virtual function. Due to its role in fault report generation and propagation it needs to be discriminated from other virtual functions.
`StovokorBase`	Base class object. These are objects that are directly derived from the SBase class only, rather than from the SComponent class. Base objects have no event channel associated with them, and cannot support a GUI. Event channel functionality is supplied by the SComponent class.
`StovokorTask`	Task object. These objects implement the Task IDL interface, and are started and stopped under the control of the system scheduler.

The mutator is called by the factory. The value to set is obtained from the object details structure.

4.3.1.4.4 Data Repository Path Methods

```
void getPath ( string& path, STOVOKOR::FileManager::ePaths category,
               const char* fname );
void getPath ( string& path, STOVOKOR::FileManager::ePaths category,
               const char* pname*, SCTime& date );
```

As described above, some modules need direct disk access to reports and data files. These methods translate filenames and file categories into complete paths. The arguments are as follows:

`path` the string that will contain the full resolved path

`category` the file category

`fname` the name of the file

`date` the date to determine the daily or monthly directory in the data repository

4.3.1.4.5 System Logger Methods

```
bool initLogger();
```

This method registers the module with the system logger and allows log messages to be sent to the logger. Before the module successfully registers with the logger it caches all log messages locally. The cache is flushed when registration succeeds.

```
virtual void Log ( int LineNo, CORBA::Short Type,
                   STOVOKOR::SBase::eLevels Level, char *logString,
                   ... );
```

This is the data logging method, and is the preferred and correct way to add debugging and other logging information to the code. (Disregard whatever you may see in existing code!)

The first parameter is the line number where the message was generated. This is a convenience and any value can be used. Typically you would use the `__LINE__` macro.

The next two parameters categorize the log call by type and severity. The types are:

```
kLogTypeNone
kLogTypeFlow
kLogTypeData
kLogTypeCondition
```

These are arranged as bit flags, so that a log message may be categorized as belonging to several types, and will thus be logged in all log files to which it applies. The severity levels are:

```
STOVOKOR::SBase::Failure
STOVOKOR::SBase::Error
STOVOKOR::SBase::Warning
STOVOKOR::SBase::Info
STOVOKOR::SBase::Debug
```

The rather long winded designation is due to the fact that these values are defined in the SBase IDL interface, so the full scope must be explicitly given.

The remaining parameters are standard `printf` arguments, with a format string and optional arguments.

The messages will be recorded in whatever target files are defined for the log level and category in the logger configuration. (This will be expanded with a browser/GUI interface soon.)

4.3.1.4.6 CORBA Object Name Resolution Methods

```
CORBA::Object_var ResolveName ( const char * pName );
CORBA::Object_ptr ResolveName ( string &name );
```

These methods resolve a name to a CORBA::Object reference. This can then be narrowed to a reference of the required type. Since module interconnections are defined by module name, these methods will be used often.

4.3.1.4.7 Configuration Parameter Accessors

```
string getCfgItem ( const string & cfgName );
bool getCfgItem( const string & cfgName, ConfigItem & cfgItem );
```

The two forms of `getCfgItem` get either the string associated with the parameter name, or the data structure containing the string and flags. The `configItem` structure is defined above. If the config item does not exist the first form returns an empty string, and the latter form returns false.

4.3.1.4.8 Event Channel Methods

Event channels are fundamental to the interoperation of Satlinx modules. The only mechanism by which a module can communicate with an upstream module is by posting events on its event channel. In addition, there is a fault channel on which all fault and event records are posted, and a system event channel on which system-wide events are posted. Every module is a listener (consumer in CORBA-speak) on the system events. Most system events are handled at the base class level, but unhandled system events are passed up to the module. The following methods provide an interface to the system-wide event channels. The interfaces to the module's own event channel are provided in the SComponent class.

```
OBEventChannelFactory::EventChannelFactory_ptr
getECFactory (const char *pnode=NULL);
```

This method returns a reference to the Orbacus event channel factory. It is used by the base classes, and would only be needed by a module needing to directly create or delete event channels, which is not recommended or needed in the architecture as it stands.

```
string getSysECName();
```

This method returns the name of the system event channel. It is used by the base classes, and is not needed by derived modules.

```
bool processSystemEvent ( const CORBA::Any& any );
```

This method provides SBase level handling of messages received on the system event channel. It is overloaded at the SComponent level. It should not be called from outside the infrastructure.

```
void sendSystemEvent ( CORBA::Short );
void sendSystemEvent ( CORBA::Any& );
```

These methods post events on the system event channel. Simple system events are defined as `CORBA::Short` in *satlinx.h*. All other events will need to be inserted into a `CORBA::Any` object before posting.

```
int getFaultBase();
```

This method returns the base for generated fault codes for the module. This is set in the `Register()` method from the FAULTBASE setup parameter. If the parameter does not exist the fault base is 100. For physical instrument objects that implement multiple similar instruments within them, particularly those implementing redundant subsystems, each like instrument will have a separate virtual instrument above it in the hierarchy. The fault base allows these multiple instruments to be distinguished from each other.

```
void SendFault ( STOVOKOR::SBase::eFaultStatus fault, eEventType type,
                 eFaultProbableCause cause, int code, const char* dF1, const
char* dF1);
```

This method creates and sends a fault record over the fault event channel. This function need not be used, as a more convenient implementation of fault management exists with the fault objects described in section 4.3.1.8.2.

```
SendFault(STOVOKOR::SBase::FaultWarning, kProcessingError, kFileMissing,
        kFileMissing, mp_topologyFile->value().c_str(), "Missing Topology file - exiting");
```

```
void SendEvent ( eEventType type, eEventProbableCause cause, int code, const
char* dF1, const char* dF2);
```

This method creates and sends an event record over the fault event channel. The parameters correspond to the fault record fields defined in the individual module design specifications. An example pulled from the FileManager would be:

```
SendEvent(kStatusEvent,kStateChange,eFileRead,name,"");
```

```
void forwardFault ( const struct STOVOKOR::SBase::FaultReport& pFR );
```

This method is provided as an upcall for the SComponent class. The SBase implementation does nothing.

4.3.1.4.9 Data Item Operations and Accessors

```
void addDataItem ( const char * name, DataItem * item );
void removeDataItem ( const char * name );
void initDataItems();
void setDataItems();
void saveDefaultXML();
void saveCustomXML();
template <typename T> bool isValid ( const string &strParamName,
                                     const T value);
bool isValueAsStringValid ( const string &strParamName,
                            const string &strValue );
void getValueAsString ( const string &strParamName,
                        string &strValue );
void getDefaultAsString ( const string &strParamName,
                          string &strValue);
bool setParamValueAsString ( const string &strParamName,
                             const string &strValue );
DataItem * getParam ( const string &strParamName );
template <typename T> bool getParamValue ( const string &strParamName,
                                           T &value );
template <typename T> bool setParamValue ( const string &strParamName,
                                           T &value );
```

These methods are all related to data items, and are described in more detail in section 4.3.3.

4.3.1.5 Virtual Member Functions for Optional Overriding

```
CORBA::Boolean isReady();
```

This is a default implementation of the SBase CORBA interface method. It returns the setting of the protected member variable bool m_bOk. This is set after the module completes it registration with the configuration manager.

4.3.1.6 Pure Virtual Member Functions for Required Overriding

4.3.1.6.1 CORBA Interface Methods

```
virtual void Exit() = 0;
```

The SBase CORBA interface `Exit()` method must be implemented for all SBase derived classes. At a minimum it must call the `shutdown()` method described above. Any additional termination processing should also be included.

4.3.1.6.2 Non-CORBA public pure virtual methods

```
virtual void ECpush ( int type, int instance, const CORBA::Any& any );
```

This method must be implemented by all modules. It is called from the event sink objects described below. Each event sink is instantiated with a unique (within the module) type and instance. The event sink provides the `push()` method required by the event channel. It than tags the data and passes to the module's servant class by calling its `ECPush()` method.

All modules have an event sink attached to the system event channel (type `kSysEvents`), and should handle module specific system events. Most modules will have one or more event sinks attached to downstream modules. The base classes handle generic system events.

The typical implementation of ECPush() would be a switch statement containing any extractions and method calls, such as the following:

```
void ECPush ( int type, int instance, const CORBA::Any& any )
{
   switch ( type )
   {
   case kSysEvents:
      int ev;
      if ( any >> ev )
      {
         handleSystemEvent ( ev ); // module specific function to handle event
      }
      break;

   case kPIEvents:                 // module specific type (based from kUserEventBase)
      struct PIXX_Telemetry telemetry;
      if ( any >> telemetry )
      {
         handleTelemetry ( instance, telemetry );  // module specific function
      }
      break;
   }
}
```

The type and instance variable are passed by the individual EventSink object that made the call.

4.3.1.6.3 Protected pure virtual methods

4.3.1.6.3.1 Configuration Completion Method

```
virtual void configDone() = 0;
```

This method is called by the base class when the initial module configuration has been completed. At this time the module has registered with the configuration manager, and loaded all its configuration data. Those data items created with names matching configuration entries have been initialized. All screens have been loaded. Module specific startup operations should now occur.

Physical instrument modules will call the PhysicalInstrument class methods to establish the connection to the device via the communications adapter. (See section 4.3.6 for details.)

Any threads that need to be started for the module must be started by this point. Once this method returns the module is now under the control of the ORB, and will only respond to remote method calls. Active objects will now be running in the context of their threads.

4.3.1.7 Protected Data and Member Functions

These methods and data are only available to modules derived from the SBase hierarchy. The data members and methods listed are the only ones that should be used by non-infrastructure classes.

4.3.1.7.1 Parameter Map and Access Methods

```
typedef map<string, DataItem*> ParameterMap;
ParameterMap::iterator PMapFirst();
ParameterMap::iterator PMapEnd();
void PMapAdd ( string name, DataItem * pItem );
```

The parameter map provides a name lookup for the DataItems in the module. Operations that need to iterate through the data items use an iterator that it is initialized with `PMapFirst()`. The end of map condition is tested by comparison with `PMapEnd()`. The following loop illustrates how this would be done:

```
for ( ParameterMap::iterator it = PMapFirst(); it != PMapEnd(); ++it )
{
   ...
}
```

4.3.1.7.2 Common Module Object Reference Accessors

```
STOVOKOR::CfgMgr_ptr       getCfgMgr();
STOVOKOR::FileManager_ptr  getFileMgr();
STOVOKOR::NodeControl_ptr  getNodeCtl();
```

The above methods return object references for the configuration manager, the file manager, and the node controller, respectively. If the user needs access to these service modules, these accessor functions should be used. If the relevant module is unavailable a nil reference is returned.

4.3.1.7.3 System Logger Trace Method

```
void  Trace ( int LineNo, int ParmCount, ... );
```

This method is part of the internal logging mechanism of the object. It inserts a series of numbers into a trace buffer for later analysis. (This is not currently being used, and is much more cryptic than the logging mechanism described below.)

4.3.1.7.4 Update Configuration Parameters from XML Data

```
void parseParameterXML ( string & xml );
```

This method takes a string containing a block of XML and updates the configuration data map with its contents. New configuration items are added if necessary. (This is of limited usefulness to derived classes.)

4.3.1.7.5 Retrieve Configuration XML

```
string getXMLData ( STOVOKOR::FMFileType fileType, string& xmlName );
```

This method retrieves a block of XML from the file manager. The `fileType` is used by the file manager to determine location. The `xmlName` identifies the data repository. (At present this is a file. Who knows, later it may be a record in a database.)

4.3.1.8 Related and Subordinant Classes

4.3.1.8.1 SCEventSink

4.3.1.8.1.1 Public Member Functions

4.3.1.8.1.1.1 Constructor

```
SCEventSink ( SComponent_impl * delegate, int type, int instance );
```

This is the constructor for the event sink object. An event sink object must be instantiated for each event channel the module needs to monitor. The delegate parameter is simply the 'this' pointer of the module object. The type determines the tag that will be passed to `ECPush()`. These must be unique within the module. The constant `kUserEventBase` defines the starting value, and additional event sinks would have incrementing type numbers. The instance parameter permits the module to handle data coming from multiple similar modules, as in the sample `ECPush()` above.

4.3.1.8.1.1.2 Attach to Event Channel

```
bool attach ( const char * chname );
```

This method attaches the event sink to the event channel. The channel must already exist, or the function returns false.

4.3.1.8.1.1.3 Create Event Channel and Attach

```
bool createAndAttach ( const char * chname );
```

This method attaches the event sink to the event channel. If the channel does not exist it is created. This is the normal way to attach an event sink to a channel since it is independent of startup sequence.

4.3.1.8.2 Class FaultObject

The fault object is an encapsulation of the fault record that is sent over the fault channel to the fault manager, as well as over module's event channel to upstream objects.

Fault objects can be created implicitly, from fault records, or explicitly with the parameters that would be passed in the fields of the fault record. The SBase class maintains a collection container object for all fault objects, and can send the records over the required event channels on demand. Once created, fault objects can be set and cleared as needed, and the action of setting and clearing the fault object results in its propagation over the relevant event channels.

4.3.1.8.2.1 Public Member Functions

4.3.1.8.2.1.1 Constructors

```
FaultObject ( SBase_impl*, SBase::eFaultStatus, eEventType,
              eFaultProbableCause, int, const char*, const char* );
```

This is the explicit constructor for a persistent fault object. The SBase pointer identifies the object that is the source of the fault. Most typically, fault objects are created in the constructor of the module's servant object, and the pointer is initialized with `this`. The remaining parameters for the constructor are the fields that will be included in the fault report. The two text strings for descriptive text (two allow for flexibility) are default strings that will be included in the fault report if no override is set. eEventType and eFaultStatus are defined in Satlinx.h

```
FaultObject ( SBase_impl*,
              const struct STOVOKOR::SBase::FaultReport* );
```

This is the implicit constructor for a transient fault object. The SBase pointer identifies the object that is the source of the fault, and will also most typically be initialized with `this`. Unlike the persistent fault object, the transient object is intended to be used to regenerate a fault received from a downstream object. A virtual instrument object, which is visible to the modules in the hierarchy above it, would use this form to generate its own version of a fault as received from a downstream physical instrument.

For the object to be created and added to the module's fault object collection the fault report must satisfy certain criteria:

- The fault has to be set, not cleared.
- The fault report is not a duplicate of a prior fault report.
- The fault code is in the range defined for public fault reports. The range is defined by the fault base setup parameter, and extends from *faultbase* to *faultbase+99*.

If the fault is cleared, and a corresponding object exists, the existing fault object is cleared and then deleted. Transient fault objects are not retained in a cleared state, unlike persistent fault objects which must be explicitly deleted in the owning objects destructor.

4.3.1.8.2.1.2 Status Accessor

```
bool isSet();
```

This method returns the current state of the fault object, set (`true`) or cleared (`false`).

4.3.1.8.2.1.3 Mutators

```
void set();
void set ( const char*, const char* );
void set ( SBase::eFaultStatus );
void set ( SBase::eFaultStatus, const char*,const char* );
```

There are four forms of the `set()` method. The simple form just sets the fault object with the severity and text messages defined in the constructor, and calls `SBase_impl::SendFault()` with the stored parameters. The second form replaces the default text fields before the fault report is sent. The new text becomes the default for future `set()` and `clear()` calls. The other two versions override the default severity of the fault object. The new fault severity becomes the default for future `set()` and `clear()` calls.

```
void clear();
void clear ( const char*, const char* );
```

There are two forms of the `clear()` method. The simple form just clears the fault object with the text messages defined in the constructor, and calls `SBase_impl::SendFault()` with the stored parameters and a severity of `STOVOKOR::SBase::FaultCleared`. The second form replaces the default text fields

before the fault report is sent. The new text becomes the default for future `set()` and `clear()` calls.

```
void send();
```

This method forces a fault report to be sent, corresponding to the current settings of the fault object.

4.3.2 Class XMLParser

4.3.2.1 Overview

In most cases the `SBase::getCfgItem()` function will suffice for most applications that need to read setup and configuration information from XML configuration files. These are loaded automatically on system startup, and there is no additional code required in order to load the correct information. For some applications, however, additional data files are required that are not addressed with the standard configuration scheme. In order to simplify access to the XML configuration and data files used by the system, it was decided to write a set of routines to allow developers to use a hierarchical convention, and to free them from having to parse the files directly. For simplicity's sake this functionality was incorporated into a new class, XMLParser. (Please note that all examples are located in section 4.3.2.4, Usage Notes, on page 115.)

The XMLParser class encapsulates the contents of one or more XML files and presents them to the user in an easy-to-use manner. All of the elements in the source XML files are organized in a map with hierarchical strings as keys. Each object in the map can be either an attribute or element object, or a key object. Nested XML statements will result in one or more key nodes being produced, which the user will have to traverse separately. Repeated occurrences of keys with the same name (hereafter referred to as 'sequences') are stored as a linked list, starting with the first occurrence. Versioning of information is supported, in that up to 254 separate levels of information can be stored for each item in the memory structure. This was done to allow the parser to overlay higher-priority information on top of existing information, while still allowing access to the lower-priority information as required. The parser can combine several XML files into a single memory structure, and can write out the its contents as formatted XML as well.

The XMLParser class does **not** preserve XML TEXT fields , since they cannot be directly addressed using a hierarchical naming convention, nor does it preserve comment ('<!--><--/>') fields (they are not passed on from the SAX parser). Most other XML

constructs following the XML 1.0 specification and compatible with the SAX 1.0 parser can be processed successfully.

When reading XML, the user specifies a file name stub for File Manager, a full filename, or a pointer to a string buffer containing XML to add into the map.

Each key, element, and attribute in the XML file is inserted into an XMLValue object's data map (sorted by the level specified in the `ParseXMLxxxx()` API), so that multiple versions of each data point can be maintained. In the event that an element or attribute is at the same level as an existing entry, the existing entry is replaced.

The currentently defined data levels(there can be up to 254 total levels) are:

```
XML_NOLEVEL
XML_CLASSINIT
XML_CLASSSAVE
XML_INSTINIT
XML_INSTSAVE
XML_ALL
```

Saving at the XML_ALL level will save all defined XML entries to the specified output, at the highest defined level for each item. Saving at a specific level will only save the items actually defined at the specified level. Thus we can merge multiple input files into a single output document, or split them back out to separate documents at will.

4.3.2.2 Public Member Functions

(Please note that all examples are located in section 4.3.2.4, Usage Notes, on page 115.)

```
XMLValue *CreateAttribute ( XMLValue *pparent, const char *pname,
                            const char *pvalue,
                            XMLLEVEL level=XML_NOLEVEL);
```

This function creates a new XML attribute under the specified parent object. The first three parameters to this function must be non-NULL, and the name and value pointers should not reference empty strings. On success, a pointer to an XMLValue object that holds the attribute is returned. This can be a new object or an existing one, in the event that a matching object already existed. If any of the data parameters are invalid, or if the allocation of the new object fails, then the function returns NULL.

```
XMLValue *CreateElement ( XMLValue *pparent, const char *pname,
                          const char *pvalue,
                          XMLLEVEL level=XML_NOLEVEL);
```

This function creates a new XML element under the specified parent object. The first three parameters to this function must be non-NULL, and the name and value pointers should not reference empty strings. On success, a pointer to an XMLValue object that

holds the element is returned. This can be a new object or an existing one, in the event that a matching object already existed. If any of the data parameters are invalid, or if the allocation of the new object fails, then the function returns NULL.

```
XMLValue *CreateKey ( XMLValue *pparent, const char *pkey,
                      XMLLEVEL level=XML_NOLEVEL,
                      const char *pname=NULL);
```

This function creates a new XML key under the specified parent object. The first two parameters to this function must be non-NULL, and the key and name pointers should not reference empty strings. When adding the key, the following rules are observed:

- If the key string **is not** already defined, then a new key object is constructed and a pointer to this object is inserted in the parent object's map.
- If the key string **is** already defined, then the original key object is located. Once found, then the **pname** parameter is examined:
- If the parameter is **non-NULL**, then the parser iterates through the list of keys with the same list, looking for a key with a matching NAME attribute.
    - If a match is found, then a pointer to the existing object is returned
    - If no match is found, then a new key item is constructed, appended to the list of keys with the same name, and a pointer to the new object is returned.
- If the parameter is **NULL**, then the parser iterates to the end of the list of keys and adds it there, since there is no basis for comparison to determine a match. A new key item is constructed, appended to the list, and a pointer to the new object is returned.

On success, a pointer to an XMLValue object that holds the new key is returned. This can be a new object or an existing one, in the event that a matching object already existed. If any of the data parameters are invalid, or if the allocation of the new object fails, then the function returns NULL. Note that the `pname` value corresponds to a NAME attribute, but it does NOT create the attribute when it is specified. It is only used to determine whether a matching key object exists.

```
bool  DeleteAttribute ( XMLValue *pparent, const char *pkey,
                        XMLLEVEL level=XML_NOLEVEL);
```

This function deletes the specified XML attribute from the specified parent key. The parent object must be a key node. The first two parameters to this function must be non-NULL, and the key pointer should not reference an empty string. The **level**

parameter is not currently used. If the attribute specified is successfully deleted, then this function returns **true**. Otherwise, it returns **false**.

```
bool  DeleteElement ( XMLValue *pparent, const char *pkey,
                      XMLLEVEL level=XML_NOLEVEL);
```

This function deletes the specified XML element from the specified parent key. The parent object must be a key node. The first two parameters to this function must be non-NULL, and the key pointer should not reference an empty string. The **level** parameter is not currently used. If the element specified is successfully deleted, then this function returns **true**. Otherwise, it returns **false**.

```
bool DeleteKey ( XMLValue *pparent, const char *pkey,
                 const char *pattrvalue=NULL,
                 const char *pattrname=NULL);
```

This function deletes the specified XML key from the specified parent key. The parent object must be a key node. The first two parameters to this function must be non-NULL, and the key pointer should not reference an empty string. The third parameter allows the developer to specify a value for the NAME attribute to match against when searching for a key, the fourth parameter allows the developer to search for an attribute with a different name than NAME. **Note that this function will remove all child objects of the key being deleted!!!** If the key specified is successfully deleted, then this function returns **true**. Otherwise, it returns **false**.

```
void  DumpKeyMap();
```

This function dumps the global key string map, which is used to speed up accesses to the various maps internal to the XMLParser. This is normally used for diagnostic purposes.

```
void  DumpMap ( PXMLMAP pmap=NULL, USHORT tabcount=0);
```

This function dumps the contents of the XMLParser's internal data maps in their entirety. This is normally used for diagnostic purposes. Specifying a NULL pointer to the XMLMAP collection will dump the entire contents of the XMLParser object, from the root XML object on down. Since the XMLMAP collections are private data members for both the XMLParser and XMLValue classes, there is no way to get a pointer to a sub-map, outside of this class object, NULL is the only valid option to pass here.

```
XMLValue *GetAttribute ( XMLValue *pparent, const char *pkey,
                         const XMLValue *pprev);
```

This function returns a pointer to the specified XML attribute for the specified parent key, or NULL if the attribute is not found. The third parameter is only relevant for key

objects, not attributes or elements. If the specified attribute is found, a pointer to the relevant XMLValue object is returned, otherwise NULL is returned.

```
XMLValue *GetElement ( XMLValue *pparent, const char *pkey,
                       const XMLValue *pprev);
```

This function returns a pointer to the specified XML element for the specified parent key, or NULL if the element is not found. The third parameter is only relevant for key objects, not attributes or elements. If the specified element is found, a pointer to the relevant XMLValue object is returned, otherwise NULL is returned.

```
XMLValue *GetItem ( XMLValue *pparent, const char *pkey,
                    const XMLValue *pprev, XMLTYPE type,
                    const char *pattrvalue=NULL,
                    const char *pattrname="NAME");
```

This item returns a pointer to an XMLValue object, with a type as specified in the type parameter. This is the base function used by the other `GetXXX()` functions. Passing XML_ALL for the type will return the next item in the parent item's map, irrespective of its type. This is a handy way to walk through an XML file with an unknown layout. Parameters 3, 5, and 6 are only valid for key-type objects, not attributes or elements. The third parameter is used to iterate key sequences. The fifth parameter allows the developer to specify a value for the NAME attribute to match against when searching for a key, and the sixth parameter allows the developer to search for an attribute with a different name than NAME. If the specified item is found, a pointer to the relevant XMLValue object is returned, otherwise NULL is returned.

```
USHORT  GetCount();
```

Returns the number of objects stored in the XMLParser object's maps. This simply returns the value returned by the `XMLValue::GetCount()` member; there is no way for the method to determine the number of items at anything other than a global level. This method is primarily used to determine if the XMLParser object contains any data, and to check for memory leaks. Since this is a static count maintained by the XMLValue class, it will return non-zero if there are ANY items allocated by ANY XMLParser. **Thus if you have two XMLParser objects instantiated, one with valid data and one empty, BOTH will return a non-zero count, even though only one has valid data associated with it**.

```
XMLValue *GetKey ( XMLValue *pparent, const char *pkey,
                   const XMLValue *pprev, const char *pattrvalue=NULL,
                   const char *pattrname="NAME" );
```

This function returns a pointer to the specified XML element for the specified parent key, or NULL if the element is not found. The third parameter is used to iterate key sequences – when reading the first item of a sequence you pass a NULL for the pprev parameter, after that you pass the last key object pointer returned by this function, and continue to do so until a NULL is returned. The fourth parameter allows the developer to specify a value for the NAME attribute to match against when searching for a key, and the fifth parameter allows the developer to search for an attribute with a different name than NAME. If the specified item is found, a pointer to the relevant XMLValue object is returned, otherwise NULL is returned.

```
bool HasChildKeys ( XMLValue *pparent );
```

This function checks if the referenced XMLValue object is a key object, and if the object has any child key objects. This is useful when walking through the object tree in the parser. If the pointer is non-NULL, the referenced object is a key object, and the key object has child keys, then this function returns **true**. Otherwise, it returns **false**.

```
bool  ParseXMLFile ( const char *ppath, XMLLEVEL level=XML_NOLEVEL,
                     XMLValue *pparent=NULL );
```

This function parses contents of the specified XML file into memory. The level parameter determines the precedence of the new data being parsed versus data that is already contained in the map. Same-precedence data will replace existing data. If the pparent parameter is specified, then the new entries are inserted into the object map for the specified key object. Otherwise it is assumed to be at the root ('\XML') level. If the XML file path pointer is non-NULL, the file exists and is accessible, the referenced object is a key object, and if the input file is parsed successfully, then this function returns **true**. Otherwise, it returns **false**.

```
bool  ParseXMLString ( const char *pxml, XMLLEVEL level=XML_NOLEVEL,
                       XMLValue *pparent=NULL );
```

This function parses the provided XML string into memory, allowing the parser to ingest XML blocks from external sources. The level parameter determines the precedence of the new data being parsed versus data that is already contained in the map. Same-precedence data will replace existing data. If the pparent parameter is specified, then the new entries are inserted into the object map for the specified key object. Otherwise it is assumed to be at the root ('\XML') level. If the XML string pointer is non-NULL, the referenced object is a key object, and if the input string is parsed successfully, then this function returns **true**. Otherwise, it returns **false**.

```
bool  Reset();
```

This function clears all of the internal maps and deletes all of the XMLValue objects currently allocated. The parser object will now be as it was when first instantiated.

```
bool  SaveXMLFile ( const char *ppath, XMLLEVEL level=XML_ALL,
                    XMLValue *pparent=NULL, bool includeparent=true,
                    bool includeseq=true);
```

This function writes the contents of the parser object in XML format to the specified file, creating the file if necessary but always overwriting prior contents. Only information at the specified level will be written out. If XML_ALL is specified, then the entire contents of the parser object will be written out at the highest level for each item. If the `pparent` parameter is specified, then the map for the specified key is used as a starting point, so that subsections of the XML can be written out. The `includeparent` flag indicates whether the parent object's key string should be written out. If the parent object is part of a sequence, setting the `includeseq` flag to `true` will cause the contents of the entire sequence to be written out, as opposed to just the contents of the parent key's object map. If the path pointer is non-NULL, the file exists and is accessible, the referenced object is a key object (or NULL), and the XML was written successfully, then this function returns `true`. Otherwise, it returns `false`.

```
bool  SaveXMLString ( std::string &xml, XMLLEVEL level=XML_ALL,
                      XMLValue *pparent=NULL, bool includeparent=true,
                      bool includeseq=true);
```

This function writes the contents of the parser object back to the supplied string. Only information at the specified level will be written out. If XML_ALL is specified, then the entire contents of the parser object will be written out at the highest level for each item. If the `pparent` parameter is specified, then the map for the specified key is used as a starting point, so that subsections of the XML can be written out. The `includeparent` flag indicates whether the parent object's key string should be written out. If the parent object is part of a sequence, setting the `includeseq` flag to `true` will cause the contents of the entire sequence to be written out, as opposed to just the contents of the parent key's object map. If the referenced object is a key object (or NULL), and the XML was written successfully, then this function returns `true`. Otherwise, it returns `false`.

4.3.2.3 Class XMLValue

4.3.2.3.1 Overview

The XMLValue class is designed to hold values parsed from the targeted XML files, and is used by the XMLParser class to store the contents of the XML files in its collection. . Each object can be either an attribute or element object (containing string data), or a key

object. A unique key string identifies each object, except for special key objects called key sequences. Key sequences are generated when the input XML file contains more than one item at a given level with the same name. XMLValue objects support versioning of their data, in that up to 254 levels of data can be stored for each attribute or element, and these values can be individually accessed using a level tag. The XMLParser class is implemented as a friend of the XMLValue class, to simplify coding and maintenance.

The XMLValue class encapsulates data for all of the objects in the XMLParser data map. Those functions that return 'std::string &' are returning a reference to the internal datastores.

4.3.2.3.1.1 Static Member Functions

```
static DWORD GetCount();
```

This function will return the current number of XMLValue objects currently allocated irrespective of who allocated it. Useful only for looking for memory leaks, as the count is not tied to the allocating objects.

4.3.2.3.1.2 Public Member Functions

```
std::string& GetKey();
```

This function will return the entire hierarchical key string for the item. An example would be '\ACU\APU\MIN_POWER'.

```
XMLLEVEL GetLevel();
```

This function will return the item's highest defined level as an XMLLEVEL enum variable.

```
std::string& GetName();
```

This function will return the item name, which is its NAME attribute. This is blank for non-key objects.

```
XMLValue * GetNext();
```

Returns the next item in the sequence. If this is the last value in a key sequence, an attribute or element node, or if this is a singleton (non-sequenced) value NULL is returned. Useful when iterating a sequence of keys.

std::string& GetShortKey();

This function retrieves the short key, which is the key string for the individual item, as opposed to the hierarchical key string used to access the item. For key string "\XML\ACU\APU\MIN_POWER" the short key would be "MIN_POWER".

std::string& GetValue(XMLLEVEL level=XML_ALL);

This function will return the string data contained in a non-key node. If the node is a key node an empty string will be returned.

bool IsLast();

If the item is the last value in a key sequence, or if the object is not a member of a key sequence this function will return `true`.

bool IsLeaf();

If the item is a non-key node this function will return `true`.

bool SetValue(const char *pvalue, XMLLEVEL level);

This function can only be used to set the string value for an attribute or element node. Attempting to set the value for a key node will return `false`.

4.3.2.4 Usage Notes

This section will endeavor to describe the methods by which the developer can create, read from, and write to XML data files using the XMLParser class.

4.3.2.4.1 Creating XML Programmatically

A typical application would be a list of data items. Here we will look at the user list, as maintained by the User List Editor. In order to create a new user list XML file, we would instantiate a parser object, or if we had one, clear its contents:

```
XMLParser parser;
parser.Reset();
```

Then we would create the root-level key:

```
XMLValue *proot;
proot = parser.CreateKey ( NULL,"XML",XML_NOLEVEL );
```

Create a new USERS key object under the new root ('XML') key:

```
XMLValue *puserskey;
puserskey = parser.CreateKey(proot,"USERS",XML_NOLEVEL);
```

Create a new USER key object under the users ('USERS') key:

```
XMLValue *puserkey;
puserkey = parser.CreateKey(puserskey,"USER",XML_NOLEVEL,"testuser");
```

Create the attributes for the USER key object that we want to include:

```
parser.CreateAttribute(puserkey,"NAME","testuser");
parser.CreateAttribute(puserkey,"ID","testid");
parser.CreateAttribute(puserkey,"AUTHORITY","testauthority");
parser.CreateAttribute(puserkey,"PASSWORD","testpassword");
```

And finally, write out the contents of the parser object to an XML file:

```
parser.SaveXMLFile("c:\\temp.xml",XML_ALL,NULL);
```

This would yield the following XML file:

```
<XML>
    <USERS>
        <USER AUTHORITY="testauthority" ID="testid" NAME="testuser"
              PASSWORD="testpassword" />
    </USERS>
</XML>
```

We could add an additional user using the following bit of code:

```
puserkey = parser.CreateKey(puserskey,"USER",XML_NOLEVEL,"newuser");
parser.CreateAttribute(puserkey,"NAME","newuser");
parser.CreateAttribute(puserkey,"ID","newid");
parser.CreateAttribute(puserkey,"AUTHORITY","newauthority");
parser.CreateAttribute(puserkey,"PASSWORD","newpassword");
```

Note that we specified the NAME attribute for the new key. Using the NAME allows the parser to differentiate between multiple keys at the same indention level. In this case if a key object with a NAME attribute of 'newuser' already exists a pointer to the existing key object would be returned, and no new object would be allocated. This is advantageous, as it allows us to use the same block of `CreateXXX()` functions to both create new objects as well as update them. The `CreateKey()` method would return a pointer to the existing key object, and the `CreateAttribute()` calls would update the values in the existing attribute objects. If the NAME parameter is not specified the parser is forced to assume that the key is unique, and it will add the key as a new entry in a key sequence. Although in this case the default behavior would be acceptable, it is good to get in the habit of having a NAME attribute for all key objects.

Saving the XML out again using SaveXMLFile() would result in the following output file:

```
<XML>
    <USERS>
        <USER AUTHORITY="testauthority" ID="testid" NAME="testuser"
              PASSWORD="testpassword" />
        <USER AUTHORITY="newauthority" ID="newid" NAME="newuser"
              PASSWORD="newpassword" />
    </USERS>
</XML>
```

4.3.2.4.2 Loading XML Data

The parser can load data from more than one source file or string, and will merge them together into a unified data map. In this way the contents of multiple files can be merged together. New information can be inserted at the root level (under the default '\\XML' key), or under the specified key object. Attribute or element object values parsed at the same level will be overwritten with subsequent values. Values written at different levels will be added to the value map for the given object. When accessing the values for a given attribute or element object, the highest level value is returned by default, but the level can be specified. In this way lower-level values that would not ordinarily be seen can be examined and modified.

4.3.2.4.3 Finding XML Data

In a simple flat XML file, where key sequences contain only attributes as child object as in our user list example, the developer could access an individual USER block using the following call:

```
puserkey = parser.GetKey(NULL, "\\XML\\USERS\\USER",NULL,"newuser");
```

By specifying a NULL for the parent key, the parser will consult a global key map as a shortcut. This map contains pointers to the **first occurrence** of the specific hierarchical string in the parser's memory maps. This example passes a value of 'newuser' and assumes we're looking for this value in an attribute called NAME, which is the default value of the following (absent) parameter. Specifying the NAME parameter allows the parser to check for a key with a matching NAME attribute value, so that we can search through key sequences. If no NAME attribute was defined for the keys, then the following form could be used to search on another attribute for the key object:

```
puserkey = parser.GetKey(NULL,"\\XML\\USERS\\USER",NULL,"testid","ID");
```

This would look for a key object with an XML attribute called 'ID' with a value of 'testid'. In the event that the key objects do not have any attributes that render them unique, the developer will have to manually search the list of key objects using the following form:

```
puserkey = NULL;

// Loop through all of the matching keys
while((puserkey=parser.GetKey(NULL,"\\XML\\USERS\\USER",puserkey)))
{
   // Check the current object or sub-objects, and break the loop if we
   // got a match on some criteria…
}
```

If the `puserkey` variable is non-NULL a match was found.

If the XML has nested blocks, accessing individual items can become challenging, and in most cases can only be accomplished through multiple calls to the parser. Assuming the following XML:

EMPLOYEE01.XML

```
<XML>
    <MANAGER NAME="Tom Smith" DESCRIPTION="Customer Service Manager">
        <EMPLOYEE NAME="Tina Jones" DESCRIPTION="Insurance Clerk"/>
        <EMPLOYEE NAME="Leslie Slivovitz" DESCRIPTION="Underwriter"/>
    </MANAGER>
    <MANAGER NAME="Janice Warren" DESCRIPTION="Accounting Manager"/>
        <EMPLOYEE NAME="Bob Peterson" DESCRIPTION="Accountant"/>
        <EMPLOYEE NAME="Mary Klemp" DESCRIPTION="Secretary">
    </MANAGER>
</XML>
```

In this case, you could not access the EMPLOYEE 'Bob Peterson' directly, since it is a sequence key object of a sequence key object. If you attempted to use the `GetKey()` call with a NULL for the first parameter, you would get a pointer to the first employee for 'Tom Smith', in this case, 'Tina Jones'. You first have to locate the entry for the MANAGER 'Janice Warren', and use the returned pointer as the parent object when searching for the EMPLOYEE:

```
XMLParser    parser;

XMLValue *pmanager, *pemployee;

parser.ParseXMLFile("EMPLOYEE01.XML",XML_NOLEVEL);
if((pmanager=parser.GetKey(NULL,"\\XML\\MANAGER",NULL,"Janice Warren")))
{
   if ( ( pemployee = parser.GetKey ( pmanager,"\\XML\\MANAGER\\EMPLOYEE",NULL,
                                      "Bob Peterson")))
   {
      // Do something with the employee key object or its child objects
   }
}
```

Iterating through the key objects would be largely the same, remembering to provide a pointer to their parent object:

```
XMLValue *pmanager=NULL,*pemployee=NULL;

// Iterate through the manager key objects
while((pmanager=parser.GetKey(NULL,"\\XML\\MANAGER",pmanager)))
{
   // Do something with the manager key object

   // Iterate through the employee key objects
   while ( ( pemployee = parser.GetKey ( pmanager,"\\XML\\MANAGER\\EMPLOYEE",
                                         pemployee)))
   {
      // Do something with the employee key object
   }
}
```

The fourth and fifth parameters to the GetKey() function bear a separate discussion, as their use can be confusing. Assume the following XML file:

```
<XML>
       <CAR BRAND='Porsche' MODEL='911' PRICE='$80000'/>
       <CAR BRAND='BMW' MODEL='328i' PRICE='$40000'/>
       <CAR BRAND='Mercedes' MODEL='560SLC' PRICE='$120000'/>
</XML>
```

If we read this into a parser object using ParseXMLFile(), the parser would create three separate key objects, stored as a sequence. If we wanted to look for a specific CAR object using its MODEL attribute, then we would have two choices: get the first CAR object and then iterate through the sequence, checking for a matching MODEL attribute value, or use the fourth and fifth parameters to the GetKey() function. In normal usage, when a key item has a NAME attribute, you can call GetKey() with the fourth parameter set to get a specific key object:

```
if((pkey=parser.GetKey(NULL,"\\XML\CAR",NULL,"911")))
{
       // Do something here…
}
```

Because the CAR key objects do not have a NAME attribute in this case, the call will fail, returning NULL instead of a pointer to the desired key object. To correct this, you can specify the **name** of the attribute to search for as the fifth parameter:

```
if((pkey=parser.GetKey(NULL,"\\XML\CAR",NULL,"911","MODEL")))
{
       // Do something here…
}
```

This will correctly return the first CAR object, as it has an attribute named MODEL with the value "911". This is useful in situations where there are no NAME attributes to search on, as well as situations where you want to ignore the NAME attribute and search for keys using different attribute names.

Note that GetKey() calls GetItem() with the fourth parameter set to XML_KEY, so the fifth and sixth parameters to GetItem() work in exactly the same manner:

```
if((pkey=parser.GetKey(NULL,"\\XML\CAR",NULL,"911","MODEL")))
```

Is equivalent to the following call to GetItem():

```
if((pkey=parser.GetItem(NULL,"\\XML\CAR",NULL,XML_KEY,"911","MODEL")))
```

4.3.2.4.4 Saving XML Data

The parser can save data to string variables, as well as data files. The data can be saved at a given level only, starting at a specified key object (so that a subsection of the XML can be saved), and at a global level, which will effectively merge data from disparate sources. Assuming we have a parser with the previous manager and employee information in it (from the EMPLOYEE01.XML file), we can merge in more information from another file:

```
EMPLOYEE02.XML:

<XML>
    <MANAGER NAME="Marge Johannsen" DESCRIPTION="Vice President">
        < EMPLOYEE NAME="Tom Smith" DESCRIPTION="Customer Service Manager"/>
        < EMPLOYEE NAME="Janice Warren" DESCRIPTION="Accounting Manager"/>
    </MANAGER>
</XML>
```

Using the function calls to read in the XML information:

```
parser.ParseXMLFile("EMPLOYEE01.XML", XML_NOLEVEL);
parser.ParseXMLFile("EMPLOYEE02.XML", XML_INSTINIT);

Note that we have parsed in the contents of EMPLOYEE02.XML at a different level.
```

Saving the contents of the parser at level XML_NOLEVEL:

```
parser.SaveXMLFile("OUTPUT.XML",XML_NOLEVEL);
```

Would produce the following output:

```
OUTPUT.XML:

<XML>
    <MANAGER NAME="Tom Smith" DESCRIPTION="Customer Service Manager">
        <EMPLOYEE NAME="Tina Jones" DESCRIPTION="Insurance Clerk"/>
        <EMPLOYEE NAME="Leslie Slivovitz" DESCRIPTION="Underwriter"/>
    </MANAGER>
    <MANAGER NAME="Janice Warren" DESCRIPTION="Accounting Manager">
        <EMPLOYEE NAME="Bob Peterson" DESCRIPTION="Accountant"/>
        <EMPLOYEE NAME="Mary Klemp" DESCRIPTION="Secretary"/>
    </MANAGER>
</XML>
```

Saving the contents of the parser at level XML_INSTINIT:

```
parser.SaveXMLFile("OUTPUT.XML",XML_INSTINIT);
```

Would produce the following output:

```
OUTPUT.XML:
<XML>
    <MANAGER NAME="Marge Johannsen" DESCRIPTION="Vice President">
        < EMPLOYEE NAME="Tom Smith" DESCRIPTION="Customer Service Manager"/>
        < EMPLOYEE NAME="Janice Warren" DESCRIPTION="Accounting Manager"/>
    </MANAGER>
</XML>
```

Saving the contents of the parser using the default level (XML_ALL):

```
parser.SaveXMLFile("OUTPUT.XML",XML_ALL);
```

Or, since XML_ALL is the default:

```
parser.SaveXMLFile("OUTPUT.XML");
```

Would produce the following output:

```
OUTPUT.XML:
<XML>
    <MANAGER NAME="Tom Smith" DESCRIPTION="Customer Service Manager">
        <EMPLOYEE NAME="Tina Jones" DESCRIPTION="Insurance Clerk"/>
        <EMPLOYEE NAME="Leslie Slivovitz" DESCRIPTION="Underwriter"/>
    </MANAGER>
    <MANAGER NAME="Janice Warren" DESCRIPTION="Accounting Manager">
        <EMPLOYEE NAME="Bob Peterson" DESCRIPTION="Accountant"/>
        <EMPLOYEE NAME="Mary Klemp" DESCRIPTION="Secretary"/>

    </MANAGER>
    <MANAGER NAME="Marge Johannsen" DESCRIPTION="Vice President">
        < EMPLOYEE NAME="Tom Smith" DESCRIPTION="Customer Service Manager"/>
        < EMPLOYEE NAME="Janice Warren" DESCRIPTION="Accounting Manager"/>
    </MANAGER>
</XML>
```

This effectively merges the contents of the two input files.

And finally, using the last three parametersof the SaveXMLxxx() functions will allow saving of subsections of the XML tree, by specifying the key object where we want to start saving, whether to include the key object in the output, and finally whether to check for key sequence siblings for the specified parent key. Thus, in order to save only the record for the manager "Janice Warren":

```
XMLValue *pmanager=parser.GetKey(NULL,"\\XML\\MANAGER","Janice Warren");

if(pmanager)
{
   bool success = parser.SaveXMLFile("c:\\output.xml", XML_ALL, pmanager, true, false);
}
```

would write out the file:

```
<XML>
    <MANAGER NAME="Janice Warren" DESCRIPTION="Accounting Manager">
        <EMPLOYEE NAME="Bob Peterson" DESCRIPTION="Accountant" />
        <EMPLOYEE NAME="Mary Klemp" DESCRIPTION="Secretary" />
    </MANAGER>
</XML>
```

Setting the fourth parameter to FALSE to prevent the parser from writing out the MANAGER key to XML:

```
bool success = parser.SaveXMLFile("c:\\output.xml",XML_ALL,pmanager,false,false);
```

would result in the following output (since sequences are set to `false`):

```
<XML>
    <EMPLOYEE NAME="Bob Peterson" DESCRIPTION="Accountant" />
</XML>
```

Setting the fifth parameter to TRUE to force the parser to process sequences:

```
bool success = parser.SaveXMLFile("c:\\output.xml",XML_ALL,pmanager,false,true);
```

would result in the following output:

```
<XML>
    <EMPLOYEE NAME="Bob Peterson" DESCRIPTION="Accountant" />
    <EMPLOYEE NAME="Mary Klemp" DESCRIPTION="Secretary" />
</XML>
```

And the last iteration, setting the last two parameters to TRUE (show parent, show sequences):

```
<XML>
    <MANAGER NAME="Janice Warren" DESCRIPTION="Accounting Manager">
        <EMPLOYEE NAME="Bob Peterson" DESCRIPTION="Accountant"/>
        <EMPLOYEE NAME="Mary Klemp" DESCRIPTION="Secretary"/>
    </MANAGER>
    <MANAGER NAME="Marge Johannsen" DESCRIPTION="Vice President">
        < EMPLOYEE NAME="Tom Smith" DESCRIPTION="Customer Service Manager"/>
        < EMPLOYEE NAME="Janice Warren" DESCRIPTION="Accounting Manager"/>
    </MANAGER>
</XML>
```

Note that the first manager, "Tom Smith", is not included, since he was first in the input XML file, before "Janice Warren". There is no sorting of data by value in the parser. Information is sorted only by the name of the key, element, or attribute, and appears in FIFO order otherwise.

4.3.3 DataItem Classes

4.3.3.1 Overview

Data items are objects used to hold the various types of data cached within a module, that have some relationship with other modules. These may be telemetry data, updated by data posted from downstream modules, internal data initialized from startup configuration files, or variables that must be displayed on and modified from a GUI.

Not all the features of the data item classes will be required for all applications, but if custom data items are to be developed all features must be supported. It is not for the data item to restrict its application.

4.3.3.1.1 Data Item Creation

Data items can be one of three main types:

Scalar Data Items — these are template classes, and can be based on any built-in data type or alias for such. Scalar data types are continuous values with a minimum and maximum. The limits may be specified in the configuration file for the object, or the system limits will be used. (Note: it is useful to generate min and max values from the system limits since the GUI may be running on a system with different limits.)

List Data Items	these are enumerated types, as well as the built-in `bool` data type. List data items are initialized after instantiation with the values they support. When a list is specified in the configuration, the supported set of values is the intersection of the programmatically specified list and that read from the configuration. This allows the implementation to specify supersets of options without risk of the parameter being set to an unsupported or unwanted value.
Custom Data Items	these are explicitly coded to the data item interface, but implementation is specific to the data type. Currently the custom data items are for strings, frequency, time and object status.

Data items can be instantiated on the stack or the heap, but it is really only practical to use heap variables (created with `new`). Data items need to insert themselves into the object's parameter map, and this linkage is not possible until the object pointer (`this`) is available. Pointers to the data items can and should be included as members of the parent object.

It would be possible to modify the data item class by adding a setOwner() method, which would allow data items to be created on the stack or as members of a class, and have ownership established later. This would require additional checking to effectively disable a data item until it is owned, and may be of limited value.

Data items store their values internally, but can mirror that value to an external variable. The external variable bypasses all the range checking logic, so it is recommended that it be a member of a `const volatile` data structure in the binary format used for propagating telemetry to other modules.

4.3.3.1.2 Data Item Access and Update

Data items may be accessed directly or by name lookup. Note that name lookup requires searching an STL map, and will involve multiple string comparisons. This is not efficient! The map is implemented more for the purpose of configuration file management and the GUI interface than for normal parameter management. (With telemetry data from multiple instruments arriving at intervals of 10s of milliseconds we cannot afford to be profligate with clock cycles!)

4.3.3.1.3 Base Class Processing

The intent of the parameter scheme, based on data items, is to locate as much of the logic for instruments and other modules in the base classes. In practice, instruments will have the most to gain from this. The operations handled completely in the base classes are:

- Initialization of data items, values and ranges, from the configuration.
- Extraction of data item values into an XML block suitable for saving to a configuration file. This can be by instance or class initialization.
- Generation of range strings for the GUI interface.
- Complete handling of the `UIRefresh()` method to update the GUI fields

Additionally helper methods are provided to simplify the posting of telemetry data to the GUI. (As noted above, the data item objects are not the most efficient mechanism for handling inter-module telemetry data. This should use the IDL personality structure with its fields mirrored to the data items.)

4.3.3.2 Class DataItem

This is an abstract base class from which the other parameter types are derived.

4.3.3.2.1 Protected data members

`string m_name;`

The name of the data item as used in the configuration files and the GUI controls.

`string m_xml;`

A string to hold the text representation of the data.

`SComponent_impl * mp_owner;`

The object that owns the data item.

`string m_list;`

A string to hold the pick list (4.3.3.5 – List Data Item) read from the configuration file.

`string m_range;`

A string to hold the range string for the GUI.

bool m_modified;

A flag indicating that the contents have been modified since the last call to the `resetModified()` method.

bool m_lifetimeModified;

A flag indicating that the contents have been modified since the last call to the `init()` method.

bool m_autoEvent;

A flag indicating that the parameter change event should be posted whenever the contents are modified.

4.3.3.2.2 Protected member functions

string getMin();

Return minimum limit string from the configuration.

string getMax();

Return maximum limit string from the configuration.

string getPermLevel();

Return the permission level for updating this data item. This is only required for GUI originated updates, and is handled automatically in the base classes. Derived classes and modules should not need to use this method.

string getListElement(int);

Return specified item from the pick list. A zero length string indicates that the item did not exist.

string getTextElement(int);

Return specified item from the custom text version of the pick list(4.3.3.5 – List Data Item). A zero length string indicates that the item did not exist.

bool hasPickList();

Returns a boolean indicating whether a pick list was in the configuration file.

```
bool hasPickText();
```

Returns a boolean indicating whether a user-friendly version of the pick list was in the configuration file. This version of the pick list is what is displayed in screen controls, and has no programmatic role.

4.3.3.2.3 Public non-virtual member functions

```
DataItem(const char*, SComponent_impl*, bool);
```

The constructor. The first argument is the name of the data item. The second is a pointer to the owning object. The final argument is a flag that indicates whether the data item is public, that is, it's made available to the Profile Editor so it can it be configured as part of a mission profile. Private data items can only be modified by the application code and its GUI.

```
~DataItem();
```

The destructor.

```
const string& name();
```

Name accessor.

```
void addFunction ( SControl_impl::PFN pFn );
```

This method associates a GUI update function with this data item. See section 4.3.4 - Scontrol and chapter 5 for more details.

```
void cfgLevel(XMLLEVEL);
XMLLEVEL cfgLevel() const;
```

These methods are the accessor and mutator for the configuration level member. These are only used by the base classes, as part of the configuration subsystem.

```
void isPublicData(bool);
bool isPublicData() const;
```

These methods are the accessor and mutator for the public data flag member. These are primarily used by the base classes, as part of the configuration subsystem.

```
void isPublished(bool);
bool isPublished() const;
```

These methods are the accessor and mutator for the published data flag member. Published data items are included in data posts to the GUI. Data items that are not required to be posted to the GUI may be set to unpublished, thus reducing the

bandwidth required when refreshing the GUI screens. By default all data items are published.

```
void isPOC(bool);
bool isPOC() const;
```

These methods are the accessor and mutator for the point-of-control data flag member. Point-of-control data items require that the GUI has point-of-control access before they can be modified from the GUI. By default all data items require GUI point-of-control access for update. See chapter 5 for GUI and POC details.

```
void setAutoEvent ( bool setting = true );
```

This method sets or clears the auto event flag described above. By default, data items are constructed with the flag cleared, so data modification events are not generated automatically. This method enables the feature if called with no argument, or sets the flag to the value of the argument passed.

```
void sendEvent(bool always = true) const;
```

This method sends a data modification event for the variable. If called with no argument the event is generated, regardless of the setting of the modified flag. If called with an argument of `false` the event is only generated if the modified flag is set.

```
void saveCfg(bool);
```

This method saves the value of the data item back to the configuration map. This is used in conjunction with the configuration subsystem and the maintenance GUI to allow configuration settings to be tweaked from the GUI and then saved back as system defaults. This method would be called as part of the implementation of a "SAVE CONFIGURATION" option from the maintenance GUI screen.

```
void permLevel(eAuthLevels);
eAuthLevels permLevel() const;
```

These methods are the accessor and mutator for the access permission level member. This is only required by the base classes as part of the authentication subsystem.

4.3.3.2.4 Public virtual member functions

The following functions provide rudimentary implementations of the methods defined, but in many cases it will be necessary to overload these when building custom data items.

```
bool isModified();
```

Modified flag accessor.

```
bool everBeenModified()
```

Lifetime modified flag accessor.

```
bool setModified();
```

Set modified flag.

```
bool resetModified();
```

Test and clear modified flag.

```
bool range(string&);
```

Retrieve GUI range string. Returns false if no range string exists.

```
bool list(string&);
```

Retrieve GUI pick list string. Returns false if no pick list string exists.

```
bool isValueAsStringValid(const string&);
```

Parses and range checks string data representation.

4.3.3.2.5 Pure virtual functions

The following functions must be overloaded and implemented in the derived classes when building custom data items.

```
void init(const char*);
```

Initialization method. The argument is a string representation of the initial value for the data item.

In addition to setting the internal value of the data item, the `init()` function can search the configuration map for additional attributes for the data item. These can define limits or option lists, as they do for scalar and list data items, and string lengths for the string data item. Custom data items can add checks for any other attributes relevant to the data type.

```
void save ( bool instance = false );
```

This method saves the current value to the internal default value and to the configuration map.

```
void restore();
```

This method restores the current value from the internal default.

```
DataItem * clone(SComponent_impl * sb) const;
```

This method creates a clone of the data item owned by another object, the object referenced by the `SComponent_impl` pointer.

```
const char* toXML();
```

Returns the current value as an XML string.

```
void getValueAsString(string&);
```

Returns the current value as a simple string.

```
void getDefaultAsString(std::string &s) const;
```

Returns the default value as a simple string.

```
bool setValueAsString(const string&);
```

Sets the current value from a simple string.

4.3.3.3 Template Class SDataItem

The SDataItem class is a template data item class for any data type that supports the assignment, and equality and inequality operators. According to C++ convention, `T` in the descriptions below corresponds to the type with which the data item is declared. For instance, for a data item declared as `SDataItem<int>`, `T` would be equivalent to `int`.

4.3.3.3.1 Protected data members

```
T m_value;
```

The current value of the data item.

```
T m_default;
```

The default value of the data item.

```
volatile T * const mp_value;
```

A pointer to the data value, whether real or shadow. The data item can be defined to encapsulate an external variable. This will allow the variable to be part of a telemetry data structure, for example. All updates of the data item are applied to the external variable, and all changes to the external variable are detected by the data item methods. If the data item is created without a pointer to an external variable then the shadow pointer references the internal store, `m_value`. When implementing custom data items derived from SDataItem care must be taken to keep the internal store and the shadow variable in sync. This is done by doing something similar to this:

```
virtual T value ( T v )
{
   if ( v != *(T*)mp_value )
   {
      *(T*)mp_value = m_value = v;
      m_lifetimeModified = m_modified = true;
      if ( m_autoEvent )
      {
         sendEvent();
      }
   }
   return m_value;
}
```

4.3.3.3.2 Public non-virtual member functions

```
SDataItem ( const char*, SComponent_impl*, bool isPublic = true );
SDataItem ( const char*, SComponent_impl*, T * const,
            bool isPublic = true);
```

The constructors. The first argument is the name of the data item. The second is a pointer to the owning object. The final argument is a flag that indicates whether the data item is public, that is, can it be configured as part of a mission profile. Private data items can only be modified from a management GUI. This argument may be omitted, in which case the data item is public.

The second form of the constructor takes an additional argument, a pointer to a variable of type T. This is the shadow variable described above.

4.3.3.3.3 Public virtual member functions

The following functions provide rudimentary implementations of the methods defined, but in many cases it will be necessary to overload these when building custom data items.

```
T value() const;
T value ( T v );
```

Value accessor and mutator. These methods access the value of the data item. In addition to reading or updating the internal store, they synchronize the internal store with the shadow variable and update the modified data flag if the value has changed. Even on a read the modified flag might be set, if the shadow variable differed from the internal store. When the internal store and the shadow variable are synchronized the shadow store takes precedence.

```
T defaultValue();
```

Default value accessor.

```
void save ( bool instance = false );
```

This method saves the current value to the default and to the configuration map.

```
void restore();
```

This method restores the current value from the default.

```
bool isModified();
```

Modified flag accessor. Prior to returning the flag it synchronizes the internal store with the shadow variable and updates the modified data flag if the value has changed. When the internal store and the shadow variable are synchronized the shadow store takes precedence.

4.3.3.3.4 Pure virtual functions

```
bool isValid()
```

Value validity status accessor. The base class has no idea what this means, so it is pure virtual and overridden in all of the specific Data Item classes. It is pure so any new custom Data Items will be required to implement this.

4.3.3.4 Template Class ScalarDataItem

The scalar data item is used to hold continuous data, constrained by minimum and maximum values. In addition to the value, limits may be specified in the configuration file. These are specified as attributes to the XML element corresponding to the data item. The MIN attribute contains a textual representation of the minimum value. The MAX attribute contains the maximum value. If either attribute is missing the data item is constrained by the host system limits.

4.3.3.4.1 Public non-virtual member functions

```
ScalarDataItem<T>(const char*,SBase_impl*, bool isPublic = true);
```

Simple constructor.

```
ScalarDataItem<T>(const char*,SBase_impl*,const T*, bool isPublic = true);
```

Constructor with shadow variable. See 4.3.3.3.2 for details.

```
DataItem * clone(SComponent_impl * sb) const;
```

This method creates a clone of the data item owned by another object, the object referenced by the `SComponent_impl` pointer. The value, range and flags are copied to the cloned data item.

```
T min() const;
T min(T t);
```

Minimum limit accessor and mutator. The minimum may only be increased from the previous setting, thus reducing the range not enlarging it.

Note that for built-in types the numeric_limits templates are used by default. For floating point types the minimum returned by these templates is the smallest positive number, not the largest negative number. If the range needs to span 0 the minimum must be defined in the setup XML since it cannot be programmatically reduced from the default.

Some investigation of numeric_limits seems to indicate that I used the wrong method to determine the minimum value. There are two flavours of minimum values, and several flags that intentify the characteristics of the type. A combination of flag tests and relevant function calls could fix the problem.

```
T max() const;
T max(T t);
```

Maximum limit accessor and mutator. The maximum may only be decreased from the previous setting, thus reducing the range not enlarging it.

```
ScalarLimits<T> setLimits ( ScalarLimits<T>);
ScalarLimits<T> getLimits();
```

Limits accessor and mutator. These methods use the ScalarLimits template class described below (section 4.3.3.4.3.1) to set minimum and maximum in a single call. As noted above, the range may only be reduced not enlarged. Both methods return the

updated limits. If the current settings and the new settings overlap the resulting range is the intersection of the two.

`bool range(string&);`

Copy the GUI range string into the passed string. Returns false if no range string exists.

4.3.3.4.2 Public virtual member functions

`void init(const char*);`

Initialization method. This searches the configuration map for an initialization element for the data item, as well as limit attributes to set its range.

`const char* toXML();`

Returns value as an XML string.

`bool isValid(T);`

Returns validity status of the data item, specifically if it is in the range `min <= value <= max`.

`bool isValueAsStringValid(const string&);`

Parses and range checks string data representation.

`void getValueAsString(string&);`

Returns current value as a simple string.

`bool setValueAsString(const string&);`

Sets current value from a simple string.

4.3.3.4.3 Related Classes

4.3.3.4.3.1 Template Class ScalarLimits

This class is defined as follows:

```
template<class T> struct ScalarLimits
{
   T min;
   T max;
};
```

It provides a simple mechanism for setting ranges on a scalar data item, and is used by the `getLimits()` and `setLimits()` methods described above.

4.3.3.5 Template Class ListDataItem

The list data item is used to hold discrete data, selected from a pick list. Typically this will be based on a C++ or CORBA enumerated data type. In addition to the value, stored in the template data type, two strings may be specified in the configuration file as attributes of the XML element. One is the list string named LIST. This contains a hard system text synonym for the values of the pick list. The other is the text string named TEXT which is a customizable textual representation of the pick list. This cannot be used internally, but is included in the GUI list strings sent to pick list screen controls, allowing the user view of the data to be customized.

4.3.3.5.1 Public non-virtual member functions

```
ListDataItem<T>(const char*, SBase_impl*, bool isPublic = true);
```

Simple constructor.

```
ListDataItem<T>( const char*, SBase_impl*, const T*,
                 bool isPublic = true);
```

Constructor with shadow variable. See 4.3.3.3.2 for details.

```
DataItem * clone(SComponent_impl * sb) const;
```

This method creates a clone of the data item owned by another object, the object referenced by the `SComponent_impl` pointer. The value, range items and flags are copied to the cloned data item.

```
void addRangeItem(T value,const char*);
```

This method adds a list item to the data item. When created, a list data item is empty and has no valid settings. The individual options must be explicitly added with this method. The first argument is the value according to the template type, and the second is the text value associated with it. A couple of examples will clarifying many aspects of the list data item setup.

The first example is a list data item based on an enumerated data type:

```
typedef enum E_PICKITEMS { Pick1, Pick2, Pick3, Pick4 } ePickItem;

mp_pickItem = new ListDataItem<ePickItem> ( "PICKITEM", this );
mp_pickItem->addRangeItem ( Pick1, "PickListItem1" );
mp_pickItem->addRangeItem ( Pick2, "PickListItem2!List Item 2" );
mp_pickItem->addRangeItem ( Pick3, "PickListItem3" );
mp_pickItem->addRangeItem ( Pick4, "PickListItem4" );
```

Here we see range items added for each enumeration. For 'Pick2", in addition to the system text, the text string adds GUI text to the range item. In this format all text

following the '!' character is the text that will be displayed on the GUI. Note also that that system text may not contain embedded space characters, but the GUI text may. Only the first '!' is used as a delimiter, so the system text cannot contain one, but the GUI text can contain one after the one used as a delimiter.

Now let's consider the role of the setup XML. For the same variable we might see

```
<PICKITEM LIST="PickListItem1,PickListItem2,PickListItem3,PickListItem5"
          TEXT="List Item One,List Item Two,List Item Three,List Item Five">
   PickListItem3
</PICKITEM>
```

This XML fragment qualifies the active option in the data item. `PickListItem4` is missing from the list attribute, so that option is not active. It could be specifically activated in code, however. `PickListItem5` appears in the `LIST` attribute, but is was not added to the data item, so that option is not supported at all. GUI text is added for all options with the `TEXT` attribute, and the XML version replaces the GUI text specified in the code.

The second example is a list item based on the C++ Boolean type. In this case the GUI text contains a color attribute to support an annunciator screen control. The '+' character is specific to the Annunciator control and would simply be part of the GUI text when rendered by any other control.(See section 5.8.1 for more details.)

```
mp_alarm = new ListDataItem<bool> ( "ALARM", this );
mp_alarm->addRangeItem ( false, "0!Clear+green" );
mp_alarm->addRangeItem ( true, "1!Alarm+red" );
```

4.3.3.5.2 Public virtual member functions

`void init(const char*);`

Initialization method. This searches the configuration map for an initialization element for the data item, as well as text attributes to set its active options and GUI representation.

`const char* toXML();`

Returns value as an XML string.

`bool list(string&);`

Copy the GUI pick list string into the passed string. Returns false if no list string exists.

`void buildList();`

This methods forces the internal pick list string to be rebuilt. This would be used after range items have been activated or deactivated as described below.

```
void getValueAsString(string&);
```

Returns the current value as a simple string.

```
bool setValueAsString(const string&);
```

Sets the current value from a simple string.

```
bool isValid();
```

Returns the validity status of data item.

```
bool isValueAsStringValid(const string&);
```

Checks that string data representation is a valid option.

```
bool isModified();
```

Modified flag accessor. If the data item is mirrored to an external variable and the external variable has been changed, the internal value is updated and the method returns true.

```
int getActiveItems ( std::list<T>& theList ) const;
```

This method populates the passed list with the active range items for the data item and returns a count of the number. This allows a module to programmatically create a validation list for another similar data item. This is particularly relevant between physical instruments and virtual instruments, where the virtual instruments list of allowable options is set by the physical instrument's capabilities.

```
void deactivateAllItems();
```

This method deactivates all the range items for a data item. Typically this is in preparation for activating range items based on a list of active range items from another module, as described above.

```
void activateItem ( const T& v );
```

This method activates the specified range item. If a corresponding range item does not exist in the data item no action occurs. This allows a data item to have its range items activated from a list containing a superset of its supported range items.

```
std::string getText ( const T& v ) const;
```

This method returns the system text associated with a specific range item.

```
T getValueFromText ( std::string s ) const;
```

This method returns the value associated with a specific system text string.

```
std::string getGUIText ( const T& v ) const;
```

This method returns the GUI displayed text associated with a specific range item.

4.3.3.6 Class FrequencyDataItem

The frequency data item is a custom data item used to hold the system-wide frequency data type. It is derived from the SDataItem template, so all its public methods are available. It can be constrained by minimum and maximum values, specified in the configuration file. These are specified as attributes to the XML element corresponding to the data item. The MIN attribute contains a textual representation of the minimum value. The MAX attribute contains the maximum value.

The data item also suuports the attribute UNITS, which can be set to HZ, KHZ, MHZ or GHZ. If units are specified in the value or limit strings these are applied to the data. If units are specified in the value string or in the UNITS attribute, these are used in the absence of units in the value or limit strings. The following example should clarify this.

```
<XML>
   <UPLINK_FREQ MIN=2000 MAX=2200 UNITS=MHZ>
      2.1GHZ
   </UPLINK_FREQ>
</XML>
```

The units for the data item are defined as MHz, so the MIN and MAX attributes are interpreted as 2000MHz and 2200MHz respectively. The value is defined with an explicit units specifier, GHz, so it is interpreted as 2.1GHz, or 2100MHz, which is within the limits defined. If no UNITS attribute is defined the units default to Hz, in the absence of an explicit units specifier.

4.3.3.6.1 Public non-virtual member functions

```
FrequencyDataItem(const char*,SBase_impl*);
```

Simple constructor.

```
FrequencyDataItem(const char*,SBase_impl*,const SCFrequency*)
```

Constructor with shadow variable. See 4.3.3.3.2 for details.

```
DataItem * clone(SComponent_impl * sb) const;
```

This method creates a clone of the data item owned by another object, the object referenced by the `SComponent_impl` pointer. The value, range and flags are copied to the cloned data item.

```
void init(const char*)
```

Initialization method. This searches the configuration map for an initialization element for the data item, as well as units and limit attributes to set its range.

```
const char* toXML()
```

Returns value as an XML string.

```
SFrequency min() const;
SFrequency min(SFrequency);
```

Minimum limit accessor and mutator. The minimum may only be increased from the previous setting, thus reducing the range not enlarging it.

```
SFrequency max() const;
SFrequency max(SFrequency);
```

Maximum limit accessor and mutator. The maximum may only be decreased from the previous setting, thus reducing the range not enlarging it.

4.3.3.6.2 Public virtual member functions

```
bool range(string&);
```

Copy the GUI range string into the passed string. Returns false if no range string exists.

```
bool isValid(Sfrequency);
```

Returns validity status of data item.

```
bool isValueAsStringValid(const string&);
```

Parses and range checks string data representation.

```
void getValueAsString(string&)
```

Returns current value as a simple string.

```
bool setValueAsString(const string&)
```

Sets current value from a simple string.

4.3.3.7 Class StringDataItem

The string data item is a simple data item used to hold standard C++ strings. It is derived from the DataItem class, so all its public methods are available. Additional classes may be derived from it if more sophisticated string handling and validation is required.

4.3.3.7.1 Protected data members

`std::string m_value;`

The current value of the data item.

`std::string m_default;`

The default value of the data item.

`std::string * const mp_value;`

A pointer to a shadow value of type C++ string. All updates of the data item are applied to the external variable, and all changes to the external variable are detected by the data item methods. If the data item is created without a pointer to an external variable then the shadow pointer references the internal store, `m_value`. When implementing custom data items derived from StringDataItem care must be taken to keep the internal store and the shadow variable in sync.

`volatile CORBA::String_var * const mp_corbastring;`

A pointer to a shadow value of type CORBA::String_var. If the external variable is part of a telemetry data structure defined in IDL this data type is more applicable than a C++ string. As the constructors below indicate, only one shadow variable is supported, and the data item automatically determines which pointer is valid. All updates of the data item are applied to the external variable, and all changes to the external variable are detected by the data item methods. If the data item is created without a pointer to an external variable then the shadow pointer references the internal store, `m_value`. When implementing custom data items derived from StringDataItem care must be taken to keep the internal store and the shadow variable in sync.

`std::string::size_type m_min;`

The minimum permitted string length.

`std::string::size_type m_max;`

The maximum permitted string length.

4.3.3.7.2 Public non-virtual member functions

```
StringDataItem( const char*, SComponent_impl*, bool);
StringDataItem( const char*, SComponent_impl*,string* const, bool);
StringDataItem( const char*, SComponent_impl*,
                CORBA::String_var* const, bool);
```

The constructors. The first form is the simple constructor, with just the item name, owner, and public flag. The other two forms include pointers to shadow variables, either a C++ string or a `CORBA::String_var` type.

```
std::string value() const;
std::string value ( std::string v );
```

Value accessor and mutator. These methods access the value of the data item. In addition to reading or updating the internal store, they synchronize the internal store with the shadow variable and update the modified data flag if the value has changed. Even on a read the modified flag might be set, if the shadow variable differed from the internal store. When the internal store and the shadow variable are synchronized the shadow store takes precedence.

```
std::string defaultValue() const;
```

Default value accessor.

4.3.3.7.3 Public virtual member functions

```
bool isValid ( const std::string& v );
```

Value validity status accessor.

```
bool isModified();
bool isModified() const;
```

Modified flag accessors for `const` and non-`const` data. Prior to returning the flag it synchronizes the internal store with the shadow variable and updates the modified data flag if the value has changed. When the internal store and the shadow variable are synchronized the shadow store takes precedence.

```
void init ( const char * valuetext );
```

Initialization method. This searches the configuration map for an initialization element for the data item, as well as limit attributes to set its minimum and maximum lengths. These are specified as the `MIN` and `MAX` XML element attributes.

```
void save ( bool instance = false );
```

This method saves the current value to the default and to the configuration map.

```
void restore();
```

This method restores the current value from the default.

```
DataItem * clone(SComponent_impl * sb) const;
```

This method creates a clone of the data item owned by another object, the object referenced by the `SComponent_impl` pointer. The value, string limits and flags are copied to the cloned data item.

```
const char * toXML();
```

Returns the current value as an XML string.

```
void getValueAsString(std::string &sValue) const;
```

Returns the current value as a simple string.

```
void getDefaultAsString(std::string &sValue) const;
```

Returns the default value as a simple string.

```
bool setValueAsString(const std::string &sValue);
```

Sets current value from a simple string.

4.3.3.8 Class TimeDataItem

The time data item is a custom data item used to hold the system-wide time data type. It is derived from the SDataItem template, so all its public methods are available. It can be constrained by minimum and maximum values, specified in the configuration file. These are specified as attributes to the XML element corresponding to the data item. The MIN attribute contains a textual representation of the minimum value. The MAX attribute contains the maximum value. If units are specified in the value or limit strings these are applied to the data. If units are specified in the value string or in the attribute UNITS, these are used in the absence of units in the value or limit strings.

4.3.3.8.1 Public non-virtual member functions

```
TimeDataItem (const char*,SBase_impl*);
```

Simple constructor.

```
TimeDataItem (const char*,SBase_impl*,const SCTime*)
```

Constructor with shadow variable.

```
DataItem * clone(SComponent_impl * sb) const;
```

This method creates a clone of the data item owned by another object, the object referenced by the `SComponent_impl` pointer. The value, range and flags are copied to the cloned data item.

```
void init(const char*)
```

Initialization method. This searches the configuration map for an initialization element for the data item, as well as limit attributes to set its range.

```
const char* toXML()
```

Returns value as an XML string.

```
TimeDataItem min() const;
TimeDataItem min(STime);
```

Minimum limit accessor and mutator. The minimum may only be increased from the previous setting, thus reducing the range not enlarging it.

```
TimeDataItem max() const;
TimeDataItem max(STime);
```

Maximum limit accessor and mutator. The maximum may only be decreased from the previous setting, thus reducing the range not enlarging it.

4.3.3.8.2 Public virtual member functions

```
bool range(string&);
```

Copy the GUI range string into the passed string. Returns false if no range string exists.

```
bool isValid(STime);
```

Returns validity status of data item.

```
bool isValueAsStringValid(const string&);
```

Parses and range checks string data representation.

```
void getValueAsString(string&)
```

Returns current value as a simple string.

```
bool setValueAsString(const string&)
```

Sets the current value from a simple string.

4.3.3.9 Class OpStateDataItem

The OpState data item is a custom data item used to hold the SComponent SOpState data type. It is derived from the SDataItem template, so all its public methods are available. It is not designed to be initialized from XML, and is primarily intended to provide a GUI interface the status class object (ie. Online/Offline and the various mode within each category).

4.3.3.9.1 Public non-virtual member functions

```
OpStateDataItem ( const char* name, SComponent_impl* sb,
                  ListDataItem<SComponent::eModes>* pM, bool isPublic = true
);

OpStateDataItem ( const char* name,SComponent_impl* sb,
                  ListDataItem<SComponent::eModes>* pM, SOpState * const pS,
                  bool isPublic = true );
```

The constructors. These follow the pattern of the other data items, with a simple constructor, and one with a shadow variable. One deviation from the pattern, however, is the addition of the pointer to a ListDataItem object. The OpStateDataItem takes care of initializing the range items of the list data item, but it requires the object to be created externally for the sake of the name, which must be named differently from the OpStateDataItem.

We could simply pass in a name, or even better, generate the mode data item name from the state data item name; a suffix would be very easy to append. Then the existence of the embedded data item could be hidden. Why didn't I think of that when I wrote it?!!

4.3.3.9.2 Public virtual member functions

```
void init ( const char * valuetext );
```

Initialization method. As noted above, this data item is not intended to be initialized from a configuration file.

```
DataItem * clone(SComponent_impl * sb) const;
```

This is not implemented for the OpStateDataItem since it is not appropriate to duplicate the status data item to another SComponent object. The function returns a hull pointer.

```
const char * toXML();
```

This is not implemented for the OpStateDataItem. The function returns a hull pointer.

```
bool list ( string &r ) const;
```

Copy the GUI pick list string into the passed string. Returns false if no list string exists.

```
void buildList();
```

This methods forces the internal pick list string to be rebuilt. The list is always the same, but this is provided for consistency with the list data item.

```
void getValueAsString(string &s) const;
```

Returns current value as a simple string.

```
void getTextAsString(string &s) const;
```

Returns the GUI text for current value as a simple string.

```
void getDefaultAsString(string &s) const;
```

There is no default value for this data item, so this method returns the current value as a simple string.

```
bool setValueAsString(const string &s);
```

Sets the current value from a simple string.

```
bool isModified();
```

Modified flag accessor. If the data item is mirrored to an external variable and the external variable has been changed, the internal value is updated and the method returns true.

```
bool isValid ( const SOpState & s ) const;
bool isValid ( const STOVOKOR::SComponent::OPSTATE & s ) const;
bool isValid ( STOVOKOR::SComponent::eOnlineModes s ) const;
bool isValid ( STOVOKOR::SComponent::eOfflineModes s ) const;
```

These are merely provided to satisfy the interface. All return `true`.

```
SOpState value();
SOpState value ( SOpState & s );
SOpState value ( const STOVOKOR::SComponent::OPSTATE & s );
SOpState value ( STOVOKOR::SComponent::eOnlineModes s );
SOpState value ( STOVOKOR::SComponent::eOfflineModes s );
```

Value accessor and mutators. These methods access the value of the data item. In addition to reading or updating the internal store, they synchronize the internal store with the shadow variable and update the modified data flag if the value has changed. Even on a read the modified flag might be set, if the shadow variable differed from the internal store. When the internal store and the shadow variable are synchronized the shadow store takes precedence.

```
bool resetModified();
```

This method synchronizes the data item with the shadow variable and reset the modified flag. The prior state of the flag is returned.

4.3.3.9.3 Related Classes

4.3.3.9.3.1 Class SOpState

This class is a wrapper for the OPSTATE union defined in the SComponent IDL as follows:

```
union OPSTATE switch ( eModes )
{
case eONLINE:  eOnlineModes  onlineMode;
case eOFFLINE: eOfflineModes offlineMode;
};
```

The main reason for the class is to provide operator overloads to facilitate its use in a ListDataItem. The operator overloads are as follows:

```
STOVOKOR::SComponent::OPSTATE& operator= ( const STOVOKOR::SComponent::OPSTATE & s );
STOVOKOR::SComponent::OPSTATE& operator= ( STOVOKOR::SComponent::OPSTATE & s );
STOVOKOR::SComponent::OPSTATE& operator= ( STOVOKOR::SComponent::eOnlineModes s );
STOVOKOR::SComponent::OPSTATE& operator= ( STOVOKOR::SComponent::eOfflineModes s );
bool operator== ( const SComponent::OPSTATE & s ) const;
bool operator== ( STOVOKOR::SComponent::eOnlineModes s ) const;
bool operator== ( STOVOKOR::SComponent::eOfflineModes s ) const;
bool operator!= ( const STOVOKOR::SComponent::OPSTATE & s ) const;
bool operator!= ( STOVOKOR::SComponent::eOnlineModes s ) const;
bool operator!= ( STOVOKOR::SComponent::eOfflineModes s ) const;
operator const STOVOKOR::SComponent::OPSTATE&() const;
operator STOVOKOR::SComponent::OPSTATE&();
operator const STOVOKOR::SComponent::OPSTATE*() const;
operator STOVOKOR::SComponent::OPSTATE*();
```

4.3.3.10 Base Class Methods for Manipulating Data Items

`void addDataItem ( const char * name, DataItem * item );`

- This is only used by the DataItem constructor itself. Should we change it to private and make DataItem a friend of SBase_impl?

`void removeDataItem ( const char * name );`

- This is only used by the DataItem destructor itself. Should we change it to private and make DataItem a friend of SBase_impl?

`void initDataItems();`

This method iterates through all data items owned by the module, and calls init() on each one. It is almost exclusively called from within SBase_impl itself, but is made public to support the configuration manager, which does not follow the normal module registration procedure.

`bool setDataItems();`

This method iterates through all data items owned by the module, and sets the value of each one to its configuration map setting.

`void saveDefaultXML();`

This method iterates through all data items owned by the module, and, for those data items with a configuration level of `XML_CLASSSAVE`, copies the value back into the configuration map.

`void saveCustomXML();`

This method iterates through all data items owned by the module, and, for those data items with a configuration level of `XML_INSTSAVE`, copies the value back into the configuration map.

```
template <typename T> bool isValid( const string &strParamName,
                                    const T value)
```

This method executes the *isValid()* method on the named parameter. It is not recommended if the identity of the parameter can be determined from the context, since it requires searching the parameter map which is not as efficient as direct object access.

```
bool isValueAsStringValid( const string &strParamName,
                           const string &strValue)
```

This method executes the *isValueAsStringValid()* method on the named parameter. It is not recommended if the identity of the parameter can be determined from the context, since it requires searching the parameter map which is not as efficient as direct object access.

```
bool setParamValueAsString( const string &strParamName,
                            const string &strValue)
```

This method executes the *setParamValueAsString()* method on the named parameter. It is not recommended if the identity of the parameter can be determined from the context, since it requires searching the parameter map which is not as efficient as direct object access.

```
DataItem * getParam ( const string &strParamName )
```

This method returns a pointer to the DataItem object for the named parameter. It is not recommended if the identity of the parameter can be determined from the context, since it requires searching the parameter map.

```
template <typename T> bool getParamValue ( const string &strParamName,
                                           T &value )
```

This method executes the *getParamValue()* method on the named parameter. It is not recommended if the identity of the parameter can be determined from the context, since it requires searching the parameter map which is not as efficient as direct object access.

```
template <typename T> bool setParamValue ( const string &strParamName,
                                           T &value )
```

This method executes the *setParamValue()* method on the named parameter. It is not recommended if the identity of the parameter can be determined from the context, since it requires searching the parameter map which is not as efficient as direct object access.

4.3.3.11 Creating New Data Item Classes

Data items can be created for any data type. The existing data items provide useful examples of how these may be implemented.

The ScalarDataItem and ListDataItem templates take care of most requirements.

4.3.3.11.1 Coding Custom Data Types for Use with Class ScalarDataItem

If a class has the following operator function overloads defined it can be implemented as a ScalarDataItem:

```
operator==
operator<
operator>
operator<=
operator>=
operator!=
operator=
```

Also, for default minimum and maximum values to be set an overload of the `numeric_limits` template class must be defined for the custom class. (See the SXL include file *limits*, and the documentation file *limits2.html* for a description of this template class.) If no overload is defined the limits must be defined in configuration XML.

4.3.3.11.2 Non-ranged Data Items Based on Class DataItem

DataItem-based classes are intended as simple containers for data that will be initialized from a configuration file or included on a GUI screen. As noted in the description of the DataItem class, the following methods must be implemented:

```
void init ( const char * valuetext );
void save ( bool instance = false );
void restore();
DataItem * clone(SComponent_impl * sb) const;
const char * toXML();
void getValueAsString(std::string &s) const;
void getDefaultAsString(std::string &s) const;
bool setValueAsString(const std::string &s);
```

Also, by convention, the various overloads of the value() methods should be implemented. If there are several data types that can be passed into the data item, value methods or conversion operators should be defined for all.

The string data item is a good example of a DataItem-based data item class.

In practice, the DataItem class is a little too basic for most data types, and requires a lot of function overrides to complete the interface. Its main merit is the lack of any requirements for operator overloads.

The StringDataItem could be SDataItem-based, really. Why wasn't it?

4.3.3.11.3 Ranged Data Items Based on Template Class SDataItem

SDataItem-based classes are intended as simple containers for data that will be initialized from a configuration file or included on a GUI screen. As noted in the description of the DataItem class, the following methods must be implemented:

```
void init ( const char * valuetext );
DataItem * clone(SComponent_impl * sb) const;
const char * toXML();
void getValueAsString(std::string &s) const;
void getDefaultAsString(std::string &s) const;
bool setValueAsString(const std::string &s);
```

Also, by convention, the various overloads of the value() methods should be implemented. The Save() and Restore() functions have been implemented in the SdataItem class. If there are several data types that can be passed into the data item, value methods or conversion operators should be defined for all.

4.3.4 Class SControl_impl

```
void addFunction( const char *szName, PFN pFn, bool needPOC = true );
void addStringTag( const char * tag, CORBA::ULongLong key = 0 );
void addStringData( const char * cmd, long value, bool final = true );
void addStringData( const char * cmd, double value, bool final=true );
void addStringData( const char * cmd, const char * value,
                    bool final = true );
void addStringData( const char * cmd, const std::string & value,
                    bool final = true );
void resetEventString();
void postEventString();
void sendEventString ( CosEventComm::PushConsumer_ptr src );
```

These methods are all associated with the SCI (SControl Interface) used by the GUI engine, and other external modules. The full usage of this class is described in 'Chapter 5 - The GUI, the SCI, and the outside world'.

4.3.4.1 Overview

This class is a 'mix-in', which is to say that it adds functionality into a class hierarchy but is not in the direct line of derivation. One feature of this method is that a pointer of this type can access members of the instantiated object that are implemented by this class without giving access to or being concerned by methods and members of the base class. A 'mix-in' has no parent class. A 'mix-in' where all the members were declared pure-virtual would be analogous to Interface Inheritance in Java.

The purpose of the SControl class is to provide the GUI interface processing functionality. This is the CORBA interface that the GUI resolves object references down to. The GUI doesn't need access to any other class methods and shouldn't be allowed to.

4.3.4.2 CORBA Interface

```
interface SControl
```

SControl is a mix-in with no parent.

4.3.4.2.1 Constants

personality

```
const string personality = "SControl";
```

4.3.4.3 Operations

```
char* getUIDef( CORBA::ULongLong key, const char* uiDefName,
                CosEventComm::PushConsumer_ptr src);
```

It is called by the GUI to request the XML definition string corresponding to the 'uiDefName' parameter. It includes the compound access key and a CORBA reference to the GUI itself for direct action, if this should be necessary. It authenticates the key and looks the screen name up in the internal map of registered screens. It calls a protected virtual function called 'prepUIDef()', which is used to dynamically generate or alter a screen before it is supplied to the GUI for rendering. An internal copy of the string is made and then it iterates through the list of registered replacement strings, making substitutions in the copy. It returns the ASCIIZ string containing the screen definition.

```
void UIRefresh( CORBA::ULongLong key, CORBA::Boolean first,
                const char* screenName );
```

It is called by the GUI to request that the back-end supply ranges and data to the GUI. It includes the compound access key that it stores in an internal holder variable for future reference. It upcalls to the other definition of 'UIRefresh()' which has a different signature and is declared pure virtual, passing the parameters through . That version is implemented by the class 'SComponent', which is derived from 'SControl'.

```
CosEventChannelAdmin::ProxyPushSupplier_ptr
ECRegister(CORBA::ULongLong key, CosEventComm::PushConsumer_ptr src );
```

It is called by the GUI to register itself with the back-end process and possibly trigger login and authentication. It includes the compound access key and a CORBA reference

to the GUI itself. In all cases but the processing of a login screen, the key is authenticated and the object reference is used to register the GUI with the EventService as a consumer. The virtual function 'ECRegistered()' is called if the user has overridden it. It returns a reference to the ProxyPushSupplier, just in case it is needed.

```
char* UIActivate( CORBA::ULongLong key, const char* actString,
                  CosEventComm::PushConsumer_ptr src);
```

It is called by the GUI to send data or commands to the back-end. It includes the compound access key, a compound data string, and a CORBA reference to the GUI itself. The compound string is disassembled into its individual update messages and flags as to whether any of them require Point-Of-Control (POC) authority. The key is authenticated and POC status of this user is assessed. 'UIPreActivate()' is called if the user has overridden it. If access is denied or POC is required but not possessed, then a flag is set to indicate that the update fails. The string fragments are iterated. For each, if the failure flag was set then the attempted access is logged, otherwise the successful access is logged and all the functions that were registered for each of those updates is called with the associated string fragment. This function assembles a string containing the names off all the individual fields whose update functions returned false. After all that is done, the virtual function 'UIActivated()' is called if the user has overridden it. If this override returns 'false', then the word 'Failed' is appended to the result string to indicate that the whole update failed. Two methods of data validation can be used: The individual fields can be validated (returning true/false) or all the fields can return 'true' and the 'UIActivated()' override can validate all of it. A combination of the methods is allowed. The result string is returned to the GUI.

```
void ECDeregister(CORBA::ULongLong key);
```

It is called by the GUI to de-register itself. The virtual function 'ECDeregistered()' is called if the user has overridden it.

```
void CheckLink() throw(CORBA::SystemException);
```

It is called by the GUI in some circumstances to ensure that the back-end connection is still valid. Its sole purpose is to exist. If the call makes it through the CORBA infrastructure, then the purpose has succeeded. If not, failure will be determined elsewhere.

4.3.4.4 Static Member Functions

```
static bool SCI_scrnCulprit( SControl_impl *obj, const char* culprit,
                             const char *Screen );
```

This method will be called during the processing of an update string from the GUI. It is the built in function which handles the identification of what button or control in the GUI caused this update to occur.

4.3.4.5 Public Member Functions

```
void addFunction( const char *szName, PFN pFn, bool needPOC = true );
```

This method is used to register a particular handler function for a particular data string.

```
void setLoginFunction ( const std::string &sName, bool isLogin );
```

This method is used to mark a registered function as being used during the login process, which alters how key authentication is done.

```
USHORT regGUICnt() { return m_usRegCnt; }
```

This method is an accessor function which returns the number of registered GUI connections.

```
void addActionTag ( const char * cmd );
```

This method will append a command to the GUI data string which is to be interpreted by the GUI Engine itself as a command.

```
void addGlobalPopupTag ( const char * xml, CORBA::ULongLong key = 0 );
```

This method will append a request to the GUI data string which will request that the Engine request and display the specified dialog. The key is forced to 0 to broadcast this request to all attached GUI's.

```
void addPopupTag ( const char * xml );
```

This method will append a command to the GUI data string which will request that the Engine request the specified dialog from this object and display it. The key is set to that of the most recently heard-from GUI, and only that GUI will process the request.

```
void addPopupTag ( const char * target, const char * xml );
```

This method will append a command to the GUI data string which will request that the Engine request the specified dialog from the object specified by "target" and display it.

The key is set to that of the most recently heard-from GUI, and only that GUI will process the request.

```
void postEventString();
```

The method will push the assembled compound string onto the event channel.

```
void addReplacement( const char *szTarget, const char *szSub );
```

This method is used to register substitution string which will be edited into a copy of any screen XML that is requested and the time of the request. This allows for the topical customization of the screens

4.3.4.6 Protected Member Functions

```
CosEventComm::PushConsumer_ptr callerIOR();
```

This method returns a duplicate of the calling GUI's CORBA object reference (IOR).

```
CORBA::ULongLong callerKey() { return m_key; }
```

This method returns the key of the currently conversing GUI.

```
void addNamedXML( const std::string &sName, const std::string &sXML );
```

This method is used to register a string of screen XML in an internal map under a unique name. If the name is a duplicate, then the original is replaced. Any '##' present in the XML is replaced with the objects name at this time.

```
void setLoginScreen ( const std::string &sName, bool isLogin );
```

This method is used to mark a registered screen as being a Login screen, which affects key authentication.

```
void addRange( const char *szItemName, const char *szRange,
               USHORT permLevel = eAUTH_OPERATOR );
```

This method is used by the user or the DataItems to append range information for a particular screen data element. The 'permLevel' parameter sets the authority level required by this data element in order to allow updates.

```
void addRange( const char *szItemName, char *szRange,
               USHORT permLevel = eAUTH_OPERATOR );
```

This method just calls the above method using a non-const string for the range info.

```
void addRange( const char *szItemName, std::string range,
               USHORT permLevel = eAUTH_OPERATOR );
```

This method just calls the above method using an STL string for the range info.

```
void addList( const char *szItemName, const char *szList,
              USHORT permLevel = eAUTH_OPERATOR );
```

This method is used by the user or the DataItems to append selection list information for a particular screen data element. The 'permLevel' parameter sets the authority level required by this data element in order to allow updates.

```
void addList( const char *szItemName, char *szRange,
              USHORT permLevel = eAUTH_OPERATOR );
```

This method just calls the above method using a non-const string for the selection list info.

```
void addList( const char *szItemName, std::string range,
              USHORT permLevel = eAUTH_OPERATOR );
```

This method just calls the above method using an STL string for the selection list info.

```
void addStringTag ( const char * tag, CORBA::ULongLong key = 0 );
```

This method begins a data-string fragment by appending the name of the screen data element and setting a MUTEX semaphore in preparation for the data portion of the fragment to be appended. The key defaults to 0 (broadcast) but can be targeted to a particular GUI.

```
void addStringTagSecure ( const char * tag );
```

This method begins a data-string fragment by appending the name of the screen data element and setting a MUTEX semaphore in preparation for the data portion of the fragment to be appended. This is called 'Secure' because the key embedded in the string will be set to that of the GUI with which we are currently conversing. Only that GUI will process this fragment.

```
void addStringData( const char * cmd, long value, bool final = true );
```

This method appends a LONG data value to the fragment and, if 'final' is 'true' releases the MUTEX. If 'final' is 'false' then the MUTEX is held.

```
void addStringData( const char * cmd, double value, bool final=true );
```

This method appends a DOUBLE data value to the fragment and, if 'final' is 'true' releases the MUTEX. If 'final' is 'false' then the MUTEX is held.

```
void addStringData( const char * cmd, const char * value,
                    bool final = true );
```

This method appends an ASCIIZ string to the fragment and, if 'final' is 'true' releases the MUTEX. If 'final' is 'false' then the MUTEX is held.

```
void addStringData( const char * cmd, const std::string & value,
                    bool final = true );
```

This method appends an STL string to the fragment and, if 'final' is 'true' releases the MUTEX. If 'final' is 'false' then the MUTEX is held.

```
void resetEventString();
```

This method will wipe out the contents of the compound data string and reset it to start over.

```
void postRangeString();
```

This method will push the assembled range onto the event channel.

```
void sendEventString ( CosEventComm::PushConsumer_ptr src );
```

This method will send the assembled data string directly to a specific GUI by making a direct call to the specified IOR's 'push' method.

```
void showScreenList();
```

This method will display a list of the registered screens. It is primarily for debugging purposes.

4.3.4.7 Virtual Member Functions for Optional Overriding

```
virtual void prepUIDef( const char* uiDefName );
```

This method is overridden when the user needs to dynamically generate the requested screen before it is supplied to the GUI engine for rendering. It has no default implementation

```
virtual void ECRegistered ( CORBA::ULongLong key,
                            CosEventComm::PushConsumer_ptr src );
```

This method is overridden when the user needs to be notified of a GUI attaching to the back-end object. This is normally required, as the system counts the number of attachments. If the back-end object must maintain different data for different viewers, the data sets could be stored in a STL map indexed by the GUI key. This has no default implementation.

```
virtual void ECDeregistered ( CORBA::ULongLong key );
```

This method is a companion to the above and is overridden when the user needs to be notified that a GUI is detaching.

```
virtual void UIPreActivate();
```

This method is called during the 'UIActivate()' process but before any of the updates are executed. It has no real default implementation.

```
virtual bool UIActivated();
```

This method is called during the 'UIActivate()' process, after all of the individual updates have been executed. It can be overridden if more complex or complete validation is required. If it returns 'true' then the update is considered to have succeeded, except for the individual fields that may have failed. It does not override the fact that they may have failed. If it returns 'false', then the whole update is considered to have failed, even if no individual field explicitly failed. If some or all of the fields work together and cannot be individually validated, you may use this method. It has a default implementation which returns 'true' if not overridden.

4.3.4.8 Pure Virtual Member Functions for Required Overriding

```
virtual void UIRefresh( CORBA::Boolean first, const char* screenName);
```

This is an internal method which is implemented by the class 'SComponent'. It is called by the publicly accessible variant (which has a different parameter signature). It causes the back-end to assemble data strings for parsing and display by the GUI. If the parameter 'first' is true, range strings will be assembled to be sent prior to the data strings and the data strings will contain ALL of the pertinent data. If the parameter is false, no range info is sent and only those data elements that have changed since the last update are sent.

```
virtual STOVOKOR::Authent::Permit
authenticate ( CORBA::ULongLong key, string & username,
               CosEventComm::PushConsumer_ptr src );
```

This method is a pure virtual which is implemented by the 'SComponent' class. This allows the methods in this class to transparently upcall to the implementation which has access to full system services for key authentication.

```
virtual void requestLogin ( CosEventComm::PushConsumer_ptr src,
                            STOVOKOR::Authent::Permit cause );
```

This method is a pure virtual which is implemented by the 'SComponent' class. This allows the methods in this class to transparently upcall to the implementation which has access to full system services to force the system to issue a login dialog in the event of an update which failed due to an expired key or lack of required POC.

4.3.5 Class SComponent_impl

4.3.5.1 Overview

4.3.5.2 CORBA Interface

```
interface SComponent inherits from STOVOKOR::SBase, STOVOKOR::SControl
```

This is the main interface for all Stovokor components. It combines the interface of the low-level base SBase and the GUI interface, and adds system management methods and data structures required by all physical and virtual instruments, virtual functions, and tasks.

Some of the methods are fully implemented in the implementation class SComponent_impl, while others are provided a minimal implementation or no implementation at all, and a module-specific implementation must be provided in the module servant class. In the description of the implementation class only those methods requiring or permitting overriding are listed. CORBA methods not described in the implementation are fully implemented by the base class.

4.3.5.2.1 Aliases

sComponentStatus

```
typedef COMPONENT_STATUS sComponentStatus;
```

sConnectorFeedback

```
typedef CONNECTOR_FEEDBACK sConnectorFeedback;
```

4.3.5.2.2 Constants

personality

```
const string personality = "SComponent";
```

4.3.5.2.3 Enums

eModes

```
enum eModes
{
    eONLINE,
    eOFFLINE
};
```

System-wide operating modes. The system controller (Node Manager) module will be able to specify the operating mode to all components.

eOfflineModes

```
enum eOfflineModes
{
    eDISCONNECTED,
    eNONOPERATIONAL,
    eINDETERMINATE,
    eNONRESPONSIVE,
    eLOCALCONTROL
};
```

Component operating mode when the system is offline. The component will report the appropriate mode depending on the system operating mode and the internal state of the component.

eOnlineModes

```
enum eOnlineModes
{
    eINITIALIZATION,
    eREADY,
    eNORMAL,
    eREDUCEDCAPACITY,
    eACQUISITION
};
```

Component operating mode when the system is online. The component will report the appropriate mode depending on the system operating mode and the internal state of the component.

eRoles

```
enum eRoles
{
    eACTIVE,
    eBACKUP,
    eNOTUSED
};
```

System-wide operating roles. The system controller (Node Manager) module will be able to specify the operating mode to all components.

4.3.5.2.4 Structs

BasicData

```
struct BasicData
{
    unsigned short heartbeatRate;
    string FWVersion;
};
```

just for getting the Heartbeat and Firmware version upstream

COMPONENT_STATUS

```
struct COMPONENT_STATUS
{
    short tag;
    OPSTATE operationalState;
    eRoles role;
    SBase::eFaultStatus faultStatus;
};
```

Component status structure, which is an aggregation of operating state (mode), role, and fault status. This structure is returned by the explicit getComponentStatus() IDL method, and is propagated on the module event channel when any field changes.

CONNECTOR_FEEDBACK

```
struct CONNECTOR_FEEDBACK
{
    boolean connected;
};
```

Structure for passing feedback about a Connector Object's connection status

4.3.5.2.5 Unions

OPSTATE

```
union OPSTATE switch(eModes)
{
case eONLINE: eOnlineModes onlineMode;
case eOFFLINE: eOfflineModes offlineMode;
};
```

Operating mode reporting structure. Use of a CORBA union ensures consistency of mode reporting. The members are:

`OnlineMode` Component operating mode when the system is in the online state.

`OfflineMode` Component operating mode when the system is in the offline state.

4.3.5.2.6 Operations

`void Initialize(in short cmdType);`

General purpose method to instruct the module to initialize itself and its underlying modules.

`void PIComm(in CAData PIData);`

PIComm is a direct call interface for the communications adapter to bypass the event channel.

`void Reset(in short cmdType);`

General purpose method to instruct the module to reset itself and its underlying modules.

`void Synchronize(in short cmdType);`

General purpose method to instruct the module to synchronize its internal data with its underlying modules.

`boolean execConfig();`

Execute the configuration

```
sComponentStatus getComponentStatus(in short cmdType);
```

This method returns the current component status in the form of the structure defined above. (The cmdType parameter may be deprecated.)

```
boolean getConfigObject(out string cfg);
```

This method creates and returns a configuration object. It returns false if there are no configurable parameters.

```
string getFWversion();
```

Retrieve the firmware version from the remote object.

```
string getGUIName();
```

Return the object's human friendly name.

```
OPSTATE getMode(in short cmdType);
```

This method returns the current operating mode. (The cmdType parameter may be deprecated.)

```
string getParameters(in boolean all);
```

Method to create an XML block from the current parameter settings.

```
eRoles getRole(in short cmdType);
```

This method returns the current operating role. (The cmdType parameter may be deprecated.)

```
string getTrueName();
```

This method returns the object's TRUE name for registering channels, since objects can be retrieved from the name server by an alias (in the case of multiple interfaces).

```
unsigned short getUnitID(in string unitName);
```

getUnitID gets the PI unit number to allow the VI to associate the telemetry and fault structures with the correct PI. Most often will default to 0 which is for the PI itself. In the case of satellite devices in a redundant system, it may be nonzero to indicate which satellite unit

```
unsigned short heartbeatRate(in unsigned short rate);
```

The module heartbeat rate is maintained by a Data Item in this class. This method is used to set the heartbeat rate in milliseconds for the component, but must be overridden. The default implementation only returns the current value.

```
boolean setParameters(in string xml);
```

Method to receive a block of XML and update the parameters from the contents.

```
void setRole(in eRoles role);
```

This method sets the current operating role.

4.3.5.3 Public Member Functions

```
SComponent_impl ( const char * name );
```

This is the only constructor for the class. The name is the system name that will be published in the CORBA Naming Service, and used in the configuration files.

```
void PostGUIRanges ( const char * itemName = 0 );
```

This method assembles and posts to the event channel the range and list strings for all published data items. This is called from within UIRefresh().If the optional argument is supplied, only the named data item's ranges are posted. See chapter 5 for more details.

```
void PostGUIData ( bool postAll = false );
```

This method assembles and posts to the event channel the range and list strings for published data items. This is called from within UIRefresh(). The optional argument, if true, causes the values of all the data items to be posted. Otherwise, only modified values are posted. See chapter 5 for more details.

```
void PostComponentStatus();
```

This method posts the m_componentStatus structure to the event channel. Modules must maintain this structure to reflect their current status, and post it using this method whenever it changes.

```
void SendEventAny ( const CORBA::Any& any );
```

This method is used to propagate arbitrary data over the event channel. The data must be inserted into a CORBA::Any object, and then passed to this function. This typically would be used to publish structured telemetry data, or other module specific structured

data. Modules monitoring the event channel must recognize the data structure to be able to use its data.

```
Authent_ptr getAuthent();
```

This method returns the object reference for the authentication manager. If the module is unavailable a nil reference is returned.

```
void  setCfgServant( SComponent_impl *in_cfgSrvnt );
```

This is only provided for the benefit of the SConfig destructor. In view of the similarity of the two classes it would seem better to make SConfig a friend of SComponent_impl, and remove this method. Also, SConfig's processRoleChange() just calls down to SComponent_impl's version: totally redundant!

4.3.5.4 Protected Data and Member Functions

```
SCStatus m_status;
```

```
ScalarDataItem<unsigned short> * mp_heartbeatRate;
```

This is a pointer to a scalar data item created in the `SComponent_impl` constructor. Its name is *HeartbeatRate* , and it should be initialized through configuration XML. All external access to the firmware version string is through this data item.

```
StringDataItem * mp_FWVersion;
```

This is a pointer to a string data item created in the `SComponent_impl` constructor. The default contents should be replaced with the module specific firmware version as defined in its design document. All external access to the firmware version string is through this data item.

```
static bool SCI_updateParameter ( SControl_impl*, const char*,
                                  const char* );
bool updateParameter ( const char * cmd, const char * target );
```

This pair of functions conform to the SControl activation model, and provide a generic means for updating variables held in data items. The method updates the value of the data item with the name that matches the `target` argument with the string data representation passed as the `cmd` argument.

```
bool authenticate ( CORBA::ULong, string &,
                    CosEventComm::PushConsumer_ptr );
```

This method is called to authenticate an SCI operation with the authentication server. This method is only required as an upcall from SControl.

Do we need to publish this, or does SControl handle it completely?

```
void requestLogin ( CosEventComm::PushConsumer_ptr,
                    STOVOKOR::Authent::Permit );
```

This method is called to force a login after a failed authentication request. This method is only required as an upcall from SControl.

4.3.5.5 Pure Virtual Member Functions for Required Overriding

4.3.5.5.1 Protected pure virtual methods

```
virtual void processRoleChange( STOVOKOR::SComponent::eRoles,from
                                STOVOKOR::SComponent::eRoles to ) = 0;
```

These methods must be implemented to respond to mode and role changes commanded by high level functions. The appropriate action is device and status dependent. All external interface to the module status is funneled through this method and the module's SCStatus object. This simplifies the implementation of module-specific role handling.

Although the method is defined as pure virtual, there is a default implementation in SComponent_impl. This may be called explicitly as

```
SComponent_impl::processRoleChange ( oldRole, newRole );
```

The reason for making the method pure virtual is to force an implementation to be provided at the module specific layer, since role changes are fundamental to the operation of Satlinx objects and must be handled correctly. The default implementation only works for infrastructure and manager objects, not instruments and tasks.

4.3.5.6 Virtual Member Functions for Optional Overriding

4.3.5.6.1 CORBA Interface Methods

```
virtual void Initialize(CORBA::Short cmdType);
```

This is any empty implementation of the CORBA interface method. Modules should override this if they have specific initialization requirements.

```
virtual void Synchronize(CORBA::Short cmdType);
```

This is any empty implementation of the CORBA interface method. Modules should override this if they need to synchronize their internal state with its downstream object, such as PI to device, or VI to PI.

```
virtual void Reset(CORBA::Short cmdType);
```

This is any empty implementation of the CORBA interface method. Modules should override this if they have any reset requirements. If appropriate the module should propagate the call to its downstream object.

```
virtual void UIRefresh( CORBA::Boolean first, const char* screenName);
```

This method is called via the SControl interface and requests an update of the data for the named screen. If the first argument is `true`, the ranges must be sent before the data. Most modules will not need to override the method, but if they do, the second argument indicates which screen is to be refreshed. See chapter 5 for more details.

```
virtual CORBA::UShort heartbeatRate(CORBA::UShort rate);
```

This is any empty implementation of the CORBA interface method. Modules should override this if they have any heartbeat rate requirements. If appropriate the module should propagate the call to its downstream object. The PhysicalInstrument base class overrides this method, so it is not an issue to PI-derived classes. Virtual instrument classes (will most) likely need to override this method to support profiles with heartbeat rate settings in them. The return value is the actual value that the heartbeat rate was set to, which may differ from the requested value due to range constraints.

- How is this handled at present for VIs? Are there telemetry rate changes in the profiles?

```
virtual void PIComm ( const STOVOKOR::CAData& PIData );
```

This is any empty implementation of the CORBA interface method. The `PhysicalInstrument` class overrides this for it derived classes. It should not be called on an object derived directly from `SComponent_impl`.

```
Virtual CORBA::UShort getUnitID( const char* unitName );
```

This is any empty implementation of the CORBA interface method.

- What is the intended use of this? It not overridden anyway at present.

```
CORBA::Boolean execConfig();
```

4.3.5.6.2 Public Virtual Methods

```
void forwardFault ( const struct STOVOKOR::SBase::FaultReport& pFR );
```

This method is provided as an upcall for the SComponent class. The SBase implementation does nothing.

```
void onHeartbeatChange();
```

This is any empty implementation of the method called whenever the component's heartbeat rate is changed. Modules should override this if heartbeat rate affects their operation of if they need to pass the rate change to a downstream object, such as VI to PI.

4.3.5.6.3 Protected Virtual Methods

```
void processScreenXML ( string &xml );

void SendFault ( STOVOKOR::SBase::eFaultStatus, eEventType,
                 eFaultProbableCause, int, const char*, const char* );

void SendFault ( STOVOKOR::SBase::eFaultStatus, eEventType,
                 eFaultProbableCause, int, const char*, const char*);
```

This method creates and sends a fault record over the fault event channel. This is an override of the SBase method. The SComponent_impl object maintains a set of active faults, sorted on severity. This makes it easy to determine the highest active fault for a module, which can then be translated into an object status. This override takes care of managing that set when faults are set or cleared.

```
void configSComponent();
```

This is an upcall from SBase, to hold any SComponent processing of the configuration after it has been loaded. Currently it is empty.

```
void refreshData();
```

What is the intended use of this? It is empty and not overridden anywhere at present, and only called from getConfigObject().

4.3.5.7 Related and Subordinant Classes

4.3.5.7.1 Class SConfig

The SConfig class is a specialization of the SComponent_impl class that is used as part of the configuration subsystem. There is no reason to directly instantiate an SConfig object, as that is handled by the `getConfigObject()` IDL method.

4.3.5.7.2 Class SCStatus

The SCStatus class is a wrapper class for the COMPONENT_STATUS IDL structure. Each module must maintain a component status object to support the interface to the rest of the system. This maintenance is through the methods defined for this class.

In addition, some objects may need to maintain multiple status objects for child objects. In these cases the additional status objects can be tagged to distinguish them from the main status object.

- Has anybody used the tagging scheme? Does it need additional support in upper layers?

- This comment appears in the code: "This also needs data items tied to the fields of the status structure. This will enable maintenance screen to monitor and modify the object mode and role, and monitor the object state and fault state." I think the data items are already there!

4.3.5.7.2.1 Public Member Functions

```
SCStatus();
SCStatus ( int tag );
```

These are the constructors for the component status object. The version with no argument creates a primary status object for the module. It is invoked when the SComponent_impl object is created, and should not be used by module developers.

The second form of the constructor is for creating additional tagged status objects for satellite objects. The tag value must be non-zero.

```
const STOVOKOR::SComponent::sComponentStatus& operator()();
```

The function operator returns a reference to the internal COMPONENT_STATUS structure. The reference is a constant, so the structure may only be read, not changed.

```
void setOwner ( SComponent_impl * owner );
```

This method initializes the owner member of the object, and enables it to create the data items associated with the status fields. It is called for the primary status object in the SComponent_impl constructor. The module must call this method for all additional status objects it creates, passing in its '`this`' pointer.

```
const ListDataItem<SComponent::eModes> * getModeItem() const;
const ListDataItem<SComponent::eRoles> * getRoleItem() const;
const OpStateDataItem                  * getStateItem() const;
```

These methods provide access to the internal data items corresponding to the various status fields.

```
void  carryOver();

void setFaultStatus ( SBase::eFaultStatus f );

SBase::eFaultStatus getFaultStatus();

void raiseFault ( STOVOKOR::SBase::eFaultStatus fault );

bool isOnline();
```

This method test the current mode of the module and returns `true` if is online.

```
bool isActive();
```

This method test the current role of the module and returns `true` if is active.

```
bool isBackup();
```

This method test the current role of the module and returns `true` if is in backup.

```
bool isNotUsed();
```

This method test the current role of the module and returns `true` if is not used.

```
void onlineMode ( STOVOKOR::SComponent::eOnlineModes s );
```

This method returns the online mode of the object.

```
void offlineMode ( STOVOKOR::SComponent::eOfflineModes s );
```

This method returns the offline mode of the object.

```
void setStatus ( STOVOKOR::SComponent::OPSTATE f );
```

This method sets the current state of the object, online or offline.

```
STOVOKOR::SComponent::OPSTATE getStatus();
```

This method returns the current state of the object, online or offline.

```
void setRole ( STOVOKOR::SComponent::eRoles f );
```

This method sets the current role of the object.

```
STOVOKOR::SComponent::eRoles getRole();
```

This method returns the current role of the object.

```
void post();
```

This method posts the component status structure on the module's event channel.

4.3.6 Class PhysicalInstrument

4.3.6.1 Overview

The physicalInstrument class is an intermediate class, derived from 'SComponent', and from which all PhysicalInstrument (PI) modules should be derived. It transparently handles the connection to the CommAdapter, which communicates directly to the device or simulator. This class implements an internal state machine to handle the stages of connection and reconnection. It manages the Online and Offline states. It also implements a timer operation which operates at the programmed 'Heartbeat' interval and is used to drive operations in the PI object. The PI code will operate on this timer-tick to issue their status queries to the device and to send data messages upstream on the event channel. This class provides automated functionality to any application modules which are derived from it. It does this by operating on several threads. The state machine runs on its own independent thread. The calls that proceed out to the device are either on the main module thread or on a CORBA thread from an inbound call. Data coming in from a device are also in a CORBA thread due to their being inbound CORBA calls. This allows calls out to the device to block while the timing loop and inbound message processing are not impeded. This class creates and manages to Timeout dataItem and a collection of Fault Objects which relate to communications failures:

mp_ConnectToCAFailed

mp_CAChannelDetached

mp_ResetFailed

mp_LostComm

mp_CmdTimeOut

mp_CmdResultFailed

mp_TransientWarning

mp_DirectedOfflineIndeterminate

4.3.6.2 Static Member Functions

```
static bool SCI_Mode( SControl_impl * pThis, const char * cmd,
                      const char * target );
```

This method is the static handler function for dealing with changes between online and offline modes. It calls 'processModeChange()' to perform the actual work.

4.3.6.3 Public Member Functions

```
PhysicalInstrument(const char * name);
```

The class constructor. It takes in the object name and passes it down to the 'SComponent' class from which this is derived. It creates the data item for the Timeout data element which will be parsed from the setup XML. It creates the fault objects which relate to communications failures.

```
virtual ~PhysicalInstrument();
```

The destructor is virtual, which will cause the whole destructor chain to be called even if the object is deleted using a base-class pointer. This just deletes the elements allocated by the constructor but doesn't do any other management.

```
CORBA::UShort heartbeatRate(CORBA::UShort rate);
```

This method is an override of the CORBA interface method in the 'SComponent' interface. It is used to set the update rate of the timer loop. This value is range-chacked by the mp_heartbeatRate dataItem, which will restrict the value to the acceptable range rather than reject out of range values. The method returns the actual value which becomes the new setting. If the input value is '0', then the method simply returns the current setting.

```
void reset(CORBA::Short cmdType);
```

This method is an override of the CORBA interface method in the 'SComponent' interface. It is currently improperly declared and thus inactive. It is used to stop, reset,

and restart the communications interface to the device. It sends a fault if the reset fails. The parameter is vestigial.

```
void PIComm( const STOVOKOR::CAData& PIData );
```

This method is an override of the CORBA interface method in the 'SComponent' interface. It is called by the CommAdapters to send data from the device connection to this object. It resets the timeout timer and upcalls to the 'processDevMsg()' method, which the user must override to process the message. The message may be directly from the CommAdapter itself regarding the state of the linkage. If that is the case, this method sets the appropriate state and sends the appropriate fault.

```
bool connect();
```

This is an internal method used by the state machine to attempt to make the device connection.

```
void stopPIStateMachine();
```

This method sets a flag which stops the state machine from ticking.

```
short PIC_SendData( long dataLen, const char* szData,
                    bool bBlock = false, const string sRspTag = "");
```

This method is called by the application level to send data down to the device. This overload is used to send a binary block of data, so it requires a length. It has a 'bBlock' with a default value of 'false' which will cause the call to be asynchronous. If the parameter value is 'true' then the call will block and next parameter MUST be used, although it has a default value of "". This value should be the value that the communication subsystem will wait for before releasing this call.

```
short PIC_SendData( const char* szData, bool bBlock = false,
                    const std::string sRspTag = "");
```

This method is called by the application level to send data down to the device. This overload is used to send an ASCIIZ string of data, from which it will get the length. It has a 'bBlock' with a default value of 'false' which will cause the call to be asynchronous. If the parameter value is 'true' then the call will block and next parameter MUST be used, although it has a default value of "". This value should be the value that the communication subsystem will wait for before releasing this call. This method actually calls the first overload after processing some parameters.

```
short PIC_SendData( const std::string sData, bool bBlock = false,
                    const std::string sRspTag = "")
```

This method is called by the application level to send data down to the device. This overload is used to send an STL string of data, from which it will get the length. It has a 'bBlock' with a default value of 'false' which will cause the call to be asynchronous. If the parameter value is 'true' then the call will block and next parameter MUST be used, although it has a default value of "". This value should be the value that the communication subsystem will wait for before releasing this call. This method actually calls the first overload after processing some parameters.

```
void PIC_SendDataDone(bool bSuccess, const std::string sRspTag);
```

The 'processDevMsg()' will receive a message from the device which it must parse. Once it determines what command the message is a response to, it must call this method with the appropriate tag to release the command which is most likely blocking. The tag that the command contained must match the tag used in this call.

```
bool isConnectedToCA();
```

This is an accessor method which returns the flag which describes the connection state.

```
void clearTOFault();
```

This method is a helper to clear the Timeout fault.

4.3.6.4 Protected Member Functions

```
void onChangeToOnlineActive();
```

The method is called internally by the state machine on the transaction to the OnlineActive state. It is only called once during the transition.

```
void onChangeToOnlineNotUsed();
```

The method is called internally by the state machine on the transaction to the OnlineNotUsed state. It is only called once during the transition.

```
void onChangeToOnlineBackup();
```

The method is called internally by the state machine on the transaction to the OnlineBackup state. It is only called once during the transition.

```
void onChangeToOfflineNotUsed();
```

The method is called internally by the state machine on the transaction to the OfflineNotUsed state. It is only called once during the transition.

```
void onChangeToOfflineFault();
```

The method is called internally by the state machine on the transaction to the OfflineFault state. It is only called once during the transition.

```
void checkFW_Version( const std::string sOldVer,
                     const std::string sNewVer);
```

This method is called by the state machine on successfully achieving a connection to the device. Synchronize is called synchronously to get the devices basic settings, including firmware version. This method is called next. If the retrieved firmware version is different from the last one used, a message is formulated with the new firmware version and the current heartbeat rate and pushed onto the event channel. A transient fault is sent signaling the version change.

```
void checkPrivateParams();
```

This method is called by the state machine on successfully achieving a connection to the device. Synchronize is called synchronously to get the devices basic settings. This method is called after 'checkFW_Version'. This method just calls isPrivateParamsGolden and sends a transient fault if the result is 'false'.

```
void processModeChange ( STOVOKOR::SComponent::eModes oldMode,
                         STOVOKOR::SComponent::eModes newMode );
```

This method is called internally to process the transition the online and offline modes. If a module is taken offline, its role changes to 'NotUsed'.

```
void processRoleChange ( STOVOKOR::SComponent::eRoles oldRole,
                         STOVOKOR::SComponent::eRoles newRole );
```

This method is called internally to process the transition between the roles 'Active', 'Backup', and 'NotUsed'. If a module's role is set to 'Active' or 'Backup' and it was offline, it is placed online first.

```
void toOnlineState( STOVOKOR::SComponent::eOnlineModes state  );
```

This is a pointless wrapper function which just changes to online mode and state.

```
void toOfflineState( STOVOKOR::SComponent::eOfflineModes state  );
```

This is a pointless wrapper function which just changes to offline mode and state.

```
virtual void disConnectCA();
```

This method forces a disconnection from the device and closes the communications channel.

```
void setNoCA();
```

This method is used by modules which fit into the system hierarchy as PI's, but have no actual device to talk to. This allows the PhysicalInstrument class to be used just to drive the heartbeat timing loop.

```
void startPIStateMachine();
```

This method is called by the application to query the name of the CommAdapter to attach to and start the state machine thread which will try to establish a connection. Because it queries a configuration value, it should not be called in to modules constructor.

```
unsigned short setIdleRate(unsigned short rate);
```

This internal method sets the idle rate, which does nothing.

```
unsigned short getIdleRate();
```

This internal method queries the internal idle rate, which isn't used.

```
unsigned short setUpdateRate(unsigned short rate);
```

This method sets the actual heartbeat rate, which is range constrained. If the set value is outside the allowed range, it will be set to the closest limit value, rather than being rejected. It sets a flag indicating the rate change and builds a message with the new value and pushes it onto the event channel and returns the value that was actually set.

```
unsigned short getUpdateRate();
```

This method just returns the current heartbeat setting.

```
bool isHeartbeatChanged();
```

This method returns the state of the flag. This will only be 'true' after the value has been changed but before the timing loop changes its update rate to comply.

```
void resetHeartbeatChangeFlag();
```

This method is a pointless wrapper function to change the value of the flag to 'false'.

```
void debugOn(bool bOn);
```

This method sets the internal mp_Debug dataItem to 'true' and posts a message to that effect on the GUI, in case the user is using it. This is an assist for the user, but has functionality in the environs of the PhysicalInstrument class.

```
bool isDebugOn();
```

This method returns the state of the mp_Debug dataItem.

```
virtual void onHeartbeatChange();
```

This method sets the heartbeatChanged flag and wakes up the timing loop. It is not used.

```
void suspendUpdate();
```

This method sets the flag which short circuits the timing loop and state machine. It causes the system to do nothing until turned back on.

```
void resumeUpdate();
```

This method restore the timing loop and state machine to their prior function.

```
bool isUpdateSuspended();
```

The method returns whether the telemetry loop has been suspended or not.

4.3.6.5 Virtual Member Functions for Optional Overriding

```
virtual void doOnlineActive();
```

This method is to be overridden if needed. It is called every time through the state machine while in the OnlineActive state and is to provide any functionality the user may require in this state.

```
virtual void doOnlineBackup();
```

This method is to be overridden if needed. It is called every time through the state machine while in the OnlineBackup state and is to provide any functionality the user may require in this state.

```
virtual void doOnlineNotUsed();
```

This method is to be overridden if needed. It is called every time through the state machine while in the OnlineNotUsed state and is to provide any functionality the user may require in this state.

```
virtual void doOfflineNotUsed();
```

This method is to be overridden if needed. It is called every time through the state machine while in the OfflineNotUsed state and is to provide any functionality the user may require in this state.

```
virtual void doOfflineCriticalFault();
```

This method is to be overridden if needed. It is called every time through the state machine while in the OfflineCriticalFault state and is to provide any functionality the user may require in this state.

```
virtual void onlineActiveToOnlineBackup();
```

This method is to be overridden if needed. It is called once on the transition from OnlineActive to OnlineBackup. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineActiveToOnlineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OnlineActive to OnlineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineActiveToOfflineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OnlineActive to OfflineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineActiveToOfflineFault();
```

This method is to be overridden if needed. It is called once on the transition from OnlineActive to OfflineFault. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineBackupToOnlineActive();
```

This method is to be overridden if needed. It is called once on the transition from OnlineBackup to OnlineActive. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineBackupToOnlineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OnlineBackup to OnlineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineBackupToOfflineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OnlineBackup to OfflineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineBackupToOfflineFault();
```

This method is to be overridden if needed. It is called once on the transition from OnlineBackup to OfflineFault. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineNotUsedToOnlineActive();
```

This method is to be overridden if needed. It is called once on the transition from OnlineNotUsed to OnlineBackup. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineNotUsedToOnlineBackup();
```

This method is to be overridden if needed. It is called once on the transition from OnlineNotUsed to OnlineBackup. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineNotUsedToOfflineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OnlineNotUsed to OfflineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void onlineNotUsedToOfflineFault();
```

This method is to be overridden if needed. It is called once on the transition from OnlineNotUsed to OfflineFault. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void offlineFaultToOnlineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OfflineFault to OnlineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void offlineFaultToOfflineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OfflineFault to OfflineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void offlineNotUsedToOnlineNotUsed();
```

This method is to be overridden if needed. It is called once on the transition from OfflineNotUsed to OnlineNotUsed. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual void offlineNotUsedToOfflineFault();
```

This method is to be overridden if needed. It is called once on the transition from OfflineNotUsed to OfflineFault. It is used to provide any functionality the user may require to affect this transition other than what the state machine handles automatically.

```
virtual CORBA::Short doSoftReset();
```

This method is to be overridden if possible. It is called during the state machine's processing to trigger a soft reset. The user must override this to provide specific functionality. The default implementation currently sleeps one second and returns the value for success.

```
virtual bool isPrivateParamsGolden();
```

This method is meant to be overridden. It is used to compare the current private parameters with the original XML defaults and return 'true' if they match and 'false' if not. The default implementaion always returns 'true'.

4.3.6.6 Pure Virtual Member Functions for Required Overriding

```
virtual void updateTelemetry(bool bBadTelemetry = false);
```

This function MUST be overridden. It is called on every tick of the timing loop, so that the application code can perform its period operations. It is passed 'true' if the is NO connection to the required device, or 'false' if there is an active connection.

```
virtual void processDevMsg(const std::string &sDevMsg);
```

This function MUST be overridden. It is called whenever an update message is received from the attached device. It is passed the message as it was received and is responsible for parsing the message into the necessary DataItems and releasing and calls that were blocked waiting for this response.

4.3.7 Class ReportManager

```
DWORD CreateReport( SComponent_impl *pcomponent, LPCTSTR pdatafile,
                    LPCTSTR ptemplate, LPCTSTR phtmlfile,
                    bool showfile, bool showwait);
```

4.3.7.1 Overview

The ReportManager class provides a simple, easy-to-use method to invoke the Report Generator application, which interfaces with Microsoft Excel in order to convert developer-supplied data files to output HTML suitable for viewing by users. All knowledge of COM/DCOM and BackOffice development is encapsulated by the ReportManager/ReportGenerator combination, simplifying construct of output reports considerably.

4.3.7.2 Public Member Functions

```
DWORD CreateReport( SComponent_impl *pcomponent, LPCTSTR pdatafile,
                    LPCTSTR ptemplate, LPCTSTR phtmlfile,
                    bool showfile, bool showwait);
```

ReportManager has only a single pubic member function, which supplies the necessary information to the Report Generator application. The developer simply specifies the calling object's pointer, as well as the paths to the three files (CSV data file, Excel Template (.xlt) file, and output HTML file). Setting the optional 'showwait' and 'showfile' parameters control the display of a progress dialog, as well as the launching of a browser to display the formatted HTML output produced by the Report Generator application.

4.3.7.3 Generating Reports

Designing reports for use with the Report Manager class is fairly straightforward, and consists of following steps:

1. Write a function in your application to output the report information to a comma-separated-value (CSV) data file.
2. Run the application to output the data file.
3. Copy the data file to C:\TEMP.
4. Rename the file to REPORT.TXT.
5. Copy the RPTTEMPLATE.XLT file from the Satlinx\Infrastructure\ReportGenerator project directory to Config\Operations\Analysis Templates directory.
6. Rename the template to something more appropriate (*<projectname>.xlt* is a good start).
7. Run Excel.
8. Load the new template (<projectname>.xlt) you just created/copied.
9. When the template loads, it should automatically load up the REPORT.TXT file you created in step (4).
10. Format the data in Excel until the results are acceptable.
11. Save the new spreadsheet **as a template** back to your <projectname>.xlt file.
12. When prompted to clear the data and allow Excel to automatically refresh the data on startup, click on 'Yes'.

Continue to make changes to your code that outputs the CSV file, and repeat 2-12 until the desired result is achieved. Once you have all of the CSV data formatted correctly, you can speed up the development of reports by editing the ReportManager.cpp file and commenting out the line that deletes the REPORT.TXT file from C:\TEMP. Doing so you can eliminate steps 1-9, and can concentrate on editing the report template file in Excel. You should not have to alter the Excel macro contained in the template file except in **very** unusual circumstances; we have yet to see a need to do so, and describing Excel macro programming is out of the scope of this document.

4.4 Infrastructure Object Modules

4.4.1 Authentication Module

4.4.1.1 Overview

The authentication module is used to authenticate external access to the system via the SControl interface. It validates both UserID/Password and PointOfControl. It will primarily be used for GUI access, but may at some point be used for script-engine control. The only system module that will make direct authentication calls will be the NodeController or some similar controlling entity. Calls to this module are otherwise automatic and the user would generally not attempt it.

The UserID database is maintained in an external file by another process, the User List Editor, which will call to the Authentication module to refresh its internal user list structures when the file has been changed.

4.4.1.2 CORBA Interface

```
interface Authent inherits from Sbase.
```

4.4.1.2.1 Constants

personality

```
const string personality = " Authent";
```

4.4.1.2.2 Enums

AuthentLevel

```
enum AuthentLevel
{
   Guest,
   Operator,
   Administrator,
   Integrator
};
```

These are the various authority levels within the system, from lowest to highest.

Permit

```
enum Permit
{
   Grant,
   BadKey,
   Timeout,
   AccessDeny,
   POCDeny
};
```

These are the possible responses:

`Grant`	Full access is granted to this UserID
`BadKey`	In invalid key has attempted to access the system. The key has a compound structure and must be properly formed
`Timeout`	An active and authenticated key has attempted access to the system but not within the required timeout. This will cause a login dialog to popup, forcing the user the revalidate their credentials
`AccessDeny`	Unknown UserID or Password doesn't match
`POCDeny`	User granted access, but not Point-Of-Control. They will have access to all data and allowed to change non-critical values, but not critical values until the current owner or POC relinquishes it

4.4.1.2.3 Structs

```
struct AuthentData
{
   unsigned long key;
   AuthentLevel maxLevel;
};
```

Contains the compound key and the permission level.

4.4.1.2.4 Operations

```
unsigned long long Login ( in string name, in string password,
                           in boolean POC, in unsigned long ip );
```

This method attempts to validate the UserID and Password. It takes in the UserID, Password, true/false for requesting POC, and the IP of the calling client (which is extracted from the inbound IOR). The permission level is associated with the UserId, but is transparently downgraded to 'Guest' if the user does not request control. The method returns the compound key, which is generated. The key has a unique number, the access level, and the IP embedded in it. This is called by the NodeController (or equivalent).

```
unsigned long long LoginNode ( in string name, in string password );
```

Unimplemented.

```
void Logout ( in unsigned long long userkey );
```

Removes the key from the map so that it can't be used again.

```
Permit Authenticate ( in unsigned long long userkey,
                      in unsigned long ip, out string username );
```

This method is used by the SControl interface to validate each inbound call. Each inbound message will have a key prepended to it. This call checks that the key is valid, current, and that the embedded IP matches the IP that's passed in. It resets the 'LastAccess' time used for timing keys out. It returns a selection from the 'Permit' enum.

```
void DeletePOC ( in unsigned long long userkey );
```

Deactivates the last UserID to have POC and reverts it to the prior UserID who relinquished it but didn't log out.

```
void RelinquishPOC ( in unsigned long long oldkey,
                     in unsigned long long newkey );
```

Relinquish Control authority without logging out. UserID stays active, but gets a new key which reflects their new `Observer` status.

```
void AccessEnable ( in boolean enable );
```

Set whether Authent can authenticate UserIDs or not. If authentication is disabled, all calls to Authenticate() return `AccessDeny`. This method is provided to allow a lockout of GUI or script control of the system during startup or shutdown

```
void RefreshUserList();
```

Causes the Authentication module to refresh its internal user map from the User File.

4.4.1.3 File Format

```
<XML>
  <USERS>
    <NUM_USERS>2</NUM_USERS>
    <USER AUTHORITY='INTEGRATOR' ID='mbaggett' NAME='Marion Baggett'
PASSWORD='abcdef'></USER>
    <USER AUTHORITY='ADMINISTRATOR' ID='axeman' NAME='Fred Flintsone'
PASSWORD='abcdef'></USER>
  </USERS>
</XML>
```

This is the file format, as created by the UserList Editor. This file resides in the 'Equipment Defaults' and should never be edited directly.

4.4.2 Configuration Manager

The configuration manager is the root of the system, and is the module that starts all others. Even though it is the startup module it is deployed as a DLL for revision management purposes. The actual startup executable is described later in section 4.5.5.

The main functions performed by the configuration manager are:

1. Verify the working directory tree, and create missing directories.
2. Load the system configuration XML file and module instance configuration XML files into internal data structures.
3. Start all modules required at startup.
4. Start deferred startup modules on demand.
5. Support module registration, and provide module specific configuration data to modules upon registration.
6. Monitor module status and restart modules that die prematurely.
7. Process system shutdown.

Some of these functions are autonomous, while others are in reponse to calls to the configuration manager's CORBA interface methods. The interface and the autonomous functions will now be discussed in some detail.

4.4.2.1 Installation Mode

When the system is started in installation mode, described in section 4.5.5.2, the Configuration Manager just performs needed housekeeping and then terminates,

without starting any modules or services. At present this is limited to step 1 described above. Future enhancements may perform additional version specific system setup functions when the Configuration Manager is started in this mode.

4.4.2.2 CORBA Interface

interface STOVOKOR::CfgMgr

```
interface CfgMgr inherits from STOVOKOR::SBase
```

This is the ConfigManager interface. It is the initial startup module of the system, and provides initial configuration information for all other modules through the registration method.

4.4.2.2.1 Aliases

ConfigItem

```
typedef s_ConfigItem ConfigItem;
```

InitialConfig

```
typedef s_InitialConfig InitialConfig;
```

These are aliases for the configuration structures descrbed below.

4.4.2.2.2 Enums

eConfigData

```
enum eConfigData
{
    eNODENAME,
    eNODECTL,
    eNODEINFO,
    eFILEMANAGER,
    eAPPNAME,
    eSOFTWAREPATH,
    eCONFIGPATH,
    eDATAPATH
};
```

Configuration data type constants. These are used to identify items in the `ConfigItem` sequence in the `InitialConfig` structure.

4.4.2.2.3 Structs

s_ConfigItem

```
struct s_ConfigItem
{
    eConfigData type;
    string dataField;
};
```

s_InitialConfig

```
struct s_InitialConfig
{
    sequence<ConfigItem> cfgItems;
    string xml;
};
```

These data structures are used to transfer system configuration information to the module when it registers. Rather than a fixed structure for the configuration items, a sequence is used to make future additions easier.

4.4.2.2.4 Operations

Activate

```
boolean Activate(in string name);
```

This is the method to request activation of a deferred startup module. The parameter is the name of the object to start.

Configured

```
void Configured(in string name, in StringSeq personalities);
```

This method is called at the end of the registration process, when the module has processed all its configuration data.

CreateComplete

```
void CreateComplete(in string name);
```

This method is called by a factory object after it has process the creation request. The module may not have been created, since a failure may have occurred, but other data cached by the configuration manager is used to determine the success of the operation.

ExitAndReboot

```
void ExitAndReboot();
```

This method instructs the configuration to shut down the software and reboot the host computer.

ExitAndRestart

```
void ExitAndRestart();
```

This method instructs the configuration to shut down the software and restart it.

Register

```
boolean Register ( in string name, in eModuleType type, in SBase sb,
                   in boolean fromFactory, out InitialConfig config );
```

This method is called by modules to announce their presence to the configuration manager, and to retrieve their startup configuration data.

Unregister

```
void Unregister(in string name);
```

This method is called by modules to announce their shutting down to the configuration manager.

4.4.2.3 Manage Working Directory Tree

The working tree is described in detail in section 2.3.2.2. The configuration manager builds this tree, or any missing fragments of it. This ensures that all required directories exist when accessed. If new subdirectories are added to the tree, the configuration manager will need to be modified to create them.

Note that this part of the code is OS specific.

4.4.2.4 Manage Configuration XML

4.4.2.4.1 Node Information File

The first piece of configuration data is contained in the node information file, "*nodeinfo.xml*". This file lives in the same directory as the startup executable, and defines the unique node name and the root paths for the configuration and data repository trees. Its contents are as follows:

```
<XML>
   <SYSTEM NODENAME='NEWNODE' CFG_PATH='D:\Satlinx\Config'
                              DATA_PATH='D:\Satlinx\Data'/>
</XML>
```

This is the first piece of configuration data loaded, and it determines where the configuration manager will look for additional configuration files.

The node information file contains a `SYSTEM` section, which has the following attributes:

`NODENAME` The CORBA object name of the node cluster itself. The group of objects comprising the instrument cluster and all the control modules register themselves in a single naming context with the CORBA naming service, which is logically similar to a directory in a file system. This allows multiple clusters to use standard names for the objects without name collisions. This name is passed to the module executables on the command line.

`CFG_PATH` The root of the configuration directory tree. This name is passed to the module objects when they register with the configuration manager.

`DATA_PATH` The root of the data repository directory tree. This name is passed to the module objects when they register with the configuration manager.

The node information file can be created by the system startup service, as described in section 4.4.5

4.4.2.4.2 System Configuration File

The system configuration is defined in XML, contained in a file named after the as the startup service executable. For obscure historical reasons the startup executable is named "*stovokor.exe*" in the standard source archive. For deployment this would typically be renamed to something relevant to the customer, or to a generic name such as "*satlinx.exe*". The startup service uses the executable name to locate the system configuration, which has the same name as the executable, but with the extension changed to "*.xml*". This file lives in the "*Equipment Defaults*" directory in the configuration tree. Thus, by default the system configuration is found in "*stovokor.xml*", and the general rule is that the for startup executable "*<start>.exe*" the configuration will be found in "*<config root>/Operations/Equipment Defaults*".

Reproduced below is a non-functional configuration file, but it includes all the fields that typically appear in such a file.

```
<XML>
   <SYSTEM CONFIG="CFGMGR" LOGGER="LOGGER" REVMGR="REVISION" FILEMGR="FILEMGR"
           NODEINFO="L1NODEINFO_MGR" NODECTL="NODE-AKM" BASEPORT="20000">
      <CORBA_ARG NAME="NameService" ARG="-ORBInitRef"
                 VALUE="NameService=corbaloc::localhost:15000/NameService"/>
      <RUN NAME="NameService" EXECUTABLE="corbaservice.exe" ARGUMENT1="NameService"
           ARGUMENT2="nameserv.exe" ARGUMENT3="-passthrough-OAport" ARGUMENT4="15000"
           USE_CORBA_ARGS="Yes" USE_VERSION_PATH="No"/>
      <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="authent.exe" NAME="AUTHENT"/>
      <MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="FAULTMGR" EXECUTABLE="factory.exe">
         <TYPE>FAULT_MGR</TYPE>
         <FACTORY_NAME>FAULTMGR_FACTORY</FACTORY_NAME>
      </MODULE>
      <MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="USERLIST_EDIT" STARTUP="No"
              EXECUTABLE="factory.exe">
         <TYPE>USERLIST_EDIT</TYPE>
         <FACTORY_NAME>USERLIST_EDIT_FACTORY</FACTORY_NAME>
      </MODULE>
   </SYSTEM>
   <FAULTMGR>
      <DEFAULTS>FaultMgrDflt</DEFAULTS>
      <SCREENS>FaultMgrScrn</SCREENS>
      <CONFIG_LOG_ENABLE>Enabled</CONFIG_LOG_ENABLE>
      <RESOURCE_LOG_ENABLE>Enabled</RESOURCE_LOG_ENABLE>
      <FAULTLOG>fault.log</FAULTLOG>
   <FAULTMGR>
</XML>
```

The configuration file contains a `SYSTEM` section, and, optionally, some module instance configuration blocks.

4.4.2.4.2.1 SYSTEM Section Attributes

The system section has the following attributes that are added to the attributes defined in the node information file:

`CONFIG` The CORBA object name of the configuration manager. The configuration manager uses this name to register itself with the CORBA naming service, and all other modules use this name to access it. This name is passed to the module executables on the command line.

`LOGGER` The CORBA object name of the system logger. The logger uses this name to register itself with the CORBA naming service, and all other modules use this name to access it. This name is passed to the module executables on the command line.

`REVMGR` The CORBA object name of the revision manager. The revision manager uses this name to register itself with the CORBA naming service, and all other modules use this name to access it. This name is passed to the module objects when they register with the configuration manager.

`FILEMGR` The CORBA object name of the file manager. The file manager uses this

name to register itself with the CORBA naming service, and all other modules use this name to access it. This name is passed to the module executables on the command line.

`NODEINFO` The CORBA object name of the node information manager. The node information manager uses this name to register itself with the CORBA naming service, and all other modules use this name to access it. This name is passed to the module objects when they register with the configuration manager.

`NODECTL` The CORBA object name of the node controller. The node controller uses this name to register itself with the CORBA naming service, and all other modules use this name to access it. This name is passed to the module executables on the command line.

`BASEPORT` The base IP port number used for the CORBA ORBs. By specifying the port number to the ORB, the object references are guaranteed to be the same across multiple incarnations of the objects. Therefore, even if an object dies and is restarted, all other modules that have its IOR will still be able to access it. Otherwise it would be necessary to code mechanisms to discard stale references and reinitialize them. This number needs to be chosen so as not to clash with any other applications that allocate a range of IP ports for their own use.

4.4.2.4.2.2 CORBA_ARG Element

These are arguments passed on the command line of all executables, and are used to set standard parameters for the ORB, such as identifying the CORBA naming and event services. They are implemented as name/value pairs.

The attributes specified in the CORBA_ARG key are:

`NAME` This is a unique name for the argument. It is not included in the command line, but is needed by the parser.

`ARG` Argument string.

`VALUE` Value string.

If both attributes are defined, the ARG contents precede the VALUE contents. The two arguments are space delimited on the command line.

4.4.2.4.2.3 RUN Element

RUN elements define commands to execute programs that do not contain Satlinx modules. These typically start required external services, such as the CORBA naming and event services.

The attributes specified in the RUN key are:

`NAME`	This is a unique name for the argument. It is not included in the command line, but is needed by the parser.
`EXECUTABLE`	The name of the executable to run.
`ARGUMENT`*n*	Multiple arguments may be defined. These must be name sequentionally, starting from ARGUMENT1. The string value of the attribute is added as a space delimited argument on the command line. Arguments are ordered on the command line by their numeric suffix, rather than by the order they occur in the XML file.
`USE_CORBA_ARGS`	If this is present and set to "No", the CORBA arguments described above are not added to the command line. By default the CORBA arguments are included on the command line.
`USE_VERSION_PATH`	If this is present and set to "No", the executable is run without a directory prefix. By default the version directory is prepended to the executable name, making the executable version specific.

4.4.2.4.2.4 MODULE Element

MODULE elements define the parameters for all the Satlinx modules.

The attributes specified in the MODULE key are:

`LOCAL`	Whether an object is created on this system or just registered here. Defaults to 'Yes'.
`USE_FACTORY`	Whether this module is created by a factory or run directly. Defaults to 'No'.

NAME	The object's name as it will be located in the map, created, and registered in the naming service.
EXECUTABLE	The name of the executable that will create the module.
STARTUP	Whether the object should be started now or deferred. Defaults to 'Yes'.
USE_DEBUG_CONSOLE	Whether this module should have a console window. Defaults to 'No'.
THREADS	How may threads the threadpool should pre-allocate. Defaults to 10.

In addition to the attributes listed above, the following parameters are sub-elements of the module element.

TYPE	The type of module to be created. As described in section 4.4.3, the object factory uses the type of the object to locate the DLL containing its implementation.
FACTORY_NAME	The name of the factory object that will create the module's CORBA object.

This could easily be changed in code to make them attributes. Would it be worthwhile at this point?

4.4.2.4.2.5 Module Instance Elements

Each module should have instance specific configuration XML. Only those that have no GUI support, and no configurable operating characteristics can do without.

The XML blocks may be contained within the system configuration file, or in separate files. This section discusses their implementation within the system configuration file. From the sample file above the instance configuration example is as follows:

```
<FAULTMGR>
   <DEFAULTS>FaultMgrDflt</DEFAULTS>
   <SCREENS>FaultMgrScrn</SCREENS>
   <CONFIG_LOG_ENABLE>Enabled</CONFIG_LOG_ENABLE>
   <RESOURCE_LOG_ENABLE>Enabled</RESOURCE_LOG_ENABLE>
   <FAULTLOG>fault.log</FAULTLOG>
<FAULTMGR>
```

Some instance variables are common to all objects, and are interpreted at the base class level. The rest are specific to the module class, and are accessed from the configuration map as required. The common instance variables are:

DEFAULTS	This is the name of the default XML configuration for the object. Typically this is common to all instances of the type, rather than to the individual instance, although it would be possible to clone multiple default files and tweak them for different instances. This approach would defeat the purpose of having separate class and instance configurations. The named configuration is to be found in an XML file in the Equipment Defaults directory in the configuration tree.
SCREENS	This is the name of the XML screen definitions for the object. This is described in more details in chapter 5. The named screen definitions are to be found in an XML file in the Screens directory in the configuration tree.
GUINAME	This is a free form text string that is saved in the configuration as the warm and fuzzy name of the object. It is available for automatic substitution in screen XML, as described in section 5.2, and is returned by the SComponent::getGUIName() CORBA interface method. It is also inserted into fault and event records posted on the fault channel.
FAULTBASE	This is the faultcode base as described in section 4.3.1. This is only required for satellite VIs, as described in section 3.4.2.

4.4.2.4.2.6 Instance Configuration Files

As noted above, each module should have instance specific configuration XML and the XML blocks may be contained within the system configuration file, or in separate files. This section discusses their implementation in separate file. The file containing the instance configuration example from the sample file above would have the name FAULTMGR.xml, and would be found in the System Setup directory in the configuration tree. Its contents would be as follows:

```
<XML>
   <DEFAULTS>FaultMgrDflt</DEFAULTS>
   <SCREENS>FaultMgrScrn</SCREENS>
   <CONFIG_LOG_ENABLE>Enabled</CONFIG_LOG_ENABLE>
   <RESOURCE_LOG_ENABLE>Enabled</RESOURCE_LOG_ENABLE>
   <FAULTLOG>fault.log</FAULTLOG>
<XML>
```

Note that the object name is now used to identify the file, and the configuration elements are directly under the root XML key. Once the instance configuration data is loaded from the file into the configuration manager's internal data structures its original location is unknown and irrelevant.

Should this last statement be true? Don't we say that the system configuration file loads at `XML_INSTINIT` level, while the instance files load at `XML_INSTSAVE` level?

4.4.2.5 Initial Module Startup

Once the system and instance configuration files are loaded into its internal module map, it executes the programs called for in the RUN and MODULE elements of the system configuration. The sequence is as follows:

1. RUN items, in the order they appear in the system configuration file.
2. Non-factory created MODULES, in alphabetical order.
3. Factories with startup modules, in alphabetical order.
4. Startup MODULES in factories, in order as the factories register with the configuration manager.

4.4.2.6 Deferred Module Startup

For modules with the `STARTUP` attribute set to 'No', the configuration manager reserves a space in its module map but performs no immediate operation. Later, when another module requests activation of the deferred module by calling the `Activate()` CORBA interface method, it searches the map for the module details. One of three conditions may obtain:

1. The object is already active.
2. The object's factory is active, but the object is not.
3. Neither the object, nor its factory, are active.

All these case are handled, and the `Activate()` call does not return until the factory is started and the object created and registered. It is up to the other module to determine that the object needs to be activated, and it is up to the activated module to know when and how to shut itself down. When shutting down it must call the `Deregister()` CORBA interface method on the configuration manager, so the internal module map can be updated, and the absence of the module will not be detected as an error.

4.4.2.7 Module Registration

Modules derived from the `SBase_impl` class will call the configuration manager's `Register()` method when they are created. This call is not automatic to the construction of the object, but the factory calls the base class method that handles module registration. If a module creates addition `SBase` derived objects that need to register with the configuration manager, the creating module must call the `DoRegister()` method on the created object to trigger its registration.

When a module registers the configuration searches its internal module map to find it. If found, its configuration XML is returned along with the system configuration items described above. If the module is not found an entry is created in the module map, and the system configuration items are returned.

Once the module completes its configuration it calls the configuration manager's Configured() method. The module is now flagged as active in the module map and reported as such to the node controller.

Two special cases exist:

1. If the module is a factory the configuration manager will issue a series of Create() calls to create the object required on that factory.

2. If the module is the node controller, a list of all modules in the map is sent to it, so it can build the dynamic GUI screens, and issue module lists on request.

4.4.2.8 Module Status Monitoring

The configuration manager has an autonomous thread that periodically pings each module. The parameters for the ping thread are configurable, as to when it starts, how often it runs, and how long it waits for a ping to timeout before posting a failure. The configuration XML for these parameters is reproduced below:

```
<XML>
   <PINGINTERVAL>5000</PINGINTERVAL>
   <PINGTIMEOUT>1000</PINGTIMEOUT>
   <PINGDELAY>60000</PINGDELAY>
</XML>
```

All these parameter are in milliseconds.

The `PINGINTERVAL` is the frequency of the ping loop. The thread will sleep for this period between each pass through the module map, calling Ping() on each registered module.

The `PINGTIMEOUT` is timeout for the Ping() call. If the call does not return before the timeout a failure will be posted. Note that the default for CORBA method calls is for an infinite timeout, once the module exists. The ping thread creates an overloaded IOR for the object with a timeout, so if the object is very busy the call may timeout, even though the module is alive and well. A single failure would indicate a dead module.

The `PINGDELAY` is an initial sleep at the start of the ping loop. The thread will sleep for this period before processing the module map. This allows time for the system to stabilize before status checking begins.

Additional parameters are hardcoded as constants in *CfgMgr_impl.h*:

```
const int kSBasePingMax      = 10;
const int kSFactoryPingMax   = 5;
const int kSComponentPingMax = 5;
```

These determine the number of consecutive failed pings that are interpreted as a total failure. The number of missed pings can be different for each type of module.

These parameters should also be made configuration items, rather that hard coded.

If a module fails to respond past the above limits it is killed and restarted by the configuration manager.

4.4.2.9 System Shutdown

There are three ways to shutdown the system, described below.

4.4.2.9.1 Standard Shutdown

The configuration manager overrides the base class `Exit()` method to perform a total system shutdown. Using the module map, it calls `Exit()` on all registered modules. After a delay it then calls the OS `kill()` function on all remaining modules.

4.4.2.9.2 Shutdown and Restart

The CORBA interface method `ExitAndRestart()` performs the same sequence of operations as the `Exit()` method, but then returns a restart code to the startup service. This then goes through the startup procedure again. If the version key file has changed from the previous run, a different release of the software will be started.

4.4.2.9.3 Shutdown and Reboot

The CORBA interface method `ExitAndReboot()` performs the same sequence of operations as the Exit() method, but then returns a reboot code to the startup service. This calls the OS reboot API function to force a full system shutdown and reboot.

4.4.3 Object Factory

4.4.3.1 Overview

The object factory is used by the Configuration Manager to create one or more of the modules listed in the 'Stovokor.xml' file. The factory is a named object like all the others and its name is registered in the NameServer, although the ConfigManager is the only module that will ever connect to it. The factory creates a mapfile of module information which is used to locate the required modules ate startup.

4.4.3.2 Factory Operation

4.4.3.2.1 Module definition for the DLLMap

The factory process uses a map file to associate the type with a specific DLL where the operative code resides. The information used to create the map and to control the object creation is declared in the 'main' source file, name 'whatever_main.cpp'. It consists of a structure filled with module identification data and a well-known entry points. The structure 'DLLMap' is defined with the following structure:

```
typedef enum { Multithreaded, Singlethreaded, SinglethreadedGroup } ThreadModel;

#define StovokorVF   STOVOKOR::StovokorVF
#define StovokorVI   STOVOKOR::StovokorVI
#define StovokorPI   STOVOKOR::StovokorPI
#define StovokorCA   STOVOKOR::StovokorCA
#define StovokorSIM  STOVOKOR::StovokorSIM
#define StovokorOps  STOVOKOR::StovokorOps
#define StovokorBase STOVOKOR::StovokorBase
#define StovokorTask STOVOKOR::StovokorTask
```

```
// The following data structure is returned to the DLLmap utility, that creates
// a directory of supported modules. This map is used by the generic factory
// executable to load the appropriate DLL and instantiate the objects.

struct sDLLMap_0
{
   ThreadModel threading;                       // Threading model for servant
   STOVOKOR::eModuleType  type;                 // ModuleType within system
   char        name[32];                        // Module name (class name)
   char        creator[32];                     // Name of creation function
   const char * personality[kDLLMapMaxPers];    // Mix-in personalities supported
};

typedef struct sDLLMap_0 DLLMap;
```

The first field, 'threading', controls how the POA that the object is created on is declared. This controls the threading environment that the individual module runs in. Multiple modules in the same factory can each have a different thread model.

The next entry, 'type', declares whether the module is a PI/VI/VF etc. This is used by the NodeController to categorize the individual modules.

The next entry, 'name', is the name by which this module will be located in the map file, invoked, and registered in the name server for other modules to gain access.

The 'creator' field is the name of the entrypoint function that the factory will locate and call to create the module.

The last entry is an array, 'personality', which contains the names of the IDL interfaces that this module exports. This should be filled by using the 'personality' static entry in the specified IDL interfaces. This ensures that typographical errors do not prevent a module from being properly identified.

The following example is from 'LNAC_PI_Miteq_main.cpp'. Note that it exports multiple interfaces:

```
#include "windows.h"

#include "SBase_impl.h"
#include "LNAC_PI_Miteq_impl.h"
#include "dllmap.h"

DLLMap   objects[] =
{
   {
      Multithreaded,
      StovokorPI,
      "LNAC_MITEQ",
      "CreateMiteqPI",
      STOVOKOR::LNAController::personality,
      STOVOKOR::LNASwitch::personality
   }
};
```

The contents of this structure are read in by the factory during the mapping process by calling a boilerplate function, which will be identical in all 'main' source files.

```
extern "C" __declspec(dllexport) const void *
ObjectDetails ( int * count, int * version )
{
   *version = kDLLMapVersion;
   *count   = sizeof objects / sizeof objects[0];
   copyPersonalities ( objects, (sizeof(objects)/sizeof(DLLMap)) );
   return (void*)objects;
}
```

The 'ObjectDetails' function returns the informational structure to the factory for processing.

4.4.3.2.2 Creating the DLLMap

Whenever the module definition information in the above structure changes, or a new module has been created that has not existed before, the mapfile must be recreated. This is done by the user executing 'factory -–map' (note double '-'). This will start a factory process which will search the current directory for any file ending in '.dll'. It will load each one in turn and attempt to call the 'ObjectDetails' function within it. If the attempt fails, it just unloads the DLL and moves on to the next one. For each one that succeeds, it assembles the returned data into an internal map containing the information for all modules, and the name of the DLL that supports each module. When complete, it writes the array out to a file called 'dllmap.txt'.

When a new system is started for the first time, obviously no map exists. The first factory started will discover this and will create one automatically

4.4.3.2.3 Module Creation

Each factory is created or controlled initially by the startup XML in the 'stovokor.xml' file. Observe the section:

```
<MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="LNA_A" EXECUTABLE="factory.exe">
   <TYPE>LNA_VI</TYPE>
   <FACTORY_NAME>LNA_VI_FACTORY</FACTORY_NAME>
</MODULE>
```

This causes the Config Manager to search for a factory called 'LNA_VI_A_FACTORY'. If it is not found, one is created as a separate process. The Config Manager calls the factory's Create() method with the object name 'LNA_A' and the type 'LNA_VI'. The request is placed on the factory's workqueue. The thread associated with the workqueue processes the request and attempts to create an object of the requested type within its own context. To accomplish this task, the factory references the internal module map constructed from the 'dllmap.txt' file. It searches for an entry corresponding to the type parameter and, if it finds one, it loads the associated DLL and attempts to locate the entry point named in the 'creator' field in the structure:

```
extern "C" __declspec(dllexport) const STOVOKOR::SBase_impl *
CreateMiteqPI ( const char * type, const char * name )
{
   STOVOKOR::LNAC_PI_Miteq_impl * pTS = 0;
   STOVOKOR::SBase_impl *rc = NULL;

   pTS = new STOVOKOR::LNAC_PI_Miteq_impl ( name );
   rc = pTS;
   copyPersonalities ( objects, (sizeof(objects)/sizeof(DLLMap)) );

   return rc;
}
```

If the function is found, it is called, which results in the creation of the module. The module itself will then call back to the Config Manager for the rest of its configuration data.

If, during module startup, the Config Manager comes across a similar section:

```
<MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="LNA_B" EXECUTABLE="factory.exe">
   <TYPE>LNA_VI</TYPE>
   <FACTORY_NAME>LNA_VI_FACTORY</FACTORY_NAME>
</MODULE>
```

Note that the factory name is the same, but the module name is different. The Config Manager will find this one already in existence, so it will direct the existing factory to make another module called 'LNA_B' within its own context. Both modules will share the same debug console, if one exists, and run in the same memory space on different threads.

4.4.3.3 CORBA Interface

`interface SFactory inherits from SBase`

Note that the SFactory is only used by the Configuration Manager. No other module should utilize this interface.

4.4.3.3.1 Constants

personality

```
const string personality = "SFactory";
```

4.4.3.3.2 Operations

void Create (in string type, in string name);

Used to create modules, using the type and name. Note that the Create() method queues a module creation request. It returns before the module is actually created. The configuration manager provides the CreateComplete() method to indicate that the module has been created.

void Delete (in string name);

Unimplemented.

void EnableDebugConsole();

Turns on the ability for the factory to show a console window. All modules created within the context of the factory will use the same console window. Default behaviour is not the show the window.

4.4.4 Fault Manager

4.4.4.1 CORBA Interface

Interface STOVOKOR::FaultManager

```
interface FaultManager inherits from STOVOKOR::SComponent
```

4.4.4.1.1 Aliases

FaultList

```
typedef sequence<SBase::FaultReport> FaultList;
```

4.4.4.1.2 Constants

personality

```
const string personality = "FaultManager";
```

4.4.4.1.3 Operations

DisableFaultPromotion

```
void DisableFaultPromotion();
```

This method suspends fault and event promotion from the Fault Manager to the Node Operations Manager. All records that would be promoted are queued internally until promotion is re-enabled.

EnableFaultPromotion

```
void EnableFaultPromotion();
```

This method enables fault and event promotion from the Fault Manager to the Node Operations Manager. All records that have been queued internally are processed before new records are promoted.

GetActiveFaults

```
short GetActiveFaults ( out FaultList fl );
```

This method gets a list of active faults from the Fault Manager. This is primarily used by the Asset Manager to initialize its topology map. Until this method is called, the Fault Manager queues fault records internally. Once it has been called, fault records that have been queued internally are processed through the Asset Manager before new records are promoted.

4.4.4.2 Operation

The fault manager receives all fault and event records posted on the system fault channel. Incoming records may be categorized as follows:

1. Event records for simple logging.
2. Fault records for simple logging.
3. Event records for logging, and promotion outside the node.
4. Fault records for logging, severity translation, and promotion outside the node.

Cases 1 and 2 are simple. All records are logged into a daily log file, a fragment of which is reproduced below:

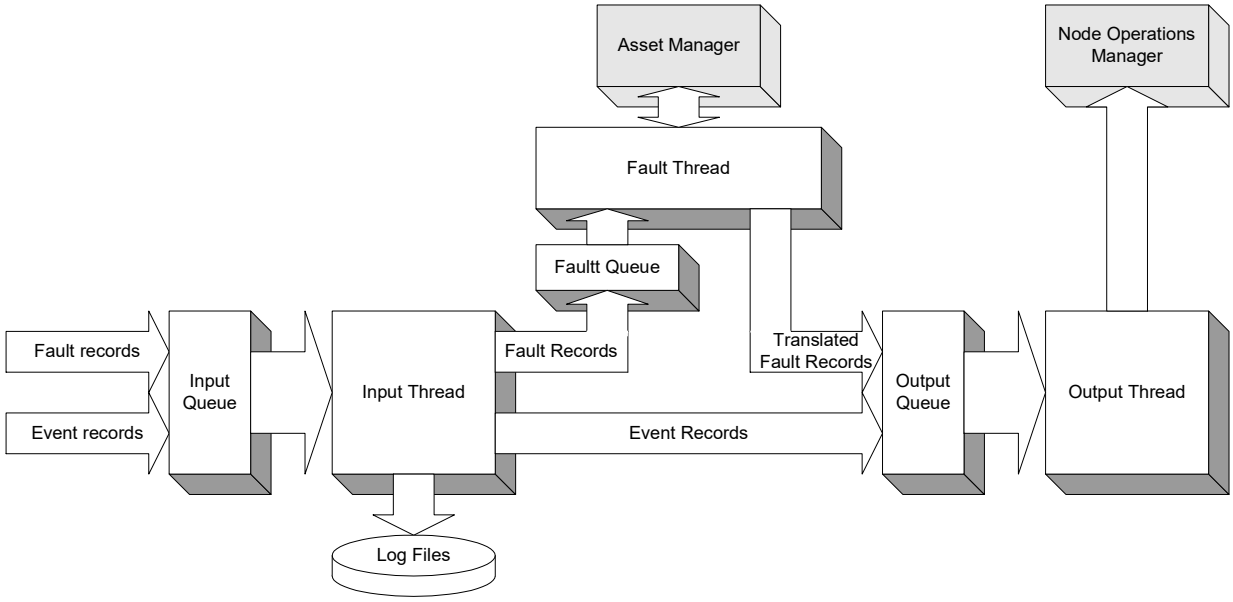
```
Status,13,2002/02/12 17:20:27.351,COMPUTER,Status Event,Resource Change,None,1,Role = Active,
Alarm,16,2002/02/12 17:20:27.351,COMPUTER,Communications,Hardware Fault,Cleared,101,CPU USAGE ALARM,
Alarm,17,2002/02/12 17:20:27.401,COMPUTER,Communications,Hardware Fault,Cleared,102,RAM USAGE ALARM,
Alarm,18,2002/02/12 17:20:27.401,COMPUTER,Communications,Hardware Fault,Cleared,103,DISK USAGE ALARM,
Alarm,19,2002/02/12 17:21:17.964,COMPUTER,Communications,Hardware Fault,Minor,101,CPU USAGE ALARM,
Status,14,2002/02/12 17:21:34.598,AUTHENT,Status Event,User Login/Logout,None,0,q,The Q Continuum
Status,15,2002/02/12 17:21:34.668,NODE-AKM,Status Event,User Login/Logout,None,0,q,LOGIN
Alarm,20,2002/02/12 17:21:58.1,COMPUTER,Communications,Hardware Fault,Minor,102,RAM USAGE ALARM,
```

Event records are recorded as "Status" lines in the file, and fault records are recorded as "Alarm" lines. This file is found in the Data Repository tree, under the month and day directories. Additionally, all fault records are recorded in a monthly fault log under the moth and "Monthly Logs" directory. The format is the same as above. Optionally, resource change events and configuration management events are recorded in monthly files in the same directory.

Case 3 involves the promotion of events. In some configurations, there is a"Node Operations Manager" module, which acts as a single point of access for second tier control into the node. If such a module is included in the configuration, the fault manager will promote certain classes of events to the "Node Operations Manager", which will then post them on its event channel to the second tier of control. All promoted events are also written to the relevant logs, before they are passed on to the "Node Operations Manager".

Case 4 involves the translation and promotion of faults. In some configurations, there is an "Asset Manager" module, which maintains a topology map of the system. This map allows it to determine the impact of faults on the overall operation of the node. Thus critical faults in non-critical components can be downgraded to their actual severity to the node. If such a module is included in the configuration, the fault manager will pass certain classes of faults to the "Asset Manager", and update the severity with the revised value returned from the Asset Manager. All faults are written to the fault log before they are translated. All translated faults are passed on to the "Node Operations Manager".

To support the detours in record processing the Fault Manager is implemented as a multi-threaded object. A graphical representation of the internal structure is as follows:



On input over the CORBA interface, all records are placed on the input queue. The input thread formats the records and logs them to the relevant log files. Fault records subject to promotion are placed on the fault queue, and event records subject to promotion are placed on the output queue.

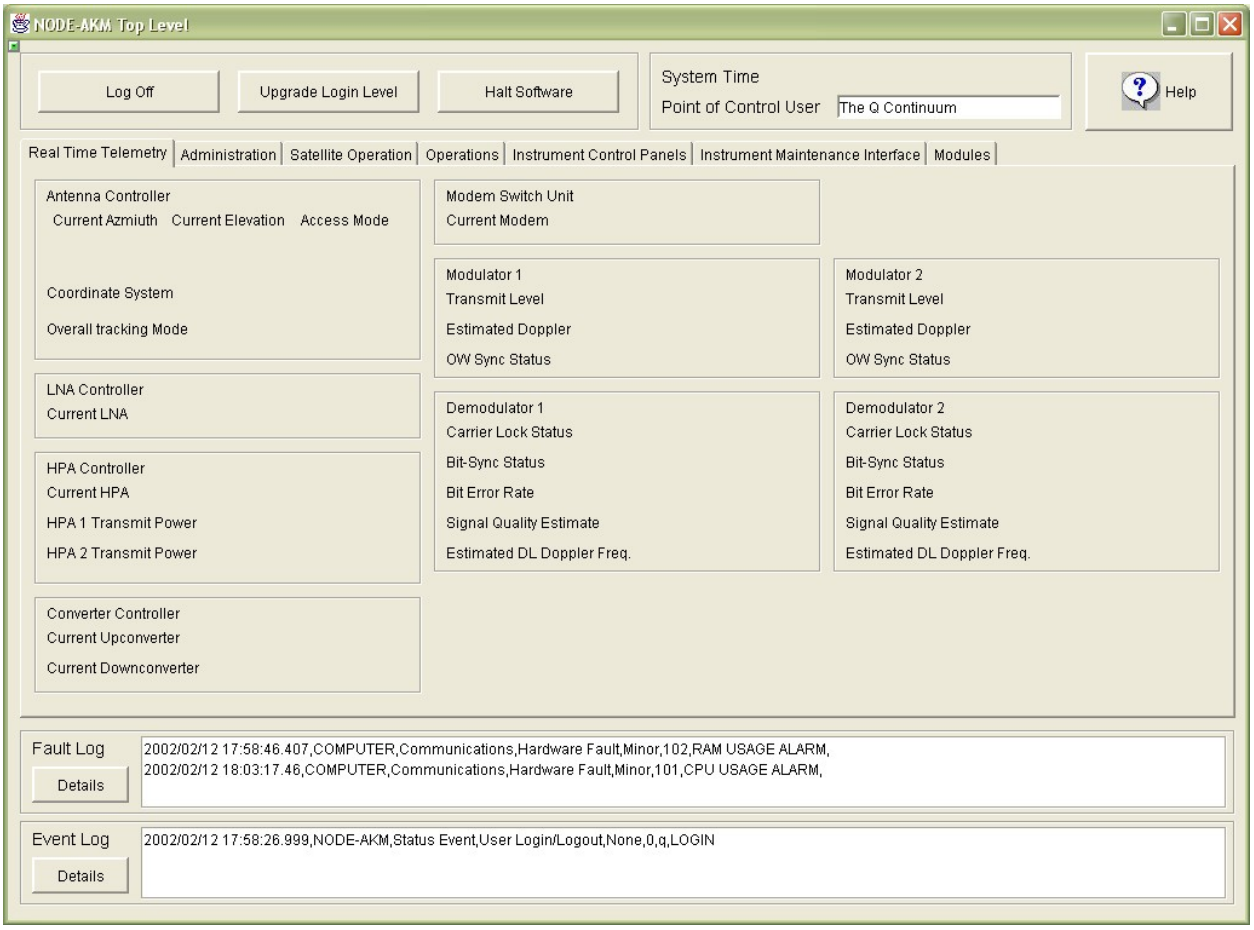
The fault thread waits until it has established a connection to the Asset Manager, and then dequeues fault records from the fault queue. Those needing translation are passed to the Asset Manager and then they are placed on the output queue. If no Asset Manager is configured for the system fault records are simply placed onto the output queue.

The output thread waits until it has established a connection to the Node Operations Manager, and then dequeues fault and event records from the output queue and forwards them to the Node Operations Manager. If no Node Operations Manager is configured for the system fault records are simply discarded.

4.4.4.3 User Interface

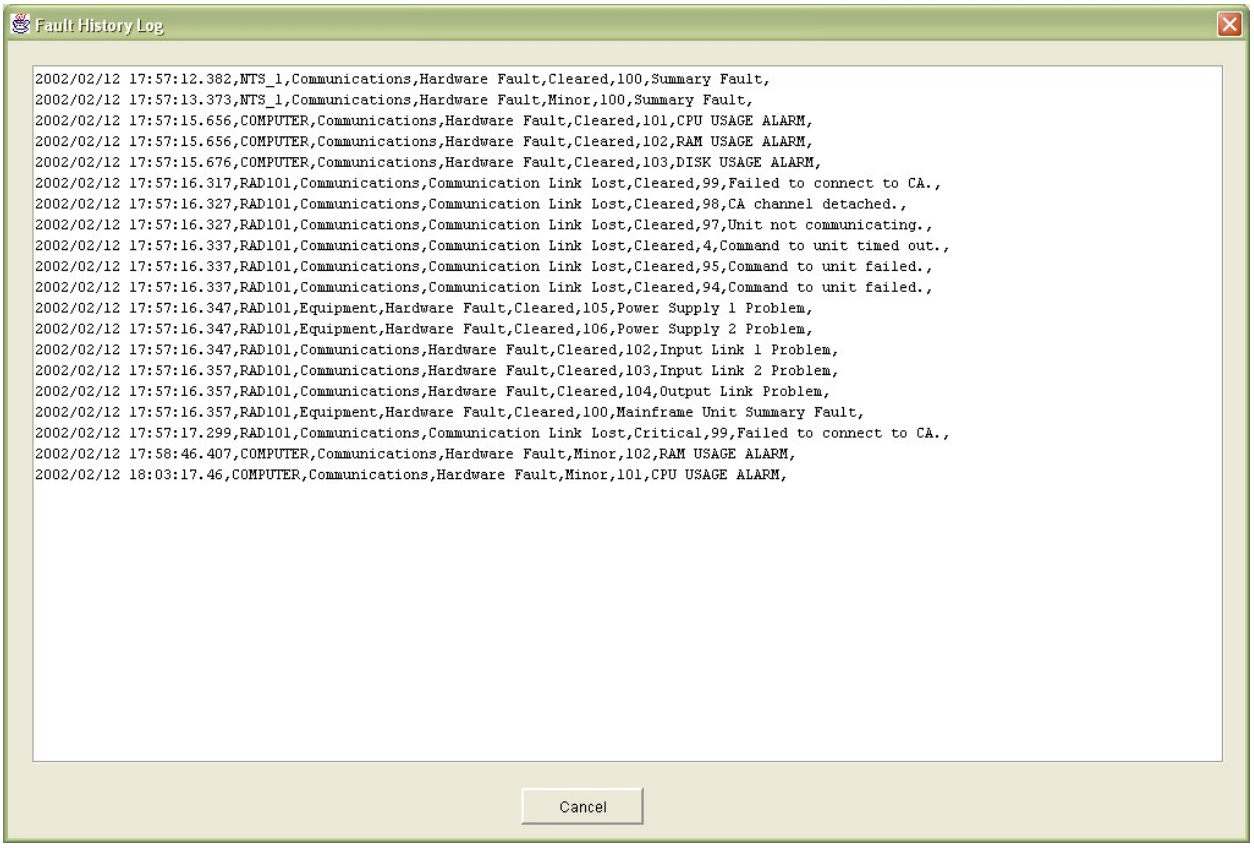
The Fault Manager provides three separate screens for viewing logs and controlling logging and reporting.

The first two screens are intended for linkage to the top-level screen of the Satlinx system. Reproduced below is a typical top-level user interface screen:

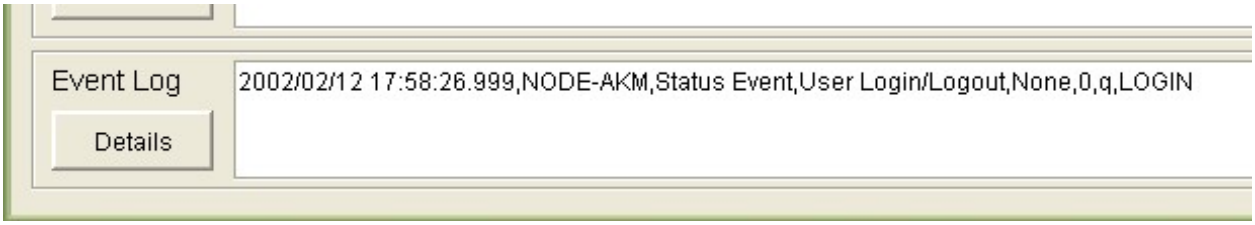


In the bottom segment of the screen are two small scrolling windows showing faults and events as they are received by the Fault Manager. Beside each one is a button that is linked to a full screen display of the log.

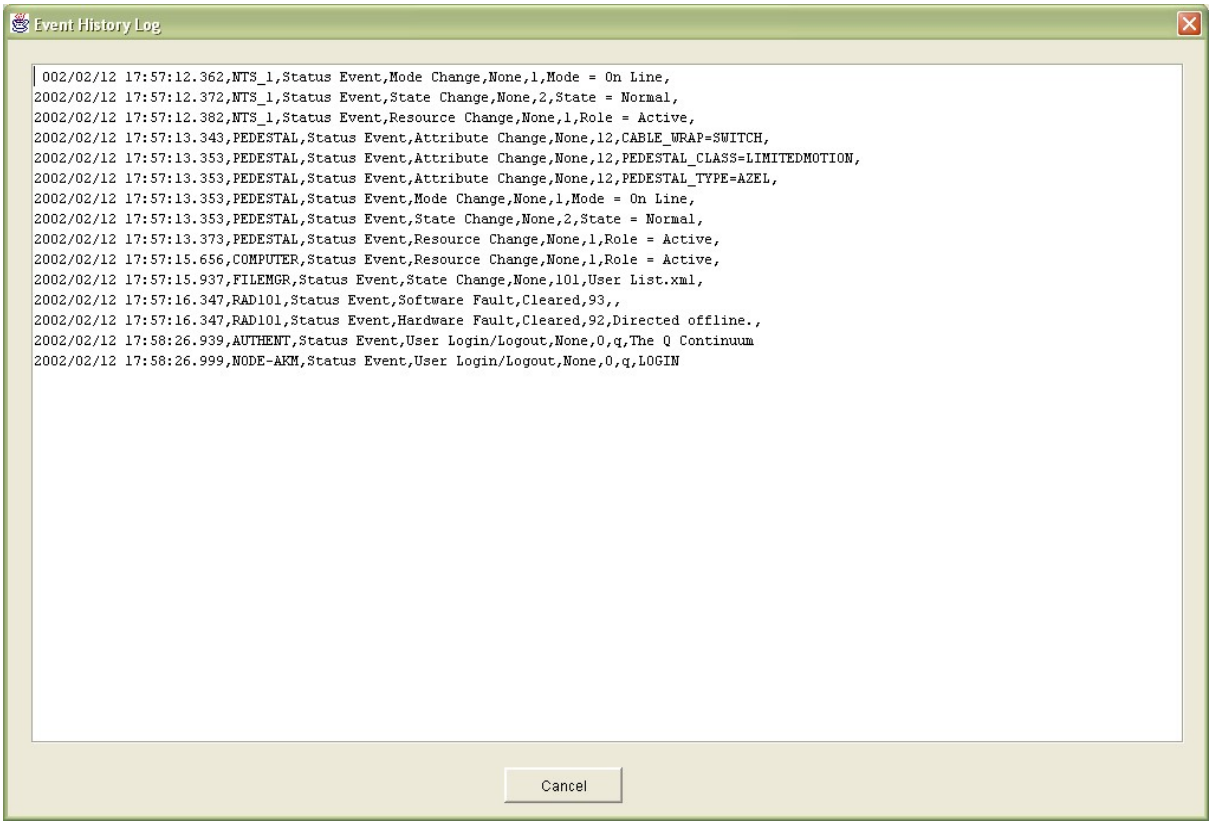
The "Details" button above would cause the Fault Log History screen to pop up:



Note that the small window on the top-level screen contained the most recent records listed in the log. The history log is scrollable back to the start of the run, up to a configurable maximum size.



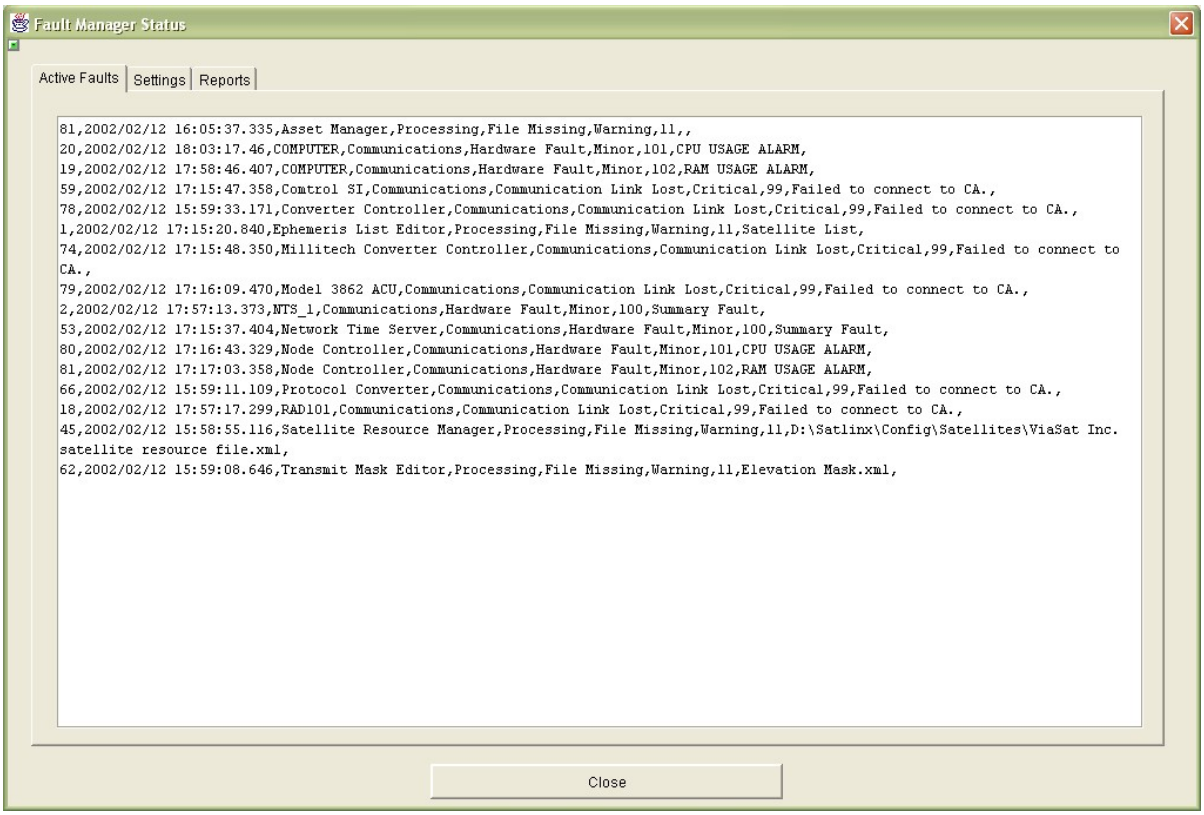
The "Details" button associated with the Event Log would cause the Event Log History screen to pop up:



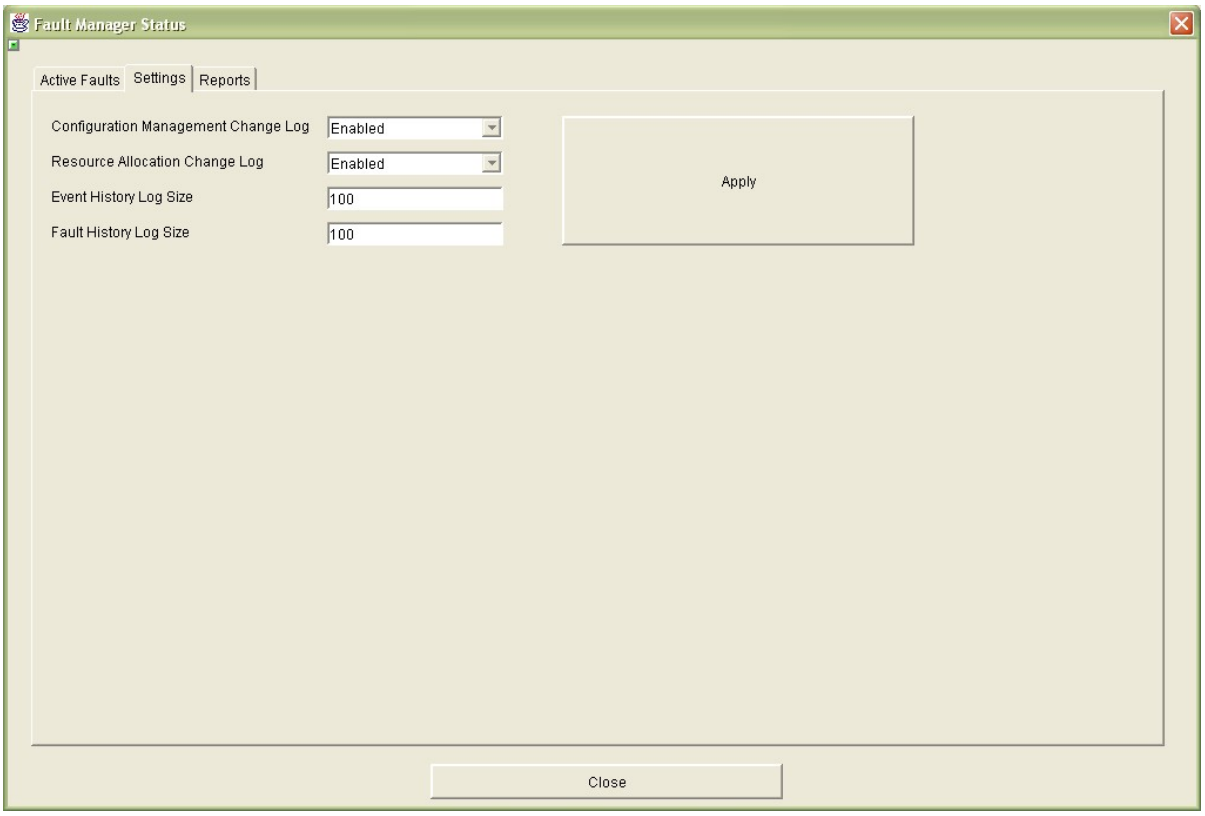
This display functions in the same way as the Fault History Log

Configuration and report generation for the Fault Manager is from the third user interface screen. This is a notebook with three tabs.

The first tab selects a display of all the currently active faults in the system. As faults are reported they are added to this list, and when they are cleared the Fault Manager removed them from the list. An example of the active fault list is shown below.



The second tab selects the configuration page for the Fault Manager, reproduced below.



Configurable parameters are as follows:

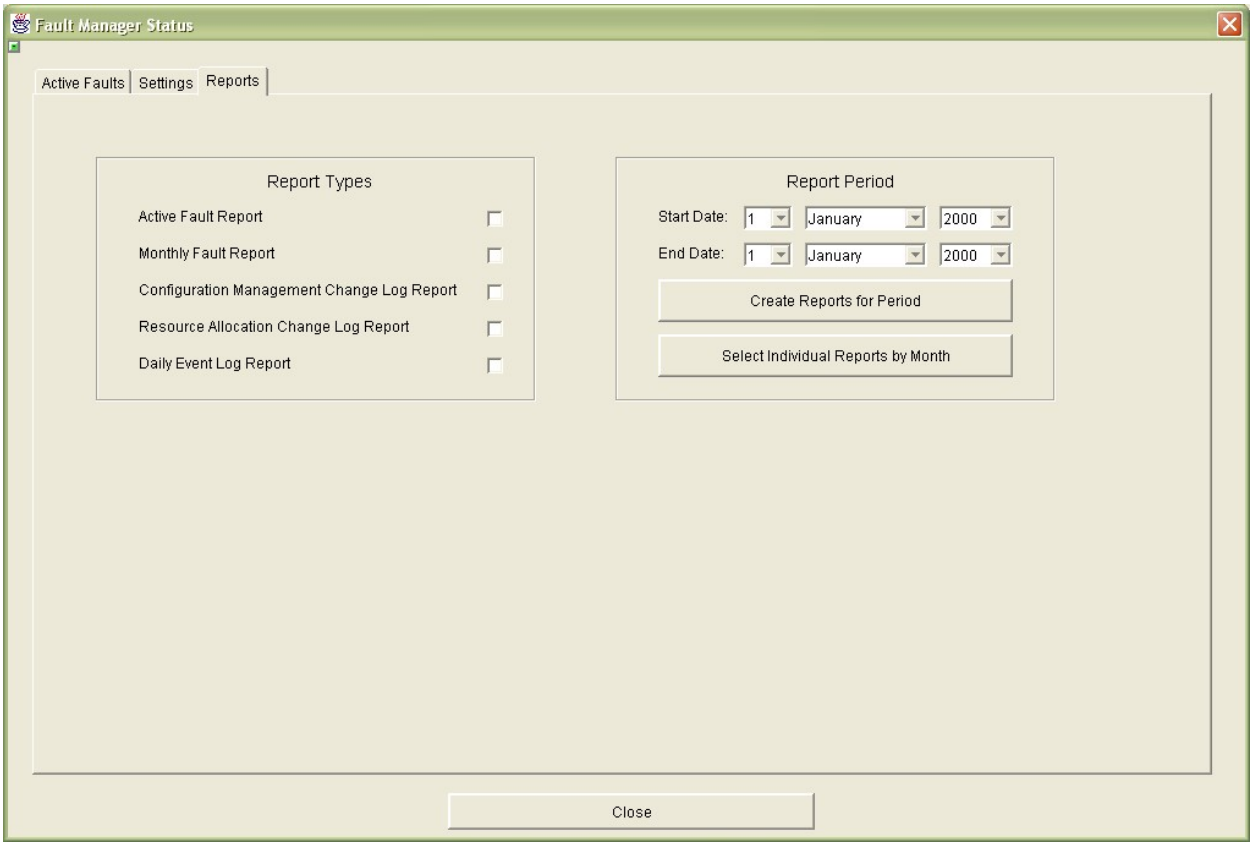
CM Change Log Enable This selects whether a Configuration Management Change Log will be created, containing this class of events. If not the CM events will only be written to the daily log.

Resource Log Enable This selects whether a Resource Allocation Change Log will be created, containing this class of events. If not the resource allocation events will only be written to the daily log.

Event History Log Size This parameter sets the size of the event history log that will be displayed on the Event History screen. After this number of records have been added to the history, additional events will cause the oldest event records to be be discarded from the history. They will still be available to review through reports.

Fault History Log Size This parameter sets the size of the fault history log that will be displayed on the Fault History screen. After this number of records have been added to the history, additional events will cause the oldest event records to be be discarded from the history. They will still be available to review through reports.

The third tab of the notebook selects the report selection and generation page, reproduced below. From it can be selected the various reports that coreespond to the different log file maintained by the Fault Managet, and the date span for the reports. These reports are generated as inputs for the ReportGenerator, which will render them as configured in the system report templates.



4.4.5 File Manager

4.4.5.1 Overview

The FileManager is used to handle almost all of the file access in the system. It is done this way so that files locations can be move without disrupting the rest of the system. It also allows the entire filestore to be moved to another machine for security or interoperability. With the exception of the 'GenericFile' operations, which will be described, each of file access methods read files from a predetermined path. The 'GenericFile' methods allow the user to specify the Enum for the specific location desired, but not give freeform paths. Filenames are specified without path. All operations are atomic and no files are held open. It has no GUI, should not be derived into anything else, and is used via its CORBA interface:

4.4.5.2 CORBA Interface

interface FileManager inherits from Sbase.

4.4.5.2.1 Aliases

ActivityIdentifierSeq

```
typedef sequence<ActivityIdentifier> ActivityIdentifierSeq;
```

4.4.5.2.2 Constants

personality

```
const string personality = "FileManager";
```

4.4.5.2.3 Enums

ePaths

```
enum ePaths
{
   eNoPath,
   eConfigPath,
   eScreenPath,
   eAdministrationPath,
   eOperationPath,
   eDefaultPath,
   eSystemPath,
   eSatellitePath,
   eMissionPath,
   eTraceLogPath,
   eReportTemplatePath,
   eReportOutputPath,
   eCalibrationPath,
   eDailyDataPath,
   eMonthlyDataPath
};
```

These are the selectors for the predetermined file paths.

eActivityType

```
enum eActivityType
{
   eNatural,
   eDirected,
   eAllActivities
};
```

For identifying activity types.

eActivityEntryType

```
enum eActivityEntryType
{
   eDataManagement,
   eHardMaintenance,
   eSoftMaintenance,
   eReserved,
   eTest,
   eTrack
};
```

Activity entry types.

4.4.5.2.4 Structs

```
struct ActivityIdentifier
{
   unsigned long      id;
   string             name;
};
```

For identifying activities by name

```
struct FMFileStats
{
   string         createtime;
   string         modtime;
   unsigned long  length;
};
```

File stats returned from the file manager.

```
struct ActivityInfo
{
   eActivityType           type;         // See above enums
   eActivityEntryType      acttype;      // "Test",
   string                  name;         // "LAN Fault Isolation Test",
   unsigned long           id;           // "25",
   string                  priority;     // "3",
   boolean                 concurrent;   // "0",
   string                  performer;    // "LAN_FAULT_ISOLATION_TEST",
   string                  profile;      // "standard",
   unsigned long           duration;     // "30"
};
```

Activity Info.

4.4.5.2.5 Operations

`ActivityIdentifierSeq GetActivityIdentifierList(in eActivityType type);`

Called to get a list of activity IDs???

`boolean GetActivityInfoByName(in string name, out ActivityInfo info);`

Gets activity info by name

```
boolean GetActivityInfoById( in unsigned long id,
                             out ActivityInfo info);
```

Gets activity info by ID.

`string GetDefaults ( in string name );`

Retrieves the contents of a defaults XML file from the preset defaults directory.

`string GetInstanceData ( in string name );`

Retireves the contents of an individual instance-data XML file from the preset instance directory.

`string GetScreens ( in string name );`

Retrieves the contents of the system screens file and the specified modules screens file from the preset screens directory. It concatenates them together into one collection of screens.

```
StringSeq GetMissionList();
```

Retrieves a list of the mission profile names from the missions directory. It is essentially a directory listing. All activities are referred to as missions even if they're just tests or static settings. If it has a profile of settings, it's called a mission.

```
string GetMission ( in string name );
```

Retrieves a specific XML profile from the missions directory

```
StringSeq GetSatelliteList();
```

Parses the "satellite list.xml" file and returns the list of satellites.

```
string GetSatellite ( in string name );
```

Reads the specific satellite file and returns the contents.

```
string GetTransmitMask ( in string name );
```

Returns the contents of the Transmit Mask file.

```
boolean PutTransmitMask ( in string name , in string mask );
```

Stores the transmit mask in the specified file

```
string GetGenericFile ( in ePaths pathenum,  in string name );
```

Reads in the contents of any kind of file, specified by name, from any one of the preset locations, specified by Enum.

```
boolean PutGenericFile ( in ePaths pathenum, in string name,
                         in string data, in boolean append );
```

Writes out any kind of file, specified by name, to any one of the preset locations, specified by enum. It allows either Append or Overwrite.

```
boolean CopyGenericFile( in ePaths srcpathenum, in string srcname ,
                         in ePaths destpathenum, in string destname );
```

Copies the contents of any kind of file, specified by name, from any one of the preset locations, specified by Enum, to any one of the preset locations, specified by enum, under a different name.

```
boolean GetGenericFileStats( in ePaths pathenum, in string name,
                             out FMFileStats stats);
```

Retrieves the filestates of any file, specified by name, from any one of the preset locations, specified by Enum.

```
string GetGenericPath( in ePaths pathenum, in string name );
```

Accepts a standard path enum and filename and returns the fully qualified path to that file. *Beware that in a distributed system, this may be useless at best and possibly disruptinve.*

```
boolean DeleteGenericFile( in ePaths pathenum, in string name );
```

Accepts a standard path enum and filename and deletes the specified file.

```
StringSeq GetGenericFileList( in ePaths pathenum, in string filter,
                              in boolean directories);
```

Accepts a standard path enum and filename wildcard filter and returns a list of matching filenames from the specified directory.

4.4.6 Logger

4.4.6.1 Overview

The Logger module is used by the entire Satlinx system to categorize and log any data the individual modules are set to send. The 'Log()' call is implemented in the 'Sbase_impl' base class. Each module is automatically attached to the Logger when started. If they fail to attach, they will queue and store any log messages until the connection is made. Application code in each module can just call its own 'Log()' functionwith the necessary parameters, which will take care of the transmission. Messages are categorized marked with a severity level by using Enums defined in the 'SBase' CORBA interface. Messages can be filtered at the source, so that messages that don't exceed the threashold are not sent. Messages can also be filtered at the destination so that messages that don't exceed the threashold are not stored. These thresholds are dynamic and are set using methods using methods in the 'SBase' CORBA interface. The Logger can be called synchronously or asynchronously, depending on threshold settings in the client. All traffic is queued internally so that even synchronous calls return immediately.

4.4.6.2 Usage

The CORBA interface of the Logger is documented here but is typically not used by the application programmer. The Logger is written to indirectly by using the 'SBase' methods and enums:

```
virtual void Log ( int LineNo, CORBA::Short Type,
                   STOVOKOR::SBase::eLevels Level, char *logString, ... );
```

This is the data logging method, and is the preferred and correct way to add debugging and other logging information to the code. (Disregard whatever you may see in existing code!)

The first parameter is the line number where the message was generated. This is a convenience and any value can be used. Typically you would use the `__LINE__` macro.

The next two parameters categorize the log call by type and severity. The types are:

```
kLogTypeNone
kLogTypeFlow
kLogTypeData
kLogTypeCondition
```

`These` are arranged as bit flags, so that a log message may be categorized as belonging to several types, and will thus be logged in all log files to which it applies. The severity levels are:

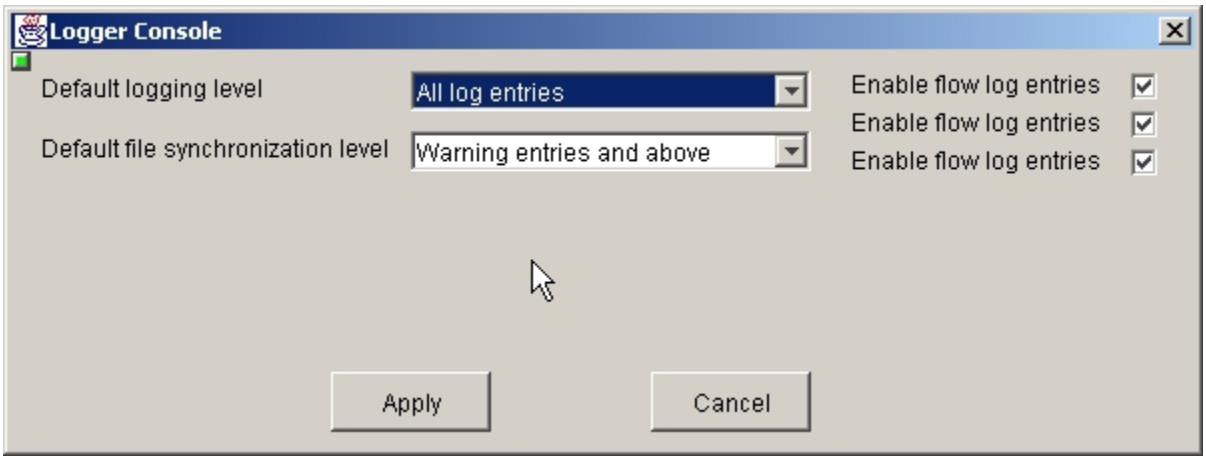
```
STOVOKOR::SBase::Failure
STOVOKOR::SBase::Error
STOVOKOR::SBase::Warning
STOVOKOR::SBase::Info
STOVOKOR::SBase::Debug
```

The rather long winded designation is due to the fact that these values are defined in the SBase IDL interface, so the full scope must be explicitly given.

The remaining parameters are standard `printf` arguments, with a format string and optional arguments.

The messages will be recorded in whatever target files are defined for the log level and category in the logger configuration. (This will be expanded with a browser/GUI interface soon.)

4.4.6.3 User Interface



This simple screen allows some basic settings to be altered.

The intent is to extend this with a dynamic page with all registered modules and their log levels. Thus each module's level can be changed from this one screen.

4.4.6.4 CORBA Interface

```
interface Logger inherits from SComponent
```

4.4.6.4.1 Constants

personality

```
const string personality = "Logger";
```

4.4.6.4.2 Operations

```
void SLog ( in short Type, in SBase::eLevels Level, in string From,
            in string logString );
```

Synchronous call from the base class of the client module. The CORBA method is declared normally and thus will complete before returning.

```
void ALog ( in short Type, in SBase::eLevels Level, in string From,
            in string logString );
```

Asynchronous call from the base class of the client module. The CORBA method is declared as 'oneway', which means the CORBA substructure will return successfully from this call immediately, regardless of whether the message was actually sent or not.

```
boolean CutTo ( in string szNewName, in boolean bClearFile );
```

A remote call to tell the Logger to cut the contents of the current log to another file. Currently unimplemented

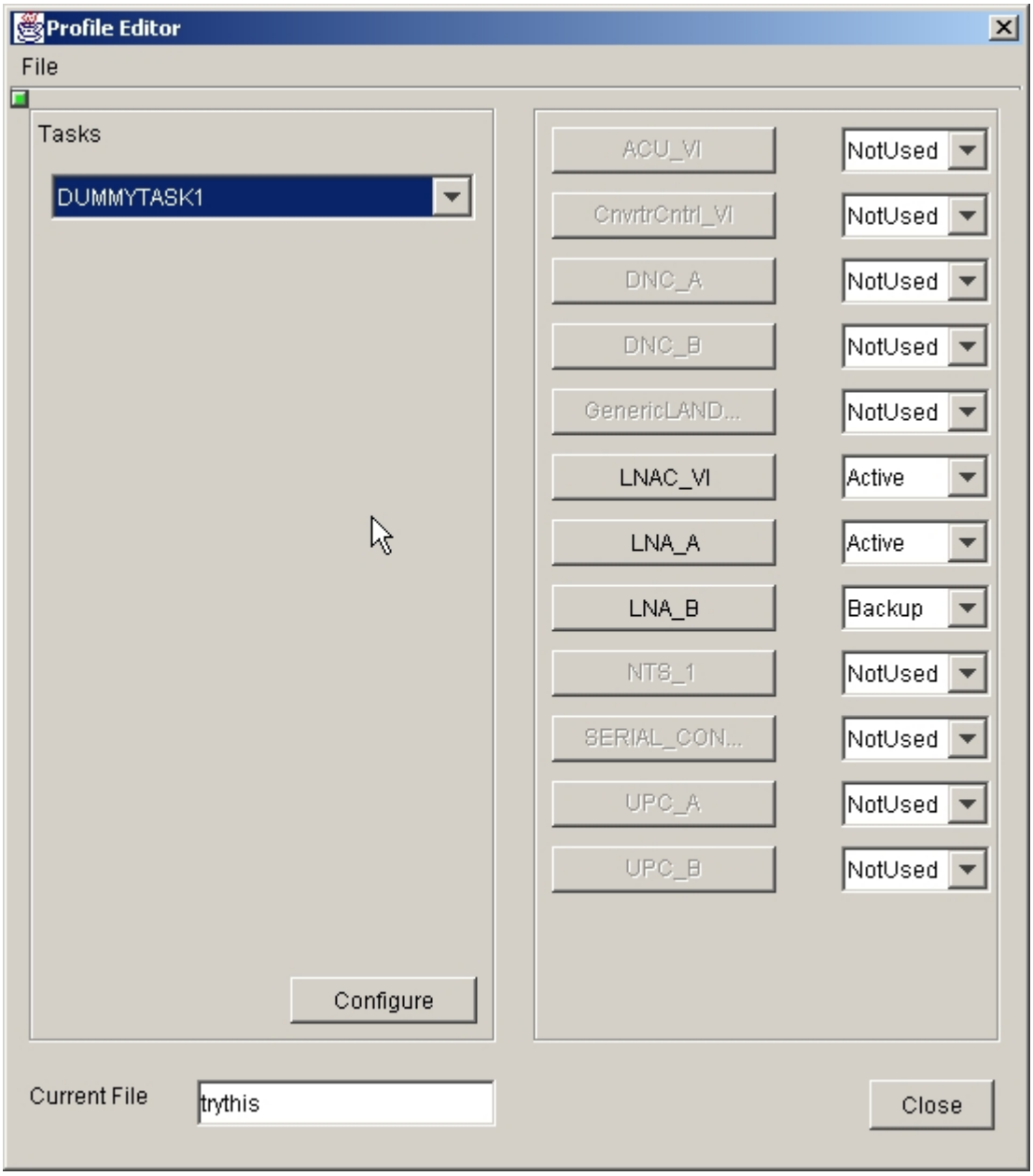
```
boolean LogInit ( inout SBase::LevelSet init );
```

Called from the base class of a client module during its initialization. It queries whether the Logger is ready for traffic and also returns the log threshold settings for the client module.

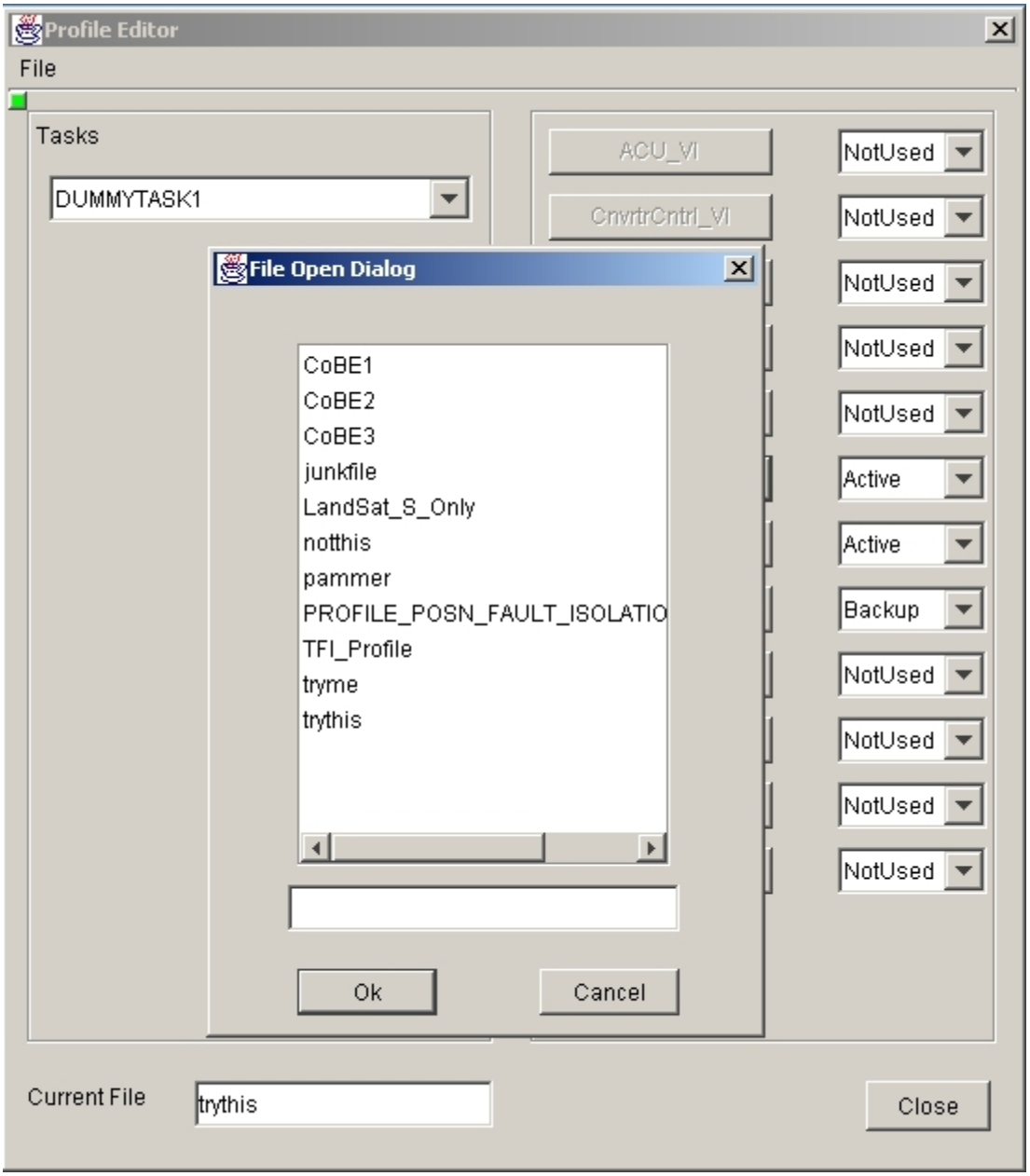
4.4.7 Profile Editor

The profile editor is used to assemble XML files that will control specific Virtual Task objects. Profiles consist of the settings for a specific VT and for the VI's that the VT will use to accomplish its work. The editor gets a list of available objects from the NodeController to populate its fields at runtime. The editor itself has no knowledge of how these objects will be used or how to set them up. Each object also has within it a subsystem designed for behavior configuration. This subsystem has it's own screen stored in the object's own screen file. It copies to itself all of the DataItems from the object that are marked as 'public'. These are the DataItems that the screen interacts with. The editor sends XML blocks to the individual configuration objects and receives back altered XML blocks that it assembles into a profile. This configuration-specific data is managed in parallel with the running object and has no effect on the current operations of the live object, it is isolated within the subsystem. No object requires custom code to provide the configuration functionality.

The editor loads the VT's into a combobox and dynamically generates a selection panel with the VI's:



This example shows the editor with the profile 'trythis' open. The file has been parsed and the active objects identified. This task, 'DUMMYTASK1', only uses 3 VI's. All VI's will be listed in the profile, but the unused ones will have their role set to 'NotUsed'. The dialog begins with the all of the controls unselected. The user must choose, from file menu, whether to create a new profile or open an existing one. None of the controls are active until this choice is made. The FileOpen and FileSave look like this:

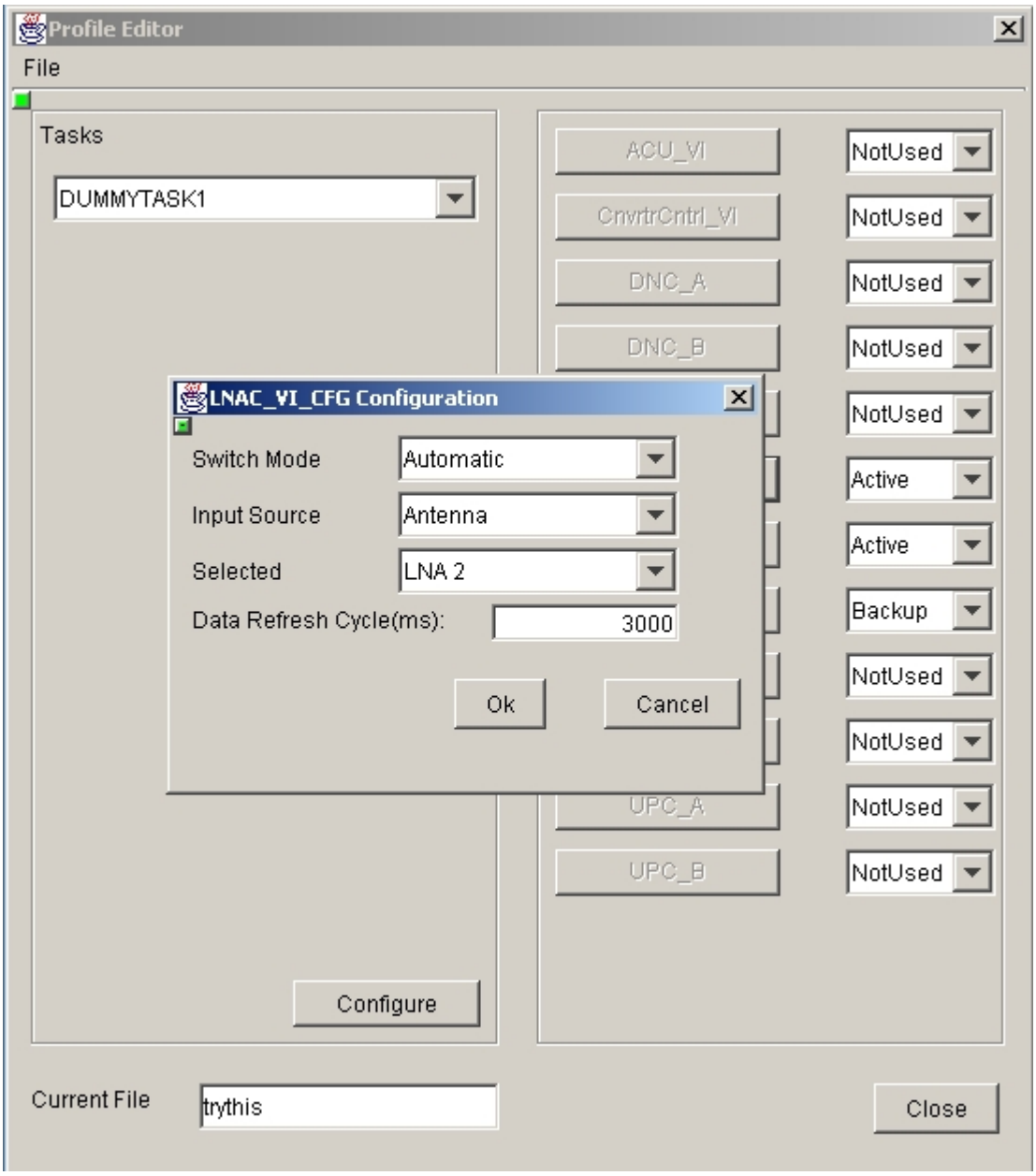


If the user elects to create a new profile, they must first select a task that will be at the heart of the profile. The task selection button is inactive at all times save for this. Once a task is chosen and the user presses the 'Configure' button in the task panel, the task selection dropdown becomes locked. The task configuration can be changed as often as desired, but the task itself is now tied to the profile. The user must choose 'New' again to choose a different task. The profile doesn't get named or committed until it is saved.

If the user elects to open an existing profile, the selection dropdown displays the task name and the editor parses the file to show what VI's are used and what their role

would be. All of the XML blocks for each of the VI's are sent down to the VI's configuration subsystem.

Pressing the pushbutton with the VI's name on it causes the VI to display it's own configuration screen, complete with the values that now reside within it's configuration subsystem:



All of the values and selections have the same range restrictions as the live object.

Once the values are chosen for all of the VIs and tasks, the profile must be saved using a dialog which operates the same as the file open dialog. When this action is chosen, the

task and all of the VI's are queried for their individual XML blocks, which are then assembled into a profile file and saved.

4.4.7.1 Configurator

The Configurator is the other half of the Profile Editor duet. It is used to apply configurations prior to a activity being initiated. It has no GUI. It has the IDL interface:

```
interface Configurator : SComponent
{
   short   loadProfile( in string profileXML );
   short   apply();
   short   gotoIdle();
   short   getStatus();
}; // end of interface
```

4.4.7.1.1 Usage

When a controlling entity like the Scheduler wishes to execute a particular task, the task is given that name of a profile to load. The task will load the XML from the profile and parse its own section, determining whether it can begin the job. If it can, it will obtain a reference to the Configurator and call 'loadProfile()' passing in the actual XML string. When 'apply()' is called, the Configurator will iterate through the individual MODULE sections, getting references to the various VI's. It will call up the configuration subsystem of each VI and pass down the XML block which the VI will validate against the allowed range for each field. If the XML passes validation, the VI's is told to execute it which copies the values from the configuration subsystem over to the live-operation side of the VI and those become the active values. The VI's role is then set. If ALL the configuration blocks are assimilated properly by the VI's, the 'apply()' method returns 'kCmdResultOK', otherwise 'kCmdResultFailure'. If 'gotoIdle()' is called on the Configurator, it will iterate through the individual MODULE sections, getting references to the various VI's and set their role to 'NotUsed'. 'getStatus()' returns the 'kCmdResultBusy' if called while the configuration is being applied or is going idle. Otherwise, it returns the success or failure of whatever that last operation had been.

4.4.7.1.2 Operations

`short   loadProfile( in string profileXML );`

Ingest the XML string and see whether it's valid

`short   apply();`

Apply the XML configuration

```
short   gotoIdle();
```

Revert all modules mentioned in the profile to their idle state.

```
short   getStatus();
```

Return the status of the last apply() or gotoIdle() command.

4.5 Infrastructure Utilities

4.5.1 CORBA Service Starter

The CORBA Service Starter is a wrapper program for starting the Orbacus services required by the Satlinx system. It accepts command line parameters specifying the service to start, and verifies that the service is not already running before it starts it.

It is normally configured as the executable in a `RUN` element in the `SYSTEM` section of the system confguration file. A typical entry for the CORBA Naming service is as follows:

```
<RUN NAME="NameService" EXECUTABLE="corbaservice.exe" ARGUMENT1="NameService"
     ARGUMENT2="nameserv.exe" ARGUMENT3="-passthrough-OAport" ARGUMENT4="15000"
     USE_CORBA_ARGS="Yes" USE_VERSION_PATH="No"/>
```

The ORB is initialized, which strips out the CORBA arguments from the command line vector. Then a command line is assembled starting with the executable name, and then the arguments in order of the numerical suffix.

There is one special form of argument, the "`-passthrough`*XXXXX*". This form allows an argument to be passed on to the command line that would normally be stripped by the ORB. In the above example the argument "`-OAport`" would be recognized by the ORB and removed from the argument vector. By prefixing the argument with "`-passthrough`" the ORB does not recognize the argument. This prefix is then stripped by the CORBA Service Starter executable before the argument is added to the command line.

Once the command line is assembled it is invoked. If it succeeds the CORBA Service Starter is replaced by the CORBA service. Otherwise it reports a failure and terminates.

4.5.2 Name Service Federator

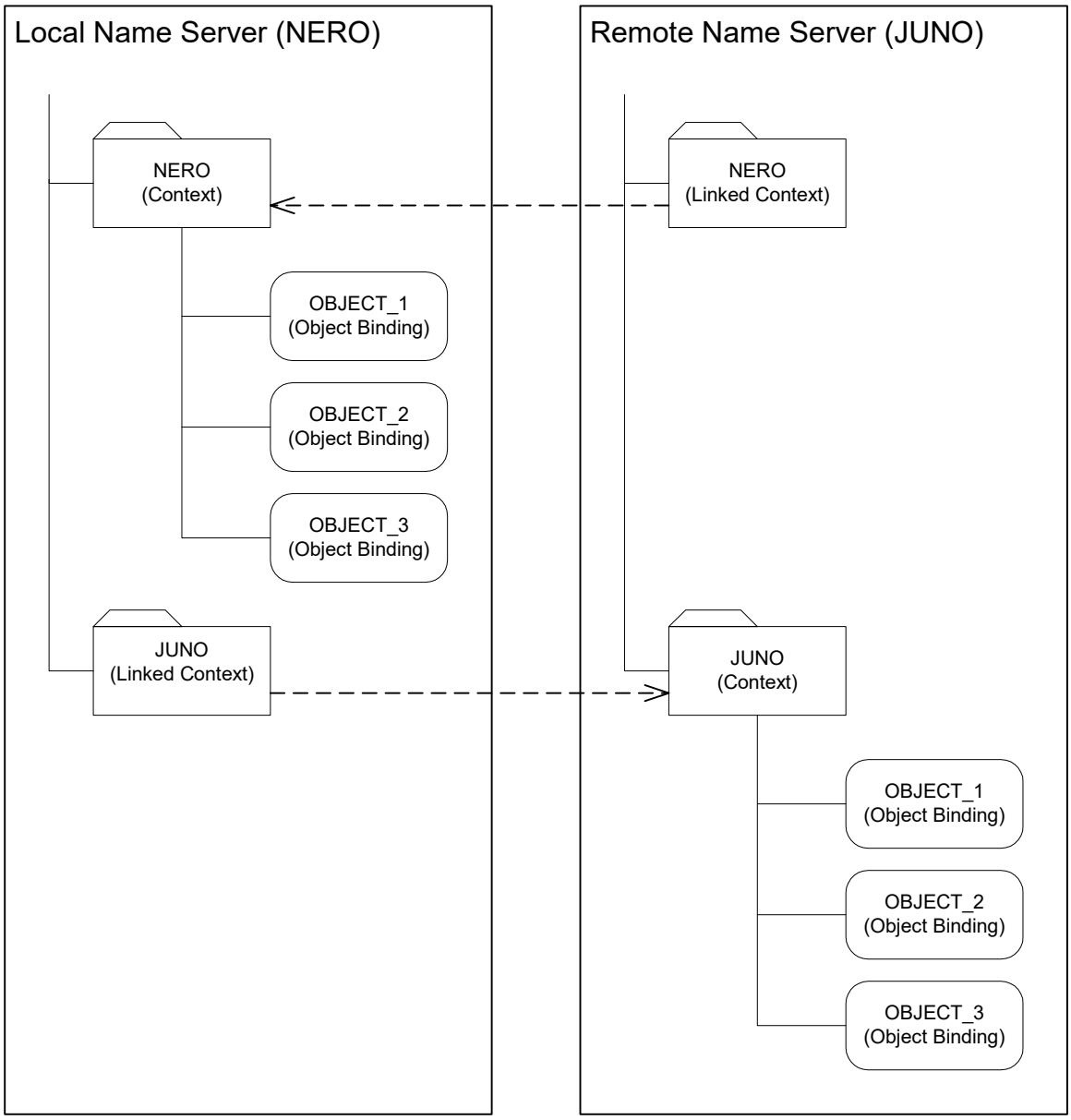
4.5.2.1 Overview

The Name Service Federator (NSFederator.exe) is an object that allows for redundant OMG Name Servers to exist on the same network. This enhances reliability, as failure of a single machine would only deny access to the failed machine and the objects residing on it, whereas the use of a single Name Server would bring the entire system down in the event that the machine hosting the Name Server were to fail. In order to understand how the Federator works, a little background on Name Servers is in order.

In all environments that use only a single Name Server, it is a single point of failure. If the Name Server fails, then all further attempts to resolve object names to get an object pointer will fail degrading system performance and likely bringing it to a halt.

To use multiple redundant Name Servers, all objects would have to bind every Name Server in the system, which would be a logistical nightmare. Considerable network resources would be consumed in keeping the contents of each Name Server synchronized with that contained in the other Name Servers. Managing the names for all of the bindings would also be an issue, since all objects across the entire system would have to be uniquely named, lest they collide. Binding the name for a new object on a Name Server where the binding already exists would replace the binding with that of the new object, effectively orphaning the existing object. Since each object across the entire system would have to be uniquely named, it would also make the task of writing generic applications more difficult, since each would have to know where it was running in order to know which object names to resolve.

Since contexts can point to other contexts stored in remote Name Servers, we can link the local Name Server to other Name Servers, without requiring the local objects to keep track of where the object is, or even being aware that object is not local. By creating a context (akin to a folder on a hard disk) on each Name Server to hold the bindings for each object local to the Name Server, as well a linked context for each node on the network, we can solve the issue with name collisions between machines. Here is a pictorial representation:



An application on NERO that wanted to gain access to "OBJECT_1" on NERO would first resolve the context "NERO", and then do a resolve on "OBJECT_1" using the

reference to the context. To access the object with the same name on JUNO it would get a reference to the "JUNO" context on the local Name Server, and then do a resolve on "OBJECT_1" using the reference to the context (which would actually be processed on the Name Server on JUNO). Thus there are no differences when attempting to access objects that are local or remote, effectively giving us location transparency. A helper function, `resolveName()` is included in the SBase base class to further simplify name resolution, and allows the use of hierarchical names for objects, such as "NERO/OBJECT_1", and "JUNO/OBJECT_1". This cleans up the code significantly and should be used wherever possible.

4.5.2.2 Usage

On startup, the Federator sends out a UDP/IP multicast datagram, in the form of an announcement message. All other Federators that hear this message will create a context in their local Name Server for the new node name, and will respond with another datagram containing their own node name (prompting every Federator that hears the responses to create/update contexts in their Name Servers in turn). In this way all of the Name Servers should create a context for each node on the network, with no intervention on the part of the users. New nodes added to the network will automatically be added as new contexts on all nodes running the Federator, and there is no need to maintain tables of machine IP addresses or perform any complicated setup on new or replacement nodes that are added into the network.

In order to support a distributed system, the user need only make sure that an entry for the NSFederator object is in the main startup XML file, and to prefix all object names with the node name for the object. The configuration XML file for this object, NSFEDERATOR.XML, contains only two lines:

```
<XML>
   <IPADDR>226.2.2.2</IPADDR>
   <NSPORT>15000</NSPORT>
</XML>
```

The IPADDR setting will default to 225.0.0.1 if not specified, and defines the multicast address which the Federator will use when sending/receiving information to/from other Federators; it must be a multicast IP address (224.0.0.1-239.255.255.255 inclusive) or the application will not work. In addition, if the system spans multiple network segments (i.e., spans one or more routers), then each router must be configured to pass multicast datagrams.

The NSPORT setting defines the port on which all Name Servers are configured to accept requests; it is assumed that all of the Name Servers in the system are configured

to use the same port address. The Federator itself is hard-coded to use port 810 for its multicast datagrams.

4.5.2.3 Caveats/Enhancements

There are a number of changes that could be made in order to increase the utility of the Federator:

- Set the port address for datagram traffic using an entry in the XML file.
- Add a timing loop to periodically send out announcements; this can increase reliability at the expense of flooding the network. Should only be considered if nodes prove unstable or the network is unreliable.
- Send the port address of the local Name Server out in the datagrams; this would allow for having different Name Server port addresses on each node on the network.
- Rewrite the sockets code to use the less efficient (read : slower) Berkeley-type socket calls for portability.

4.5.3 Report Generator

The Report Generator is called automatically by the ReportManager class and has no other interface or usage. See section 4.3.7 for more details.

4.5.4 Shutdown Initiator

The Shutdown Initiator is a small program that invokes the `Exit()` method on a Satlinx CORBA object, by default the object named "`CFGMGR`". This is the usual name for the Configuration Manager. When the Configuration Manager executes the `Exit()` method it shuts down the system. The syntax for the command is:

```
ctl <nodename> <objectname>
```

The first argument is the name of the node, or name context, the object is registered in, and the second is the name of the object.

Although it is most often used to shut down the system, the command can be use to cause any module to terminate, which can be useful for testing. Also, it is a useful template for writing other test programs to call other CORBA methods on objects.

4.5.5 Startup Service

The Startup Service is the executable that starts the whole Satlinx system. By default, it runs with no command line parameters and takes all configuration details from system files.

The input files for the startup service are:

`NODEINFO.xml`	The node information file containing paths to the data and configuration directory trees. See section 4.4.2.3.1 for more details.
`<name>.ver`	The version file, specifying the version directory tree containing the executables for the system.

The name of the startup program as built is stovokor.exe. This is just a working name and it can be changed to any desired name. The version file name and the system configuration file name are based on this name, as follows:

	Executable name	Version file name	Configuration file name
Rule	./<*name*>.exe	./<*name*>.ver	<*config*>/Operations/Equipment Defaults/<*name*>.xml

This allows multiple configurations to reside in the same tree. With multiple copies of the startup program, each named uniquely, each configuration can be invoked by running the required executable. In a customer installation this ability has limited value, but in a development context it can be very useful and convenient.

4.5.5.1 Version File

The version file contains the relative path to the version specific executables. An example of its contents follows:

```
../1.0
```

The executable tree is described in detail in section 2.3.2.2. This file provides a path from the directory containing the startup executable and the version file to the version directory.

4.5.5.2 Command Line Arguments

The optional command line parameters enable the startup service to override the configuration in the files, as well as to create the configuration files in installation mode.

The full syntax for the command line parameters follows:

`stovokor [vpath [--install cfgpath datapath]]`

`vpath`	The relative path to the version specific executables. It has exactly the same format as the contents of the version file.
`configpath`	The absolute path to the root of the configuration directory tree described in section 2.3.2.3.
`datapath`	The absolute path to the root of the data directory tree described in section 2.3.2.4.

The "`--install`" flag runs the startup service in installation mode. In this mode the version file and the node information file are created from the command line parameters, and the Configuration Manager DLL is loaded and run in installation mode.

- At present, the installation mode hard codes the node name to NEWNODE in the node information file. It may be desirable to add an additional, perhaps optional, argument to set the node name in the node information file. This would allow the system to be fully configured for skeletal operation after the installation process is completed.

5 The GUI, the SCI, and the outside world.

5.1 Overview

The GUI interface was designed to be run remotely and to allow for multiple simultaneous connections. This required that there be a standard interface and that the GUI be decoupled from the back-end operations. As such, the back-end code must be written with the understanding that it is not necessarily interacting with one and only one GUI. Data interchange to and from the GUI takes place in batches, and all data updates out to the GUI are intended to be broadcast to all of them for each to display as it sees fit. Different users may be viewing different screens with different representations of the same data. In all but very specific circumstances, the back-end must concentrate on content but NOT on presentation. The screens will have been independently specified to control their own presentation. This separation also required that security and Point of Control arbitration be built into the communications subsystem.

The GUI is designed with multiple parallel connections to individual back-end objects. It is designed to allow the creation of a vertical hierarchy of windows that are modal. It allows the creation of a top window and child dialogs (screens). Each screen has one connection to each back-end object for which it has controls or fields. Multiple screens connecting to the same back-end objects each have their own connection. Each connection object has a list of controls or field that it services. None of these have any knowledge of any of the others. The GUI engine itself has no knowledge of what the controls are. They are Java Beans that adhere to a proprietary interface and are specified by name in the screen definition XML. This means that new types of controls can be added to the system without having to alter either the GUI engine or the back-end code. Just provide the new Bean and change the name in the screen definition XML. With the exception of the object naming and placement attributes, all of the instructions for each one are interpreted by the bean itself with little or no help from the engine. This allows for near total flexibility in object control. All beans are (or should be) able to restrict the values of data entered to a specified range or list of discrete values. There are two types of bean operation, data-entry and data-selection. Either of these may be used for display only. Data-entry types operate like an entry field, while data-selection types operate like a listbox or combobox. Data-entry types accept a specification of what the datatype is and what the acceptable range of values are, and will not accept anything outside that range. Data-selection types have a programmable list of selectable values which is changeable under program control. The format and syntax of the screen XML will be described in later in the chapter.

There are two methods of driving the GUI, the low level interface, and the high level DataItem interface. They CAN be used together, but there are specific issues that will be discussed later in the chapter.

The Engine can also be sent URL's to be passed over to Internet Explorer for remote display of HTML pages generated or resident on the server.

5.2 Screen XML Syntax

All of the screen definitions for an object reside in a single file which is specified in the object's default or instance XML. The format is to surround each window or dialog definition with a Screen wrapper of the form:

```
<XML>
<SCREEN>
   dialog1 stuff with name
</SCREEN>
<SCREEN>
   dialog2 stuff with name
</SCREEN>
</XML>
```

All of the dialogs and windows specified in the screens file will be automatically registered for the user. The object names are important, as that is how the object will be referenced. Target specifications should use the '##' tag so that they are replaced automatically with the current object name.

Any string can be put in the XML bracketed by '%'s. The user would then use addReplacement("key", "newString") to register replacement information. The edit is done at the time the XML is requested by the Engine and it is done in a copy. This is done so that the key-string is preserved in the source XML and the user can change the registered value on the fly, each time, and the screen will reflect the new value on the next getUIDef().

The complete XML for the development node controller GUI looks as follows:

```
<XML>
   <SCREEN>
      <DISPLAY name='LOGIN' title='Login to ##' x='462' y='351' cx='200' cy='130'
               level='0'>
         <LABEL title='Username' x='5' y='5' cx='80' cy='20'/>
         <CONTROL target='##' name='A-username' control='SEditField' x='95' y='5'
                  cx='85' cy='20'/>
         <LABEL title='Password' x='5' y='30' cx='80' cy='20'/>
         <CONTROL target='##' name='B-password' control='SPassword' x='95' y='30'
                  cx='85' cy='20'/>
         <BUTTON title='OK' x='5' y='60' cx='80' cy='30' default='true'/>
         <BUTTON title='Cancel' x='100' y='60' cx='80' cy='30'/>
      </DISPLAY>
   </SCREEN>
   <SCREEN>
      <DISPLAY title='## - %User%' name='TOPLEVEL' x='200' y='200' cx='1024' cy='768'
               icon='duke2.gif'>
         <PANEL x='20' y='20' cx='532' cy='75'>
            <BUTTON title='Exit' x='20' y='20' cx='100' cy='35'/>
            <CONTROL target='##' name='SYSHALT' text='HALT!'
                     control='SPushButton' fgcolor='black' bgcolor='red' x='140'
                     y='20' cx='100' cy='35'/>
            <CONTROL target='##' name='SHUTDOWN' text='Shutdown'
                     control='SPushButton' bgcolor='yellow' fgcolor='black' x='260'
                     y='20' cx='100' cy='35'/>
            <CONTROL target='##' name='RELOAD' text='Reload XML'
                     control='SPushButton' fgcolor='black' bgcolor='green' x='380'
                     y='20' cx='100' cy='35'/>
         </PANEL>
         <PANEL x='572' y='20' cx='432' cy='75'>
            <LABEL title='System Time' font='14' x='20' y='12' cx='0' cy='0'/>
            <CONTROL target='SYSCLOCK' name='Time' control='SEditField' x='120' y='10'
                     cx='100' cy='25' readonly='true'/>
         </PANEL>
         <NOTEBOOK x='20' y='115' cx='984' cy='608'>
            <PAGE title='An Apology Page'>
               <LABEL title='An Apology Page' x='10' y='10' cx='150' cy='20'/>
            </PAGE>
            <PAGE title='Summary'>
               <LABEL title='Summary Page' x='10' y='10' cx='150' cy='20'/>
            </PAGE>
            <REMOTEPAGE Target='##' name='MODULES'/>
            <PAGE title='Topology'>
                    <LABEL title='Topology Page' x='10' y='10' cx='150' cy='20'/>
                </PAGE>
            </NOTEBOOK>
       </DISPLAY>
    </SCREEN>
    <SCREEN>  → if the remote page is specified in the file, it needs a unique name
       <PAGE title='Topology' name='topologypage'>
          <LABEL title='Topology Page' x='10' y='10' cx='150' cy='20'/>
       </PAGE>
   </SCREEN>
 </XML>
```

Notice that this contains both the login screen and the main screen. This is for example use only. The login screens are only used by the NodeController.

5.3 Engine Operations

The top level, referred to as level-0, will be automatically created as a frame window. All child windows will be created as dialogs. Some elements are directly connected to back-end control objects. All windows and dialogs are named and that name is how the object will be registered, so it must be a static name that other code and controls will know about.

Locally created and managed objects are:

Windows and Dialogs – called DISPLAY

- Panels
- Labels
- Buttons
- Notebooks

The syntax for creating displays is thus:

```
<XML>
    <DISPLAY name='##' title='%whose% SysClock' x='100' y='100' cx='300'
             cy='200' [bgcolor='gray' fgcolor='red' icon='bob.jpg' level='2']>
…
    </DISPLAY>
</XML>
```

The WINDOW keyword will cause a new window to be created and shown. It is modal and must dismissed before antecedent windows can be accessed again. The name is the object name and must be unique among windows. If this window instruction is sent again, identified by name, then the existing one will be depopulated and repopulated with the new instructions. It is not destroyed in the process because this would orphan the next any other window in the hierarchy that uses this one as its parent. The title is what will be shown in the titlebar. The '##' is a reserved tag and will be replaced ONCE with the back end objects name. It is valid to name the primary dialog for an object after itself. The %xx% notation allows the developer to replace this with anything else. The replacement is registered on command and done when the XML is requested by the engine. This allows the developer to change the requested edit when desired and it will show the next time the dialog is displayed. Helper functions are provided. The x, y, cx, and cy specify window placement and sizing. Other parameters are optional. If they are understood, they will be obeyed. Otherwise they are simply discarded with no notice. All of the optional parameters are passed to the Java object responsible for the commanded action, so enhancing that one object to allow more parameters will not

destabilize the system. In this case the optional parameters that are understood by windows and dialogs are foreground and background color. They root-level window also understands 'icon' and must be passed a jpg, gif, or bmp file. Ico files do not work. The level specifies the required permission level for this screen. No restrictions are made on the screen itself but all fields on this screen which don't explicitly specify their own requirement inherit this one. This is the minimum security level the user must possess before being allowed to update any fields. The security levels are Login – only login screens use this, Observer – can look but not touch, Operator – can update all standard fields, Administrator – can update operational but not system setup fields, Integrator – can change ANYTHING. All windows and dialogs have a default panel upon which elements are placed.

Menus and menu items are created using the following syntax:

```
<MENUBAR>
   <MENU title='File'>
      <ITEM title='Dialog' Target='SYSCLOCK|Primary'/>
      <ITEM title='Command' Target='SYSCLOCK|doit:action=push'/>
      <ITEM title='Exit' Target='exit'/>
   </MENU>
</MENUBAR>
```

All menus must be on a menubar and all menuitems must be on a menu. The title is the text to be displayed. The target is the action to be taken. In the first case this tells the Engine to retrieve the XML named 'Primary' from the target SYSCLOCK. In the second case this tells the Engine to send the specific command to the target object. In the third case it is using a reserved word 'exit'. Only these parameters are used with menus.

Other panels are created using:

```
<PANEL x='0' y='0' cx='0' cy='0' [fgcolor='green' bgcolor='cyan']>
```

This will create a bordered panel bounded by x,y,cx,cy and with the specified colors. Its location is relative to the window on which is placed. It must be terminated by a </PANEL> after all of the elements which are to be placed on the panel are specified. All of those elements are positioned relative to the panel. The colors are optional

Labels are placed using the following

```
<LABEL title='current time' x='10' y='10' cx='70' cy='25' font='14' [fgcolor='red'
icon='smilie.jpg'] />
```

The object is unnamed and the title is the text to be shown, size and position are relative to the panel on which they are placed. They inherit their colors from that panel, but the text color can be changed with the fgcolor command. The font size can be specified as above. The label can also be used to place an icon on the screen by specifying an image file to use. Supported types are jpeg and gif. Text and image are mutually exclusive.

Notice the specification is terminated with a '/'. Labels can not contain any other elements, and so are specified in their entirety.

Buttons are specified in two different manners. The first are local to the engine and are specified thus:

```
<BUTTON name='Apply' title='SaveThis' x='100' y='75' cx='50' cy='25'
        [fgcolor='green' bgcolor='cyan' font='14' default='true']/>
```

The title specifies the button text and can be any text, while the name must be on of the following reserved words:

Ok	Gather and send all data and dismiss the window
Apply	Gather and send all data, but don't dismiss the window
Cancel	Discard all data and dismiss the dialog
Close	Discard all data and close the window (may also include '=name') If the name is present it will verify that this is the top window before closing it.
Restore	Reset all fields to accept data and request a data refresh
Reload	Cause the whole back-end to reload all of its XML screen definitions and redisplay the current window with the new specification
Revert	Can be used on a button, but is really an internal command to pop all windows off the stack back to the top level.
Exit	Close GUI – NO MATTER WHERE

Data management issues will be discussed further on. Positions are relative to the parent panel, colors and font are optional and default will cause this to be the default button, which is actuated by the enter key, regardless of focus.

Notebooks are complex controls made up of immediately specified pages and remotely specified pages, which are deferred momentarily. Each page is to be treated as if it were a separate dialog.

```
<NOTEBOOK x=’20’ y=’115’ cx=’984’ cy=’608’>
    <PAGE title='An Apology Page'>
        <LABEL title='An Apology Page' x=’10’ y=’10’ cx=’150’ cy=’20’/>
    </PAGE>
        <PAGE title='Summary'>
            <LABEL title='Summary Page' x=’10’ y=’10’ cx=’150’ cy=’20’/>
        </PAGE>
        <REMOTEPAGE Target='##' name='MODULES'/>
        <PAGE title='Topology'>
            <LABEL title='Topology Page' x=’10’ y=’10’ cx=’150’ cy=’20’/>
        </PAGE>
    </PAGE>
</NOTEBOOK>
```

The notebook is only specified with size and placement, which is relative to the window on which it is placed. Each page is simply given the title to its tab. All contents on the page are specified identically to other panels. RemotePages specify a back-end object to query and the name of the dialog to request. The dialog is retrieved and placed in the notebook at the specified location. The remote page can be dynamically generated (which is why it's delayed) or it can be in screen file because it's dynamically requested, in which case it also needs a unique name for registration in the map. In the latter case, the notebook wouldn't have the remotepage specifier in it, this would be generated dynamically with the list of pages to be loaded. The Engine will then go load the page into the most recently created notebook (!). If you're going to have more than 1 notebook per dialog, make sure the one with the dynamic pages is the last one created. Color and font are optional for the pages.

Objects with connections:

The Engine allows for two different types of connections to back-end objects. There are the graphic objects directly connected to back-end objects, referred to as Controls, and objects called "Connections" which are attachments to the same event channel as a Control would be but do not have an attached graphic. The graphically oriented controls will receive all messages, but only the first connection to each back-end object will execute commands while all of them will accept data messages. Connections can be instantiated for any existing back-end object, even when that object might never have a dialog to display. Controls are written as JavaBeans, and thus are very flexible and can be added to the system without alteration to the Engine. They are called by name and instantiated dynamically.

Controls are created thus:

```
<CONTROL target='##' name='Time' data='Time' control='SEditField'
         x=’100’ y=’10’ cx=’100’ cy=’25’/>
```

Target, name, data, control, x, y, cx, cy are the only required parameters. Target is the name of the back-end object to connect to. It must be a name registered in the

nameserver and must resolve to an SControl type reference. Name is the object name and must be unique on this dialog, but not necessarily the whole system. 'data' is the name of the data item in the back-end object that is registered for automated processing. Control is the name of the Bean to be instantiated. The Engine does not know about these in advance. The placement is relative. All other parameters are passed to the bean for interpretation, so we'll have to discuss each Bean individually. All optional parameters will follow the same syntax across all Beans.

Connection objects reside solely within the engine and are used to convey pushed commands to the engine when no graphics may be present. They can be created on the fly, but exist for the duration of the session. Connections are intended to stay active for the duration of the operation, while Controls only exist while they are being used. Connection objects are primarily intended to be used by objects that may need to command the GUI to do something, such as popup a warning box, but may not have a dialog open at the time. The Fault Manager is a good example. It has no dialogs at all, but may need to warn the user of critical faults. The syntax for a Connection is:

```
<XML>
    <CONNECTION Target='FAULT_MGR'/>
</XML>
```

This is the total startup for the system. It creates a Connection to the node controller. The Engine holds an authentication key that is passed with every interchange. In this case the NodeController determines that the current key, being uninitialized, is invalid, so its first action is to push a login dialog to whoever is listening… our Connection. This how the interplay with the system begins. The authentication keys expire periodically and the engine and whichever object is connected to next will execute the same login process.

The GUI engine receives data updates asynchronously while the user is doing whatever it is that users do. To prevent any field from updating while the user is trying to change its value, when the user clicks in an entryfield or selects a value in a selection type field, that field is locked and will no longer accept updates from the back end. Leaving an entryfield without entering any data will unlock the field or clicking a 'Restore' button will unlock all of them. A successful update of the data in the back-end will unlock the fields for further activity.

5.4 Data Interchange Basics

By convention, all function prototypes described in these sections are paraphrased. The spelling will be correct, but the parameters listed will usually be limited to those germane to the discussion. There are a few exceptions where a full and complete

prototype is given, but these are obvious. The full prototypes are otherwise described in detail in their own chapters.

For all data elements which will be connected to the Engine, an 'addFunction()' call is used to register a handler function for each element. The handler function must be a static member function of the following prototype:

```
static bool SCI_handler( SControl_impl *obj, const char* value, const char *source );
```

The same function can be used for any and all of them if desired. The last parameter contains the name of the data element from which the data was extracted, which can be used in processing. The first parameter provides a pointer to the actual object. We must use a static function because C++ won't let us extract the address of a member function within a class instance. We provide the static function with a pointer to the object so that it can modify member variables or call member functions, both using a "static_cast<classtype *>". The second parameter provides the value from the GUI, in ASCIIZ string form.

The GUI begins the process by being given the name of a target object and a screen name to request from that object. It makes the requested CORBA connection, and then calls 'ECRegister()' which authenticates the GUI's key. Upon success, this GUI is registered to accept asynchronous updates from the back-end and 'ECRegistered()' is called. This is a virtual placeholder function which the programmer is meant to override if he/she needs to track connections (in addition to the system's own tracking). The GUI then calls 'getUIDef()' on object with the name of the screen to be fetched. This call first (after authentication) calls 'prepUIDef()'. This is a virtual placeholder function which the programmer is meant to override if he/she needs to generate or modify the screen XML before it is supplied to the GUI. The '%x%' replacements are done next and then the XML string is returned to the GUI. The GUI next calls 'UIRefresh(true)' on each back-end object to cause the object to send relevant data to the GUI. The parameter 'true' means this is the first for this screen, so the back-end object is expected the send out the range control information before the data. The GUI will call 'UIRefresh(false)' after each time it sends data down to the back-end so that the back-end can refresh whatever new values should now be reflected.

The GUI sends data to the back-end objects by calling 'UIActivate(string)' with a string containing all necessary data from the topmost screen aggregated together. This function will disassemble the aggregated strings into its component parts and make the individual data updates via the registered functions. If any of those updates fail, the function can return 'false' which will cause the GUI to consider that field 'not updated' and it will not get unlocked on completion. After all field updates are actuated (successfully or not), 'UIActivated()' is called. This is a virtual placeholder function

which the programmer can override if they need to do further data validation. If several data elements are interrelated, then the individual update function may or may not return 'false' on failure while this function can return 'false' to signify the failure of the entire process. If the instigator (the 'culprit') of the update was an 'Ok' button, the GUI's mode of operation would be to dismiss the dialog if all went well or retain it if anything failed. If the 'culprit' was an Apply button, the GUI's mode of operation would be to retain the dialog, unlock the fields for which the update was successful, and.request a data update by calling 'UIRefresh(false)';

5.5 The SCI Protocol

The SCI protocol is a text-based protocol primarily used for communications with the GUI engine, although it can be used for any other module as long as the proper conventions are used. It rather prosaically stands for SControl Interface, as SControl is the name of the code module that handles it.

The protocol string begins with a key value and groups of ID, tag and value sets. The protocol is two-way and contains the same elements, although the meaning of the key is slightly depending on the direction.

From the back-end to the GUI:

```
"key:0; namefield:value=none|bgcolor=red;agefield:value=99|bgcolor=red;"
```

-or-

From the GUI to the back-end:

```
"key:2376482736482;culprit=Apply;namefield:value=bob;agefield:value=32;"
```

The key value is typically generated by the(an) Authentication process, although there are two special purpose values. The GUI process should begin by using a key of '0' for the first query. This will trigger a login screen to be sent. The GUI process will automatically go to using a key of '1' for dealing with the login screen. The '0' key will ALWAYS elicit a login screen and the key of '1' can ONLY be used for dealing with the login screen. The back-end process has a choice of keys to use for outbound messages. It can use a '0' key to broadcast the message to all attached GUI's, or it can use a specific key which will only be processed by the GUI which has specifically received that key. After the NodeController has finished authenticating the user, a key is generated for this user and is sent to that specific GUI by a direct call with an identification tag of 'keyset'. The IP address of the source-machine for every CORBA call is accessible to the recipient. This is used internally during the validation process. The generated key has

the IP address of the GUI process embedded in it. The embedded value is compared to that of the actual caller to ensure that security keys can't be spoofed. Any failure in this process is logged, including the address of the offending machine. Keys are inserted automatically on both ends of the conversation and the user/programmer typically shouldn't worry about this unless specifically trying to control the targeting of a particular string.

The next series of entries following the key are data and control information. They consist of a field/dataitem name, followed by a colon ':', and a variable number of controls per field. They are formatted as 'control=value', ie. 'value=bob'. More than 1 per field can be specified, although they must be unique and must be delimited by a bar '|' and terminated by a semicolon ';'. In the first example, the back-end process has instructed the 'namefield' field to display the string 'none' and set its background color to red since 'none' is an invalid name. The same is done with the 'agefield'.

The second example shows the input from the GUI after the user has entered the required data. It shows a special entry which is processed internally, the 'culprit'. This entry contains the name of the button or control that caused the data to be sent. Most simple screens will have one 'Apply' or 'Ok' button, but it is possible to have several of these button types with different names which may be processed differently by the back-end code. The GUI thus informs the back-end code as to which control caused the action to be taken. The rest of the string follows the established pattern, although the GUI will only send one entry per field, the value. The GUI only sends values that have been changed by the user, unless the field is specifically setup otherwise.

Data is sent to the Engine as aggregated strings of tags and values. Commands can be sent interleaved with data. Ranges are created in their own subsystem and are sent separately from the data.

5.6 The Low Level Interface

The low-level interface allows the direct manipulation of the Range and Data strings to allow for direct and detailed manipulation of the GUI. The API for this interface is fully documented in section 4.1.3.

The 'UIRefresh(bool first)' call, which is used called by the GUI to have data sent to it, should be overridden (in the current scenario… this may change). If the parameter 'first' is true, the ranges should be assembled and the 'postRangeString()' call should be made here before the data is processed and sent. This will set the range restrictions and populate the selection type controls when the screen is first drawn.

All of the following should be done with the overriding 'UIRefresh()'.

Range strings can be registered for each data element using the 'addRange(target, string[, permLevel])' or 'addList(target, string[, permLevel])' calls. The 'permLever' is a optional parameter. It allows the back-end to set a higher authority requirement on an individual field than it was already set for. It defaults to 'Operator' level if not set. The 'postRangeString()' is used to send the whole aggregated string up to the GUI when it is completed.

It is assumed that, even in a multithreaded object, only one thread will be assembling and posting ranges to the GUI, since it is a specific function. Therefore no 'ThreadSafe' techniques have been used for ranges. Data strings, however, might be assembled by multiple reactive threads. As such, there are protected.

Data strings are assembled using 'addStringTag()' and various 'addStringData()' overloads. Each piece of data or command in the aggregated string can be targeted to a different GUI or broadcast. The system handles the automatic insertion of keys into the string when the programmer uses this feature. The 'addStringTag()' call sets the name of the data field into the buffer and sets a mutex semaphore. This prevents any other thread from appending data strings to the main buffer until this transaction is complete. The 'addStringTag()' also has a additional parameter which is the key of the specific GUI that this piece of data can be targeted to, if used. If it isn't used, it defaults to '0' and broadcasts to all GUI's. The 'addStringData()' calls add data messages to the string that has already been tagged with GUI data control that will handle it. There are various overloads for all the basic data types. There is an addition parameter with a default value of 'true' which will cause the semaphore to be released. If the programmer is sending compound instructions to any GUI control, this parameter should be used and set to false on all but the last one ***for that control***.

'addActionTag()' can be used to add commands to the string which are meant directly for the GUI engine itself to interpret. These would be programmatic button pushes. The action string can be the URL for an HTML page served up by a web server, in which case it will be passed over to Internet Explorer for display.

'addPopupTag()' can be used to add commands to the string to tell the GUI to request and display a new dialog on top of the current one.

'postEventString()' will send the completed aggregation up to the GUI and reset the subsystem for the next use.

5.7 The DataItem Interface

The Data Items described in section 4.3.3 are a linkage between the XML data files, the back-end code, and the GUI. If the XML fields, the data items and the screen fields (in screen XML) are each named the same, then the data management becomes nearly (and in some case completely) automatic.

If using DataItems, 'UIRefresh()' should NOT be overridden except in specific circumstances where the programmer has issues where direct control of the GUI is required and he/she is adequately knowledgeable of the interplay. If the function MUST be overridden to provide the required functionality, be sure to call the 'SControl::UIRefresh()' within it unless you REALLY (REALLY) know what your doing!

Data Items, (class dataItem) are created on the heap in the object's constructor using 'new'. This adds them to the parameter map. For scalar dataItems, the most permissive ranges can be added here. For list dataItems, the total number of possible selections should be added here. During object registration, the default and instance XML. These can contain more restrictive and fewer list choices, but cannot open up existing ranges or add new list items. The dataItems have built in range processing and will automatically do validation of any data that the GUI or the user tries to insert into them.

The 'addFunction()' call is still used, but now there is a convenience function of this name within the dataItem which accepts only the function pointer and uses the dataItem's name to register it. This helps prevent the names getting out of sync during the development process when they may be in flux. The SCI functions are used the same but there is one built-in, intended for use with dataItems where no custom validation or action is required. It is called SCI_UpdateParameter. If the programmer simply uses this function pointer in the addFunction call, all data updates from the GUI to that dataItem become completely automatic and no further action need be taken.

The built in 'UIRefresh()' will handle assembling the range and data strings automatically by iterating through all the dataItems, which each have the ability to produce their own segment of the string. If the programmer needs to mix high and low-level operations, he/she can override 'UIRefresh()' to add their own functionality. The rules described in the section 'The Low Level Interface' still hold true except that 'postRangeString() and postEventString() should not be done. Rather, the 'SControl::UIRefresh(bool first)' should be called as the last operation, passing to it the parameters the were passed into the override.

5.8 The Beans

The current crop of Control Beans follows. The notation given is an example and does not express all of the settings for each option. Entries shown in [ ] are optional in the XML and can also be sent in the datastream.

5.8.1 SAnnunciator:

This is a state indicator. It is a multistate text and color display control. It has no data entry or sense capability and is for display only. It accepts color, font, and list. The states are specified 'list=Ok+green, Warning+yellow, Bad+red' or 'list=0!Ok+green,1!Warning+yellow,2!Bad+red' as an example. The indicator strings consist of the text to be shown and the background color it's to be shown on. The first form will populate with whatever the choices are and those are the string sent to show a state. The second form will populate the control with the second string in each pair while keying off the first.

```
<CONTROL Target='##' name='States' data='States' control=' SAnnunciator ' x='100' y='10'
cx='100' cy='25' [list='0!Ok+green,1!Warning+yellow,2!Bad+red' font='14']/>
```

5.8.2 SBar:

This is a signal strength display bar. It is for data display only. It accepts color, font, and range. The range is specified as a span 'min!max'. The tag command directs it to show the value as text within the bar.

```
<CONTROL Target='##' name='Sig' data='Time' control='SBar' x='100' y='10' cx='100' cy='25'
[bgcolor='gray' fgcolor='green' font='14' range='0!50' tag='true']/>
```

5.8.3 SCheck:

This is a checkbox control. It accepts color and font, as well as readonly and range. The range is specified as a discrete picklist, called 'list', that is used to populate to control. It is specified as 'list=Off,On' or 'list=0!Off,1!On'. The first form will populate with whatever the choices are and those are what is returned on query. The second form will populate the control with the second string in each pair while keying off the first. The first strings are used for item selection and are what is returned when gathered. Alternately the control can be set to 'Active' mode, where the selection is sent immediately and will not be gathered. The 'always' flag makes sure that it's always sent. Level specifies the minimum privilege level in order for the user to access this field.

```
<CONTROL Target='##' name='Checker' data='setOn' control='SChecker' x='100' y='110' cx='100'
cy='25' [fgcolor='black' bgcolor='yellow' font='14' list='Off,On' readonly='true'
active='true' level='2']/>
```

5.8.4 SChooser:

This is a drop-down listbox, also called a combo-box. It accepts color and font, as well as readonly and range. The range is specified as a discrete picklist, called 'list', that is used to populate to control. It is specified as 'list=0,1,2,3,4' or as '0!zero,1!one,2!two,3!three,4!four'. The first form will populate with whatever the choices are and those are what is returned on query. The second form will populate the control with the second string in each pair while keying off the first. The first strings are used for item selection and are what is returned when gathered. Alternately the control can be set to 'Active' mode, where the selection is sent immediately and will not be gathered. The 'always' flag makes sure that it's always sent. Level specifies the minimum privilege level in order for the user to access this field.

```
<CONTROL Target='##' name='Chooser' data='Time' control='SChooser' x='100' y='110' cx='100'
cy='25' [fgcolor='black' bgcolor='yellow' font='14' list='0!one,1!two,2!three,3!four,4!five'
rows='4' readonly='true' active='true' always='true' level='2']/>
```

5.8.5 SEditfield:

An editfield control directly connected to a back-end object and receiving data updates from it. When the mouse is clicked in it, it stops displaying remote data in preparation for typed data entry. If the field is exited without entering any data, then it begins displaying remote data again. If data has been entered, then it waits for that data to be gathered by the engine. It accepts color and font commands as well as 'readonly=true' which makes it a display only field. It accepts 'type=I/R/S' for Int, Real, or String which is the default. This affects range control. 'Range=x!y' allows a numeric range span to be specified. Range is checked by the object before any data can be gathered by the engine. All ranges are promoted to Double internally. 'format=x.xx' can specify decimal precision and layout to format the data to when it is shown or gathered. Ranges are displayed to the user as a ToolTip. The entry data is not passed back to the back-end until the Engine gathers it to send in a single block. Typically no data is sent unless the user has entered new data into the field. The 'always' flag makes sure that it's always sent. Level specifies the minimum privilege level in order for the user to access this field.

```
<CONTROL Target='##' name='Time' data='Time' control='SEditField' x='100' y='10' cx='100'
cy='25' [bgcolor='gray' fgcolor='black' type='R' format='0.00' range='2100.000!3100.00'
readonly='false' always='true' level='2']/>
```

5.8.6 SFrequency:

This is a compound control consisting of an entry field and a combobox. It is used for entering frequency data and the combobox allows you to change the display scale on the fly. When the mouse is clicked in the entryfield, it stops displaying remote data in preparation for typed data entry. If the field is exited without entering any data, then it begins displaying remote data again. If a selection is made in the combobox, it changes the display scale and stops accepting remote data. It can be reactivated be clicking in the associated entryfield and then back out again without entering any data. If data has been entered, then it waits for that data to be gathered by the engine. It accepts color and font, as well as readonly and range. The range is specified as a numeric span, ie. 'Range=x!y'. Range is checked by the object before any data can be gathered by the engine. All ranges are promoted to Double internally and the internal representation of the value is in Hz. This is scaled to the proper level when the data is displayed or gathered. The 'cx' size specifier is required but ignored. The 'always' flag makes sure that it's always sent. The control is always a minimum of 130 pixels wide , but may be wider. Level specifies the minimum privilege level in order for the user to access this field.

```
<CONTROL Target='##' name='Freq' data='SBandFreq' control=' SFrequency ' x='100' y='110'
cx='170' cy='25' [fgcolor='black' bgcolor='yellow' font='14' range='2100MHZ!2400MHZ'
readonly='true' always='true' level='2']/>
```

5.8.7 SLister:

This is a scrolling listbox that can be used for data selection or display. It accepts color and font, as well as readonly and range. The range is specified as a discrete picklist, called 'list', that is used to populate to control. It is specified as 'list=0,1,2,3,4' or as 'list=0!zero,1!one,2!two,3!three,4!four'. The first form will populate with whatever the choices are and those are what is returned on query. The second form will populate with the second string in each pair while keying off the first. The first strings are used for item selection and are what is returned when gathered. Alternately the control can be set to 'Active' mode, where the selection is sent immediately and will not be gathered. The 'always' flag makes sure that it's always sent. Level specifies the minimum privilege level in order for the user to access this field.

```
<CONTROL Target='##' name='Lister' data='Time' control='SLister' x='100' y='110' cx='100'
cy='25' [fgcolor='black' bgcolor='yellow' font='14' list='0!one,1!two,2!three,3!four,4!five'
rows=4 readonly='true' active='true' always='true' level='2']/>
```

5.8.8 SMultiLine:

An multiline editfield (MLE) control directly connected to a back-end object and receiving data updates from it. When the mouse is clicked in it, it stops displaying remote data in preparation for typed data entry. If the field is exited without entering any data, then it begins displaying remote data again. If data has been entered, then it waits for that data to be gathered by the engine. It accepts color and font commands as well as 'readonly=true' which makes it a display only field. It accepts 'type=I/R/S' for Int, Real, or String which is the default. This affects range control. 'Range=x!y' allows a numeric range span to be specified. Range is checked by the object before any data can be gathered by the engine. All ranges are promoted to Double internally. 'format=x.xx' can specify decimal precision and layout to format the data to when it is shown or gathered. Ranges are displayed to the user as a ToolTip. Range and format are dubious concepts in an MLE, so be aware that they may go away. "Append='true'" allows the MLE to append the new text to the existing text. "dupes='true'" works with append. It defaults to 'false' and does not allow duplicate entries to accumulate in the MLE. If set to 'true', it will allow duplicates. The entry data is not passed back to the back-end until the Engine gathers it to send in a single block. Typically no data is sent unless the user has entered new data into the field. The 'always' flag makes sure that it's always sent. Level specifies the minimum privilege level in order for the user to access this field.

```
<CONTROL Target='##' name='Time' data='Time' control='SMultiLine' x='100' y='10' cx='100'
cy='25' [bgcolor='gray' fgcolor='black' type='R' format='0.00' range='2100.000!3100.00'
readonly='false' always='true' level='2' append='true' dupes='true']/>
```

5.8.9 SScrollMLE:

An version of the multiline editfield (MLE) control with vertical scrolling capabilities. Its command set is exactly the same as the regular SMultiLine.

5.8.10 SPassword:

The SPassword is a simplified version of the SEditfield which allows the entry of '*' screened text. It does not display any other data nor accept range information. The 'always' flag makes sure that it's always sent. Level specifies the minimum privilege level in order for the user to access this field.

```
<CONTROL Target='##' name='Time' data='Time' control='SPassword' x='100' y='10' cx='100'
cy='25' [bgcolor='gray' fgcolor='black' font='14' always='true' level='2']/>
```

5.8.11 SPushButton:

Is a pushbutton that will send an immediate message to its target object. It accepts color and font, 'enabled=true/false' to activate or deactivate the button. The button can also specify 'popup=dlgname' which will cause the Engine to request a dialog of this name from the target instead of sending a push notification. An alternate for is "popup='targ2|dlgname'" which gets the dialog from a different target object.

```
<CONTROL Target='##' name='Time' data='Time' control='SPushbutton' x='100' y='10' cx='100'
cy='25' [bgcolor='gray' fgcolor='black' font='14' text='Pushme' enabled='true' popup='
whoozit| Oops']/>
```

5.8.12 SToggleButton:

A bi-state, toggling pushbutton that can send an immediate message to its target object or hold the selection. It accepts color and font, 'enabled=true/false' to activate or deactivate the button. The control can be set to 'Active' mode, where the selection is sent immediately and will not be gathered. The 'always' flag makes sure that it's always sent. Level specifies the minimum privilege level in order for the user to access this field. The control always shows the same text, until it is explicitly changed. It has 3 color selection which all default to whatever the bgcolor is. "offcolor='red'" sets what the color selection should be when the button is 'off'. "oncolor='green'" sets what the color selection should be when the button is 'on'. "intcolor='yellow'" sets what the color selection should be when the button state has been selected but not confirmed (interim color). This selection is operationally optional and only used when the color is set and the button is set 'active'. The operational values are '0' for off and '1' for on.

```
<CONTROL Target='##' name='Power' data='Power' control=' SToggleButton ' x='100' y='10'
cx='100' cy='25' [bgcolor='gray' fgcolor='black' font='14' text='Power' enabled='true'
active='true' always='true' level='2' offcolor='red' intcolor='yellow' oncolor='green']/>
```

5.9 System Screens

There is a collection of standard helper dialogs such as warning boxes and yes/no dialogs that reside in a system screens file. These screen blocks will be sent to the object and registered automatically, just as the object specific dialogs are. The developer does not have to explicitly provide them. The developer does, however, have to provide the functional code to drive the standard screens that will be used by each individual module.

The included screens are:

OpenDlg - A file-open dialog

SaveDlg - A file-save dialog. Same format as OpenDlg but different fieldnames

Wait - A wait box with a programmable message field

Config - A simple Config dialog for simple VI's that don't provide their own.

5.10 Writing New Screen Control Beans

New data beans are easy to write within some guidelines. The must implement the 'SDataInterface' Java interface:

```
public interface SDataInterface
{
   public   void     setup( SPushee dataSource, String szName );
   public   void     setGeometry( int x, int y, int cx, int cy );
   public   void     setContents( String input );
   public   void     addActionActor( ActionListener who );
   public   String   extract( boolean yesNo );
   public   void     unHook();
   public   void     setLevel( short level );
   public   void     setUserLevel( short level );
   public   String   getName();
   public   void     unDo();
}
```

Any other functions implemented with the class are incidental, but all of these function MUST be implemented because other parts of the GUI Engine calls them to perform specific functions:

'setup()' is used to finish what is not done in the constructor and to register some handler functions.

'setGeometry()' is used to size the object because there are possible implementation restrictions on the sizing that the Engine wouldn't know about. An example of this is the Sfrequency bean, which has a minimum size and must adjust itself to account for the built-in combobox which is part of the control.

'setContents()' is the function where all data and commands come into the bean to be interpreted. By convention, 'value', 'fgcolor', and 'bgcolor' are some of the standard commands that all should parse and handle. Others are handled as needed for the function of the bean.

'addActionActor()' is vestigial in the current version and should be removed.

'extract()' is used by the Engine to gather any data entered into the bean.

'unHook()' is used by the Engine to detach this bean from the data flow system.

'setLevel()' is used by the display classes (SWindow and Sdialog) to have the bean inherit the displays security level until it receives its own.

'setUserLevel()' is used by the data flow system to inform each bean what the current users security level is so the bean can act accordingly.

'getName()' is used by the system to query each object's name while iterating through an anonymous list of beans.

'unDo()' is used by the data flow system to undo selections where the data update was set to immediate and the update failed validation for some reason.

Here is an example shell of a bean that needs to be filled in. It is functionally incomplete and erroneous, but is here to show the basic idea and structure. If writing a new Bean, please review the existing package for sources close to the required functionality and work from that. The shell source follows:

```
========================================
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class SNewBean extends OriginalJavaControl implements SDataInterface
{
   private String             fieldName;
   private SPushee            _dataSource;
   private boolean            accessFlag;
   private boolean            dataEntered;
   private int                unDoer;
   private boolean            readOnly = false;
   private boolean            setReadOnly = false;
   private boolean            active = false;
   private short              permLevel = -9;
   private short              userLevel = 0;

   public SLister(  )
   {
      accessFlag = true;
      dataEntered = false;
   }

   public void setup( SPushee dataSource, String szName )
   {
      fieldName = new String( szName );
      _dataSource = dataSource;
   }
```

```
   public void setGeometry( int x, int y, int cx, int cy )
   {
      setBounds( x, y, cx, cy );
   }

   public void setContents( String input )
   {
      int    i = input.indexOf( '=' ) + 1;
      String setting = input.substring( i );

      if ( accessFlag == true )
      {
         if ( input.startsWith( "value" ) )
         {
            // set your internal data value here

            return;
         }
         else if ( input.startsWith( "bgcolor" ) )
         {
            this.setBackground( new SColorMap().select( setting ) );
            return;
         }
         else if ( input.startsWith( "fgcolor" ) )
         {
            this.setForeground( new SColorMap().select( setting ) );
            return;
         }
      }

      if ( input.startsWith( "readonly" ) )
      {
         setReadOnly = ( setting.equalsIgnoreCase( "true" ) ? true : false );
         if ( userLevel<permLevel )
            readOnly = true;
         else
            readOnly = setReadOnly;
         dataList.setEnabled( readOnly ? false : true );
      }
      else if ( input.startsWith( "font" ) )
      {
         float fPoints = Float.parseFloat( setting );
         setFont( getFont().deriveFont( fPoints ));
      }
      else if ( input.startsWith( "level" ) )
      {
         permLevel = (short) java.lang.Math.max( java.lang.Short.parseShort( setting ),
permLevel );
      }
   }

   public void addActionActor( ActionListener who )
   {
      // there are no actions here
      return;
   }
```

```
   public String extract( boolean yesNo )
   {
      String returner;

      if ( yesNo )
         returner = fieldName + ":value=" + contents + ";";
      else
         returner = "";

      if ( yesNo == false )
      {
         accessFlag = true;
         dataEntered = false;
      }

      return returner;
   }

   public void unHook()
   {
      _dataSource.removeDisplayObject( fieldName, this );
   }

   public void setLevel( short level )
   {
      if ( permLevel == -9 )
         permLevel = (short) java.lang.Math.max( level, permLevel );
   }

   public void setUserLevel( short level )
   {
      userLevel = level;
      if ( userLevel<permLevel )
         readOnly = true;
      else
         readOnly = setReadOnly;
      field.setEnabled( readOnly ? false : true );
   }

   public String getName()
   {
      return fieldName;
   }

   public   void     unDo()
   {
      // fill this with how to undo
   }
}
=========================================
```

5.11 GUI Build and Packaging

The 'gui.dsp' Visual Studio project file simple call the 'gui.mak' makefile. The build is completely automatic and results in two 'JAR' files, which are JavaARchives. The first is the 'SatLinx.jar' which is the Engine and its operating code. The second is 'SatBeans.jar' which only contains the data beans. On some systems, the batchfile 'jarit.bat' will exhibit a failure during the 'jar –i SatLinx.jar' or 'jar –i SatBeans.jar'

commands. This is an optional performance step and its failure does not inhibit the system.

5.12 GUI Execution

The GUI process is started by executing 'java –jar SatLinx.jar [%1 [%2 [%3]]]'. There is a batchfile called 'runjar.bat' which just contains the 'java –jar SatLinx.jar' and passes the parameters. All parameters are options, but must be used in order and preceding parameter must be specified in order to specify later ones. The first parameter is the name of the system where the NodeController is running. It defaults to 'localhost'. The second parameter is the name of the NodeController module on that host. It currently defaults to 'NODE-AKM'. The third parameter is the name of an initial XML file to parse, giving initial screen instructions or declaring an initial connection. It defaults to an internal setup that simply connects to the NodeController, which does the rest.

6 Source Control

6.1 The Development Project

The development tree stored within the Visual SourceSafe archive contains all of the source code and all versions of any extra packages used to build the system. The entire tree should never be indiscriminately pulled.

6.1.1 Updating Third Party Libraries

For all of the external libraries that are used, all historical versions are maintained within the SourceSafe archive. Each directory tree is maintained in its entirety with the version number as a component of the name of the older versions, ie. OOC.403, OOC.405 and OOC. In the example, 'OOC' would be the currently used directory. The user should never indiscriminately pull the entire source tree from the archive, as a large volume of superceded code will be pulled and may confuse the development environment.

When a new version of any of these external packages is to be brought in, the prior version should be renamed to include the version number in the name. Then the newer version should be added to the archive under the base name. OOC would become OOC.410 and version 4.20 would be uploaded as OOC. The working directory for both the old and new versions should be checked so they don't coincide.

6.1.2 Adding New Modules

In order to create a new SatLinx module, a new subdirectory should be created under the directory appropriate to the function of the new module. This is accomplished by right-clicking on the proper parent directory in SourceSafe and selecting 'Create Project' and then giving it the name of the new module. Then identify a similar project and copy their DSP and Makefiles into the new directory **ON THE WORKSTATION.** Copying files within SourceSafe can be problematic and a fair amount of damage has been done in the past by doing this with insufficient care. Once the files are copied, rename them appropriately and add them to the newly created project in SourceSafe. Now check them out so that they are writable and there is a historical record of the changes to be made.

6.1.2.1 Visual Studio Project File (.DSP)

The Visual Studio Project (DSP) files contain the individual project file and build settings. They are **VERY** subject to damage by allowing the IDE to change the settings. They have been set up and standardized to have the same build settings. Please edit them by hand and only do so to add files to the project.

The top section of the file contains project name information, followed by build information for the release and debug versions. After that are the file inclusions, followed by the custom build sections where the commands to build the IDL reside. This last section is superfluous and may be removed **if the makefile has been properly implemented**, since this function is handled by the makefile architecture before the DSP file is executed. The custom build section may not exist in the file you are working with!

Bring the file up in a text editor and begin. Change the necessary project name instances throughout the top section. Change the target names in the two build sections. You should avoid changing anything else in the build sections unless absolutely necessary. If you chose the source well, all the libraries and DLLs should be the same. Next, find the section titled '# Begin Group "Source Files"' and add or remove file names from the next section. Last, find the section titled '# Begin Group "IDL Files"' which will be after the end of the 'Source Files' group. If you have built a proper makefile, you may remove the custom build section if it exists. It is STRONGLY suggested that a makefile be created for each module, with the proper IDL references in it. This is a convenience now, but will be a necessity when building releases from source control. It also helps to resolve dependencies, which relying on DSP files cannot do.

6.1.2.2 Makefile system

As noted in section 2.3.4, it is necessary to implement a makefile for each new project so that the IDL files can be compiled into C++ stubs and skeletons. To accomplish this it is necessary to make the following additions to the makefile system:

1. Create a module makefile

2. Add the module directory to the relevant infrastructure makefile. Note that the module makefile and the module directory differ only in the "*.mak*" extension of the makefile.

3. Create a library dependency fragment if the module is a statically linked library or implicitly loaded DLL.

6.1.2.2.1 Creating a Module Makefile

The module makefile should be created from an existing makefile, or from a template such the VI template reproduced below.

```
# ****************************************************************
# Stovokor Module Makefile
# Module: Personality_VI
# Author: Andrew Manison
# Date:   9/27/2000
# Rev:
# ****************************************************************

# ****************************************************************
# Define module name. This is used elsewhere for module unique
# parameters so make sure there is no clash with any other module.
# ****************************************************************
MODULE_NAME = Personality_VI

# ****************************************************************
# Check for required macros definitions.
# ****************************************************************
!ifndef STOVOKOR_TOP
!error STOVOKOR_TOP not set in the environment or on the command line.
!endif
!if "$(MODULE_NAME)" == ""
!error MODULE_NAME not set.
!endif

# ****************************************************************
# Define subsystem dependencies.
# ****************************************************************

#DEBUG=yes
IS_DLL=yes
IS_SCOMPONENT=yes

# ****************************************************************
# Include the directory definitions
# ****************************************************************
!include $(STOVOKOR_TOP)\config\Make.dirs
```

```
# ************************************************************************
# List the sources and intermediate sources used comprising the module.
# ************************************************************************
SOURCES = Personality_VI_impl.cpp \
          Personality_VI_main.cpp

IDLSRC1 = Personality_VI.idl

IDLGEN1 = $(IDLMEZZ)\Personality_VI.h $(IDLMEZZ)\Personality_VI.cpp \
          $(IDLMEZZ)\Personality_VI_skel.h $(IDLMEZZ)\Personality_VI_skel.cpp

IDLSRC2 = $(IDLDIR)\Personality.idl

IDLGEN2 = $(IDLCPP)\Personality.h $(IDLCPP)\Personality.cpp \
          $(IDLCPP)\Personality_skel.h $(IDLCPP)\Personality_skel.cpp

IDLSRC  = $(IDLSRC1) $(IDLSRC2)

IDLGEN  = $(IDLGEN1) $(IDLGEN2)

TARGETS = $(OUTDIR)\Personality_VI.dll

# ************************************************************************
# Define the objects to be linked.
# ************************************************************************
OBJECTS = $(INTDIR)\Personality.obj         \
          $(INTDIR)\Personality_skel.obj    \
          $(INTDIR)\Personality_VI.obj      \
          $(INTDIR)\Personality_VI_skel.obj \
          $(INTDIR)\Personality_VI_impl.obj \
          $(INTDIR)\Personality_VI_main.obj

# ************************************************************************
# Include the build rules
# ************************************************************************
!include $(CFGDIR)\Make.rules

# ************************************************************************
# Target (link) dependencies
# ************************************************************************

$(TARGETS) : $(OBJECTS)
       @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo idl"
       @cmd /c "cd $(STOVOKOR_TOP) & $(MAKE) /nologo $(LIBDEPS)" > nul
       @$(LINK) @<<
  $(LINKFLAGS) $(LINK32_FLAGS) $(OBJECTS) $(LIBS)
<<

# ************************************************************************
# Installation dependencies
# ************************************************************************

install::
       @echo Copying $(OUTDIR)\Personality_VI.dll to $(vbindir)\Personality_VI.dll
       @copy $(OUTDIR)\Personality_VI.dll $(vbindir) > nul

# ************************************************************************
# Include the generated dependencies
# ************************************************************************
!IF EXISTS(".depend")
!INCLUDE ".depend"
!ELSEIF "$(DEBUG)" == "yes"
!MESSAGE Warning: cannot find ".depend" for $(MODULE_NAME)
!ENDIF
```

This template is for a virtual instrument, which is a module based upon a mix-in IDL interface. The details of the module makefile macros are described in section 2.3.4.8. This template assumes that the module is based upon the following source files:

Personality.idl	The mix-in IDL definition.
Personality_VI.idl	The mezzanine IDL definition to combine the mix-in with the base class definitions.
Personality_VI_impl.h	The module implementation header file.
Personality_VI_impl.cpp	The module implementation source file.
Personality_VI_main.cpp	The module startup file.

The two IDL files are listed in the IDLSRCx macros, and combined in the macro IDLSRC. The generated stubs and skeletons are listed in the IDLGENx macros, and combined in the macro IDLGEN. The remaining source files are listed in the SOURCES macro.

The objects files generated by compiling all the C++ sources are listed in the macro OBJECTS.

The target file (or files, for implicitly loaded DLLs) are listed in the macro TARGET.

Once these macros are correctly defined the makefile is complete.

6.1.2.2.2 Updating The Infrastructure Makefile

The infrastructure makefile includes the macro SUBDIRS that lists all the module subdirectories to be built. The subdirectory for the new module must be added to the list. Note the name of the subdirectory must be the same as the makefile without the '*.mak*' extension.

6.1.2.2.3 Creating A Library Dependency Fragment

If the module is a library that is to be linked at build time with other modules, as opposed to being explicitly loaded by a factory, it needs a dependency fragment. An example of one is shown in section 2.3.4.4. This must be saved with a unique name and the extension '.mkx' in the config subdirectory of the development tree.

6.2 The Base Configuration Project

This set of subdirectories is the base from which all custom configurations are produced.

6.2.1 Base Configuration File

The base configuration version of the system configuration file is called *Base_Stovokor.xml*. This file contains entries for the standard infrastructure modules and single entries for all of the module types in the system. It should be copied over to the custom installation directory and edited as needed.

The infrastructure portion of the file is reproduced below:

```
<XML>
   <SYSTEM CONFIG="CFGMGR" LOGGER="LOGGER" REVMGR="REVISION" FILEMGR="FILEMGR"
           NODEINFO="L1NODEINFO_MGR" NODECTL="NODE-AKM" BASEPORT="20000">
      <CORBA_ARG NAME="NameService" ARG="-ORBInitRef"
                 VALUE="NameService=corbaloc::localhost:15000/NameService"/>
      <CORBA_ARG NAME="EventService" ARG="-ORBInitRef"
                 VALUE="DefaultEventChannelFactory=
                 corbaloc::localhost:10000/DefaultEventChannelFactory"/>
      <RUN NAME="NameService" EXECUTABLE="corbaservice.exe" ARGUMENT1="NameService"
           ARGUMENT2="nameserv.exe" ARGUMENT3="-passthrough-OAport" ARGUMENT4="15000"
           USE_CORBA_ARGS="Yes" USE_VERSION_PATH="No"/>
      <RUN NAME="EventService" EXECUTABLE="corbaservice.exe"
           ARGUMENT1="DefaultEventChannelFactory" ARGUMENT2="eventserv.exe"
           ARGUMENT3="-passthrough-OAport" ARGUMENT4="10000" USE_CORBA_ARGS="Yes"
           USE_VERSION_PATH="No"/>
      <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="NSFEDERATOR.exe"
              NAME="NSFEDERATOR"/>
      <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="logger.exe" NAME="LOGGER"/>
      <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="authent.exe" NAME="AUTHENT"/>
      <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="revmgr.exe" NAME="REVISION"/>
      <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="filemgr.exe" NAME="FILEMGR"/>
      <MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="NODE-AKM" EXECUTABLE="factory.exe">
         <TYPE>NC_VANILLA</TYPE>
         <FACTORY_NAME>NODE_FACTORY</FACTORY_NAME>
      </MODULE>
      <MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="FAULTMGR" EXECUTABLE="factory.exe">
         <TYPE>FAULT_MGR</TYPE>
         <FACTORY_NAME>FAULTMGR_FACTORY</FACTORY_NAME>
      </MODULE>
```

For each new module created, an entry must be added to the system configuration file. The following fragment illustrates what would need to be added:

```
<MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="NEW_FUNC" EXECUTABLE="factory.exe">
   <TYPE>NEW_FUNC</TYPE>
   <FACTORY_NAME>NEW_FUNC_FACTORY</FACTORY_NAME>
</MODULE>
```

Note that the module's name is the same as its type. This would not necessarily be the case in a real system, but it simplifies the organization of the system configuration file and the module instance files.

The system configuration file must end by closing the SYSTEM element and the top-level XML element, as follows:

```
   </SYSTEM>
</XML>
```

6.2.2 Equipment Default Files

These files should contain the specific default settings for the individual object types. No data for any specific instance should be here, only default values and ranges:

```
<XML>
   <HeartbeatRate MIN='500' MAX='30000'>2000</HeartbeatRate>
   <TimeOut>3500</TimeOut>
   <MAX_LNAS>2</MAX_LNAS>
   <SwitchMode>0</SwitchMode>
   <LocalMode>0</LocalMode>
   <INSTALLED_LNAS>2</INSTALLED_LNAS>
   <LNA LIST='1,2' TEXT='LNA 1,LNA 2'></LNA>
   <INPUTSRC LIST='A,T' TEXT='Antenna,Test'></INPUTSRC>
   <Gain>5</Gain>
</XML>
```

Note the LNA selection that contains the available selection but no value.

6.2.3 Default Instance Files

There should be on instance file for each type of object. There should be no duplicates or actual instances here, just one per type. These files contain prototype data that will be used in the actual instances to set specific values, name, and connections.

The name of the default instance file must be the same as its type, just as the module element added to the system configuration file described above. This file will be renamed to the actual instance name when the base configuration is copied to a new customer configuration tree.

At least, the instance file should contain the following elements:

- DEFAULTS
- SCREENS

- GUINAME

Physical instruments should also contain:

- FW_Version
- COMMADAPTER

Redundant instrument VIs should contain:

- FAULTBASE

Other elements are unique to the specific module implementation. A sample instance file is shown below.

```
<XML>
   <SCREENS>LNAC_PI_MITEQ_Screens</SCREENS>
   <DEFAULTS>LNAC_PI_MITEQ_Defaults</DEFAULTS>
   <COMMADAPTER>CASOCKET</COMMADAPTER>
   <GUINAME>Miteq LNA Controller</GUINAME>
   <UNITADDRESS>A</UNITADDRESS>
   <FW_Version>Version 1.3A</FW_Version>
   <LNA>1</LNA>
   <INPUTSRC>A</INPUTSRC>
</XML>
```

6.2.4 Screen Files

All of the screen files should reside here in their original form. They are aggregation of all screens specific to the module. It does not contain system-wide screen definitions. It is not directly parseable XML, but is dissembled into parseable block by the 'SComponent' baseclass:

```
<SCREEN>
  screen 1 block
</SCREEN>
<SCREEN>
  screen 2 block
</SCREEN>
<SCREEN>
  screen 3 block
</SCREEN>
<SCREEN>
  screen 4 block
</SCREEN>
```

6.3 Customer Configuration Projects

Custom configurations are created by copying the source tree within Visual SourceSafe and mapping the original and the copy to the same directory on the local filesystem. Instance files will then need to be renamed and/or copied based on the instrument

collection. The 'Base_Stovokor.xml' will be copied to 'Stovokor.xml' in the 'Equipment Defaults' subdirectory.

6.3.1 Selection of Components

Modules come in three flavors: System(required), Support(required), and Operational(optional).

6.3.1.1 Required Components

The Required components are either structural or platform-related. The required System modules are built as executables. They are run directly and must be present in all systems:

```
NameService
EventService
NSFEDERATOR.exe – only used in distributed systems
logger.exe
authent.exe
revmgr.exe – not yet implemented
filemgr.exe
```

The required Support modules are started by the factory after the first group have been started:

```
NodeController
FaultManager
NodeInfoManager
NodeOperationsManager
SchedulerEphemeris
ManagerAssetManager
```

6.3.1.2 Optional Operational Components

All of the optional modules are operational and relate to device control, usage, or communications.

6.3.1.3 System Configuration File

The system configuration file will need to be edited to reflect the requirement equipment and module mix in the custom system.

The top section of the system configuration file must remain untouched, except for possible port number changes or module name changes in the 'SYSTEM' line. 'USE_DEBUG_CONSOLES' is also optional and defaults to 'Yes':

```
<SYSTEM CONFIG="CFGMGR" LOGGER="LOGGER" REVMGR="REVISION" FILEMGR="FILEMGR"
        NODEINFO="L1NODEINFO_MGR" NODECTL="NODE-AKM" BASEPORT="20000"
        USE_DEBUG_CONSOLES="Yes">
   <CORBA_ARG NAME="NameService" ARG="-ORBInitRef"
              VALUE="NameService=corbaloc::localhost:15000/NameService"/>
   <CORBA_ARG NAME="EventService" ARG="-ORBInitRef" VALUE=
   "DefaultEventChannelFactory=corbaloc::localhost:10000/DefaultEventChannelFactory"/>
   <RUN NAME="NameService" EXECUTABLE="corbaservice.exe" ARGUMENT1="NameService"
        ARGUMENT2="nameserv.exe" ARGUMENT3="-passthrough-OAport" ARGUMENT4="15000"
        USE_CORBA_ARGS="Yes" USE_VERSION_PATH="No"/>
   <RUN NAME="EventService" EXECUTABLE="corbaservice.exe"
        ARGUMENT1="DefaultEventChannelFactory" ARGUMENT2="eventserv.exe"
        ARGUMENT3="-passthrough-OAport" ARGUMENT4="10000" USE_CORBA_ARGS="Yes"
        USE_VERSION_PATH="No"/>
   <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="NSFEDERATOR.exe"
           NAME="NSFEDERATOR"/>
   <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="logger.exe" NAME="LOGGER"/>
   <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="authent.exe" NAME="AUTHENT"/>
   <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="revmgr.exe" NAME="REVISION"/>
   <MODULE LOCAL="Yes" USE_FACTORY="No" EXECUTABLE="filemgr.exe" NAME="FILEMGR"/>
```

This is the portion of the `SYSTEM` section creates the Satlinx infrastructure. All the rest of the modules are started by the factory after the first group have been started. The required ones are used to create the Satlinx platform on top of the Stovokor system. The rest are related to Satlinx platform operations.

The XML block for starting a factory module looks like:

```
<MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="FAULTMGR" EXECUTABLE="factory.exe">
   <TYPE>FAULT_MGR</TYPE>
   <FACTORY_NAME>FAULTMGR_FACTORY</FACTORY_NAME>
</MODULE>
```

The configuration file is discussed in more detail in section 4.4.2.4.2.

6.3.2 Topology

The topology file is used be the AssetManager to determine equipment allocation and redundancy. The topology map lists all line-replaceable-units (LRUs) in a hierarchy, and allows a fault in an LRU to be translated to a fault status for the node as a whole. Below is a sample topology file.

```
<XML>
   <TOPNODE NAME='NodeOpManager' USERNAME='Terminal Asset Manager'
            HIGHER='L2ASSETMANAGER' OUTPUT='L1_NODE_MANAGER' INPUT='L1_NODE_MANAGER'
            MAXFAULT='CRITICAL' USAGE='SPF' ITEMNUMBER='1' SCREENITEM='1'
            VERSION='ViaSat,1.0' CLASS='NODE' TYPE='L1ASSETMANAGER' INSTANCE='1' >
      <TOPGROUP NAME='CONTROLS'  USERNAME='Terminal Controls'
                HIGHER='NodeOpManager' OUTPUT='L1_ASSET_MANAGER'
                INPUT='L1_ASSET_MANAGER' MAXFAULT='MINOR' USAGE='SPF' ITEMNUMBER='2'
                SCREENITEM='2' VERSION='ViaSat,1.0'
                CLASS='GROUP' TYPE='CONTROLS' INSTANCE='1' >
         <TOPLRU NAME='COMPUTER' USERNAME='Terminal Computer' HIGHER='CONTROLS'
                 OUTPUT='TERMINAL_NETWORK' INPUT='TERMINAL_NETWORK' MAXFAULT='MINOR'
                 TYPE='COMPUTER' USAGE='SPF' ITEMNUMBER='3' SCREENITEM='21'
                 VERSION='ViaSat,1.0' CLASS='LRU' INSTANCE='1' />
```

```
    <TOPLRU NAME='SOFTWARE' USERNAME='Terminal Software' HIGHER='CONTROLS'
            OUTPUT='TerminalEquipment' INPUT='TerminalEquipment'
            MAXFAULT='WARNING' TYPE='SOFTWARE' USAGE='SPF' ITEMNUMBER='4'
            SCREENITEM='22' VERSION='ViaSat,1.0' CLASS='LRU' INSTANCE='1' />
    <TOPLRU NAME='SERIAL_CONVERTER_VI' USERNAME='Protocol Converter'
            HIGHER='CONTROLS' OUTPUT='TerminalEquipment' INPUT='TERMINAL_NETWORK'
            MAXFAULT='MINOR' TYPE='PROTOCONV' USAGE='SPF' ITEMNUMBER='5'
            SCREENITEM='23' VERSION='ViaSat,1.0' CLASS='LRU' INSTANCE='1' />
    <TOPLRU NAME='FIU' USERNAME='Facilities Interface' HIGHER='CONTROLS'
            TYPE='FIU' OUTPUT='SOFTWARE' INPUT='FacilitySensors' MAXFAULT='MINOR'
            USAGE='SPF' ITEMNUMBER='6' SCREENITEM='24' VERSION='ViaSat,1.0'
            CLASS='LRU' INSTANCE='1' />
    <TOPLRU NAME='NTS_1' USERNAME='Network Time Server' HIGHER='CONTROLS'
            OUTPUT='TERMINAL_NETWORK' INPUT='GPS_Receiver' MAXFAULT='MINOR'
            TYPE='NTS' USAGE='SPF' ITEMNUMBER='7' SCREENITEM='25'
             VERSION='ViaSat,1.0' CLASS='LRU' INSTANCE='1' />
    <TOPLRU NAME='MSU' USERNAME='Modem Switch Unit' HIGHER='CONTROLS'
            OUTPUT='Modems' TYPE='MSU' INPUT='ATM' MAXFAULT='MINOR' USAGE='SPF'
            ITEMNUMBER='8' SCREENITEM='26' VERSION='ViaSat,1.0' CLASS='LRU'
            INSTANCE='1' />
    <TOPLRU NAME='CnvrtrCntrl_VI' USERNAME='Converter Controller'
            HIGHER='CONTROLS' OUTPUT='Converters' INPUT='SOFTWARE'
            MAXFAULT='MINOR' USAGE='SPF' TYPE='CONVC' ITEMNUMBER='9'
            SCREENITEM='27' VERSION='ViaSat,1.0' CLASS='LRU' INSTANCE='1' />
    <TOPLRU NAME='HPAC_VI' USERNAME='HPA Controller' HIGHER='CONTROLS'
            TYPE='HPAC' OUTPUT='HPAs' INPUT='SOFTWARE' MAXFAULT='MINOR'
            USAGE='SPF' ITEMNUMBER='10' SCREENITEM='28' VERSION='ViaSat,1.0'
            CLASS='LRU' INSTANCE='1' />
    <TOPLRU NAME='LNAC_VI' USERNAME='LNA Controller' HIGHER='CONTROLS'
            TYPE='LNAC' OUTPUT='LNAs' INPUT='SOFTWARE' MAXFAULT='MINOR'
            USAGE='SPF' ITEMNUMBER='11' SCREENITEM='29' VERSION='ViaSat,1.0'
            CLASS='LRU' INSTANCE='1' />
</TOPGROUP>
<TOPGROUP NAME='TRANSMIT' USERNAME='Transmit Equipment' HIGHER='CONTROLS'
          OUTPUT='RFOUT' INPUT='ATMIN' MAXFAULT='CRITICAL' TYPE='TRANSMIT'
          USAGE='SPF' ITEMNUMBER='12' SCREENITEM='3' VERSION='ViaSat,1.0'
          CLASS='GROUP' INSTANCE='1' >
    <TOPLRU NAME='MOD_VI_1' USERNAME='Modulator 1' HIGHER='TRANSMIT'
            OUTPUT='UPCONV' INPUT='ATMIN' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
            ITEMNUMBER='13' SCREENITEM='31' VERSION='ViaSat,1.0' CLASS='LRU'
            TYPE='MOD' INSTANCE='1' />
    <TOPLRU NAME='MOD_VI_2' USERNAME='Modulator 2' HIGHER='TRANSMIT'
            OUTPUT='UPCONV' INPUT='ATMIN' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
            ITEMNUMBER='14' SCREENITEM='32' VERSION='ViaSat,1.0' CLASS='LRU'
            TYPE='MOD' INSTANCE='2' />
    <TOPLRU NAME='UPC_A' USERNAME='Up-Converter 1' HIGHER='TRANSMIT' OUTPUT='HPA'
            INPUT='MOD' MAXFAULT='CRITICAL' USAGE='REDUNDANT' ITEMNUMBER='15'
            SCREENITEM='33' VERSION='ViaSat,1.0' CLASS='LRU' TYPE='UPCONV'
            INSTANCE='1' />
    <TOPLRU NAME='UPC_B' USERNAME='Up-Converter 2' HIGHER='TRANSMIT' OUTPUT='HPA'
            INPUT='MOD' MAXFAULT='CRITICAL' USAGE='REDUNDANT' ITEMNUMBER='16'
            SCREENITEM='34' VERSION='ViaSat,1.0' CLASS='LRU' TYPE='UPCONV'
            INSTANCE='2' />
    <TOPLRU NAME='HPA_A' USERNAME='Power Amplifier 1' HIGHER='TRANSMIT'
            OUTPUT='RFOUT' INPUT='UPCONV' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
            ITEMNUMBER='17' SCREENITEM='35' VERSION='Xicom,2.53' CLASS='LRU'
            TYPE='HPA' INSTANCE='1' />
    <TOPLRU NAME='HPA_B' USERNAME='Power Amplifier 2' HIGHER='TRANSMIT'
            OUTPUT='RFOUT' INPUT='UPCONV' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
            ITEMNUMBER='18' SCREENITEM='36' VERSION='Xicom,2.53' CLASS='LRU'
            TYPE='HPA' INSTANCE='2' />
</TOPGROUP>
```

```
      <TOPGROUP NAME='RECEIVE' USERNAME='Receive Equipment' HIGHER='CONTROLS'
                OUTPUT='ATM' INPUT='RFIN' MAXFAULT='CRITICAL' TYPE='RECEIVE'
                USAGE='SPF' ITEMNUMBER='19' SCREENITEM='4' VERSION='ViaSat,1.0'
                CLASS='GROUP' INSTANCE='1' >
         <TOPLRU NAME='LNA_A' USERNAME='Low-Noise Amplifier 1' HIGHER='RECEIVE'
                  OUTPUT='DNCONV' INPUT='RFIN' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
                  ITEMNUMBER='20' SCREENITEM='41' VERSION='Miteq,0.0' CLASS='LRU'
                  TYPE='LNA' INSTANCE='1' />
         <TOPLRU NAME='LNA_B' USERNAME='Low-Noise Amplifier 2' HIGHER='RECEIVE'
                  OUTPUT='DNCONV' INPUT='RFIN' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
                  ITEMNUMBER='21' SCREENITEM='42' VERSION='Miteq,0.0' CLASS='LRU'
                  TYPE='LNA' INSTANCE='2' />
         <TOPLRU NAME='DNC_A' USERNAME='Down-Converter 1' HIGHER='RECEIVE'
                  OUTPUT='DEMOD' INPUT='LNA' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
                  ITEMNUMBER='22' SCREENITEM='43' VERSION='Vendor,0.0' CLASS='LRU'
                  TYPE='DNCONV' INSTANCE='1' />
         <TOPLRU NAME='DNC_B' USERNAME='Down-Converter 2' HIGHER='RECEIVE'
                  OUTPUT='DEMOD' INPUT='LNA' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
                  ITEMNUMBER='23' SCREENITEM='44' VERSION='Vendor,0.0' CLASS='LRU'
                  TYPE='DNCONV' INSTANCE='2' />
         <TOPLRU NAME='DEMOD_VI_1' USERNAME='Demodulator 1' HIGHER='RECEIVE'
                  OUTPUT='ATMOUT' INPUT='DNCONV' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
                  ITEMNUMBER='24' SCREENITEM='45' VERSION='Vendor,0.0' CLASS='LRU'
                  TYPE='DEMOD' INSTANCE='1' />
         <TOPLRU NAME='DEMOD_VI_2' USERNAME='Demodulator 2' HIGHER='RECEIVE'
                  OUTPUT='ATMOUT' INPUT='DNCONV' MAXFAULT='CRITICAL' USAGE='REDUNDANT'
                  ITEMNUMBER='25' SCREENITEM='46' VERSION='Vendor,0.0' CLASS='LRU'
                  TYPE='DEMOD' INSTANCE='2' />
      </TOPGROUP>
      <TOPGROUP NAME='POSITION' USERNAME='Positioning Equipment' HIGHER='CONTROLS'
                OUTPUT='SATELLITE' INPUT='EPHEMERIS' MAXFAULT='MAJOR' USAGE='SPF'
                TYPE='POSITION' ITEMNUMBER='26' SCREENITEM='5' VERSION='ViaSat,1.0'
                CLASS='GROUP' INSTANCE='1' >
         <TOPLRU NAME='VIACU' USERNAME='Antenna Control Unit' HIGHER='POSITION'
                  OUTPUT='PEDESTAL' INPUT='EPHEMERIS' MAXFAULT='MAJOR' USAGE='SPF'
                  TYPE='ACU' ITEMNUMBER='27' SCREENITEM='51' VERSION='ViaSat,1.0'
                  CLASS='LRU' INSTANCE='1' />
         <TOPLRU NAME='PEDESTAL' USERNAME='Pedestal' HIGHER='POSITION'
                  OUTPUT='ANTENNA' INPUT='ACU' MAXFAULT='MINOR' USAGE='SPF'
                  TYPE='PEDESTAL' ITEMNUMBER='28' SCREENITEM='52' VERSION='ViaSat,1.0'
                  CLASS='LRU' INSTANCE='1' />
      </TOPGROUP>
      <TOPGROUP NAME='MISC' USERNAME='Miscellaneous Equipment' HIGHER='CONTROLS'
                OUTPUT='TerminalEquipment' INPUT='TerminalEquipment' MAXFAULT='MINOR'
                USAGE='SPF' ITEMNUMBER='29' SCREENITEM='6' VERSION='ViaSat,1.0'
                CLASS='GROUP' TYPE='MISC' INSTANCE='1' >
         <TOPLRU NAME='SONETCONV' USERNAME='SONET Converter' HIGHER='MISC'
                  OUTPUT='TBD' INPUT='TBD' MAXFAULT='MINOR' USAGE='SPF' ITEMNUMBER='30'
                  SCREENITEM='61' VERSION='ViaSat,1.0' CLASS='LRU' TYPE='MISC'
                  INSTANCE='1' />
         <TOPLRU NAME='DEICER' USERNAME='Deicing System' HIGHER='MISC' OUTPUT='TBD'
                  INPUT='DIOBOX' MAXFAULT='MINOR' USAGE='SPF' ITEMNUMBER='31'
                  SCREENITEM='62' VERSION='TBS,0.0' CLASS='LRU' TYPE='MISC'
                  INSTANCE='1' />
         <TOPLRU NAME='ANTENNA' USERNAME='Antenna' HIGHER='MISC' OUTPUT='SATELLITE'
                  INPUT='PEDESTAL' MAXFAULT='MINOR' USAGE='SPF' ITEMNUMBER='32'
                  SCREENITEM='63' VERSION='TBS,0.0' CLASS='LRU' TYPE='ANTENNA'
                  INSTANCE='1' />
      </TOPGROUP>
   </TOPNODE>
</XML>
```

6.3.2.1 Creating the Topology File

At present no tool exists for creating the topology map. A template file must be used as the basis, and all groups and LRUs added according to the configuration of the node.

6.3.3 Instance File Modifications

The 'System Setup' directory holds the instance data and will initially hold a single entry for each TYPE of object, named for that type. It needs to be copied if there are multiple instances of this object type and renamed for the object instance names. The files then need to be edited to change the 'GUINAME' attribute (which affects the display name on the screen) and CommAdapter connection information. Any initial values need to be put in now, as well as range adjustments if the different instances have different roles.

6.3.4 Developing Specific GUI Content

Installation specific GUI content can be created, but care must be taken not the change the 'data' names in the XML. This is the only connection to the back-end code. It is not just there for show. The back-end modules will almost always (editors being the exception) not know HOW the data is being presented. Controls can be moved and dialogs can even be separated into parent-child, or combined, or other reorganized without disrupting the operation. 'BUTTON' or 'MENU/MENUITEM' controls can be added to reference child dialogs. These are interpreted by the GUI engine, not the back-end. If a new bean control becomes availabl.e in the 'SatBeans.jar', simply change the old 'control' attribute to reference the new bean by name and change whatever other control attributes are necessary. When a screen definition is altered, typically the change will take effect the next time the module is started, not when the screen is redisplayed. The exception is if there is a 'SPushButton' control named 'RELOAD'. This is a built-in feature which will cause the back-end module to reread the XML screens file and reregister the new screen XML.

6.3.5 Configuration for Auto and Manual Startup

The Startup Service has a very simple interface, and is typically started from the command line in a console window, or from a desktop shortcut. Normal operation, as described in section 4.5.5, requires no arguments. The desktop shortcut is the ismplest option for manual startup.

Automatic startup following user login can easily be configured by adding the shortcut to the user's startup folder. If the user is configured as an automatic login user the program will be run whenever the host system is rebooted.

The ideal automatic startup option has not yet been implemented or verified. This is to make the startup service suitable for use as a Windows NT service. Minimal changes should be necessary to make this work.

6.3.6 Configuration for Remote GUI Support

There isn't any, it's a runtime parameter, which I've added to section 5.

7 Future Enhancements and Refactoring Objectives

7.1 Portability

Isolate and wrap Windows API calls. These are primarily directory and process management functions.

7.2 System

More actively and definitively define the retrieval of ranges from the PI to the VI.

7.3 Physical Instrument

Further clean up, getting rid of unused variables like idle rate or actually using it.

Remove 'virtual' from things which shouldn't be virtual, such as half of Physical Instrument.

Get rid of mp_Debug and friends.

Get rid of onHeartbeatChange or use it.

'Reset' method is wrong case and thus won't override the CORBA method.

7.4 Persistence

Should enhance the non-persistent support and update those objects which should be transient but aren't because they were developed before this spec was done.

7.5 Startup

System startup needs to be more organized and the Logger startup needs to be more robust.

Re-engineer system startup for better stability

- System currently difficult to start due to chaotic service interactions

Runjar needs to be updated for 3 parameters.

Enhance deferred startup and shutdown

- Add base class methods for controlled shutdown
- Provide notification of deferred status

7.6 Build

Remove vestigial IDL build commands from the DSP files.

Add autocheckout for build from CM to Makefile Architecture

7.7 ModuleMap

m_useDebugConsoles is uninitialized in class ModuleMap in ModMap.h

7.8 SBase_impl

PMapAdd() should be changed to private. No-one should need to mess with it!

7.9 GUI Engine

Using existing graphing and reporting libraries in Java

- Enhance platform independence and reduce dependency on other tools

7.10 Connector Classes

Re-engineer connectors

- Call CfgMgr for transparent startup of deferred modules
- Make them pervasive and enhance system consistency

- Streamline for single use
- Add optional keepalive?
- Retrofit all modules to use connectors

7.11 Asset and Configuration Management

Finish equipment pool allocation and resource contention

- Asset manager and topology system need to be completed and wired in
- The rest of the system needs to be looked at for interactions with the Asset Manager

7.12 System Generation Tools

Module requirement resolution

GUI generator

Topology Builder tool

7.13 System Logger

Implement planned enhancements:

- Modify log levels individually
- Extraction of log subsets into separate files
- Develop Screen

7.14 Stovokor.exe

Modify Stovokor.exe to make it able to run as an NT service

7.15 Revision Manager

Never implemented