

UNCLASSIFIED

SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION

APPENDIX C: CORE FRAMEWORK IDL



FINAL / 15 May 2006

Version 2.2.2

Prepared by:

JTRS Standards
Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS)
Space and Naval Warfare Systems Center San Diego
53560 Hull Street, San Diego CA 92152-5001

Distribution Statement A - Approved for public release; distribution is unlimited (15 May 2006)

REVISION SUMMARY

Version	Revision
1.0	Initial Release
1.1	Updated IDL to reflect SCAS changes made for v1.1; updated comments.
2.0	Incorporate approved Change Proposals, numbers 175, 245, 277, 278, 282, 311, 336, 345.
2.1	Incorporate approved Change Proposals, numbers 142, 175, 245, 277, 278, 282, 306, 311, 336, 345, 360.
2.2	Incorporate approved Change Proposals, numbers 138, 496, 509
2.2.1	Incorporate approved Change Proposals, numbers 15, 77, 26, 44, 45, 70, 74, 101, 102
2.2.2	Updated IDL Reduced comment text Incorporated Change Proposals SCA-CCM 44, 178, 202, and 210

TABLE OF CONTENTS

APPENDIX C CORE FRAMEWORK IDL	C-1
C.1 Core Framework IDL.....	C-1
C.2 PortTypes Module.....	C-35
C.3 StandardEvent Module.	C-36

APPENDIX C CORE FRAMEWORK IDL

The CF interfaces are expressed in CORBA IDL. Any IDL compiler for the target language of choice may compile the generated IDL.

The CF interfaces are contained in the CF CORBA module. Additionally, IDL modules are provided for interfaces that extend the *Port* interface by defining basic data sequence types. The StandardEvent CORBA Module contains the standard event types to be passed via the event service.

Attachment 1 to this appendix contains this same IDL.

C.1 CORE FRAMEWORK IDL

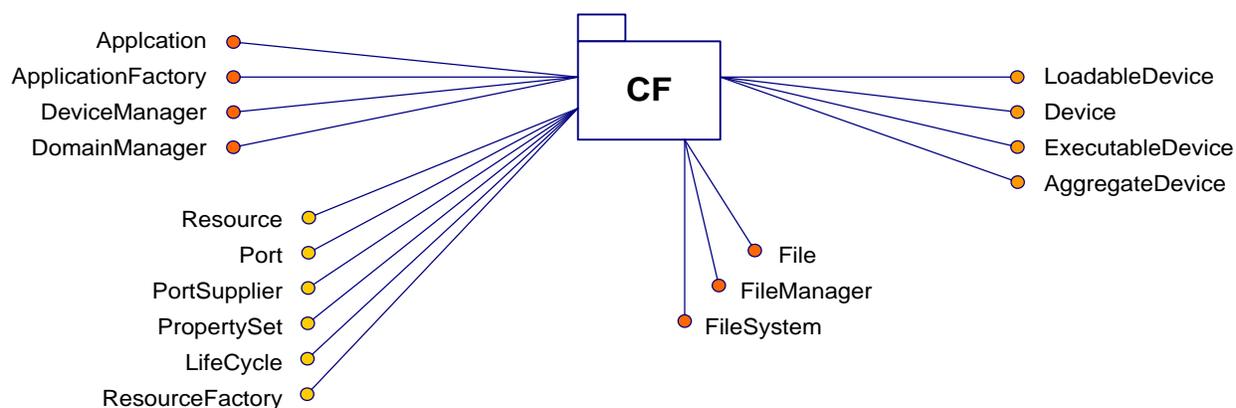


Figure C-1: CF CORBA Module

```
//Source file: CF.idl
```

```
#ifndef __CF_DEFINED
#define __CF_DEFINED
```

```
module CF {
```

```
    interface Device;
    interface File;
    interface Resource;
    interface Application;
    interface ApplicationFactory;
    interface DeviceManager;
```

```
    /* This type is a CORBA IDL struct type which can be used
    to hold any CORBA basic type or static IDL type. */
```

```
    struct DataType {
        /* The id attribute indicates the kind of value and
        type. The id can be an UUID string, an integer string, or a name
        identifier. */

        string id;

        /* The value attribute can be any static IDL type or
        CORBA basic type. */

        any value;
    };

    /* This exception indicates an invalid component profile
    error. */

    exception InvalidProfile {
    };

    /* The Properties is a CORBA IDL unbounded sequence of CF
    DataType(s), which can be used in defining a sequence of name and
    value pairs. */

    typedef sequence <DataType> Properties;

    /* This exception indicates an invalid CORBA object
    reference error. */

    exception InvalidObjectReference {
        string msg;
    };

    /* This type is a CORBA unbounded sequence of octets. */

    typedef sequence <octet> OctetSequence;

    /* This type defines a sequence of strings */

    typedef sequence <string> StringSequence;

    /* This exception indicates a set of properties unknown by
    the component. */

    exception UnknownProperties {
        CF::Properties invalidProperties;
    };

```

```
/* DeviceAssignmentType defines a structure that associates
a component with the device upon which the component is executing
on. */
```

```
struct DeviceAssignmentType {
    string componentId;
    string assignedDeviceId;
};
```

```
/* The IDL sequence, DeviceAssignmentSequence, provides a
unbounded sequence of 0..n of DeviceAssignmentType. */
```

```
typedef sequence <DeviceAssignmentType>
DeviceAssignmentSequence;
```

```
/* This enum is used to pass error number information in
various exceptions. Those exceptions starting with "CF_E" map to
the POSIX definitions. The "CF_" has been added to the POSIX
exceptions to avoid namespace conflicts. CF_NOTSET is not defined
in the POSIX specification. CF_NOTSET is an SCA specific value
that is applicable for any exception when the method specific or
standard POSIX error values are not appropriate.) */
```

```
enum ErrorNumberType {

    CF_NOTSET,
    CF_E2BIG,
    CF_EACCES,
    CF_EAGAIN,
    CF_EBADF,
    CF_EBADMSG,
    CF_EBUSY,
    CF_ECANCELED,
    CF_ECHILD,
    CF_EDEADLK,
    CF_EDOM,
    CF_EEXIST,
    CF_EFAULT,
    CF_EFBIG,
    CF_EINPROGRESS,
    CF_EINTR,
    CF_EINVAL,
    CF_EIO,
    CF_EISDIR,
    CF_EMFILE,
    CF_EMLINK,
    CF EMSGSIZE,
```

```
CF_ENAMETOOLONG,  
CF_ENFILE,  
CF_ENODEV,  
CF_ENOENT,  
CF_ENOEXEC,  
CF_ENOLCK,  
CF_ENOMEM,  
CF_ENOSPC,  
CF_ENOSYS,  
CF_ENOTDIR,  
CF_ENOTEMPTY,  
CF_ENOTSUP,  
CF_ENOTTY,  
CF_ENXIO,  
CF_EPERM,  
CF_EPIPE,  
CF_ERANGE,  
CF_EROFS,  
CF_ESPIPE,  
CF_ESRCH,  
CF_ETIMEDOUT,  
CF_EXDEV  
};
```

```
/* The InvalidFileName exception indicates an invalid file  
name was passed to a file service operation. The message provides  
information describing why the filename was invalid. */
```

```
exception InvalidFileName {  
    CF::ErrorNumberType errorNumber;  
    string msg;  
};
```

```
/* The CF FileException indicates a file-related error  
occurred. The message provides information describing the error.  
*/
```

```
exception FileException {  
    CF::ErrorNumberType errorNumber;  
    string msg;  
};
```

```
/* This type defines an unbounded sequence of Devices. */
```

```
typedef sequence <Device> DeviceSequence;
```

```
/* The AggregateDevice interface provides aggregate behavior
that can be used to add and remove Devices from a parent device.
This interface can be provided via inheritance or as a "provides
port". Child devices use this interface to add or remove
themselves from parent device when being created or torn-down. */
```

```
interface AggregateDevice {
```

```
    /* The readonly devices attribute contains a list of
devices that have been added to this device or a sequence length
of zero if the device has no aggregation relationships with other
devices. */
```

```
    readonly attribute CF::DeviceSequence devices;
```

```
    /* The addDevice operation provides the mechanism to
associate a device with another device. */
```

```
    void addDevice (
        in CF::Device associatedDevice
    )
        raises (CF::InvalidObjectReference);
```

```
    /* The removeDevice operation provides the mechanism to
disassociate
a device from another device. */
```

```
    void removeDevice (
        in CF::Device associatedDevice
    )
        raises (CF::InvalidObjectReference);
```

```
};
```

```
/* The FileSystem interface defines the CORBA operations to
enable remote access to a physical file system. */
```

```
interface FileSystem {
```

```
    /* This exception indicates a set of properties unknown
by the FileSystem object. */
```

```
    exception UnknownFileSystemProperties {
        CF::Properties invalidProperties;
    };
```

```
    /* This constant indicates file system size. */

    const string SIZE = "SIZE";

    /* This constant indicates the available space on the
file system. */

    const string AVAILABLE_SPACE = "AVAILABLE_SPACE";

    /* The FileType indicates the type of file entry. A file
system can have PLAIN or DIRECTORY files and mounted file systems
contained in a FileSystem. */

    enum FileType {
        PLAIN,
        DIRECTORY,
        FILE_SYSTEM
    };

    /* The FileInformationType indicates the information
returned for a file. */

    struct FileInformationType {
        string name;
        CF::FileSystem::FileType kind;
        unsigned long long size;
        CF::Properties fileProperties;
    };

    typedef sequence <FileInformationType>
FileInformationSequence;

    /* The CREATED_TIME_ID is the identifier for the created
time file property. */
    const string CREATED_TIME_ID = "CREATED_TIME";

    /* The MODIFIED_TIME_ID is the identifier for the
modified time file property. */
    const string MODIFIED_TIME_ID = "MODIFIED_TIME";

    /* The LAST_ACCESS_TIME_ID is the identifier for the
last access time file property. */
    const string LAST_ACCESS_TIME_ID = "LAST_ACCESS_TIME";

    /* The remove operation removes the file with the given
filename. */
```

```
void remove (
    in string fileName
)
    raises (CF::FileException,CF::InvalidFileName);

    /* The copy operation copies the source file with the
specified sourceFileName to the destination file with the
specified destinationFileName. */

void copy (
    in string sourceFileName,
    in string destinationFileName
)
    raises (CF::InvalidFileName,CF::FileException);

    /* The exists operation checks to see if a file exists
based on the filename parameter. */

boolean exists (
    in string fileName
)
    raises (CF::InvalidFileName);

    /* The list operation provides the ability to obtain a
list of files along with their information in the file system
according to a given search pattern. */

CF::FileSystem::FileInformationSequence list (
    in string pattern
)
    raises (CF::FileException,CF::InvalidFileName);

    /* The create operation creates a new File based upon
the provided file name and returns a File to the opened file. */

CF::File create (
    in string fileName
)
    raises (CF::InvalidFileName,CF::FileException);

    /* The open operation opens a file for reading or
writing based upon the input fileName. */
```

```
CF::File open (
    in string fileName,
    in boolean read_Only
)
    raises (CF::InvalidFileName,CF::FileException);

    /* The mkdir operation creates a file system directory
based on the directoryName given. */

void mkdir (
    in string directoryName
)
    raises (CF::InvalidFileName,CF::FileException);

    /* The rmdir operation removes a file system directory
based on the directoryName given. */

void rmdir (
    in string directoryName
)
    raises (CF::InvalidFileName,CF::FileException);

    /* The query operation returns file system information
to the calling client based upon the given fileSystemProperties'
ID. */

void query (
    inout CF::Properties fileSystemProperties
)
    raises (CF::FileSystem::UnknownFileSystemProperties);

};

    /* The File interface provides the ability to read and write
files residing within a distributed FileSystem. A file can be
thought of conceptually as a sequence of octets with a current
filePointer describing where the next read or write will occur. */

interface File {

    /* The IOException exception indicates an error occurred
during a read or write operation to a File. The message is
component-dependent, providing additional information describing
the reason for the error. */
```

```
exception IOException {
    CF::ErrorNumberType errorNumber;
    string msg;
};

/* This exception indicates the file pointer is out of
range based upon the current file size. */

exception InvalidFilePointer {
};

/* The readonly fileName attribute contains the file
name given to the FileSystem open/create operation. */

readonly attribute string fileName;

/* The readonly filePointer attribute contains the file
position where the next read or write will occur. */

readonly attribute unsigned long filePointer;

/* Applications require the read operation in order to
retrieve data from remote files. */

void read (
    out CF::OctetSequence data,
    in unsigned long length
)
    raises (CF::File::IOException);

/* The write operation writes data to the file
referenced. */

void write (
    in CF::OctetSequence data
)
    raises (CF::File::IOException);

/* The sizeof operation returns the current size of the
file. */

unsigned long sizeof ()
    raises (CF::FileException);

/* The close operation releases any OE file resources
associated with the component. */
```

```
void close ()
    raises (CF::FileException);

    /* The setFilePointer operation positions the file
pointer where next read or write will occur. */

void setFilePointer (
    in unsigned long filePointer
)
    raises
(CF::File::InvalidFilePointer,CF::FileException);

};

    /* A ResourceFactory can be used to create and tear down a
Resource. */

interface ResourceFactory {

    /* This exception indicates the resourceID does not
exist in the ResourceFactory. */

    exception InvalidResourceId {
};

    /* This exception indicates that the shutdown method
failed to release the ResourceFactory from the CORBA environment
because the Factory still contains Resources. The message is
component-dependent, providing additional information describing
why the shutdown failed. */

    exception ShutdownFailure {
        string msg;
};

    /* The CreateResourceFailure exception indicates that
the createResource operation failed to create the Resource. The
message is component-dependent, providing additional
information describing the reason for the error. */

    exception CreateResourceFailure {
        CF::ErrorNumberType errorNumber;
        string msg;
};
};
```

```
    /* The readonly identifier attribute contains the unique
    identifier for a ResourceFactory instance. */
```

```
    readonly attribute string identifier;
```

```
    /* The createResource operation provides the capability
    to create Resources in the same process space as the
    ResourceFactory or to return a Resource that has already been
    created. This behavior is an alternative approach to the Device's
    execute operation for creating a Resource. */
```

```
    CF::Resource createResource (
        in string resourceId,
        in CF::Properties qualifiers
    )
        raises (CF::ResourceFactory::CreateResourceFailure);
```

```
    /* In CORBA there is client side and server side
    representation of a Resource. This operation provides the
    mechanism of releasing the Resource in the CORBA environment on
    the server side when all clients are through with a specific
    Resource. The client still has to release its client side
    reference of the Resource. */
```

```
    void releaseResource (
        in string resourceId
    )
        raises (CF::ResourceFactory::InvalidResourceId);
```

```
    /* In CORBA there is client side and server side
    representation of a ResourceFactory. This operation provides the
    mechanism for releasing the ResourceFactory from the CORBA
    environment on the server side. The client has the responsibility
    to release its client side reference of the ResourceFactory. */
```

```
    void shutdown ()
        raises (CF::ResourceFactory::ShutdownFailure);
```

```
};
```

```
    /* Multiple, distributed FileSystems may be accessed through
    a FileManager. The FileManager interface appears to be a single
    FileSystem although the actual file storage may span multiple
    physical file systems. */
```

```
    interface FileManager : FileSystem {
```

```
    /* The Mount structure identifies the FileSystems
mounted within the FileManager. */

    struct MountType {
        string mountPoint;
        CF::FileSystem fs;
    };

    /* This type defines an unbounded sequence of mounted
FileSystems. */
    typedef sequence <MountType> MountSequence;

    /* This exception indicates a mount point does not exist
within the FileManager */
    exception NonExistentMount {
    };

    /* This exception indicates the FileSystem is a null
(nil) object reference. */
    exception InvalidFileSystem {
    };

    /* This exception indicates the mount point is already
in use in the FileManager. */
    exception MountPointAlreadyExists {
    };

    /* The mount operation associates a FileSystem with a
mount point (a directory name). */

    void mount (
        in string mountPoint,
        in CF::FileSystem file_System
    )
        raises
(CF::InvalidFileName,CF::FileManager::InvalidFileSystem,CF::FileMa
nager::MountPointAlreadyExists);

    /* The unmount operation removes a mounted FileSystem
from the FileManager whose mounted name matches the input
mountPoint name. */

    void unmount (
        in string mountPoint
    )
        raises (CF::FileManager::NonExistentMount);
```

```
    /* The getMounts operation returns the FileManager's
    mounted FileSystems. */

    CF::FileManager::MountSequence getMounts ();

};

    /* This interface provides operations for managing
    associations between ports. An application defines a specific
    Port type by specifying an interface that inherits the Port
    interface. */

    interface Port {

        /* This exception indicates one of the following errors
        has occurred in the specification of a Port association. */

        exception InvalidPort {
            unsigned short errorCode;
            string msg;
        };

        /* This exception indicates the Port is unable to accept
        any additional connections. */

        exception OccupiedPort {
        };

        /* The connectPort operation makes a connection to the
        component identified by the input parameters. The connectPort
        operation establishes only half of the association; therefore two
        calls are required to create a two-way association. A port may
        support several connections. */

        void connectPort (
            in Object connection,
            in string connectionId
        )
            raises (CF::Port::InvalidPort,CF::Port::OccupiedPort);

        /* The disconnectPort operation breaks the connection to
        the component identified by the input parameters. */
    }
};
```

```
void disconnectPort (
    in string connectionId
)
    raises (CF::Port::InvalidPort);

};

/* The LifeCycle interface defines the generic operations
for initializing or releasing instantiated component-specific data
and/or processing elements. */

interface LifeCycle {

    /* This exception indicates an error occurred during
component initialization. The messages provide additional
information describing the reason why the error occurred. */

    exception InitializeError {
        CF::StringSequence errorMessages;
    };

    /* This exception indicates an error occurred during
component releaseObject. The messages provide additional
information describing the reason why the error occurred. */

    exception ReleaseError {
        CF::StringSequence errorMessages;
    };

    /* The purpose of the initialize operation is to provide
a mechanism to set an object to an known initial state. */

    void initialize ()
        raises (CF::LifeCycle::InitializeError);

    /* The purpose of the releaseObject operation is to
provide a means by which an instantiated component may be torn
down. */

    void releaseObject ()
        raises (CF::LifeCycle::ReleaseError);

};
```

```
    /* The TestableObject interface defines a set of operations
that can be used to test component implementations. */
```

```
interface TestableObject {
```

```
    /* This exception indicates the requested testid for a
test to be performed is not known by the component. */
```

```
    exception UnknownTest {
};
```

```
    /* The runTest operation allows components to be
"blackbox" tested. This allows Built-In Tests to be implemented
which provides a means to isolate faults (both software and
hardware) within the system. */
```

```
    void runTest (
        in unsigned long testid,
        inout CF::Properties testValues
    )
    raises
```

```
(CF::TestableObject::UnknownTest, CF::UnknownProperties);
```

```
};
```

```
    /* The PropertySet interface defines configure and query
operations to access component properties/attributes. */
```

```
interface PropertySet {
```

```
    /* This exception indicates the configuration of a
component has failed (no configuration at all was done). The
message provides additional information describing the reason why
the error occurred. The invalid properties returned indicates the
properties that were invalid. */
```

```
    exception InvalidConfiguration {
        string msg;
        CF::Properties invalidProperties;
    };
```

```
    /* The PartialConfiguration exception indicates the
configuration of a Component was partially successful. The invalid
properties returned indicates the properties that were invalid.
*/
```

```

exception PartialConfiguration {
    CF::Properties invalidProperties;
};

/* The purpose of this operation is to allow id/value
pair configuration properties to be assigned to components
implementing this interface. */

void configure (
    in CF::Properties configProperties
)
    raises
(CF::PropertySet::InvalidConfiguration,CF::PropertySet::PartialCon
figuration);

/* The purpose of this operation is to allow a component
to be queried to retrieve its properties. */

void query (
    inout CF::Properties configProperties
)
    raises (CF::UnknownProperties);

};

/* The DomainManager interface is for the control and
configuration of the radio domain. */

interface DomainManager : PropertySet {

    /* This exception is raised when an Application
installation has not completed correctly. The message provides
additional information describing the reason for the error. */

    exception ApplicationInstallationError {
        CF::ErrorNumberType errorNumber;
        string msg;
    };
    exception ApplicationAlreadyInstalled {
    };

    /* This type defines an unbounded sequence of
Applications. */

    typedef sequence <Application> ApplicationSequence;

```

```
    /* This type defines an unbounded sequence of
ApplicationFactories. */

    typedef sequence <ApplicationFactory>
ApplicationFactorySequence;

    /* This type defines an unbounded sequence of
DeviceManagers. */
    typedef sequence <DeviceManager> DeviceManagerSequence;

    /* This exception indicates the application ID is
invalid. */
    exception InvalidIdentifier {
};

    /* This exception indicates the registering Device's
DeviceManager is not registered in the DomainManager. A Device's
DeviceManager has to be registered prior to a Device registration
to the DomainManager. */

    exception DeviceManagerNotRegistered {
};

    /* This exception is raised when an Application
uninstallation has not completed correctly. The message provides
additional information describing the reason for the error. */

    exception ApplicationUninstallationError {
        CF::ErrorNumberType errorNumber;
        string msg;
    };

    /* This exception indicates that an internal error has
occurred to prevent DomainManager registration operations from
successful completion. The message provides additional information
describing the reason for the error. */

    exception RegisterError {
        CF::ErrorNumberType errorNumber;
        string msg;
    };

    /* This exception indicates that an internal error has
occurred to prevent DomainManager unregister operations from
successful completion. The message provides additional information
describing the reason for the error. */
```

```
exception UnregisterError {
    CF::ErrorNumberType errorNumber;
    string msg;
};

/* This exception indicates that a registering consumer
is already connected to the specified event channel. */

exception AlreadyConnected {
};

/* This exception indicates that a DomainManager was not
able to locate the event channel. */

exception InvalidEventChannelName {
};

/* The NotConnected exception indicates that the
unregistering consumer was not connected to the specified event
channel. */

exception NotConnected {
};

/* The readonly domainManagerProfile attribute contains
a profile element with a file reference to the DomainManager
Configuration Descriptor (DMD) profile. */

readonly attribute string domainManagerProfile;

/* The deviceManagers attribute is read-only containing
a sequence of registered DeviceManagers in the domain. */

readonly attribute
CF::DomainManager::DeviceManagerSequence deviceManagers;

/* The applications attribute contains a list of
Applications that have been instantiated in the domain. */

readonly attribute CF::DomainManager::ApplicationSequence
applications;

/* The readonly applicationFactories attribute contains
a list with one ApplicationFactory per application (SAD file and
associated files) successfully installed. */
```

```
    readonly attribute
CF::DomainManager::ApplicationFactorySequence
    applicationFactories;

    /* The readonly fileMgr attribute contains the
DomainManager's FileManager. */

    readonly attribute CF::FileManager fileMgr;

    /* The readonly identifier attribute contains a unique
identifier for a DomainManager instance. The identifier is
identical to the domainmanagerconfiguration element id attribute
of the DomainManager's Descriptor (DMD) file. */

    readonly attribute string identifier;

    /* The registerDevice operation is used to register a
Device for a specific DeviceManager in the DomainManager's Domain
Profile. */

void registerDevice (
    in CF::Device registeringDevice,
    in CF::DeviceManager registeredDeviceMgr
)
    raises (CF::InvalidObjectReference,CF::InvalidProfile,
CF::DomainManager::DeviceManagerNotRegistered,
CF::DomainManager::RegisterError);

    /* The registerDeviceManager operation is used to
register a DeviceManager, its Device(s), and its Services. */

void registerDeviceManager (
    in CF::DeviceManager deviceMgr
)
    raises (CF::InvalidObjectReference,CF::InvalidProfile,
CF::DomainManager::RegisterError);

    /* The unregisterDeviceManager operation is used to
unregister a DeviceManager component from the DomainManager's
Domain Profile. A DeviceManager may be unregistered during run-
time for dynamic extraction or maintenance of the DeviceManager.
*/
```

```
void unregisterDeviceManager (
    in CF::DeviceManager deviceMgr
)
    raises (CF::InvalidObjectReference,
           CF::DomainManager::UnregisterError);

/* The unregisterDevice operation is used to remove a
device entry from the DomainManager for a specific DeviceManager.
*/

void unregisterDevice (
    in CF::Device unregisteringDevice
)
    raises (CF::InvalidObjectReference,
           CF::DomainManager::UnregisterError);

/* The installApplication operation is used to register
new application software in the DomainManager's Domain Profile. */

void installApplication (
    in string profileFileName
)
    raises (CF::InvalidProfile,CF::InvalidFileName,
           CF::DomainManager::ApplicationInstallationError,
           CF::DomainManager:: ApplicationAlreadyInstalled);

/* The uninstallApplication operation is used to
uninstall an application and its associated ApplicationFactory
from the DomainManager. */

void uninstallApplication (
    in string applicationId
)
    raises (CF::DomainManager::InvalidIdentifier,
           CF::DomainManager::ApplicationUninstallationError);

/* The registerService operation is used to register a
service for a specific DeviceManager with the DomainManager. */

void registerService (
    in Object registeringService,
    in CF::DeviceManager registeredDeviceMgr,
    in string name
)
    raises (CF::InvalidObjectReference,
           CF::DomainManager::DeviceManagerNotRegistered,
           CF::DomainManager::RegisterError);
```

```
    /* The unregisterService operation is used to remove a
service entry from the DomainManager for a specific DeviceManager.
*/
```

```
void unregisterService (
    in Object unregisteringService,
    in string name
)
    raises (CF::InvalidObjectReference,
           CF::DomainManager::UnregisterError);
```

```
    /* The registerWithEventChannel operation is used to
connect a consumer to a domain's event channel. */
```

```
void registerWithEventChannel (
    in Object registeringObject,
    in string registeringId,
    in string eventChannelName
)
    raises (CF::InvalidObjectReference,
           CF::DomainManager::InvalidEventChannelName,
           CF::DomainManager::AlreadyConnected);
```

```
    /* The unregisterFromEventChannel operation is used to
disconnect a consumer from a domain's event channel. */
```

```
void unregisterFromEventChannel (
    in string unregisteringId,
    in string eventChannelName
)
    raises (CF::DomainManager::InvalidEventChannelName,
           CF::DomainManager::NotConnected);
```

```
};
```

```
    /* The ApplicationFactory interface class provides an
interface to request the creation of a specific type of
Application in the domain. The Software Profile determines the type
of Application that is created by the ApplicationFactory. */
```

```
interface ApplicationFactory {
```

```
    /* This exception is raised when the parameter
DeviceAssignmentSequence contains one or more invalid Application
component-to-device assignment(s). */
```

```
exception CreateApplicationRequestError {
    CF::DeviceAssignmentSequence invalidAssignments;
};

/* This exception is raised when a create request is
valid but the Application is unsuccessfully instantiated due to
internal processing errors. The message provides additional
information describing the reason for the error. */

exception CreateApplicationError {
    CF::ErrorNumberType errorNumber;
    string msg;
};

/* This exception is raised when the input
initConfiguration parameter is invalid. */

exception InvalidInitConfiguration {
    CF::Properties invalidProperties;
};

/* The name attribute contains the name of the type of
Application that can be instantiated by the ApplicationFactory. */

readonly attribute string name;

/* The readonly identifier attribute contains the unique
identifier for an ApplicationFactory instance. The identifier is
identical to the softwareassembly element id attribute of the
ApplicationFactory's Software Assembly Descriptor file. */

readonly attribute string identifier;

/* This attribute contains the application software
profile that the factory uses when creating an application. The
string value contains a profile element with a file reference to
the SAD */

readonly attribute string softwareProfile;

/* The create operation is used to create an Application
within the system domain. */
```

```

    CF::Application create (
        in string name,
        in CF::Properties initConfiguration,
        in CF::DeviceAssignmentSequence deviceAssignments
    )
    raises
(CF::ApplicationFactory::CreateApplicationError,
    CF::ApplicationFactory::CreateApplicationRequestError,
    CF::ApplicationFactory::InvalidInitConfiguration);

};

    /* The PortSupplier interface provides the getPort operation
    for those objects that provide ports. */

    interface PortSupplier {

        /* This exception is raised if an undefined port is
        requested. */

        exception UnknownPort {
        };

        /* The getPort operation provides a mechanism to obtain
        a specific consumer or producer Port. A PortSupplier may contain
        zero-to-many consumer and producer port components. */

        Object getPort (
            in string name
        )
        raises (CF::PortSupplier::UnknownPort);

    };

    /* The Resource interface provides a common interface for
    the control and configuration of a software component. */

    interface Resource : LifeCycle, TestableObject, PropertySet,
    PortSupplier {

        /* This exception indicates that an error occurred
        during an attempt to start the Resource. The message provides
        additional information describing the reason for the error. */

```

```
exception StartError {
    CF::ErrorNumberType errorNumber;
    string msg;
};

/* The StopError exception indicates that an error
occurred during an attempt to stop the Resource. The message
provides additional information describing the reason for the
error. */

exception StopError {
    CF::ErrorNumberType errorNumber;
    string msg;
};

/* The readonly identifier attribute shall contain the
unique identifier for a resource instance. */

readonly attribute string identifier;

/* The start operation is provided to command a Resource
implementing this interface to start internal processing. */

void start ()
    raises (CF::Resource::StartError);

/* The stop operation is provided to command a Resource
implementing this interface to stop all internal processing. */

void stop ()
    raises (CF::Resource::StopError);
};

/* The Device interface defines additional capabilities and
attributes for any logical Device in the domain. */

interface Device : Resource {

    /* This exception indicates that the device is not
capable of the behavior being attempted due to the state the
Device is in. */

    exception InvalidState {
        string msg;
    };
};
```

```
    /* The InvalidCapacity exception returns the capacities
that are not valid for this device. */
```

```
    exception InvalidCapacity {
```

```
        /* The message indicates the reason for the invalid
capacity. */
```

```
        string msg;
```

```
        /* The invalid capacities sent to the
allocateCapacity operation.*/
```

```
        CF::Properties capacities;
```

```
    };
```

```
    /* This enumeration type defines a Device's
administrative states. The administrative state indicates the
permission to use or prohibition against using the Device. */
```

```
    enum AdminType {
```

```
        LOCKED,
        SHUTTING_DOWN,
        UNLOCKED
```

```
    };
```

```
    /* This enumeration type defines a Device's operational
states. The operational state indicates whether or not the object
is functioning. */
```

```
    enum OperationalType {
```

```
        ENABLED,
        DISABLED
```

```
    };
```

```
    /* This enumeration type defines the Device's usage
states. */
```

```
    enum UsageType {
```

```
        IDLE,
        ACTIVE,
        BUSY
```

```
    };
```

```
    /* The readonly usageState attribute contains the
Device's usage state The usageState indicates whether or not a
device is actively in use at a specific instant, and if so,
whether or not it has spare capacity for allocation at that
instant. */
```

```
    readonly attribute CF::Device::UsageType usageState;
```

```
    /* The administrative state indicates the permission to
use or prohibition against using the device. The adminState
attribute contains the device's admin state value. */
```

```
    attribute CF::Device::AdminType adminState;
```

```
    /* The operationalState attribute contains the device's
operational state. The operational state indicates whether or not
the device is functioning. */
```

```
    readonly attribute CF::Device::OperationalType
operationalState;
```

```
    /* The softwareProfile attribute is the XML description
for this logical Device. The softwareProfile attribute contains a
profile DTD element with a file reference to the SPD profile file.
*/
```

```
    readonly attribute string softwareProfile;
```

```
    /* The label attribute is the meaningful name given to a
Device. */
```

```
    readonly attribute string label;
```

```
    /* The compositeDevice attribute contains the object
reference of the AggregateDevice with which this Device is
associated or a nil CORBA object reference if no association
exists. */
```

```
    readonly attribute CF::AggregateDevice compositeDevice;
```

```
    /* The allocateCapacity operation provides the mechanism
to request and allocate capacity from the Device. */
```

```
    boolean allocateCapacity (
        in CF::Properties capacities
    )
        raises (CF::Device::InvalidCapacity,
CF::Device::InvalidState);

    /* The deallocateCapacity operation provides the
mechanism to return capacities back to the Device, making them
available to other users. */

    void deallocateCapacity (
        in CF::Properties capacities
    )
        raises (CF::Device::InvalidCapacity,
CF::Device::InvalidState);

};

    /* The Application interface provides for the control,
configuration, and status of an instantiated application in the
domain. */

    interface Application : Resource {

        /* The ComponentProcessIdType defines a type for
associating a component with its process ID. This type can be
used to retrieve a process ID for a specific component. */

        struct ComponentProcessIdType {
            string componentId;
            unsigned long processId;
        };

        /* The ComponentProcessIdSequence type defines an
unbounded sequence of components' process IDs. */

        typedef sequence <ComponentProcessIdType>
ComponentProcessIdSequence;

        /* The ComponentElementType defines a type for
associating a component with an element. */

        struct ComponentElementType {
            string componentId;
            string elementId;
        };
    };
};
```

```
    /* This type is an unbounded sequence of
ComponentElementTypes. */

    typedef sequence <ComponentElementType>
ComponentElementSequence;

    /* This attribute contains the list of components'
Naming Service Context within the Application for those components
using CORBA Naming Service. */

    readonly attribute
CF::Application::ComponentElementSequence
    componentNamingContexts;

    /* This attribute contains the list of components'
process IDs within the Application for components that are
executing on a device. */

    readonly attribute
CF::Application::ComponentProcessIdSequence
    componentProcessIds;

    /* The componentDevices attribute shall contain a list
of devices which each component either uses, is loaded on or is
executed on. Each component (componentinstantiation element in the
Application's software profile) is associated with a device. */

    readonly attribute CF::DeviceAssignmentSequence
componentDevices;

    /* This attribute contains the list of components' SPD
implementation IDs within the Application for those components
created. */

    readonly attribute
CF::Application::ComponentElementSequence
    componentImplementations;

    /* This attribute is the XML profile information for the
application. The string value contains a profile element with a
file reference to the SAD. */

    readonly attribute string profile;
```

```
    /* This name attribute contains the name of the created
Application. The ApplicationFactory interfaces's create operation
name parameter provides the name content. */
```

```
    readonly attribute string name;
};
```

```
    /* This interface extends the Device interface by adding
software loading and unloading behavior to a Device. */
```

```
interface LoadableDevice : Device {
```

```
    /* This LoadType defines the type of load to be
performed. The load types are in accordance with the code element
within the softpkg element's implementation element. */
```

```
    enum LoadType {

        KERNEL_MODULE,
        DRIVER,
        SHARED_LIBRARY,
        EXECUTABLE
    };
```

```
    /* The InvalidLoadKind exception indicates that the
Device is unable to load the type of file designated by the
loadKind parameter. */
```

```
    exception InvalidLoadKind {
};
```

```
    /* The LoadFail exception indicates that an error
occurred during an attempt to load the device. The message
provides additional information describing the reason for the
error. */
```

```
    exception LoadFail {
        CF::ErrorNumberType errorNumber;
        string msg;
    };
```

```
    /* The load operation provides the mechanism for loading
software on a specific device. The loaded software may be
subsequently executed on the Device, if the Device is an
ExecutableDevice. */
```

```
void load (
    in CF::FileSystem fs,
    in string fileName,
    in CF::LoadableDevice::LoadType loadKind
)
    raises (CF::Device::InvalidState,
           CF::LoadableDevice::InvalidLoadKind,
           CF::InvalidFileName, CF::LoadableDevice::LoadFail);

    /* The unload operation provides the mechanism to unload
software that is currently loaded. */

void unload (
    in string fileName
)
    raises (CF::Device::InvalidState, CF::InvalidFileName);

};

    /* This interface extends the LoadableDevice interface by
adding execute and terminate behavior to a Device. */

interface ExecutableDevice : LoadableDevice {

    /* The InvalidProcess exception indicates that a
process, as identified by the processID parameter, does not exist
on this device. The message provides additional information
describing the reason for the error. */

    exception InvalidProcess {
        CF::ErrorNumberType errorNumber;
        string msg;
    };

    /* This exception indicates that a function, as
identified by the input name parameter, hasn't been loaded on this
device. */

    exception InvalidFunction {
    };

    /* This type defines a process number within the system.
The process number is unique to the Processor operating system
that created the process. */

    typedef long ProcessID_Type;
```

```
    /* The InvalidParameters exception indicates that input
parameters are invalid for the execute operation. Each
parameter's ID and value must be a valid string type. The
invalidParms is a list of invalid parameters specified in the
execute operation. */
```

```
exception InvalidParameters {
    CF::Properties invalidParms;
};
```

```
    /* The InvalidOptions exception indicates the input
options are invalid on the execute operation. The invalidOpts is
a list of invalid options specified in the execute operation. */
```

```
exception InvalidOptions {
    CF::Properties invalidOpts;
};
```

```
    /* The STACK_SIZE_ID is the identifier for the
ExecutableDevice's execute options parameter. */
```

```
const string STACK_SIZE_ID = "STACK_SIZE";
```

```
    /* The PRIORITY_ID is the identifier for the
ExecutableDevice's execute options parameters. */
```

```
const string PRIORITY_ID = "PRIORITY";
```

```
    /* The ExecuteFail exception indicates that an attempt
to invoke the execute operation on a device failed. The message
provides additional information describing the reason for the
error. */
```

```
exception ExecuteFail {
    CF::ErrorNumberType errorNumber;
    string msg;
};
```

```
    /* The terminate operation provides the mechanism for
terminating the execution of a process/thread on a specific device
that was started up with the execute operation. */
```

```
void terminate (
    in CF::ExecutableDevice::ProcessID_Type processId
)
    raises (CF::ExecutableDevice::InvalidProcess,
           CF::Device::InvalidState);
```

```

    /* The execute operation provides the mechanism for
    starting up and executing a software process/thread on a device.
    */

```

```

    CF::ExecutableDevice::ProcessID_Type execute (
        in string name,
        in CF::Properties options,
        in CF::Properties parameters
    )
    raises (CF::Device::InvalidState,
           CF::ExecutableDevice::InvalidFunction,
           CF::ExecutableDevice::InvalidParameters,
           CF::ExecutableDevice::InvalidOptions,
           CF::InvalidFileName,
           CF::ExecutableDevice::ExecuteFail);

```

```
};
```

```

    /* The DeviceManager interface is used to manage a set of
    logical Devices and services. */

```

```

    interface DeviceManager : PropertySet, PortSupplier {

```

```

        /* This structure provides the object reference and name
        of services that have registered with the DeviceManager. */

```

```

        struct ServiceType {
            Object serviceObject;
            string serviceName;
        };

```

```

        /* This type provides an unbounded sequence of
        ServiceType structures for services that have registered with the
        DeviceManager. */

```

```

        typedef sequence <ServiceType> ServiceSequence;

```

```

        /* The deviceConfigurationProfile attribute contains the
        DeviceManager's profile, a profile element with a file reference
        to the DeviceManager's Device Configuration Descriptor (DCD)
        profile. */

```

```

        readonly attribute string deviceConfigurationProfile;

```

```
    /* The fileSys attribute contains the FileSystem
associated with this DeviceManager or a nil CORBA object reference
if no FileSystem is associated with this DeviceManager. */
```

```
    readonly attribute CF::FileSystem fileSys;
```

```
    /* The identifier attribute contains the instance-unique
identifier for a DeviceManager. The identifier is identical to
the deviceconfiguration element id attribute of the
DeviceManager's Device Configuration Descriptor (DCD) file. */
```

```
    readonly attribute string identifier;
```

```
    /* The label attribute contains the DeviceManager's
label. The label attribute is the meaningful name given to a
DeviceManager. */
```

```
    readonly attribute string label;
```

```
    /* The registeredDevices attribute contains a list of
Devices that have registered with this DeviceManager or a sequence
of length zero if no Devices have registered with the
DeviceManager. */
```

```
    readonly attribute CF::DeviceSequence registeredDevices;
```

```
    /* The registeredServices attribute contains a list of
Services that have registered with this DeviceManager or a
sequence of length zero if no Services have registered with the
DeviceManager. */
```

```
    readonly attribute CF::DeviceManager::ServiceSequence
registeredServices;
```

```
    /* The registerDevice operation provides the mechanism
to register a Device with a DeviceManager. */
```

```
    void registerDevice (
        in CF::Device registeringDevice
    )
        raises (CF::InvalidObjectReference);
```

```
    /* This operation unregisters a Device from a
DeviceManager. */
```

```
void unregisterDevice (
    in CF::Device registeredDevice
)
    raises (CF::InvalidObjectReference);

    /* The shutdown operation provides the mechanism to
    terminate a DeviceManager, unregistering it from the
    DomainManager. */

    void shutdown ();

    /* The registerService operation provides mechanisms to
    register a Service with a DeviceManager and its DomainManager. */

    void registerService (
        in Object registeringService,
        in string name
    )
        raises (CF::InvalidObjectReference);

    /* This operation provides mechanisms to unregister a
    Service from a DeviceManager and its DomainManager. */

    void unregisterService (
        in Object unregisteringService,
        in string name
    )
        raises (CF::InvalidObjectReference);

    /* The GetComponentImplementationId operation returns
    the SPD implementation ID that the DeviceManager interface used to
    create a component. */

    string GetComponentImplementationId (
        in string componentInstantiationId
    );

};

};

#endif
```

C.2 PORTTYPES MODULE.

This CORBA Module contains a set of unbundled CORBA sequence types based on CORBA types not in the CF CORBA Module.

```
//Source file: PortTypes.idl

#ifndef __PORTTYPES_DEFINED
#define __PORTTYPES_DEFINED

module PortTypes {

    /* This type is a unbounded sequence of booleans. */
    typedef sequence <boolean> BooleanSequence;

    /* This type is a unbounded sequence of characters. */
    typedef sequence <char> CharSequence;

    /* This type is a unbounded sequence of doubles. */
    typedef sequence <double> DoubleSequence;

    /* This type is a unbounded sequence of longlongs. */
    typedef sequence <long long> LongLongSequence;

    /* This type is a unbounded sequence of longs. */
    typedef sequence <long> LongSequence;

    /* This type is a unbounded sequence of shorts. */
    typedef sequence <short> ShortSequence;

    /* This type is a unbounded sequence of unsigned long longs.
*/
    typedef sequence <unsigned long long> UlongLongSequence;

    /* This type is a unbounded sequence of unsigned longs. */
    typedef sequence <unsigned long> UlongSequence;

    /* This type is a unbounded sequence of unsigned shorts. */
    typedef sequence <unsigned short> UshortSequence;

    /* This type is a unbounded sequence of floats. */
    typedef sequence <float> FloatSequence;

};

#endif
```

C.3 STANDARDEVENT MODULE.

The StandardEvent module contains the types necessary for a standard event producer to generate standard SCA events as depicted in Figure C-2.

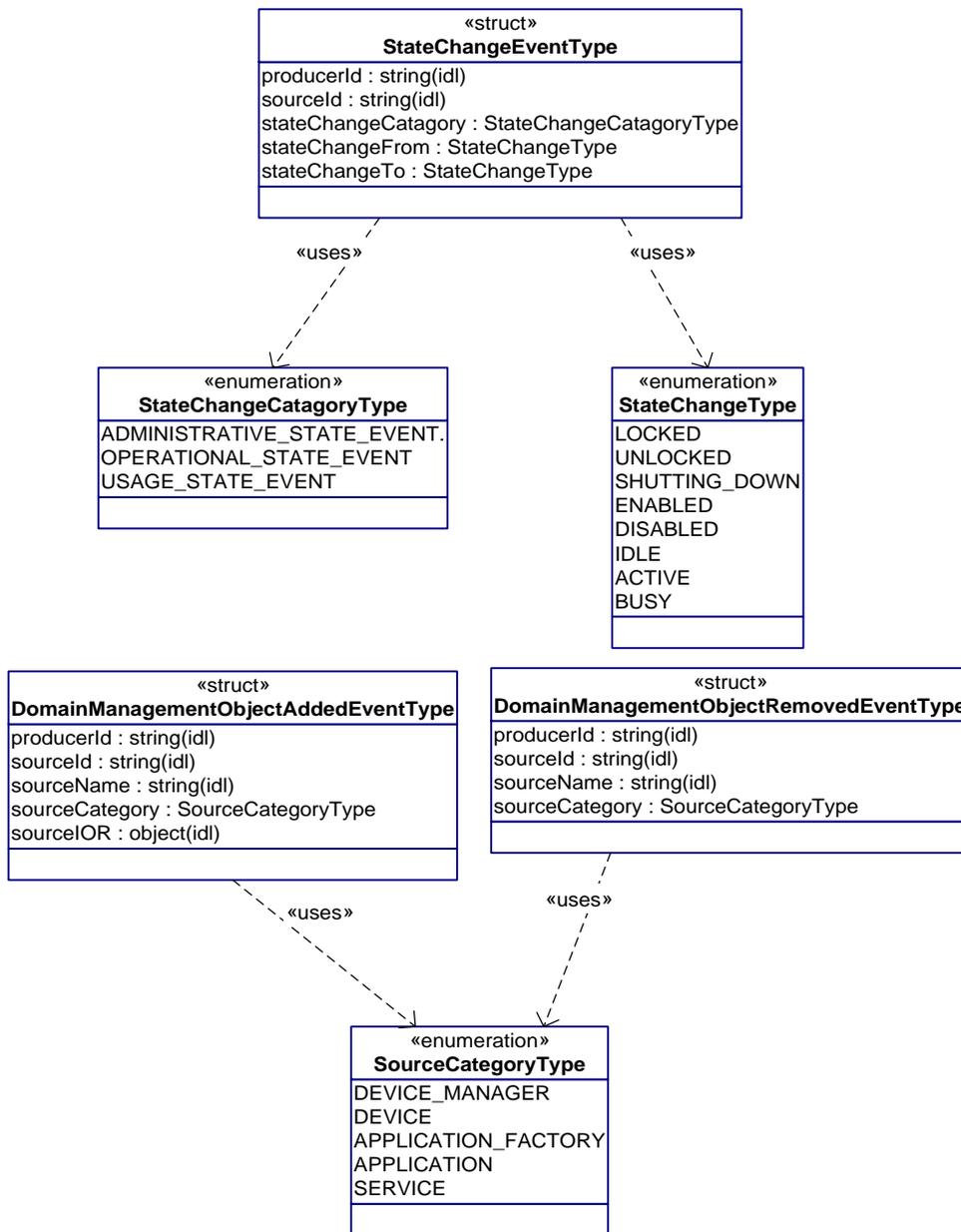


Figure C-2: StandardEvent Module

```

//Source file: StandardEvent.idl

#ifndef __STANDARDEVENT_DEFINED
#define __STANDARDEVENT_DEFINED

```

```
module StandardEvent {

    /* Type StateChangeCategoryType is an enumeration that is
    utilized in the StateChangeEventType. It is used to identify the
    category of state change that has occurred. */

    enum StateChangeCategoryType {

        ADMINISTRATIVE_STATE_EVENT,
        OPERATIONAL_STATE_EVENT,
        USAGE_STATE_EVENT
    };

    /* Type StateChangeType is an enumeration that is utilized in
    the StateChangeEventType. It is used to identify the specific
    states of the event source before and after the state change
    occurred. */

    enum StateChangeType {

        LOCKED,
        UNLOCKED,
        SHUTTING_DOWN,
        ENABLED,
        DISABLED,
        IDLE,
        ACTIVE,
        BUSY
    };

    /* Type StateChangeEventType is a structure used to indicate
    that the state of the event source has changed. The event producer
    will send this structure into an event channel on behalf of the
    event source. */

    struct StateChangeEventType {
        string producerId;
        string sourceId;
        StandardEvent::StateChangeCategoryType
stateChangeCategory;
        StandardEvent::StateChangeType stateChangeFrom;
        StandardEvent::StateChangeType stateChangeTo;
    };
};
```

```
/* Type SourceCategoryType is an enumeration that is utilized
in the DomainManagementObjectAddedEventType and
DomainManagementObjectRemovedEventType. Is used to identify the
type of object that has been added to or removed from the domain.
*/
```

```
enum SourceCategoryType {
```

```
    DEVICE_MANAGER,
    DEVICE,
    APPLICATION_FACTORY,
    APPLICATION,
    SERVICE
```

```
};
```

```
/* Type DomainManagementObjectRemovedEventType is a structure
used to indicate that the event source has been removed from the
domain. The event producer will send this structure into an event
channel on behalf of the event source. */
```

```
struct DomainManagementObjectRemovedEventType {
    string producerId;
    string sourceId;
    string sourceName;
    StandardEvent::SourceCategoryType sourceCategory;
};
```

```
/* Type DomainManagementObjectAddedEventType is a structure
used to indicate that the event source has been added to the
domain. The event producer will send this structure into an event
channel on behalf of the event source. */
```

```
struct DomainManagementObjectAddedEventType {
    string producerId;
    string sourceId;
    string sourceName;
    StandardEvent::SourceCategoryType sourceCategory;
    Object sourceIOR;
};
```

```
};
```

```
#endif
```