



Introduction to the Communication Services Framework



Executive Summary

The Communication Services Framework (CSF) is software that can facilitate development of sophisticated communications applications. The CSF provides developers with a high-level, object-oriented interface that abstracts much of the underlying native board interface, thereby allowing the developer to focus on application development. Remoting capabilities enable the development of distributed telecommunications applications.

The CSF is provided in Open Source source code format so that developers may create applications to which they own the rights. Developers are free to distribute their applications in accordance with the Berkeley Software Distribution (BSD) licensing model. The CSF makes use of third-party components that also conform to the BSD licensing model, but it is the responsibility of application developers to ensure compliance with all applicable licensing.

The CSF is available as a SourceForge project that can be downloaded from <http://sourceforge.net/projects/commsvcfw>.



Table of Contents

Communications Services Framework.....	2
CSF Platforms	2
Operating Systems	2
Third-Party Components.....	2
Dialogic-Specific Components	2
Communications Services Framework Structure	3
Application Development Considerations	6
High-Level Considerations	6
Component Test Harnesses	8
Configuration Files.....	9
Logging	9
Minimum Components	9
Creating New Components	12
Putting It All Together	13
Extensions, Enhancements, and Making Your Own Application.....	13
New Functions to Existing Classes	13
Adding Existing Hardware Components to Your Application	15
Appendix A: Third-Party Components	16
Appendix B: Component Test Harnesses	17
Appendix C: Demonstration Programs	18
Appendix D: Environment Variables	18
Appendix E: Design Pattern Details.....	19
Appendix F: Installation and Configuration	20
Appendix G: CreateManagedObjects() Examples.....	24
Appendix H: Configuration File Sections	25
Appendix I: Custom Logging Implementations	28
Acronyms	29
For More Information.....	29

Communication Services Framework

The CSF is of an object-oriented design, providing the developer with a rich set of classes that implement the wide array of capabilities provided by both Dialogic® telecommunications boards and open, standards-based boards from other suppliers. Since it uses best-known methods (BKMs) in design and is provided as Open Source code, the CSF facilitates extensions and enhancements to its capabilities. The CSF also supports both the Windows® and Linux operating systems and was designed for maximal portability between the two platforms.

The design methodology used in implementing the CSF seeks to allow application development and framework enhancement by “cutting and pasting” modules that embody BKMs for application development.

The CSF is provided as-is, such that developers using it will assume full responsibility for supporting any applications they develop. Dialogic provides limited assistance in the form of application notes such as this one, as well as online discussion forums (see <http://www.dialogic.com/forums/category-view.asp>).

The CSF uses an object-oriented design written in C++, so developers using it need to have experience in these disciplines. The projects are built using Microsoft® Visual Studio® and the code is provided as a .dsw project workspace for the Windows operating system platform. Project makefiles are available for the Linux operating system platform.

By providing a high-level, object-oriented interface, the CSF enables developers to more easily create communications applications because they do not need to know the specifics of the underlying native platform application programming interface (API). However, CSF is not intended to eliminate the need for the developer to fully understand the fundamental operation, configuration, and integration into the deployment telecommunications environment of the Dialogic® telecom products or third-party products used in conjunction with the CSF.

CSF Platforms

The CSF is a set of C++ components, embedded in a C++ namespace, that make use of third-party, Open Source components and object-oriented design patterns. Dialogic does not directly support the CSF and third-party components; all support comes from the respective websites from which the components are downloaded.

Operating Systems

The CSF supports the Linux and Windows operating systems that are supported by Dialogic® System Release software. (For more information on the System Release software, visit http://www.dialogic.com/products/ip_enabled/ip_boards.htm.)

The CSF uses a directory called Compatibility to segregate operating-system-specific functionality into a single directory structure. This increases readability by eliminating the overhead associated with platform-specific preprocessor definitions. Functionality that requires platform-specific implementations is abstracted using a generic function name. (For information on framework compatibility, see the Compatibility section.)

Third-Party Components

The CSF makes extensive use of third-party components. For more information on these, see Appendix A.

Dialogic-Specific Components

Dialogic® Telecom Board and Dialogic® Host Media Processing Software Support

Most framework components are designed to work with Dialogic® telecom boards and Dialogic® Host Media Processing (HMP) software. Dialogic offers its products in several form factors that can be combined to create telecom platforms of varying functionality and density. Dialogic® HMP software performs media processing tasks on general-purpose servers without using specialized hardware.

The CSF has classes to support the following Dialogic® hardware:

- **Network interfaces** – Dialogic® Loop Start Interface (LSI), Dialogic® Global Call, Dialogic® Springware T-1 CAS, H.323, SIP boards, Dialogic® Integrated (DI) boards
- **Station Sets** – Dialogic® Modular Station Interface (MSI), Dialogic® High-Density Station Interface (HDSI), DI boards
- **Voice** – Springware boards and Dialogic® DM3 boards
- **Fax** – Springware and DM3
- **Conferencing** – MSI, Dialogic® Conference Bridge (DCB), DM3 conferencing, Dialogic® Continuous Speech Processing Technology, and Dialogic® conferencing boards.

- **Host Media Processing** — Dialogic® Host Media Processing Software Release 1.x for Windows and Linux or later

Dialogic® System Release Software

Dialogic System Release software provides the drivers and libraries used in conjunction with Dialogic® telecom hardware. The CSF has been designed to be used with Dialogic System Release 5.x for Windows and Linux or later.

Communications Services Framework Structure

The CSF distribution consists of framework components, component test harnesses, and demonstration programs. This section describes the structural concepts and the pieces of the CSF.

Project Workspace

The CSF workspace consists of component test harness projects and demonstration program projects. Component test harness projects are simple programs intended to test the functionality of the components included in the project. Demonstration program projects show specific functionality and implementation concepts, as well as serve as the basis for performance testing or as telecom utilities.

As the developer creates new framework components, it is generally the case that the components are integrated into a component test harness prior to their integration into a demonstration program project. The component test harness is used to verify the correct component operation including inter-component communication. After components are verified in their respective test harness, the developer can integrate them into a demonstration program, usually with minimal effort. For a detailed list of component test harnesses included in the CSF, see Appendix B: Component Test Harnesses.

Demonstration programs are created to show specific functionality and sample code and to create simple telephony systems. For a detailed list of the demonstration programs available in CommSvcFw version 0.5.5., see Appendix C: Demonstration Programs. For up-to-date information on demonstration programs, visit the CSF SourceForge project website (see <https://sourceforge.net/projects/commsvcfw>).

Classes and File Relationships

In general, framework classes are declared and defined in separate files. The exceptions to this design guideline are the classes that are completely implemented in a header file, such as CDlgcTerminationParameterTable, or when multiple structures and/or classes are implemented in the same file as a class that uses the aforementioned structures and/or classes. While having many separate files results in a larger number of files in the overall framework, this approach provides more flexibility when a class is added or removed from projects or the CSF. Files that are part of a higher-level component are all contained in the same directory. This allows for easy modification of a project by adding or removing entire folders to or from a project.

Environment Variables

Framework installation instructions are provided in Appendix C of this document. During the configuration of the CSF, environment variable creation is required. Environment variables are used by the CSF to decouple the location of include and library files from the successful compilation and linking of framework projects. For a list of environment variables used by the CSF, see Appendix D.

Preprocessor Definitions

The CSF occasionally uses preprocessor definitions to include or exclude certain source code modules and/or functionality from components and existing projects. To alleviate any overhead of determining the inclusion and exclusion of components and functionality, preprocessor definitions are grouped into the \Compatibility\FWConfig.h file.

These preprocessor definitions are used in the CSF:

- **`__SR511FP1_Or_Greater__`** — Used to specify the inclusion or exclusion of modules that require Dialogic System Release 5.1.1 Feature Pack 1, Dialogic HMP software version 1.x or later. This preprocessor definition can be found in the \Compatibility\FWConfig.h file. This preprocessor definition is targeted to be phased out in CommSvcFw version 1.5.0, to be replaced with the `GC_VOIP_SUPPORT` preprocessor definition. Refer to the CSF SourceForge project website (<https://sourceforge.net/projects/commsvcfw>) for recent version information.
- **`JRTPLIB_RTPSTACK`** — Used to specify the inclusion or exclusion of the J RTP RTP Library for

VoIP audio streaming. Refer to the JRTP RTP Library section for information on the JRTP Library. This preprocessor definition is set via the project settings, but is targeted to be moved to `\Compatibility\FWConfig.h` in `CommSvcFw` version 1.5.0. Refer to the CSF SourceForge project website (<https://sourceforge.net/projects/commsvcfw>) for recent version information.

- **RVRTP_RTPSTACK** — Used to specify the inclusion or exclusion of the RadVision RTP Library for VoIP audio streaming. Refer to the RadVision RTP Stack section for information on the RadVision RTP Library. This preprocessor definition is set via the project settings in projects that use the RadVision RTP stack.

Other preprocessor definitions are used at a component level. These definitions, which are set in the respective component header file, are not discussed further in this application note.

Theory of Operations

The CSF is based on an event-driven state machine. Architecturally, the CSF is comprised of three major components: event sources, event sinks, and event processors that dispatch events from sources to sinks.

The CSF contains many abstract classes, including classes for devices, such as telephony network interfaces, and states, such as a network call state machine. The abstract class provides the public interface that defines how the concrete class specializations are accessed. Concrete specializations of the abstract device classes are used to implement the primitive instructions to the underlying telecommunications platform that are required to complete the public interface operations of the abstract class. The CSF uses design patterns as the cornerstone of event processing, both to eliminate error-prone finite state machine tables and to provide the communication between loosely coupled devices that engage in the tightly coupled interactions often associated with telecommunications applications.

The `main()` function of an application based on the CSF is used to start at least one event processor, logger, and application. The CSF requires that applications be a specialization of an application class called `ADlgcApplication`. The `main()` function instantiates the event processor(s), logger(s), and specialized application class. These objects will execute their state machines until an exit command is issued. When the exit command is

issued by the system administrator, the `main()` function shuts down the application(s), logger(s), and event processor(s) that it instantiated.

All framework solutions deliver a single system event processor and a single system logger to a single telephony application. The `main()` function provides the application with an event processor and a logger. The application manages *resources* that are specified in the application configuration file. During the startup of the application, each managed resource is provided with an event processor and logger. The managed resources receive communications and commands via the event processor provided by the application.

Managed resources include any C++ class that can generate and/or react to events from the event processor. Managed resources can notify other resources of specific events via the event processor. Technology-specific event sources deliver events, event notifications, and commands to the system event processor(s). The system event processor delivers events, event notifications, and commands to the destination managed resources, called targets. Each managed resource has a state machine that is defined at compile time via the resource's abstract class. Managed resource state machines must include a minimum of initialize, open, and close functions.

The application is responsible for managing its own resources. These resources are referred to as *managed objects*. Managed objects interact independently from all other managed objects. The behavior of a managed object is specified by the state machine that is defined by the collection of states and state connections that are implemented in the abstract class state machine. Communication between managed objects is accomplished via the event processor in a form of event notifications or the issuance of commands.

The event processor uses the Command design pattern to unify events, event notifications, and class function calls. The event processor is a single event queue that serializes commands from numerous command sources and processes them with a worker thread. To avoid data synchronization issues, each command target is associated with one (and only one) event processor so that all commands are handled in a single thread.

To process a command in an event processor, a command object is obtained from a pool and queued to the target. The target queues the command to the event processor,

which pushes the command into an asynchronous processing queue. The event processor worker thread pops commands from the queue and calls the command object's `Execute()` function. The command object's `Execute()` function implementation provides stimulus to the command target object.

Command target behavior evolves as commands are issued. This is accomplished via the *State* pattern. The State pattern is used for verification that commands delivered to an object are appropriate for the current state of the object. The target receives the command and delegates execution of the command to its current state. If the command is valid for the current state, the state delegates control back to the target. If the command is not valid for the current state, an error condition is logged.

Events are specializations of a command and are delivered to an event handler. An event handler is a specialization of the command target. Various scenarios require a command target other than the original event target to request event notification of a particular event on a particular managed resource. When this happens, the third-party resource registers with the particular event handler for notification when the specific event has occurred. Using the Observer pattern, the third party is referred to as an *observer*, and the resource it is observing is the *observer subject*. When resources receive events, they may notify observers of the particular event. Each observer notification results in an event that is queued to the event processor. The command targets respond to the notification after the command is pulled from the event processor queue and delivered to the target.

Now that we have described the basic operation of the CSF at a high level, we consider a lower-level description of the CSF design. The following sections provide an overview of the *design patterns* used for basic framework operations. You can find developer-level detail on the design patterns as used in the CSF in Appendix E: Design Pattern Details.

State Pattern

The State pattern eliminates the need for the CSF to use look-alike conditional or large-case statements to define a C++ class state machine. The state machine of a class is implemented using a finite number of state classes. A state transition of a class is accomplished by setting the class member `_state` to point to a different state object. As

the state of an object changes, the behavior of the object changes by allowing only specific commands to be valid in each specific state class. Inserting a new state, or modifying valid transitions for existing states, can be accomplished by defining a new state class or modifying the class member `state` of an existing class.

The CSF uses abstract classes to define the interface to a concrete class as well as to maintain the concrete class state machine. The abstract classes allow an application to easily use differing concrete subclasses of a particular abstract class without significant changes. The concrete classes contain the primitive implementations of the interface that is specified in the abstract super class(es).

Observer Pattern

The Observer pattern allows the CSF to implement classes that are decoupled from other classes with which the framework may communicate. An event or change to an observer subject can be propagated to one-to-many observers that have registered for notification of the specific event. The subject is not required to make assumptions regarding who or what the observers it communicates with are. The Observer pattern allows the CSF to decouple classes that are not required for basic functionality of the classes. Although these classes may be required to communicate frequently for application-specific functionality, loose coupling of the CSF classes provides a mechanism to easily insert disparate classes into unrelated application domains.

Command Pattern

The Command pattern is used for the unification of events, event notifications, and function call classes. The Command pattern requires classes to implement a common interface to support high-level operations that are built upon primitive operations. This common interface allows the event processor to invoke events, event notifications, and function calls in the same manner.

Command Processing

The CSF event processor facilitates the Command, Observer, and State patterns. The Command pattern provides the means for the unification of events, event notifications, and function calls into a single event processing queue. An event is a specialization of a *command*; an event handler is a specialization of a *command target*. The Observer pattern provides

communication between application layers and interclass communications. The State pattern maintains deterministic class behavior.

The event processor receives events and commands from event sources. Event sources may be the Dialogic® Standard Runtime Library (SRL), VoIP stacks, IP communication pipes (that is, sockets), or other classes that notify observers of events. Event sources package individual events and commands into a pooled *framework event object*. The data of the event is packaged into a pooled *framework event data object*, which is then passed into the *event object*. The event object is then posted to the event processor assigned to the event source. Each event processor has one thread for processing commands and events. Implementation of thread pooling is a future goal that is not implemented. Such an implementation would likely take the form of a single dispatcher that dispatches commands and events to an available thread from a pool of threads.

When the event processor pulls a command or event out of its queue, it calls the common interface (the `Execute()` function) that is specified in the Command pattern. The implementation of a command depends upon the specialization of the command class. The implementation for an event is a call to the target's `HandleEvent()` function. When the command target (or event handler in the case of an event) receives the command or event, it delegates the command to the target's current state as specified in the State pattern. The state validates the correctness of receiving the specific command or event given its current state. If the command or event is valid for the current state, the state object delegates the work back to the target for processing via a primitive function call. If the command or event is not valid for the current state, the abstract implementation of the state class is used to log an error condition.

In some cases, an event results in observer notifications for the specific event. Observer notifications can also be used as an external interface to other objects. Objects receive observer notifications via the event processor in the same way that events are received. After events are handled by the target event handler, the event is unused and returned to the event pool for future use. The use of smart pointers to return the object to the event pool when it goes out of scope is a proposed enhancement that is not implemented.

Error conditions in the CSF generate a simple error message that is logged by the target's logger. A negative value is returned by the state to the calling function. Exception and Error handling are areas a framework user will want to harden for a production application.

Namespace

The CSF is encapsulated in a C++ namespace called `DlgcFW2`. References to the CSF include the classes, structures, and variables defined within the scope of the `DlgcFW2` namespace. Demo programs and component test harnesses have been implemented and some are shipped with the CSF distributions as sample code. Classes defined outside the `DlgcFW2` namespace are not included in the CSF.

Application Development Considerations

High-Level Considerations

A conceptual-level understanding of the CSF is essential to efficiently designing applications based on it. Before you have a complete understanding of the CSF, you can use the source code from the demonstration programs or component test harnesses as the foundation for elementary applications. You can become familiar with the CSF by executing existing programs with a debugger and reading the application trace log file. Experience has shown that comparing the log files generated by executing the existing component test harness projects with the source code to see how the functionality was implemented is an effective way to become familiar with the CSF operations.

Managed objects are software components that are required for the correct operation of a framework-based application. These objects are managed by the `ADlgcApplication` specializations. The `ADlgcApplication` specialization will not enter the *running* state unless all managed objects are successfully opened. After each managed object is opened, it will enter the *idle* state – ready to respond to commands, including unsolicited events (for example, a call offered event). The CSF contains a class called `ADlgcManagedObject` as the super class for classes that are specializations of both `ADlgcEventHandler` and `ADlgcObserverSubject`. Some examples of existing `ADlgcManagedObject` specializations are listed in Table 1.

ADlgcManagedObject	Description
ACSInterface	Component that provides an interface for remote controlling of managed objects
ADlgcApplication	Abstract application component
ADlgcDspRes	Abstract DSP resource component for conferencing
ADlgcFax	Abstract component for fax resources
ADlgcIPStream	Abstract component for IP streaming
ADlgcNet	Abstract component for network devices
ADlgcSipCall	Abstract component for SIP communication
ADlgcVox	Abstract component for voice devices

Table 1. ADlgcManagedObject Specializations

Framework-based applications use a configuration file to determine which types of managed objects to instantiate for a particular application configuration. Other managed objects that may be required by specific applications not specified in the configuration file may be specified at compile time.

The main() function starts the event processor(s) and logger(s) before it instantiates the ADlgcApplication specialization(s). Following the instantiation of the ADlgcApplication specialization, the main() function initializes the ADlgcApplication specialization via the ADlgcApplication::Initialize() function. The ADlgcApplication object then uses the State pattern to execute ADlgcApplication::PrimInitialize(), which creates managed objects. The CSF following provides a basic implementation:

```
int ADlgcApplication::PrimInitialize()
{
    EventProcessor(CDlgcEventProcessor::Instance("DLGC_APP"));
    Logger(CDlgcLogger::Instance("SYSTEM_LOG"));
    EventProcessor()->Logger(Logger());
    EventProcessor()->RegisterForNotification(this,
        DLGC_EVENT_PROCESSOR_STARTED);
    EventProcessor()->RegisterForNotification(this,
        DLGC_EVENT_PROCESSOR_STOPPED);
    CreateManagedObjects();
    return 0;
}
```

The ADlgcApplication specialization(s) should implement the CreateManagedObjects() function to parse the configuration file(s) and instantiate the appropriate managed objects as specified in the configuration file. The CreateManagedObjects() function may also instantiate other objects that may be required by the application but are not specified in the configuration file. These other objects may be singletons or managed objects that are always required by the application, regardless of the system configuration (for example, an IP socket or sockets). Most component test harnesses use an elementary component called a CallFlow, which is not included in the DlgcFW2 namespace. The CallFlow component envelopes network and/or media components, as well as pointers to other objects that are required to execute the component test harness managed objects. These CallFlow implementations are not meant to be used in complex application implementations, but rather used to quickly create simple applications (for example, implementations of CreateManagedObjects() function used in some existing component test harnesses and demonstration programs; see Appendix G: CreateManagedObjects() Examples for more information).

Following the return of `ADlgcApplication::CreateManagedObjects()` function, the `ADlgcApplication` component opens all the managed objects it contains. If all the managed objects, event processor(s), and logger(s) are successfully opened, a `DLGC_APPLICATION_STARTED` event notification is generated. The `ADlgcApplication` specialization implements the `ApplicationStarted()` function to kick off application-specific behavior. The elementary example implementation, which uses `CallFlow` objects to envelope and control network and media devices, iterates through each of its `CallFlow` objects and calls the virtual `CallFlow::OnApplicationStarted()` function. This is shown below:

```
void CHdsiObjApp::ApplicationStarted()
{
    LogMessage("ApplicationStarted", "", DEBUG_LOG_LEVEL);
    CDlgcCallflowList::iterator i;
    for ( i = _callFlowList.begin();
          i != _callFlowList.end();
          i++ )
    {
        (*i)->OnApplicationStarted( this );
    }
}
```

A complex application needs to do more in its `ApplicationStarted()` implementation. When designing any specific application, it is important to understand when and where components are instantiated, opened, and readied for general use. Managed objects and singletons can generally be instantiated in the `CreateManagedObject()` function. All managed objects are opened by the `ADlgcApplication` object after they are created. If all managed objects successfully open, the application starts all the components. After managed objects and singletons are started, the `ApplicationStarted()` implementation performs the remaining tasks required to prepare all components for the application running state. The application remains in the application running state until the system administrator initiates a controlled shutdown by using the <CTRL>+C keystrokes.

Component Test Harnesses

The component test harness projects all use either `CallFlow` or `BasicCall` components to maintain a specific application behavior. These components are not suggested as a BKM, but intended for use in creating simple application functionality. Demonstration programs do not use the `CallFlow` or `BasicCall` components. Instead, they use more complex and efficient mechanisms to control application behavior. Each `CallFlow` or `BasicCall` implementation is a specialization of the `ADlgcApplication` component and usually contains the following:

- **Network device** — One network device per `CallFlow` or `BasicCall` class. Each call flow maintains application-specific behavior and interfaces for the network device.
- **Voice device** — One network device per `CallFlow` or `BasicCall` class. Each call flow maintains application-specific behavior and interfaces for the voice device.
- **Event processor** — One event processor per `CallFlow` or `BasicCall` class. The event processor is used to send and receive commands, events, and event notifications.
- **Logger** — One logger per `CallFlow` or `BasicCall` class. This is used to generate messages to a log file.
- **Application-specific functions, classes, and/or pointers** — Other components required for specific application behavior. These components might be socket-based command acceptors, resource manager singletons, or graphical user interface components.

Configuration Files

The existing framework projects use a configuration file to specify configuration options during application start-up. The Common directory contains the classes used to parse the configuration file and pass information to the application. The configuration file is parsed inside the `ADlgcApplication::CreateManagedObjects()` function by configuration classes designed to parse specific configuration file sections. Creating a new configuration file section is accomplished by creating a configuration file section and configuration file item pair of classes, then adding the new configuration file section label to the `ADlgcConfigurationItem` header and implementation files. The configuration file section class is used to parse its section of the specified file. The configuration file item class holds the contents of a line item specified in its configuration file section. Configuration file parsing uses two guidelines: a semicolon signifies a commented line; an end-of-line character signifies the end of a line item.

You can find a list of configuration file sections in Appendix H: Configuration File Sections.

Logging

Framework logging is designed using the Bridge pattern. The Bridge pattern allows for the decoupling of an abstraction from the implementation, enabling the abstraction and the implementation to vary independently. This is convenient because logging is an area that organizations may want to customize to their own needs. The Bridge pattern allows the implementation to change without changing the application or framework levels.

The default framework logging implementation uses the `log4cplus` open-source library. `Log4cplus` uses a configuration file, called `log4cplus.properties`, to specify maximum file sizes, daily file rollover information, and many other configurable properties. For more information on `log4cplus`, refer to the `Log4cplus` section.

An alternative to the default logging implementation is a simple logging solution that creates a log file during application startup and continues to append messages to the log file until the application is shut down. Both logging implementations have a similar output. The creation of custom logging implementations, which requires minimal effort, is detailed in Appendix I: Custom Logging Implementations.

Minimum Components

A minimum set of components is required for any application or component test harness based on the CSF. The minimum components maintain the minimum requirements and design guidelines of the CSF. These components are:

- **Application** — The Application component includes the abstract application and application state classes as well as concrete application states and the `CDlgcManagedObject` class. These classes form the Application component and provide the mechanism to application and application resource initialization, startup, and shutdown.
- **Common** — The Common component includes the abstract configuration item class as well as the concrete configuration section and configuration item classes. These classes form the Common component and provide the mechanism to enable configuration file functionality for framework-based applications.
- **Compatibility** — The Compatibility component includes the classes required to provide seamless portability between the Windows® and Linux operating systems.
- **DlgcCommand** — The `DlgcCommand` component includes the abstract network, voice, fax, and time division multiplex (TDM) routing commands as well as the concrete command classes that enable the queuing of managed object function calls. Although this component is not always required for the single threaded event processor component, it is required after the multi-threaded event processor is implemented in `CommSvcFw` version 1.5.0. See the CSF SourceForge project website (<https://sourceforge.net/projects/commsvcfw>) for version information.
- **Event Processor** — The Event Processor component includes the abstract command, command target, event, event data, event handler, and event source classes as well as the concrete event, event processor, and SRL classes. These classes form the Event Processor component, which drives the event driven asynchronous applications based on the CSF.
- **Observer** — The Observer component is comprised of the abstract observer subject class required for Observer Notifications.
- **Resource Manager** — The Resource Manager component includes the abstract resource class and the concrete resource manager class. These classes form the Resource Manager component that manages

the availability and reservation of managed resources. In addition, the Resource Manager component provides query interfaces for identifying individual resources based on resource type, ID, name, and resource group.

- **Registration** — The Registration component is comprised of a concrete registration class that maintains a registry of event handlers that have registered to receive event notifications for a particular observer subject.

Application

The Application component is discussed at a high level in the previous section and in more detail in the `DlgcFW2::ADlgcApplication` section. The `ADlgcApplication` class handles the initialization, startup, and shutdown of applications and their managed objects.

You may want to take note of possible enhancements for this component. For example, when all managed objects are not opened successfully, the application stays in the opening state and does not reach the running state. Error and exception handling in the CSF should be considered for hardening before releasing a production application. Another potential improvement is the shutdown procedure. You may want to consider providing a graceful shutdown instead of a kill-style shutdown feature, as well as restart functionality.

Common

The Common component is discussed at a high level in the Configuration Files section. The overhead required for adding new common subcomponents is minimal. In general, adding a new section to a configuration file requires two classes, one to parse the configuration file section, and another to pass the parsed information to the application specialization. The configuration file parsing classes are very simple. The classes find their respective section label in the specified configuration file and parse each individual line as individual section line items. The line item ends at an end-of-line character and a semicolon marks a comment. Each line item passes the parsed line item data to a configuration item, which is then passed to the application. It is the responsibility of the application specialization to understand the semantics of each individual piece of data.

Compatibility

The Compatibility directory is used to group platform specific implementations into a single directory. Framework components outside the Compatibility directory refer to platform-specific implementations using a generic function name. The Compatibility directory then contains the generic function signatures, operating system preprocessor definitions, and operating-system-specific implementations for each generic function. The CSF uses the Compatibility directory to support these functions:

- **<CTRL>+C Handler** — Initiates application shutdown via the system keyboard.
- **Current Thread Identification** — Provides current thread identifier to calling function.
- **Device User I/O** — Enables applications to install and retrieve user-defined I/O functions on a per-channel device basis.
- **File Management** — Performs file open and close functionality.
- **Host IP Addressing** — Returns local IP address to calling function.
- **Telecom Event Handling** — Sets Telecom event handler callback function.
- **Logging Messages** — Provides log file message formatting.
- **Sleep** — Puts a thread to sleep for a specified amount of time.
- **Time Interval Conversion** — Returns the number of seconds between now and the specified time.
- **Time Stamps** — Provides time stamp formatting.
- **Timer Support** — Timer service generates an event in a specified amount of time.

DlgcCommand

The `DlgcCommand` component uses the Command pattern to implement a mechanism for the queuing of function calls. The `DlgcCommand` component is comprised of network, voice, fax, and SCbus commands. These `ADlgcCommand` specializations replace individual function calls on managed objects. The `DlgcCommand` component is discussed in more detail in the Command Pattern section.

Event Processor

The Event Processor component is discussed at a high level throughout this application note. There is further discussion of the Event Processor component in the Command Pattern section. In general, the event processor is a single event queue that uses a single thread to process events. Events are posted via the `PostEvent()` function, commands are queued via the `QueueCommand()` function, and the event processor processes one item at a time. The event processor is converted to a multi-threaded dispatcher, with the goal for implementation being version 1.5.0 of the CSF. Refer to the CSF SourceForge project website (<https://sourceforge.net/projects/commsvcfw/>) for recent version information.

The following source code shows the single queue implementation of the event processing algorithm:

```
void CDlgcEventProcessor::ProcessQueue()
{
    ADlgcCommand* event = NULL;
    PostEvent(CDlgcEvent::GetAvailable(this, DLGC_EVENT_PROCESSOR_STARTED));
    while (_runFlag || (_eventQueue.size() > 0))
    {
        boost::mutex::scoped_lock lock(_semaphoreMutex);
        {
            event = NULL;
            if (_eventQueue.empty())
                _semaphoreCondition.wait(lock);
            {
                boost::mutex::scoped_lock lock(_criticalSection);
                if (!_eventQueue.empty())
                {
                    event = _eventQueue.front();
                    _eventQueue.pop();
                }
            }
        }
        if (event)
        {
            event->Execute();
        }
        else
        {
            LogMessage("ProcessQueue", "Running", "NULL event");
        }
    }
    DirectNotifyObservers(DLGC_EVENT_PROCESSOR_STOPPED);
}
```

Observer

The Observer component is used when an object needs to be notified of an event or change on another object without needing to know who or what that other object is. Observer notifications provide an interface for decoupled inter-object communication.

Resource Manager

The `CDIgcResourceManager` class is a singleton that manages the availability and reservation of managed resources. In addition to interfaces for managing and reserving resources, it provides query interfaces for identifying individual resources based on resource type, ID, string identifier, and resource group ID. Queries resulting in lists of resources are also available against the resource pool by resource type and reservation status.

For resources to be managed, they need to be a subclass of `ADIgcResource`. A dynamic cast should be used on resources obtained from the resource manager for conversion to an appropriate type. The type associations are:

- `NET_RESOURCE_TYPE`
- `VOX_RESOURCE_TYPE`
- `CONF_RESOURCE_TYPE`

Concrete subclasses of `ADIgcResource` may implement static class helper functions to eliminate the need for the application layer to deal with the dynamic casts.

Resource reservation is not enforced. It is by common agreement that only objects that perform a successful reserve operation may control the resource. The typical scenario for using the resource manager is:

- An object “monitors” a resource by querying the device to obtain a pointer to the resource and registering for observer notifications. (Any number of other objects may also monitor the resource.)
- An object “controls” a resource by reserving it. Only the first reserve operation succeeds.
- An object relinquishes control by performing an unreserved operation.
- No action is necessary to abandon monitoring a resource other than to unregister for observer notifications.

Registration

The Registration component, `CDIgcRegistration`, is a singleton used to obtain a pointer to an event handler object based on an ID. SRL devices use the device handle for registration; all other objects use the object memory address. A main use of the Registration component is by the SRL event handler to find the event target with that device handle. It is also used in the CSP Stream UIO and `ec_stream()` callback functions for the same purpose. The Vovida SIP integration also has a registration singleton object, `CDIgcVovidaSipRegistration`, that maps SIP call leg information to SIP call objects.

Creating New Components

The creation of new components can occur after one has an understanding of framework functionality and object-oriented design. Singletons are used when only one instance of the component should be instantiated per application. Managed objects are used when many instances of the component should be instantiated per application, and if the object is both an event source and event handler. Otherwise, if the component is only an event source or only an event handler, it should be designed accordingly. In general, the place to instantiate new components is in the `ADIgcApplication::CreateManagedObjects()` function.

Inter-object communication should be implemented with observer notifications instead of designing classes to have pointers to objects that should be decoupled from each other. The coupling of classes that should be independent from each other results in excessive overhead when minor functionality changes are required in an application.

If there are multiple specializations of a particular class, then an abstract class should be created. The abstract class provides the public interface to all specializations. The abstract class should also maintain the state machine for all specializations. In this way, the specialization implementation requires only a minimal effort.

Following the design and implementation of new components, a component test harness should be created to verify that the new component(s) communicate correctly with the event processor, logger, application, and other components. The component test harness also provides a mechanism to test minor functionality changes, as well as the verification that various specializations of the new abstract component communicate with the application consistently.

Putting It All Together

Creating a project that requires the functionality already provided with existing framework components is a simple matter. Create the project, add the appropriate folders and files, set up the project settings, and compile. If required functionality is missing in the CSF, the CSF design guidelines should provide the direction to solve most basic issues.

One way to create a new component is to first determine the category to which the new component belongs (event source, event handler, managed object, singleton). After determining the component category, copy and paste an existing object of the same category and begin a Find-and-Replace session to rename the copied class to the new class name. Remove unnecessary interfaces and implement the required interfaces for the new component.

Extensions, Enhancements, and Making Your Own Application

Creating a component test harness can be very different than creating a hardened production application. Component test harnesses or elementary telephony applications can follow most design guidelines illustrated in existing projects. Making a hardened production application is a larger task. Error and exception handling is an important area to examine. Extensions and/or enhancements to existing component(s) are generally less of an undertaking if you follow the existing guidelines.

Existing components have a simple state machine. Adding more states and a larger or more complex interface may require new functionality. The CSF is designed and implemented using the spiral-iterative approach, so most implementations contain only the functionality required for the task at hand.

New Functions to Existing Classes

Adding new functions to existing classes requires a few steps. If the new function will be required for multiple specializations, then the public function interface is specified in the abstract class and the abstract state class. A protected function interface is also added to the abstract class as a pure virtual function call. The protected function interface should be the same as the public, but the function name is preceded by a “Prim” to signify that the function call is a primitive function call. The abstract state class function call has the same interface as the abstract class function call, but the parameters are preceded with a pointer to the abstract class. The abstract implementation of the public function call requires control to be delegated to the current state. This is shown below:

```
int ADlgcVox::Play( const char* playfile, CDlgcTerminationParameterTable* tpt )
{
    LogMessage("Play", playfile, DEBUG_LOG_LEVEL);
    if (_active)
        return _state->Play( this, playfile, tpt );
    else
        return -1;
}
```

The implementation of the abstract state class is an error condition. The CSF's default error condition implementation is shown below:

```
int ADlgcVoxState::Play( ADlgcVox* subject, const char* playfile,
CDlgcTerminationParameterTable* tpt)
{
    subject->LogMessage("Play", "invalid command for state", ERROR_LOG_LEVEL);
    return -1;
}
```

States that are valid for the new function call must have an overriding implementation that delegates control back to the calling object. An example of this follows:

```
int CDlgcVoxIdleState::Play( ADlgcVox* subject, const char* playfile,
CDlgcTerminationParameterTable* tpt)
{
    subject->SetStateToPlaying();
    if ( subject->PrimPlay(playfile, tpt) == -1 )
    {
        char tbuf[ 128];
        sprintf(tbuf, "failure playing: %s", playfile);
        subject->LogVoiceDeviceError("Play", tbuf);
        subject->SetStateToIdle();
        return -1;
    }
    return 0;
}
```

After reviewing the example, note that a new state or states may be necessary to support the added functionality. If this is the case, then you must add a new state pointer(s) to the abstract class. The new states are instantiated and destroyed in the same place that the other states are instantiated and destroyed – in the constructor and destructor, respectively. The `SetStateToNewState()` function(s) must be added to the abstract class and implemented in the states that the new function is valid. The specializations of the abstract class must implement the pure virtual primitive functions to support the new functionality. In the example above, the specializations would have to implement `PrimPlay()`. This is shown below:

```
int CDlgcVox::PrimPlay(const char* file, CDlgcTerminationParameterTable* tpt)
{
    LogMessage("PrimPlay", file, DEBUG_LOG_LEVEL);
    if (_active)
    {
        int fd;
        if ((fd = DlgcFW2::fw_fileopen(file, O_BINARY | O_RDONLY)) == -1)
        {
            LogMessage( "PrimPlay", "fw_fileopen error",
                ERROR_LOG_LEVEL);
            return -1;
        }
        . . .
        if (tpt)
        {
            DvTpt = *tpt;
        } else {
            DvTpt.Clear();
            DvTpt.AddTermParameter( DX_MAXDTMF, 1, TF_MAXDTMF, 0 );
        }
        DV_TPT* pTpt = DvTpt.Dv_Tpt().get();

        if (dx_playiottdata(ID(), &_iott, pTpt, &_xpb, EV_ASYNC) == -1)
    }
}
```

```
        {
            LogMessage(    "PrimPlay - dx_playiottdata",
                          ATDV_ERRMSGP(ID()), ERROR_LOG_LEVEL);
            return -1;
        }
    return 0;
}
```

Adding Existing Hardware Components to Your Application

Adding existing hardware components to an application requires minimal overhead. To add support for existing hardware components, add the call to instantiate the new component to the `CreateNetDevice()`, `CreateVoxDevice()`, `CreateFaxDevice()`, etc., function. The `CreateManagedObjects()` function parses the configuration file and blindly passes the information to the appropriate creation function. Match the protocol specified in the creation function with the protocol specified in the configuration file to signal the instantiation of the new component. Finally, include the appropriate header files, add the appropriate folder(s) to the existing project, and add the new folder path(s) to the project settings so include files can be located during compilation.

Appendix A: Third-Party Components

Boost Thread Library

The CSF makes use of the open-source Boost Library for threading and synchronization support.

The Boost website states, “ Boost provides free peer-reviewed portable C++ source libraries. We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use.

We aim to establish “existing practice” and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee’s Library Technical Report (TR1) as a step toward becoming part of a future C++ Standard. More Boost libraries are proposed for the upcoming TR2.” For more information, see <http://www.boost.org>.

Log4cplus

The log4cplus open-source library is used to provide a default logging service to the CSF. The CSF also contains an alternative logging implementation, but the log4cplus library provides the flexibility and functionality desirable for many framework-based applications.

The log4cplus website states, “log4cplus is a simple to use C++ logging API providing thread-safe, flexible, and arbitrarily granular control over log management and configuration. It is modeled after the Java log4j API.” For more information, see <https://sourceforge.net/projects/log4cplus>.

Vovida SIP Stack

The CSF supports the open-source Vovida SIP stack as a reference implementation of a SIP stack. This stack can be used as-is, enhanced, or replaced with the developer’s stack of choice.

The Vovida.org website states, Vovida is “a communications community site dedicated to providing a forum for open-source software used in datacom and telecom environments. While we have predominantly included software to date that we have created, we are looking forward to adding more software from all of you as it is submitted to us for inclusion. If you are working on any type of software for broadband, wideband, or narrowband

environments for cable, xDSL, wireless, or other media, please contact us.” For more information, see <http://www.vovida.org>.

JRTP RTP Library

The CSF supports the open-source JRTP Library as a reference implementation of an RTP stack. This can be used as-is, enhanced, or replaced with the developer’s RTP stack of choice.

The JRTP Library website states, “ The library offers support for the Real-time Transport Protocol (RTP), defined in RFC 3550. It makes it very easy to send and receive RTP packets and the RTCP (RTP Control Protocol) functions are handled entirely internally. For more detailed information you should take a look at the documentation included in the package.” For more information, see <http://research.edm.uhasselt.be/~jori/page/index.php?n=CS.Jrtplib>.

Curl Library

The CSF uses the Curl Library for date and time parsing as part of its logging services.

The Curl Library website states, “curl is a command line tool for transferring files with URL syntax, supporting FTP, FTPS, HTTP, HTTPS, SCP, SFTP, TFTP, TELNET, DICT, FILE and LDAP. curl supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, cookies, user+password authentication (Basic, Digest, NTLM, Negotiate, kerberos...), file transfer resume, proxy tunneling and a busload of other useful tricks.” For more information, see <http://curl.haxx.se>.

RadVision RTP Stack

The CSF supports the RadVision RTP stack as a reference implementation of a production-quality RTP stack.

The RadVision RTP stack is only available after completing a non-disclosure agreement with RadVision. Contact your RadVision representative to find out more about obtaining the framework implementation for RadVision RTP. For more information, see <http://www.radvision.com/>.

Appendix B: Component Test Harnesses

Component Test Harness	Description
2bctObj	Component test harness for testing of two B-channel transfer components.
2bctSim	Two B-channel transfer simulation when a switch capable of two B-channel transfer is not available.
ConfigFile	Component test harness for verification of correct configuration file parsing.
DtiT1Obj	Component test harness for verification of correct CDlgcDtiT1Net digital circuit –switched network interface operation.
EventProcessor	Component test harness for verification of correct event processing operation.
FaxObj	Component test harness for verification of the correct operation of ADlgcFax and CDlgcFax fax classes.
GcObj	Component test harness for verification of the correct operation of all Dialogic® Global Call components, as well as components used for the specification of VoIP codec and call management.
HdsiObj	Component test harness for verification of the correct operation of High Density Station Interface (HDSI) and Dialogic® Integrated (DI) station set components.
Logging	Component test harness for the verification of the correct operation of logging services.
LsiObj	Component test harness for the verification of the correct operation of the CDlgcLsiNet loop start interface class.
MsiObj	Component test harness for verification of the correct operation of Modular Station Interface (MSI) station set components.
Observer	Component test harness for verification of the correct operation of observer notification generation and handling.
Registration	Component test harness for verification of the correct operation of observer notification registration and handling.
RemoteObj	Component test harness for verification of the correct operation of components related to the control of devices over an IP connection. These components include CRemoteNet, CRemoteVox, CCSContext, ACSInterface, ACSInterfaceContext, CRemoteContextCmd classes.
ResourceManager	Component test harness for verification of the correct operation of resource management.
ResponseSim	Component test harness for the simulation of pre-connect and post-connect termination reasons. This component test harness provides pre-connect simulations for busy, no circuit special information tone (SIT), operator intercept SIT, vacant code SIT, and reorder SIT. Post-connect simulations are for positive answering machine detection, positive voice detection, and fax detection.
RltObj	Component test harness for the verification of Release Link Trunk (RLT) call transfer components.
RvRtpVovidaSipObjCsp	Component test harness for the verification of RadVision RTP stack using the Vovida SIP stack and CSP components. This component test harness is only available after a non-disclosure agreement (NDA) is signed with RadVision.
SimObj	Component test harness for the verification of the correct operation of application level components. Simulation classes have proven to be very useful for the testing of application level components and are used to simulate hardware. Simulation components include CDlgcSimVox, CDlgcSimNet, and CDlgcSimFax.
VovidaSipCallsSink	Component test harness used to terminate inbound calls for simulation, analog, VoIP, and Dialogic Global Call network devices. This component test harness can be configured at compile time to drop connected calls after a specified amount of time or retain connected calls until the caller initiates the drop.
VovidaSipCallSource	Component test harness used to generate outbound calls for simulation, analog, VoIP, and Dialogic Global Call network devices. This component test harness can be configured at compile time to retain connected calls until the remote side disconnects or drop connected calls after a specified amount of time.
VovidaSipMsgObj	Component test harness for the verification of the correct operation of SIP messaging components. This component test harness uses the open-source Vovida SIP stack and can be used to test VovidaSIP components.

Appendix B: Component Test Harnesses (cont.)

Component Test Harness	Description
VovidaSipObjCsp	Component test harness for the verification of the correct operation of CSP components using the open-source Vovida SIP and JRTP stacks.
VovidaSipObjIpvs	Component test harness for the verification of the correct operation of IPVS components using the Vovida SIP stack.

Appendix C: Demonstration Programs

Demonstration programs are created to show specific functionality and sample code and to create simple telephony systems. This appendix shows a list of the demonstration programs available in CSF version 0.5.5. For up-to-date information, visit the CSF SourceForge project website (<https://sourceforge.net/projects/commsvcfw>).

Demonstration Program	Description
ConfObj	Simple conference server application that provides scheduled and dynamic conference resource management using Dialogic® Global Call Software and/or VoIP networks. The demonstration program with an application note, <i>Media Server Solution Recipe: Conferencing Applications</i> , is available (see the <i>For More Information</i> section).
GcMsiGateway	Simple gateway used for switching between modular station interface devices and Dialogic Global Call network devices.
VovidaSatellitePbx	Gateway using the MEGACO call model to perform switching between network devices supported by the CSF (simulation, analog, isdn, T-1 CAS, H.323, Vovida SIP, SIP under Dialogic Global Call, and Dialogic Global Call network). The VovidaSatellitePbx demo uses the VovidaSipGateway as its base source code, and provides remote controlling of devices via IP using proprietary command syntax. The command syntax can be replaced with different command syntax.
VovidaSipGateway	A simple Vovida SIP gateway that uses the Vovida SIP stack with the JRTP stack for audio streaming. Voice channel switching is performed between analog, SIP, and station set network devices.
RvRtpVovidaSipGateway	A simple Vovida SIP gateway that uses the RadVision RTP stack for audio streaming. The RvRtpVovidaSipGateway uses the VovidaSipGateway as the source code baseline, but the RTP components are replaced with more production ready components to provide higher quality audio streaming. This gateway application requires that a non-disclosure agreement with RadVision be instituted.

Appendix D: Environment Variables

Environment Variable	Description
BOOST_ROOT	Specifies the location of the root Boost directory
BOOST_DEBUG_THREADLIB	Specifies the location of the debug Boost thread library file
BOOST_RELEASE_THREADLIB	Specifies the location of the release Boost thread library file
CURL_ROOT	Specifies the location of the root CURL directory
CURL_LIB	Specifies the location of the CURL library file
FW2_RTP_INC_PATH	Specifies the location of the RTP stack include files
FW2_RTP_LIB_FILE	Specifies the name of the RTP stack library
VOVIDA_ROOT	Specifies the location of the root Vovida directory
VOVIDA_LIB	Specifies the name of the Vovida library file
LOG4CPLUS_ROOT	Specifies the location of the root log4cplus directory

Appendix E: Design Pattern Details

Dialogic has implemented many design patterns in this framework. This appendix describes in detail how the State, Observer, Command, and Singleton patterns are implemented.

State Pattern

The State design pattern is implemented in abstract classes in the CSF. The abstract class definition contains the public interface, the state class pointers, and protected pure virtual primitive operations that coincide with the public interface. These pure virtual operations include event handling functions as a secondary effect of the original function call.

In parallel with the abstract class definition is an abstract state class definition, with a public interface that very closely matches the abstract class definition. The abstract class constructor instantiates the appropriate state class object for each state class pointer. The public interface functions of the abstract class are implemented in the class declaration, and each calls the current state class function that mirrors the abstract class function called by the original function call.

DlgcFW2::ADlgcApplication

The DlgcFW2::ADlgcApplication abstract class defines the interface for application startup and shutdown. Specializations of the ADlgcApplication class receive event processor(s) and logging service(s) from the ADlgcApplication class client. Individual ADlgcApplication specializations maintain services and resources for an individual application.

DlgcFW2::ADlgcManagedObject

The ADlgcApplication class creates managed objects in the CreateManagedObjects() function. The ADlgcManagedObject is a stateless class but requires the implementation of HandleEvent(), Initialize(), Open(), and Close() functions. These functions initiate the State patterns of the objects discussed in the State Pattern section.

DlgcFW2::ADlgcNet

The DlgcFW2::ADlgcNet abstract class defines the interface for call control and device management. The ADlgcNet specialization primitive implementation is abstracted behind the state pattern, decoupling the application from the network device configuration.

DlgcFW2::ADlgcVox

The DlgcFW2::ADlgcVox abstract class defines the interface for voice devices. Voice devices are used for digit and tone detection and generation and for playing and recording files. The simulation voice device, called CDlgcSimVox, is used to generate termination events in response to public interface calls.

DlgcFW2::ADlgcSCbus

The DlgcFW2::ADlgcSCbus abstract class defines the interface for CTbus communication. CTbus communication is used for transmitting and receiving audio to and from the CTbus.

Command Pattern

The Command pattern is used to unify events, event notifications, and queued function calls. The classes that are implemented according to the Command pattern are implemented using a common public interface function, Execute(). This pattern decouples the method to invoke the operation from the implementation that performs the operation.

Event source and event handlers are critical for the operation of the CSF. The ADlgcObserverSubject class is the only specialization of the ADlgcEventSource class. Event Handlers must have the ability to retrieve events from a runtime event source (for example, the SRL), post events to the event processor, and handle events delivered from the event processor. The ADlgcEventHandler interface specifies a pure virtual function, called HandleEvent(), that must be implemented by event handler specializations.

Observer Pattern

The ADlgcObserverSubject class is a specialization of the ADlgcEventSource class. Observers may register with an observer subject for notification when a particular event or action has occurred. Each observer notification identifier is associated with a state change specific to the concrete observer subject. When the state change associated with an observer notification occurs, the concrete observer subject notifies all observers registered for that notification. The notification is an event posted to the observer's event processor. The public interface to the ADlgcObserverSubject class provides the mechanism for event handler classes to register and unregister for notification of a particular event using the RegisterForNotification() and Unregister() functions,

respectively. Event notification identifiers are located in the EventProcessor\DlgcEventDefs.h file.

Appendix F: Installation and Configuration

This appendix provides instructions for installing and configuring the CSF platform for the Windows® and Linux operating systems. Use the exact versions of third-party, open-source components, even if a later version is available on the third-party website. Using later-than-specified versions could introduce new installation or configuration steps and may cause problems during system configuring.

Linux Framework System Configuration

The Curl Library is provided with Red Hat Linux 7.2 and 7.3. The JRTP Library is not required for ATS FW2 installations on Linux. The order of installation for Linux follows.

1. Red Hat Linux
2. Dialogic System Release software
3. Boost Library
4. Vovida SIP stack
5. Log4cplus
6. CSF source code

Install Red Hat Linux

Refer to the Red Hat Linux installation instructions.

Install LiS Streams

Refer to LiS Streams installation instructions. LiS is not necessary for Dialogic® System Release 6.0 CompactPCI.

Install Dialogic® System Release Software

Refer to Dialogic System Release installation instructions.

Configure Framework for Linux

Install Boost Library

1. Obtain Boost Libraries from <http://www.boost.org> and download version 1.29.0 to a directory that will be referred to as <BOOST_ROOT>.
2. In the <BOOST_ROOT> directory, unpack the Boost tar file via:


```
tar xvfz boost_1_29_0.tar.gz
```
3. Build a jam utility by changing to the

<BOOST_ROOT>/tools/build/jam_src and execute make.

4. Copy bin.linuxx86/jam to /usr/local/bin or other convenient directory or (easier) obtain and install the jam RPMs via: `rpm -ivh boost-jam-3.1-1.i386.rpm`.
5. Copy the jam binary to <BOOST_ROOT> and change directory to <BOOST_ROOT>.
6. Build the Boost libraries via:


```
jam -sBOOST_ROOT=. -sTOOLS="gcc"
```

Note: Be very careful of the above syntax, especially about not putting a “space” between the `-s` option and its argument.

7. A successful build of the Boost Libraries will result in the following output:


```
...failed updating 43 targets...
   ...skipped 24 targets...
   ...updated 637 targets...
```

Install Vovida

1. Obtain Vovida SIP Stack from <http://www.vovida.org> and download version 1.4.0 to a directory that will be referred to as <VOVIDA_ROOT>.
2. In the <VOVIDA_ROOT> directory, unpack the Vovida tar file via:


```
tar xvfz sip-1.4.0.tar.gz
```
3. Build Vovida User Agent, by calling `make ua` from <VOVIDA_ROOT>.
4. It may also be necessary to do a `make sdp` from the sdp directory.

Install CFS

1. Unpack the CSF into a directory that will be referred to as <FW2_ROOT>.
2. For all makefiles, check and set (assuming dirs are set as per instructions):


```
DEV_ROOT=/root/boost_1_29_0
BOOST_ROOT=/root/boost_1_29_0
FW_DIR=${BOOST_ROOT}/Framework2_0.5.2
```

Windows®

Install Windows® Operating System

The CSF has been verified for these operating systems:

- Microsoft® Windows® 2000® Professional version 5.0.2195 Build 2195
- Microsoft® Windows® XP Professional version 5.1.2600 Service Pack 1 Build 2600

Install Microsoft® Development Environment

The CSF has been verified for these development environments:

- Microsoft® Visual Studio® version 6, Service Pack 5 (MSVC 6.0).
Note: Service Pack 5 *must* be installed.
- Microsoft® Development Environment 2003, Version 7.1.3088 and Microsoft .NET® Framework 1.1 Version 1.1.4322 (MSVC 7.1).

Install Dialogic® System Software and SDK for Windows®

Perform a custom installation selecting from these additional packages:

- ISDN
- Dialogic® Global Call
- Dialogic® DM3
- Dialogic® Continuous Speech Processing Technology
- Third-party software

Configure Framework for Windows®

Decompress the CommSvcFw zip file to a directory that will become your <FW2_ROOT> directory. **Note:** This will not become an environment variable.

Install Boost Library

The instructions listed in this section are located in <FW2_ROOT>\Boost_win2k.txt.

Download the Boost Library distribution from <http://www.boost.org> and unzip into a directory that will become the BOOST_ROOT directory and environment variable. This is normally Boost_n_n_n where n_n_n is the release number; for example, Boost_1_29_0 environment variable becomes:

```
BOOST_ROOT=C:\boost_1_29_0
```

Build Boost Jam Utility

1. Building bjam requires win32 environment variables set in the same command prompt window where bjam will be compiled:
 - [MSVC 6.0] <MSVC Install Directory>\Bin\VCVARS32.BAT
 - [MSVC 7.1] <MSVC Install Directory>\Vc7\Bin\VCVARS32.BAT
2. Change directory to <BOOST_ROOT>\tools\build\jam_src. If you are using MSVC 7.1, type “set JAM_TOOLSET=VC7”
3. Build the bjam utility via nmake -f builds\win32-visualc.mk. Success yields: “...updated 39 targets...”
4. Copy bjam.exe from <BOOST_ROOT>\tools\build\jam_src\bin.ntx86 to the <BOOST_ROOT> directory.
5. Modify <BOOST_ROOT>\jamfile
6. Uncheck read-only in the file properties list.
7. Exclude building the regex and python libraries by changing the jamfile to the following:

```
project-root ;
#subinclude libs/regex/build ;
subinclude libs/thread/build ;
#subinclude libs/python/build ;
```

Build Boost Threads Library

1. In the command window, change to <BOOST_ROOT> and execute the following to build the Boost threads library: bjam “-sTOOLS=msvc”. Success yields: “...updated xxx targets...”, where:
 - [boost_1_28_0] xxx is 201
 - [boost_1_29_0] xxx is 244
2. Set up Boost environment variables. The CSF uses three environment variables to locate the Boost include and library files. The default environment variables should be:


```
BOOST_ROOT=C:\boost_1_29_0
BOOST_DEBUG_THREADLIB=%BOOST_ROOT%\libs\thread\build\bin\libboost_thread.lib\msvc\debug\runtime-link-dynamic\threading-multi BOOST_RELEASE_THREADLIB=%BOOST_ROOT%\libs
```

```
\thread\build\bin\libboost_
thread.lib\msvc\release\runtime-
link-dynamic\
threading-multi
```

3. Verify project settings. Enable RTTI in project settings with the “Project Settings” → “C/C++” tab → “C++ Language” → “Enable Run-Time Type Information (RTTI)” checkbox.

Install Curl Library

Instructions are below and in <FW2_ROOT>\Curl_win2k.txt.

1. Obtain the CURL source code from <http://curl.haxx.se/>.
2. Unzip the distribution into a directory that will become the CURL_ROOT directory and environment variable, example “C:\Curl\curl-7.10.3” becomes:

```
CURL_ROOT=C:\Curl\Curl-7.10.3
```

Build Curl Library

1. Open the “<CURL_ROOT>\lib\curl.lib.dsw” MSVC workspace. [MSVC 7.1] Click “Yes To All” when prompted to convert all source files to the current compiler version.
2. Change “Project Settings” → “C/C++” → “Code Generation” → “Use run-time Library:” from “Debug Multithreaded” to “Debug Multithreaded DLL” for debug configurations and from “Multithreaded” to “Multithreaded DLL” for release configurations. This is equivalent to changing compiler switches \MTd to \MDd and \MT to \MD.
3. Build.
4. Make sure that the <CURL_ROOT>\lib\<Project Type>\libcurl.dll and <CURL_ROOT>\lib\<Project Type>\libcurl.lib are in a directory that is in your path. In this statement, <Project Type> refers to either “Debug” or “Release” — depending on the project version built.
5. Set up Curl Environment Variables. CSF uses two environment variables to locate the Curl Library files. The variables are as follows for a standard environment:

```
CURL_ROOT=C:\Curl\Curl-7.10.3
```

6. A second environmental variable is used to identify which build directory to use for linking the libraries. The variable is set as follows for a standard debug or release environment:

```
CURL_LIB=%CURL_ROOT%\lib\Debug
CURL_LIB=%CURL_ROOT%\lib\Release
```

Install JRTP Library

Instructions are below and in <FW2_ROOT>\JrtpLib_Win2k.txt.

1. Download the JrtpLib RTP Stack source code from <http://research.edm.uhasselt.be/~jori/page/index.php?n=CS.JrtpLib>
2. Unzip the distribution into a directory that will become the FW2_RTP_INC_PATH directory and environment variable, example “C:\jrtp-lib-2.7” becomes:

```
FW2_RTP_INC_PATH=C:\jrtp-lib-2.7
```
3. Open the Microsoft project workspace: <FW2_RTP_INC_PATH>\jrtp-lib-2.7.dsw. [MSVC 7.1] Click “Yes To All” when prompted to convert all source files to the current compiler version.
4. Build the desired configuration. It is suggested that the debug configuration is initially used.
5. Set up JRTP environment variables. The CSF uses two environment variables, FW2_RTP_INC_PATH and FW2_RTP_LIB_FILE, to locate the RTP stack include and library files. Variables are as follows for a typical environment:

```
FW2_RTP_INC_PATH=C:\jrtp-lib-2.7
FW2_RTP_LIB_FILE=C:\jrtp-lib-2.7\Debug\jrtp-lib-2.7.lib
```
6. To use the JrtpLib stack in CSF projects, JRTP_LIB_RTPSTACK must be defined in the “Project Settings” → “C/C++” → “Preprocessor Definitions”.

Install Vovida SIP Stack

Instructions are below and in <FW2_ROOT>\Vovida_Win2k.txt.

1. Obtain the Vovida SIP Stack source code from <http://www.vovida.org>. There is also a win32 precompiled User Agent. However, the full source is needed to build the libraries.

- Unzip the distribution into a directory that will be the VOVIDA_ROOT directory and environment variable; for example, "C:\Vovida\sip-1.4.0" becomes:
`VOVIDA_ROOT=C:\Vovida\sip-1.4.0`

Install Log4cplus

- Obtain the distribution from <http://log4cplus.sourceforge.net>.
- Unzip distribution into a convenient directory that will become LOG4CPLUS_ROOT.
- From the installation directory, execute `./configure`.
- Type "make".
- Type "make install".

Build the Vovida Libraries

- Open the <VOVIDA_ROOT>\Vocal.dsw MSVC Workspace. [MSVC 7.1] Click "Yes To All" when prompted to convert all source files to the current compiler version.
- For all projects in vocal.dsw, except adt, change the "debug" project settings using "Project Settings" → "C/C++" → "Code Generation" → "Use run-time Library:" from "Debug Multithreaded" to "Debug Multithreaded DLL". For all projects, except adt and spd2, change release project settings from "Multithreaded" to "Multithreaded DLL". This is equivalent to changing compiler switches \MTd to \MDd and \MT to \MD. Do not change single threaded projects to multi-threaded projects.
- Verify that Run Time Type Info is enabled. This is equivalent to changing compiler switches to include "/GR".
- Set the "ua files" project as the active project and build the desired configuration. It is suggested that the debug configuration be used initially.
- Copy <VOVIDA_ROOT>\contrib\win32\pthread\obj.debug.Win32.i686\pthread.dll to a directory in your path, or include <VOVIDA_ROOT>\contrib\win32\pthread\obj.debug.Win32.i686 in the path.
- Set up Vovida Environment Variables. The CSF uses an environment variable to locate the Vovida SIP

Stack include and library files. The variable is as follows for a standard environment:

```
VOVIDA_ROOT=C:\Vovida\sip-1.4.0
```

A second environmental variable is used to identify which build directory to use for linking in the libraries. The variable is as follows for a standard debug environment:

```
VOVIDA_LIB="obj.debug.Win32.i686"
```

For a standard release configuration, the variable is as follows:

```
VOVIDA_LIB="obj.nodebug.Win32.i686"
```

Install Log4cplus

- Instructions are in <FW2_ROOT>\log4cplus_Win2k.txt.
- Obtain distribution from <http://log4cplus.sourceforge.net>.
- Unzip the distribution into a directory that will be the LOG4CPLUS_ROOT directory; for example, "C:\log4cplus\log4cplus-1.0".
- Make source code changes specified in section 3 of <FW2_ROOT>\log4cplus_Win2k.txt.
- Open the MSVC++ workspace <LOG4CPLUS_ROOT>\msvc6\log4cplus.dsw and build the log4cplus_dll project for both debug and release configurations.
- Copy <LOG4CPLUS_ROOT>\msvc6\Debug\log4cplused.dll and <LOG4CPLUS_ROOT>\msvc6\Release\log4cplus.dll to a directory in your path.

Appendix G: CreateManagedObjects() Examples

HdsiObj::CreateManagedObjects()

CreateManagedObjects() is used in the HDSI component test harness to create station set network devices, voice devices, and their CDlgcCallFlow classes that handle the HdsiObj application behavior.

```

void CHdsiObjApp::CreateManagedObjects()
{
    char tempMsg[ 512];
    DlgcFW2::ADlgcNet* netobj = NULL;
    DlgcFW2::ADlgcVox* voxobj = NULL;
    CDlgcCallflow* callFlowObj = NULL;

    LogMessage("CreateManagedObjects", "", DEBUG_LOG_LEVEL);
    DlgcFW2::CDlgcLogger* appLogger =
        DlgcFW2::CDlgcLogger::Instance("SYSTEM_LOG");
    appLogger->SetLogLevel(DEBUG_LOG_LEVEL);
    appLogger->Start();
    _deviceConfiguration->ParseConfiguration("HdsiObjCfg.txt");
    DlgcFW2::CDlgcDeviceConfigurationList::iterator idev;
    for ( idev = _deviceConfiguration->DeviceList()->begin();
        idev != _deviceConfiguration->DeviceList()->end();
        idev++)
    {
        sprintf( tempMsg, "[ DEVICES] %d %s %s %s %s %s",
            (*idev)->GroupID(),
            (*idev)->Protocol(),
            (*idev)->NetDevice(),
            (*idev)->VoxDevice(),
            (*idev)->Info1(),
            (*idev)->Info2() );
        LogMessage("CreateManagedObjects", tempMsg, DEBUG_LOG_LEVEL);
        netobj = CreateNetDevice(
            (*idev)->Protocol(),
            (*idev)->NetDevice(),
            (*idev)->Info1(),
            (*idev)->Info2() );

        if (netobj != NULL)
        {
            netobj->Logger( appLogger );
            _netObjList.push_back(netobj);
        }
        voxobj = CreateVoxDevice(
            (*idev)->Protocol(),
            (*idev)->VoxDevice() );
        if (voxobj != NULL)
        {
            voxobj->Logger( appLogger );
            _voxObjList.push_back(voxobj);
        }
    }
}

```

```

        // Place the destination address in tempMsg
        sprintf( tempMsg, "%s", (*idev)->Info2());
        callFlowObj = new CDlgcCallflow( netobj, voxobj,
tempMsg, EventProcessor() );
        if ( callFlowObj != NULL )
        {
            callFlowObj->Logger( appLogger );
            _callFlowList.push_back( callFlowObj );
        }
        callFlowObj = NULL;
    }
}

```

Appendix H: Configuration File Sections

Most component test harness and demonstration programs use a configuration file to specify the hardware configuration and other project information. The configuration files are located in the working directory of each project that uses a configuration file. The naming convention of the configuration file is based on the project name. Some examples are shown in Table 2.

Project Name	Project Working Directory	Configuration File Name
ConfObj	\ConfObj	ConfObjCfg.txt
GcObj	\GcObj	GcObjCfg.txt
GcMsiGateway	\GcMsiGateway	GcMsiMgCfg.txt
HdsiObj	\HdsiObj	HdsiObjCfg.txt
ResponseSim	\ResponseSim	ResponseSimCfg.txt
VovidaSatellitePbx	\VovidaSatellitePbx	VovidaSpbxCfg.txt
VovidaSipGateway	\VovidaSipGateway	VovidaSipMgCfg.txt

Table 2. Project Name, Directory, and Configuration File Examples

The actual contents of a configuration file are project-specific. While there is consistency across most projects, each project may have slight variations based on the project requirements and functionality. The suggested way to determine how configuration file entries are used by a specific application is to review the source code for the specific project. Building on the example shown in Table 2, Table 3 lists the ADlgcApplication specializations important to look at to determine how the configuration file contents are used.

Project Name	ADlgcApplication Specialization
ConfObj	ConfObjApp.cpp
GcObj	GcObjApp.cpp
GcMsiGateway	CDlgcGcMsiGateway.cpp
HdsiObj	HdsiObjApp.cpp
ResponseSim	ResponseSimApp.cpp
VovidaSatellitePbx	CDlgcVovidaSatellitePbxApp.cpp
VovidaSipGateway	CDlgcVovidaSipGatewayApp.cpp

Table 3. Project and ADlgcApplication Specialization Examples

Projects are provided with a basic configuration file located in the project working directory. These files allow configuration of most of the implemented configuration file objects, or at least enough of them to get one started with a trial run of the project.

In general, projects that use a configuration file use a [DEVICES] section. This section is used to specify the application configuration.

Devices Configuration File Section

The [DEVICES] configuration file section is parsed by the `CDlgcDeviceConfiguration` class. Line items are passed to `CDlgcDeviceConfigurationItem` classes. The `CDlgcDeviceConfigurationItem` class provides a mechanism to specify device group, device type or protocol, network device name, voice device name, and two additional information fields. The additional information fields are used differently depending on the application or device type. This is the only section used by every framework project that includes devices and configuration file functionality. The additional information fields are commonly used to specify ANI, DNIS, Caller Identification, or a Call Information Group in the case of SIP/H.323 under Dialogic Global Call. All existing framework-based applications use the same convention to specify device type and protocol. The device type or protocol convention is listed in Table 4.

Protocol	Instantiated Class(es)
csp	CDlgcCSPStream
	CDlgcSipNet
dtiT1	CDlgcDtiT1Net
gc	CDlgcGcNet
gc:H323	CDlgcGcH323Net
gc:isdn	CDlgcGcNet
gc:lsi	CDlgcGcNet
gc:qsig	CDlgcGcQsigNet
gc:SIP	CDlgcGcSipNet
hdsi	CDlgcHdsiNet
imp	CDlgcIPVStream
	CDlgcSipNet
lsi	CDlgcLsiNet
msi	CDlgcMsiNet
sim	CDlgcSimNet

Table 4. Configuration File Protocol-To-Class Mapping

The protocol and device type naming convention is considered to be at the application level. The protocol-to-instantiated class mapping is determined in the application specialization `CreateNetDevice()` and `CreateVoxDevice()` functions, which are called by the `CreateManagedObjects()` function.

Voice Device Configuration File Section

The [VOXDEVICES] configuration file is parsed by the `CDlgcVoxDeviceConfiguration` class. Line items are passed to `CDlgcVoxDeviceConfigurationItem` classes. This section is used to place voice devices into groups for applications that perform dynamic routing of voice devices. The `CDlgcVoxConfigurationItem` class provides a mechanism to specify voice device groups and the device names that belong in the groups.

Conference Device Configuration File Section

The [CONFDEVICES] configuration file section is parsed by the CDlgcConfDeviceConfiguration class. Line items are passed to CDlgcConfDeviceConfigurationItem classes. The CDlgcConfDeviceConfigurationItem class provides a mechanism to specify conference device name, device type, and one extra information field. This section is used in the conferencing demonstration application.

Directory Configuration File Section

The [DIRECTORY] configuration file section is parsed by the CDlgcDirectory class. This section is used in all the gateway-based applications to specify extension to call destination mappings for directory look-up tables.

Routing Configuration File Section

The [ROUTING] configuration file section is parsed by the CDlgcGroupRouter class. This section is used in all the gateway-based applications to specify extension to device group mappings for call routing and call device/destination specifications.

Options Configuration File Section

The [OPTIONS] configuration file section is parsed by the CDlgcOptionsConfiguration class. Line items are passed to CDlgcOptionsConfigurationItem classes. The CDlgcOptionsConfigurationItem class uses a key/value combination to specify various application or device options. Examples of the use of this section include specifying the number of rings before no answer or options for starting the Dialogic® Global Call library.

Responses Configuration File Section

The [RESPONSES] configuration file section is parsed by the CDlgcResponseConfiguration class. Line items are passed to CDlgcResponseConfigurationItem classes. The response section is used by the response simulator to specify the type of pre/post connect call progress termination reason. The CDlgcResponseConfigurationItem class is used to specify group identifiers, ring delay, ring length, and response type.

Call Information Configuration File Section

The [CALLINFORMATION] configuration file section is parsed by the CDlgcCallInformationConfiguration class. Line items are passed to CDlgcCallInformationConfigurationItem classes. The CDlgcCallInformationConfigurationItem class provides a mechanism to specify call information group, call destination address, transmit audio codec, and receive audio codec for each line item. This section is used in some component test harnesses. The transmit and receive audio codec information specified in these line items is a group identifier that is detailed in the Codec configuration section.

Codec Configuration File Section

The [CODEC] configuration file section is parsed by the CDlgcCodecConfiguration class. Line items are passed to CDlgcCodecConfigurationItem classes. The CDlgcCodecConfigurationItem classes provide a mechanism to specify audio codec configurations by group name. Each group is used to specify the audio capability, capability type, audio direction, payload type, frames per packet, and voice activity detection flag information. This section is used to specify codec information for VoIP calls.

Appendix I: Custom Logging Implementations

This appendix lists the steps to create a new logging implementation.

1. Create the custom logging class as a subclass of ALogger. An example follows:

```
class CCustomLogger : public DlgcFW2::ALogger
{
    ...
public:
    // Required functions that are pure virtual in ALogger
    bool IsRunning();
    void HandleEvent(DlgcFW2::CDlgcEvent* event);
    int Start();
    int Stop();
    void LogMessage( char *logmsg,
        DlgcFW2::LogLevelType logLevel = DlgcFW2::DEBUG_LOG_LEVEL, const int msgID = -
        1);
    // Optional functions that are virtual in ALogger
    void SetLogLevel(DlgcFW2::LogLevelType logLevel);
    ...
}
```

2. Define the custom logger by adding an item to the LoggerType enum in DlgcLogDefs.h. An example follows:

```
enum LoggerType
{
    LegacyLogger,
    Log4cplusLogger,
    CustomLogger
};
```

3. Add the custom logger as an option for the CDlgcLogger constructor. An example follows:

```
CDlgcLogger::CDlgcLogger(LoggerType loggerType)
{
    switch(loggerType)
    {
    case LegacyLogger:
        _loggerImpl = new CLegacyLogger();
        break;
    case Log4cplusLogger:
        _loggerImpl = CLogger::Instance();
        break;
    // add the custom logger as an option
    case CustomLogger:
        _loggerImpl = new CCustomLogger();
        break;
    }
```

```

    default:
        _loggerImpl = new CLegacyLogger();
        break;
    }
}

```

4. Set the default logger in your application to be the custom logger. Do this before any loggers are created. An example follows :

```
DlgcFW2::CDlgcLogger::DefaultLoggerType(DlgcFW2::CustomLogger);
```

Acronyms

API	Application Programming Interface
BKM	Best-Known Method
BSD	Berkeley Software Distribution licensing model
CSF	Communication Services Framework
DCB	Dialogic Conference Bridge
DI	Dialogic Integrated
HDSI	High Density Station Interface
LSI	Loop Start Interface
MSI	Modular Station Interface
NDA	Non-Disclosure Agreement
RLT	Release Link Trunk
RTP	Realtime Transport Protocol
SIP	Session Initiation Protocol
SIT	Special Information Tone
SRL	Standard Runtime Library
TDM	Time Division Multiplex

For More Information

A Zip file containing the source code for this application note can be downloaded at <https://sourceforge.net/projects/commsvcfw>

Media Server Solution Recipe: Conferencing Applications — <http://www.dialogic.com/goto/?8708>

Curl Library — <http://curl.haxx.se>

Dialogic System Release Software — http://www.dialogic.com/products/ip_enabled/ip_boards.htm

Dialogic Developer Resource Center — <http://www.dialogic.com/drc/default.htm>

J RTP Library — <http://research.edm.uhasselt.be/~jori/page/index.php?n=CS.Jrtplib>

Log4cplus — <https://sourceforge.net/projects/log4cplus>

RadVision RTP — <http://www.radvision.com/>

Vovida — <http://www.vovida.org>

To learn more, visit our site on the World Wide Web at <http://www.dialogic.com>.

Dialogic Corporation

9800 Cavendish Blvd., 5th floor
Montreal, Quebec
CANADA H4M 2V9

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH PRODUCTS OF DIALOGIC CORPORATION OR ITS SUBSIDIARIES ("DIALOGIC"). NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Dialogic may make changes to specifications, product descriptions, and plans at any time, without notice.

Dialogic is a registered trademark of Dialogic Corporation. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

Microsoft, Windows, and Visual Studio are registered trademarks of Microsoft Corporation in the United States and/or other countries. Other names of actual companies and products mentioned herein are the trademarks of their respective owners. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement their concepts or applications, which licenses may vary from country to country.

This document discusses various open source products, systems and releases. Dialogic is neither responsible for your decision to use open source in connection with Dialogic products including without limitation those referred to herein, nor is Dialogic responsible for any present or future effects such usage might have, including without limitation effects on your products, business, or intellectual property rights.